Handlers of Higher-Order Effectful Operations

ZHIXUAN YANG, Imperial College London, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom

This paper studies the design of programming languages with handlers of *higher-order effectful operations* – effectful operations that may take in computations as arguments or return computations as output. We present and analyse a core calculus with higher-kinded impredicative polymorphism, handlers of higher-order effectful operations, and possibly general recursion. The distinctive design choice of this calculus is that handlers are carried by lawless raw monads, while the computation judgements still satisfy the monadic laws judgementally. We present the calculus with a logical framework and give denotational models of the calculus using realizability semantics. We prove closed-term *canonicity* and *parametricity* (for the recursion-free fragment of the language) using synthetic Tait computability and a novel form of the $\top\top$-lifting technique.

1 Introduction

1.1 What Are Higher-Order Effects and Handlers?

1.1.1 Motivating Higher-Order Effects. One view of Plotkin and Pretnar [2009, 2013]'s *effect handlers* is that they are a language feature that empowers the programmer to freely extend the programming language with new syntax, and to interpret the syntax compositionally where needed using effect handlers. The syntax that is possible to be added is restricted to be in the form of a *generic operation*, which takes in a parameter of some type P and returns a result of some type A[1]. In a call-by-value calculus parameterised by a set Σ of operations, the typing rule for invoking an operation is simply

$$\frac{o : (P, A) \in \Sigma \qquad \Gamma \vdash_\Sigma v : P}{\Gamma \vdash_\Sigma o_v : A}$$

In the original calculus of Plotkin and Pretnar, the parameter type P and the return type A must be both *ground types*, such as integers or Booleans, which do not involve *computations* directly or indirectly. The reason for this restriction is that in the denotational model, the semantics of computations depends on the signatures of the operations, so if the signatures of the operations also involve computations, they would form a mutual recursion and greatly complicate the semantics.

Nonetheless, it is common for programming languages to have effectful operations that take in computations as arguments or return computations as results. Typical examples include exception catching $try\ \{p\}\ catch\ \{h\}$, which has as arguments the exception-raising program p and the exception-handling program h; parallel composition $par\ \{p\}\ \{q\}$, which takes in programs p and q to be executed in parallel; scoped resource acquisition $with\ r\ \{p\}$, which opens/closes a resource r, e.g. a file, before/after entering the scope of a program p. Such 'higher-order operations' may be implemented as effect handlers, and indeed exception handling was a primary motivation for Plotkin and Pretnar's proposal of effect handlers.

However, as analysed by Wu et al. [2014], implementing higher-order operations as handlers causes the loss of compositionality that is enjoyed by ordinary operations. For example, if the programmer decides to write a program q using exception throwing and catching, and if exception catching $try\ \{p\}\ catch\ \{h\}$ is implemented as effect handling $\textsc{handle}\ p\ \textsc{with}\ \{throw \mapsto h\}$ in the program q, then the programmer cannot give alternative semantics to exception catching in

[1]It is also common to formulate operations in the form of an *algebraic operation* $o_v(a.\ t)$ where the return value is bound as a variable in a 'continuation' t. Generic operations and algebraic operations are equivalent [Plotkin and Power 2003].

Authors' Contact Information: Zhixuan Yang, Department of Computing, Imperial College London, United Kingdom, s.yang20@imperial.ac.uk; Nicolas Wu, Department of Computing, Imperial College London, United Kingdom, n.wu@imperial.ac.uk.

q (e.g. after p throws an exception and h is executed, p gets *re-tried* again), because there is no mechanism for the programmer to reinterpret the effect handling construct HANDLE p WITH h.

To give better treatment of higher-order operations in the framework of algebraic effects and handlers, a number of authors have studied *higher-order algebraic effects* and their handlers [Bach Poulsen and van der Rest 2023; Frumin et al. 2024; Piróg et al. 2018; van den Berg and Schrijvers 2024; van den Berg et al. 2021; Wu et al. 2014; Yang et al. 2022; Yang and Wu 2023]. Note, however, the precise technical meanings of 'higher-order (algebraic) effects' vary in the cited papers, although they share the connotation of *effectful operations that may take in computations as arguments and/or return computations as results*.

1.1.2 The Quick-and-Dirty Approach. To begin with, there is *nothing* stopping the language designer simply removing the restriction on the parameter types P and return types A to be ground types, as far as only the type system and operational semantics are concerned. Indeed, most follow-up work on effect handlers does not have this restriction. When denotational semantics is concerned, this relaxation necessitates solving *mixed-variant* recursive equations between the semantics of computation types and operation signatures, which can be done using techniques from *domain theory*, as demonstrated by Bauer and Pretnar [2014] and Kiselyov et al. [2021] using classical domain theory, and more recently by Frumin et al. [2024] using synthetic guarded domain theory.

Simply removing the ground-type restriction in the type system is a 'quick-and-dirty' approach to higher-order algebraic effects, which does not reveal the inherent structure in higher-order operations. For example, suppose that $try\ \{p\}\ catch\ \{h\}$ is implemented as an operation *catch* with two computation parameters in this way, and *throw* is implemented as a nullary operation with no parameters as usual, a (deep) handler $\{catch\ p\ h\ k \mapsto \cdots ; throw \mapsto \cdots\}$ for them would accept *unhandled* computations p and h as arguments that may invoke *catch* and *throw*, in contrast to the continuation k for which *catch* and *throw* are already handled. This gives the programmer full flexibility on how to deal with the computation parameters p and h but undermines handlers as a *structured programming construct* that can be reasoned about effectively. After all, what makes (deep) handlers stand out among powerful control operators is their simple mental model to programmers – a native form of *catamorphism/fold* that replaces operation calls in the program being handled with the corresponding handler clauses.

1.1.3 The Structured Approach. To expose the inherent structure in higher-order operations more sharply, several authors have proposed a number of refined definitions of (subsets of) higher-order operations: *scoped operations* [Bosman et al. 2024; Matache et al. 2025; Piróg et al. 2018; Yang et al. 2022], *latent operations* [van den Berg et al. 2021], *hefty operations* [Bach Poulsen and van der Rest 2023], and the general frameworks by Wu et al. [2014], Yang and Wu [2023] and van den Berg and Schrijvers [2024]. Regardless of the technical differences in their proposals, the common idea is that the signature of a higher-order operation can take a different form from ordinary operations, and the carrier of a handler (usually called an algebra in this line of work) for a higher-order operation does not have to be a type anymore, and usually is a type constructor.

Taking scoped operations for example, in the formulation of Bosman et al. [2024], the signature of a scoped operation $s : (P, S)$ consists of two types: P still means that the operation s takes in a parameter of type P, but the type S no longer means that the operation returns a value of type S. Instead, it means that the operation s delimits S-many scopes; for example, S would be the two-element type for $try\ \{\cdots\}\ catch\ \{\cdots\}$ because it delimits two scopes. The typing rule for invoking scoped operations in a call-by-value programming language is

$$\frac{s : (P, S) \in \Sigma \qquad \Gamma \vdash_\Sigma v : P \qquad \Gamma, x : S \vdash_\Sigma c : A}{\Gamma \vdash_\Sigma s_v\ \{x.\ c\} : A} \tag{1}$$

where the term c represents S-many computations that the operation s takes in as arguments; these computations can return an *arbitrary* type A, which will also be the type of the whole operation call. This rule here is a simplified version of the calculus by Bosman et al. [2024], but it is the essence.

Because a call to a scoped operation $s : (P, S)$ is polymorphic in the return type A, a handler for the scoped operation s will also need to be carried by a type constructor M rather than just a type. A possible typing rule for handling a scoped operation s is

$$\frac{\Gamma \vdash_\Sigma p : A \qquad \Gamma, a : \alpha \vdash_\Sigma r : M\ \alpha \qquad \Gamma,\ v : P,\ c : S \to M\ \alpha \vdash_\Sigma h : M\ \alpha}{\Gamma \vdash_\Sigma \textsc{handle}\ p\ \textsc{with}\ \{\textsc{val}\ a \mapsto r; s\ v\ c \mapsto h\} : M\ A} \tag{2}$$

in which α is a free type variable. The practical difference between this approach and the quick-and-dirty approach mentioned above is that (1) the types P and S in the signature of a scoped operation can still be ground types, so the denotational semantics involves no mixed-variance recursive equations, and reasoning principles such as fusion laws are still available [Yang et al. 2022], and (2) a handler h of s always works with recursively handled computations $c : S \to M\ \alpha$, so the programmer's mental model of handling scoped operations s can still be a fold that replaces every return value $\textsc{val}\ x$ with the term r and every operation call to s with the corresponding handler clause h, including those calls nested in the scopes of other calls such as $s_v\ \{x.\boxed{s_u\ \{y. \cdots\}}\}$.

1.2 Interaction of Effect Handlers and Sequential Composition

1.2.1 A Problem with Sequential Composition. The rules above for invoking (1) and handling (2) scoped operations are still not the end of the story. Any effectful language should have a construct for sequential composition of computations, such as $\textsc{let}$-bindings. A question then is *how handling should interact with sequential composition.* Concretely, suppose that we have a computation

$$\frac{\Gamma \vdash_\Sigma s_v\{x.\ c\} : A \qquad \Gamma, a : A \vdash_\Sigma d : B}{\Gamma \vdash_\Sigma \textsc{let}\ a = s_v\{x.\ c\}\ \textsc{in}\ d : B}$$

that is the sequential composition of a scoped operation $s_v\{x.\ c\}$ followed by another computation d. What should be the result of applying the handling construct (2) to this computation? Applying the handler recursively to $s_v\{x.\ c\}$ and d would give us two terms of type

$$\Gamma \vdash_\Sigma \cdots : M\ A \qquad \text{and} \qquad \Gamma, a : A \vdash_\Sigma \cdots : M\ B \tag{3}$$

and our goal is to have a term of type $M\ B$. Now we have two ways to proceed:

(i) asking M to additionally come with a monadic bind $\gg\!= : M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta$, using which we can combine the recursively handled results (3) into $M\ B$.

(ii) modifying the handler rule (2) so that the handler of s takes in a continuation parameter

$$\frac{\cdots \qquad \Gamma,\ v : P,\ c : S \to M\ \alpha,\ \boxed{k : \alpha \to M\ \beta} \vdash_\Sigma h' : M\boxed{\beta}}{\Gamma \vdash_\Sigma \textsc{handle}\ p\ \textsc{with}\ \{\textsc{val}\ a \mapsto r; s\ v\ c\ \boxed{k} \mapsto h'\} : M\ A}$$

and the recursively handled result $\Gamma, a : A \vdash_\Sigma \cdots : M\ B$ of d is supplied as the continuation parameter k to the handler when handling the operation $s_v\{x.\ c\}$.

To programming language designers familiar with effect handlers, approach (ii) may appear as the more natural one, since there is already a similar continuation parameter for handlers of generic/algebraic operations in Plotkin and Pretnar's design. Indeed, Bosman et al. [2024]'s calculus of scoped effects and handlers follows this approach, but let us not commit to this choice too quickly; instead, let us first analyse the connections and differences between these two approaches.

From an algebraic point of view, the difference in these two approaches lies in their views of the *universal property* of effectful computations. Crudely speaking, approach (i) views computations

with operation s as the initial object among *monads* M equipped with operations of type $\forall\alpha.\ P \to (S \to M\ \alpha) \to M\ \alpha$, whereas approach (ii) views computations with s as the initial object among plain type constructor M equipped with operations of type $\forall\alpha\ \beta.\ P \to (S \to M\ \alpha) \to (\alpha \to M\ \beta) \to M\ \beta$ and $\forall\alpha.\ \alpha \to M\ \alpha$.

1.2.2 An Analogy to Lists. The contrast between these two views is analogous to the following more familiar situation. The type of lists $[A]$ of elements of type A has the universal property that $[A]$ together with the empty list $nil : 1 \to [A]$ and $cons : A \to [A] \to [A]$ is initial among all types B equipped with functions $1 \to B$ and $A \to B \to B$. But the type $[A]$ has another universal property: first of all, it can be equipped with a monoid structure of $nil : 1 \to [A]$ and list concatenation $+\!\!+ : [A] \to [A] \to [A]$, and this monoid with the function $(\lambda x.\ cons\ x\ nil) : A \to [A]$ has the property of being the free monoid over the type A (which means that for every monoid M with a function $f : A \to M$, there is a unique monoid homomorphism $h : [A] \to M$ such that $h\ (cons\ x\ nil) = f\ x$ for all $x \in A$). Therefore the same type $[A]$ can be equipped with different algebraic structures, giving rise to different universal properties. It is pointless to ask which of the universal properties is 'the correct one' for the type $[A]$ *per se*. The right question should be – which class of algebraic structures are we interested in when using $[A]$, monoids M with functions $A \to M$ or types B with functions $1 \to B$ and $A \to B \to B$?

Similarly, the right question to ask about approaches (i) and (ii) above should be – when programming with scoped effects, are we interested in (i) monads M with operations of type $\forall\alpha.\ P \to (S \to M\ \alpha) \to M\ \alpha$ or (ii) type constructors M equipped with operations of type $\forall\alpha\ \beta.\ P \to (S \to M\ \alpha) \to (\alpha \to M\ \beta) \to M\ \beta$ and $\forall\alpha.\ \alpha \to M\ \alpha$?

1.2.3 Sequential Composition Are Operations. To this question, we advocate for the answer (i). Our rationale is that sequential composition *ought to be* an operation for a notion of computation, rather than only a meta-level operation that the initial object 'accidentally' support. This view is supported by the fact that in practice it is not uncommon to consider equations involving both effectful operations and sequential composition. For example, in the study of process algebra [Bergstra and Klop 1985], the distributivity equations

$$(p + q); r = (p; r) + (q; r) \qquad\qquad p; (q + r) = (p; q) + (p; r)$$

between nondeterministic choice and sequential composition are usually considered (and most models of concurrency satisfy the former but not the latter). If sequential composition is not an operation in the algebraic theory of the effect of concurrency, these two equations would not be expressible, and it would be meaningless to ask whether a handler of concurrency satisfies these two equations if we followed approach (ii).

The point that we are raising here holds regardless of whether the programming language formally checks equations on effectful operations – even in simply typed programming languages, programmers usually have equations informally in their mind and reason about effectful programs with these equations [Gibbons and Hinze 2011]. Also, our point is not specifically about higher-order operations either: even for algebraic/generic operations like nondeterministic choice $p + q$, we may already want to consider equations about the interaction of the effectful operations with sequential composition. And this is impossible in standard algebraic effects [Plotkin and Power 2002] because sequential composition is not an operation in the theory of the effect, but only an operation that the free algebras determined by the algebraic theory 'happens to have'.

1.2.4 A Problem with Laws. As the final twist of our discussion of higher-order effect handlers in this section, we discuss how we should deal with the monadic laws. As we advocated above, our handlers of higher-order algebraic effects will be carried by monads, which are type constructors

$M : \mathit{Type} \to \mathit{Type}$ with operations for returning $\mathit{ret} : \forall\alpha.\ \alpha \to M\ \alpha$ and sequential composition $\gg\!= : \forall\alpha\ \beta.\ M\ \alpha \to (\alpha \to M\ \beta) \to M\ \beta$ satisfying the monadic laws

$$\mathit{ret}\ a \gg\!= k = k\ a \qquad m \gg\!= \mathit{ret} = m \qquad (m \gg\!= k) \gg\!= k' = m \gg\!= (\lambda x.\ k\ x \gg\!= k') \tag{4}$$

asserting that *ret* is the left and right identity of $\gg\!=$ and $\gg\!=$ is associative. If our effectful programming language has dependent types – in particular, identity types – we can demand every handler to come with proofs of the equations (4), and the type checker can check the proofs mechanically. But if our languages does not have dependent types, it will be impossible for us to mechanically check if these laws are satisfied. In this case our programming language can only ask handlers to be carried by 'raw monads' $\langle M, \mathit{ret}, \gg\!=\rangle$ that do not necessarily satisfy these laws. Therefore, in the absence of dependent types, we as the language designer will not enforce the programmer to supply lawful monads, but there is a closely related but different question:

Question 1.1. When handlers of computations are carried by raw monads not necessarily satisfying monadic laws (4), can we still make LET-bindings and VAL-returning of the computation judgements satisfy the following monadic laws judgementally?

$$(\textsc{let}\ x = \textsc{val}\ a\ \textsc{in}\ k)\ \equiv\ k\ a \qquad (\textsc{let}\ x = m\ \textsc{in}\ \textsc{val}\ x)\ \equiv\ m \tag{5}$$

$$(\textsc{let}\ y = (\textsc{let}\ x = m\ \textsc{in}\ k\ x)\ \textsc{in}\ k'\ y)\ \equiv\ (\textsc{let}\ x = m\ \textsc{in}\ (\textsc{let}\ y = k\ x\ \textsc{in}\ k'\ y)) \tag{6}$$

It might seem that the answer to the this question would be unavoidably negative, since the computations of our language are handled into raw monads that do not necessarily satisfy the monadic laws, and syntax shall only satisfy the equations that are satisfied in *all* semantic model. This would be rather unfortunate because these laws are arguably the most fundamental algebraic properties for a notion of computation, and they are needed by programmers to reason about effectful programs and compilers to do optimisation, for example, to rewrite a computation that invokes no effectful operations to the form $\textsc{val}\ v$ of returning a pure value v.

1.3 Contributions of This Paper

In this paper we show that the answer to Question 1.1 is actually positive, provided that we are willing to *not* have the commutativity of handlers and LET-bindings:

$$\textsc{handle}\ (\textsc{let}\ x = m\ \textsc{in}\ k\ x)\ \textsc{with}\ h\ \equiv\ (\textsc{handle}\ m\ \textsc{with}\ h) \gg\!= \lambda x.\ \textsc{handle}\ k\ x\ \textsc{with}\ h$$

We present a core calculus for higher-order effects and handlers that we call System $\mathrm{F}^{\mathrm{ha}}_{\omega}$ (Section 2). This calculus extends Girard's [1986] System F_{ω} with higher-order effects in the fine-grain call-by-value style. Specifically, the signature of a higher-order algebraic effect is given as a higher-order functor $H : (\mathit{Type} \to \mathit{Type}) \to (\mathit{Type} \to \mathit{Type})$ in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ following Yang and Wu [2023] and van den Berg and Schrijvers [2024]'s categorical frameworks. Every signature H has a corresponding judgement of computations that supports LET-bindings, VAL-returning, invoking H-operations, and being handled with raw monads equipped with H-operations.

As promised, the computation judgements of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ satisfy the equations (5) and (6), and we show that the design of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ 'works' by establishing the following meta-theoretic properties about $\mathrm{F}^{\mathrm{ha}}_{\omega}$:

(1) In Section 3, a denotational model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is given based on realizability, establishing the *consistency* of the equational theory of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ (Theorem 3.2) and provides a way to translate $\mathrm{F}^{\mathrm{ha}}_{\omega}$ terms to untyped computational models such as λ-calculus or Turing machines (Theorem 3.3). The key idea in the construction of this model is to use continuation-passing-style (CPS) transformation to reconcile the mismatch between lawful computation judgements and lawless handlers in Question 1.1. An extension of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ with general recursion is also considered and the realizability model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is extended to support recursion (Theorem 3.5) using techniques from *synthetic domain theory* [Longley and Simpson 1997].

(2) In Section 4, a logical relation model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is constructed using the method of *synthetic Tait computability* [Sterling 2021] and a novel version $\top\top$*-lifting* [Lindley and Stark 2005]. From this model we obtain *canonicity* (Theorem 4.1) and *parametricity* (Remark 4.13) of closed $\mathrm{F}^{\mathrm{ha}}_{\omega}$-terms, which also imply the *adequacy* of the realizability model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ (Corollary 4.12).

Moreover, a methodological character of this paper is its heavy use of modern type-theoretic and category-theoretic tools to study the language $\mathrm{F}^{\mathrm{ha}}_{\omega}$. Although this paper is too short to serve as a fully satisfactory introduction to these tools, we hope that there is still some pedagogical value in this paper by showing how these tools are used coherently to present and analyse a polymorphic programming language that is not too simplistic or complicated. We hope this can contribute to making these powerful abstract tools more accessible to working programming language theorists.

2 A Core Calculus for Higher-Order Effect Handlers

In this section we present the type theory that we call System $\mathrm{F}^{\mathrm{ha}}_{\omega}$, an extension of System F_{ω} with handlers of higher-order effectful operations. Instead of in the more traditional way of defining the calculus by a grammar and typing rules, in this paper we use a *logical framework* to present $\mathrm{F}^{\mathrm{ha}}_{\omega}$. Logical frameworks are type theories designed for defining other logics, and they usually provide useful general results for theories definable in the framework. In particular, the logical framework that we will use frees us from manipulating variables and substitutions manually and provides a notion of *semantic models* of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ automatically.

The structure of this section is as follows. In Section 2.1, we briefly introduce the logical framework that we will use. In Section 2.2, we define Girard's [1986] System F_{ω} in the logical framework, which is going to be the basis of our language $\mathrm{F}^{\mathrm{ha}}_{\omega}$. In Section 2.3, we add computational judgements to $\mathrm{F}^{\mathrm{ha}}_{\omega}$, giving us $\mathrm{F}^{\mathrm{ha}}_{\omega}$. Finally, in Section 2.4, we consider an extension of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ with general recursion.

2.1 A Logical Framework

The logical framework (LF) that we will use is the one informally introduced by Sterling [2021] in his PhD thesis, which is called LccLF by Yang [2025] in his more formal treatment of this framework. This LF has been used by a number of authors in the study of various type theories [Grodin et al. 2024; Niu et al. 2022; Sterling and Angiuli 2021; Sterling and Harper 2021, 2022]. We aim to be self-contained about the LF in this paper, but our introduction is unavoidably rather terse and we refer the reader to Sterling [2021, Chapter 1] and Yang [2025] if needed.

Language 2.1. The logical framework LccLF is a dependent type theory with

- a universe $\mathbb{J}$ that is closed under the type formers of extensional Martin-Löf type theory (the unit type 1, Σ-types, Π-types, extensional equality types $x = y$);
- outside the universe $\mathbb{J}$, the LF has the unit type 1, Σ-types, and *restricted* Π-types $\Pi\ A\ B$ where the domain type A must be in $\mathbb{J}$ (so $\Pi\ \mathbb{J}\ (\lambda\_.\ \mathbb{J})$ will not be a valid type).

Notation 2.2. We adopt some notation similar to Agda [Agda Developers 2025] when working with dependent type theories. Dependent function types are written as $(a : A) \longrightarrow B$ where a may occur in B, or $A \longrightarrow B$ when B does not depend on $a : A$. Iterated Σ-types are written as records of fields with labels. Implicit function types $\{a : A\} \longrightarrow B$ are used when the arguments can inferred. Things whose names are irrelevant are denoted by the wildcard '_'.

The way to define an object logic/type theory in the LF is to write a *signature*, which is a sequence of variable declarations in the LF, where the type of each variable may depend on the preceding declarations (so formally, a signature is exactly a context in the LF). The idea is the *judgements-as-types* principle as follows [Harper et al. 1993; Martin-Löf 1987]:

(1) *Judgement forms* of the object theory are declared as LF-functions $A \to \mathbb{J}$ into the universe $\mathbb{J}$. For example, the judgement form of a proposition being true, traditionally written as $\vdash P$ true, is declared as $true : prop \to \mathbb{J}$, assuming some $prop : \mathbb{J}$ is already declared.
(2) *Inference rules* for object-theory judgements are declared as LF-functions; for example, the declaration $and\text{-}intro : (P, Q : prop) \to true\ P \to true\ Q \to true\ (P \wedge Q)$ says that the judgement of $P \wedge Q$ being true can be derived from both P and Q being true.
(3) *Judgemental equalities* of the object-theory can be treated in two ways: (i) they can be declared as judgements in $\mathbb{J}$ just like other judgements, or (ii) they can be declared using the (extensional) equality types of the LF. Logical frameworks following the former approach are sometimes called *syntactic logical frameworks*, and those following the latter are called *semantic logical frameworks*; see Harper [2016] and Sterling [2021, §0.1.2.2] for a comparison between them. We will follow the semantic approach, which has the advantage that there is no need to have the tedious congruence rules for the judgemental equalities w.r.t. all constructs of the object theory, since equality types in the LF are always respected.

These points will be demonstrated concretely in the example of defining Girard's [1972; 1986] System F_ω in the LF below. Compared to the traditional 'gamma-and-turnstile' presentation, there are three advantages of using LccLF to present our language $\mathrm{F}^{\mathrm{ha}}_\omega$:

(1) It provides a compact type-theoretic notation to present the rules of $\mathrm{F}^{\mathrm{ha}}_\omega$, and by using *higher-order abstract syntax* (HOAS), standard components of $\mathrm{F}^{\mathrm{ha}}_\omega$ such as contexts and substitutions can be dealt with automatically.
(2) It provides a notion of *models* of $\mathrm{F}^{\mathrm{ha}}_\omega$ in any locally cartesian closed category (LCCC) $\mathscr{C}$. By using the internal language of $\mathscr{C}$, models of $\mathrm{F}^{\mathrm{ha}}_\omega$ can be defined in a type-theoretic manner.
(3) It provides a *classifying category* for $\mathrm{F}^{\mathrm{ha}}_\omega$, which is an LCCC $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_\omega$ such that models of $\mathrm{F}^{\mathrm{ha}}_\omega$ in any LCCC $\mathscr{C}$ are equivalent to LCCC-functors $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_\omega \to \mathscr{C}$. Applying category-theoretic tools to the category $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_\omega$, such as Yoneda embedding and Artin gluing, we can do logical-relation proofs for $\mathrm{F}^{\mathrm{ha}}_\omega$ in a convenient type-theoretic language.

2.2 The Signature of System F_ω

In the following, we present the signature of $\mathrm{F}^{\mathrm{ha}}_\omega$ in two steps: we first define Girard's [1972; 1986] System F_ω in the LF (Language 2.1), and in the next section we bring in computations.

2.2.1 Kinds. System F_ω has the following declarations for *kinds*:

$$ki : \mathbb{J} \qquad el : ki \to \mathbb{J} \qquad ty : ki \qquad \_\Rightarrow_k\_ : ki \to ki \to ki \tag{F_ω-1}$$

where we have a judgement ki for kinds, a family of judgements el for elements of kinds, a base kind $ty : ki$ whose elements will be *types*, and function kinds $k_1 \Rightarrow_k k_2$. These declarations correspond to the following things in the traditional presentation: the declaration $ki : \mathbb{J}$ corresponds to a judgement '$\Gamma \vdash \cdots$ kind' of something being a kind; The LF-type $el\ k : \mathbb{J}$ for some kind $k : ki$ corresponds to the judgement '$\Gamma \vdash \cdots : k$' of something being an element of a kind; the two declarations ty and $\_\Rightarrow_k\_$ correspond to two inference rules for constructing kinds:

$$\frac{}{\Gamma \vdash ty\ \mathrm{kind}} \qquad\qquad \frac{\Gamma \vdash k_1\ \mathrm{kind} \qquad \Gamma \vdash k_2\ \mathrm{kind}}{\Gamma \vdash k_1 \Rightarrow_k k_2\ \mathrm{kind}}$$

Elements of the base kind $ty : ki$ include a unit type *unit*, a two-element type *bool*, function types $A \Rightarrow_t B$, and impredicative polymorphic function types $\bar{\forall}\ k\ A$ where k can be of any kind:

$$unit : el\ ty \qquad bool : el\ ty \qquad \_\Rightarrow_t\_ : el\ ty \to el\ ty \to el\ ty$$
$$\bar{\forall} : (k : ki) \to (el\ k \to el\ ty) \to el\ ty \tag{F_ω-2}$$

We use the $\forall$ symbol with a bar for the polymorphic function type so that we will not confuse it with meta-level universal quantification later.

Elements of function kinds are specified using *higher-order abstract syntax* (HOAS) via an isomorphism to functions in the LF:

$$\Rightarrow_k\text{-}iso : \{k_1, k_2 : ki\} \to el\ (k_1 \Rightarrow_k k_2) \cong (el\ k_1 \to el\ k_2) \tag{F_ω-3}$$

where the type $\cong$ of isomorphisms between two LF-types A and B is the following record type:

$$\begin{aligned}&\textsc{record}\ A \cong B\ \textsc{where}\\ &\quad fwd : A \to B\\ &\quad bwd : B \to A\\ &\quad \_ \quad : (a : A) \to bwd\ (fwd\ a) = a\\ &\quad \_ \quad : (b : B) \to fwd\ (bwd\ b) = b\end{aligned}$$

Let us unpack the declaration (F_ω-3) a bit and see how it corresponds to the more traditional presentation. Given two kinds $k_1, k_2 : ki$, the record $el\ (k_1 \Rightarrow_k k_2) \cong (el\ k_1 \to el\ k_2)$ consists of four fields: the forward-direction function $el\ (k_1 \Rightarrow_k k_2) \to (el\ k_1 \to el\ k_2)$ says that whenever we have an element of the kind $k_1 \Rightarrow_k k_2$ and an element of the kind k_1, we can construct an element of the kind k_2. This corresponds to the following rule in the traditional presentation:

$$\frac{\Gamma \vdash F : k_1 \Rightarrow_k k_2 \qquad \Gamma \vdash A : k_1}{\Gamma \vdash F\ A : k_2}$$

The backward direction $(el\ k_1 \to el\ k_2) \to el\ (k_1 \Rightarrow_k k_2)$ takes in an LF-function as its argument. In HOAS, LF-functions correspond to adding new variables to the context of the object theory, and applications of LF-functions correspond to substitutions in the object theory, so the traditional counterpart of the backward direction is

$$\frac{\Gamma, \alpha : k_1 \vdash F : k_2}{\Gamma \vdash \lambda\alpha.\ F : k_1 \Rightarrow_k k_2}$$

The other two fields of the isomorphism $el\ (k_1 \Rightarrow_k k_2) \cong (el\ k_1 \to el\ k_2)$ assert that these two directions are mutual inverses, and this is exactly η- and β-rules of $k_1 \Rightarrow_k k_2$:

$$\frac{\Gamma \vdash F : k_1 \Rightarrow_k k_2}{\Gamma \vdash (\lambda A.\ F\ A) \equiv F : k_1 \Rightarrow_k k_2} \qquad \frac{\Gamma, \alpha : k_1 \vdash F : k_2 \qquad \Gamma \vdash \alpha : k_1}{\Gamma \vdash (\lambda\alpha.\ F)\ A \equiv F[A/\alpha] : k_2}$$

In summary, we have used a single LF-declaration (F_ω-3) to express what would be four rules in the traditional presentation of functions. We will use this technique a lot in our specification of F_ω^{ha}.

2.2.2 Types. For terms of types, there is a judgement $tm : el\ ty \to \mathbb{J}$. Terms of (polymorphic) function types and the unit type are still specified by HOAS:

$$\begin{aligned} tm : el\ ty \to \mathbb{J} \qquad & unit\text{-}iso : tm\ unit \cong 1\\ \Rightarrow_t\text{-}iso : \{A, B : el\ ty\} \to tm\ (A \Rightarrow_t B) &\cong (tm\ A \to tm\ B)\\ \bar{\forall}\text{-}iso : \{k : \_\}\ \{A : \_\} \to tm\ (\bar{\forall}\ k\ A) &\cong ((\alpha : el\ k) \to tm\ (A\ \alpha))\end{aligned} \tag{F_ω-4}$$

For the two-element type, we only include two terms tt and ff:

$$tt : tm\ bool \qquad ff : tm\ bool \tag{F_ω-5}$$

This completes the signature of System F_ω. We have set it up in a minimal way for simplicity. More useful types/kinds, such as a Boolean type with the correct universal property, products and lists, can be added easily and can be found in Appendix A.

$$
\begin{array}{l}
\mathit{tyco} : \mathit{ki} \quad \text{-- type constructors} \\
\mathit{tyco} = (\mathit{ty} \Rightarrow_k \mathit{ty}) \\
\mathit{fmap\text{-}ty} : (F : \mathit{el}\ \mathit{tyco}) \to \mathit{el}\ \mathit{ty} \\
\mathit{fmap\text{-}ty}\ F = \bar{\forall}\ \mathit{ty}\ (\lambda\alpha.\ \bar{\forall}\ \mathit{ty}\ (\lambda\beta.\ (\alpha \Rightarrow_t \beta) \Rightarrow_t (F\ \alpha \Rightarrow_t F\ \beta))) \\
\textsc{record}\ \mathit{RawFunctor} : \mathbb{J}\ \textsc{where} \\
\quad 0 \quad : \mathit{el}\ \mathit{tyco} \\
\quad \mathit{fmap} : \mathit{tm}\ (\mathit{fmap\text{-}ty}\ 0) \\
\textsc{record}\ \mathit{RawMonad} : \mathbb{J}\ \textsc{where} \\
\quad 0 \quad : \mathit{el}\ \mathit{tyco} \\
\quad \mathit{ret} \quad : \mathit{tm}\ (\bar{\forall}\ \mathit{ty}\ (\lambda\alpha.\ \alpha \Rightarrow_t 0\ \alpha)) \\
\quad \mathit{bind} : \mathit{tm}\ (\bar{\forall}\ \mathit{ty}\ (\lambda\alpha.\ \bar{\forall}\ \mathit{ty}\ (\lambda\beta.\ 0\ \alpha \Rightarrow_t (\alpha \Rightarrow_t 0\ \beta) \Rightarrow_t 0\ \beta))) \\
\mathit{trans} : (F, G : \mathit{el}\ \mathit{tyco}) \to \mathit{el}\ \mathit{ty} \\
\mathit{trans}\ F\ G = \bar{\forall}\ \mathit{ty}\ (\lambda\alpha.\ F\ \alpha \Rightarrow_t G\ \alpha) \\
\textsc{record}\ \mathit{RawHFunctor} : \mathbb{J}\ \textsc{where} \\
\quad 0 \quad : \mathit{el}\ (\mathit{tyco} \Rightarrow_k \mathit{tyco}) \\
\quad \mathit{hfmap} : (F : \mathit{RawFunctor}) \to \mathit{tm}\ (\mathit{fmap\text{-}ty}\ (0\ (F\ .0))) \\
\quad \mathit{hmap} \ : (F, G : \mathit{RawFunctor}) \to \mathit{tm}\ (\mathit{trans}\ (F\ .0)\ (G\ .0)) \\
\qquad\quad \to \mathit{tm}\ (\mathit{trans}\ (0\ (F\ .0))\ (0\ (G\ .0)))
\end{array}
$$

Fig. 1. Derived concepts in System F_ω

Example 2.3. Let us see an example of a program of F_ω defined in the LF. Writing *app* and *abs* for the forward and backward directions of the isomorphism $\Rightarrow_t$-*iso* respectively, and *App* and *Abs* for the two directions of $\bar{\forall}$-*iso*, the Church numeral of 2 is

$$
\begin{array}{l}
\mathit{two} : \mathit{tm}\ (\bar{\forall}\ (\lambda\alpha.\ (\alpha \Rightarrow_t \alpha) \Rightarrow_t \alpha \Rightarrow_t \alpha)) \\
\mathit{two} = \mathit{Abs}\ (\lambda\alpha.\ \mathit{abs}\ (\lambda f.\ \mathit{abs}\ (\lambda x.\ \mathit{app}\ f\ (\mathit{app}\ f\ x))))
\end{array}
$$

2.2.3 Derived Concepts. Later we will see that signatures of higher-order effectful operations in $\mathrm{F}^{\mathrm{ha}}_\omega$ are given as higher-order endofunctors $(\mathit{ty} \Rightarrow_k \mathit{ty}) \Rightarrow_k (\mathit{ty} \Rightarrow_k \mathit{ty})$ over F_ω-functors $\mathit{ty} \Rightarrow_k \mathit{ty}$, and handlers in $\mathrm{F}^{\mathrm{ha}}_\omega$ always have a monad structure. These derived concepts, such as functors and monads, are essentially the same as the definitions in Haskell, and are collected in Figure 1, which will be ingredients for our computation judgements in the next step. Note that because F_ω does not have equality types, the equational laws for functors/monads are not included in these definitions (just like in Haskell), so they are called *raw functors/monads*.

Notation 2.4. We will sometimes suppress the field accessor *.0* of raw functors/monads in Figure 1 for readability, so given $F : \mathit{RawFunctor}$ and $X : \mathit{el}\ \mathit{ty}$, we may write $F\ X$ for $F\ .0\ X$.

2.3 Computation Judgements

Now we are ready to add computation judgements to F_ω to obtain $\mathrm{F}^{\mathrm{ha}}_\omega$.

2.3.1 Computations. We follow the *fine-grain call-by-value* (FGCBV) approach [Lassen 1998; Levy et al. 2003]. For each $H : \mathit{RawHFunctor}$ and $A : \mathit{el}\ \mathit{ty}$, there is a judgement $\mathit{co}\ H\ A$ for *computations* of A-values with effectful operations specified by H:

$$
\mathit{co} : (H : \mathit{RawHFunctor}) \to (A : \mathit{el}\ \mathit{ty}) \to \mathbb{J} \qquad (\mathrm{F}^{\mathrm{ha}}_\omega\text{-}1)
$$

The judgement has the following two rules for pure computations and sequential composition of computations respectively:

$$\begin{aligned} &\mathit{val} : \{H, A\} \to \mathit{tm}\ A \to \mathit{co}\ H\ A \\ &\mathit{let\text{-}in} : \{H, A, B\} \to \mathit{co}\ H\ A \to (\mathit{tm}\ A \to \mathit{co}\ H\ B) \to \mathit{co}\ H\ B \end{aligned} \tag{$\mathrm{F}^{\mathrm{ha}}_{\omega}$-2}$$

The interaction of *val* and *let-in* is axiomatised by the following judgemental equalities, which are exactly the equations (5) and (6) from the introduction and are essentially the monadic laws:

$$\begin{aligned} &\mathit{val\text{-}let} \quad : \{H, A, B\} \to (a : \mathit{tm}\ A) \to (k : \mathit{tm}\ A \to \mathit{co}\ H\ B) \to \mathit{let\text{-}in}\ (\mathit{val}\ a)\ k = k\ a \\ &\mathit{let\text{-}val} \quad : \{H, A\} \to (m : \mathit{co}\ H\ A) \to \mathit{let\text{-}in}\ m\ \mathit{val} = m \\ &\mathit{let\text{-}assoc} : \{H, A, B, C\} \to (m_1 : \mathit{co}\ H\ A) \\ &\qquad \to (m_2 : \mathit{tm}\ A \to \mathit{co}\ H\ B) \to (m_3 : \mathit{tm}\ B \to \mathit{co}\ H\ C) \\ &\qquad \to \mathit{let\text{-}in}\ (\mathit{let\text{-}in}\ m_1\ m_2)\ m_3 = \mathit{let\text{-}in}\ m_1\ (\lambda a.\ \mathit{let\text{-}in}\ (m_2\ a)\ m_3) \end{aligned} \tag{$\mathrm{F}^{\mathrm{ha}}_{\omega}$-3}$$

2.3.2 Thunks. We also introduce a new type former $\mathit{th}\ H\ A$ for *thunks* of computations of A-values with effects of H, whose terms are isomorphic to computations:

$$\mathit{th} : \mathit{RawHFunctor} \to \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty} \qquad \mathit{th\text{-}iso} : \{H, A\} \to \mathit{tm}\ (\mathit{th}\ H\ A) \cong \mathit{co}\ H\ A \tag{$\mathrm{F}^{\mathrm{ha}}_{\omega}$-4}$$

The two directions of the isomorphism *th-iso* will be called $\Uparrow$ and $\Downarrow$ respectively:

$$\Uparrow : \mathit{tm}\ (\mathit{th}\ H\ A) \to \mathit{co}\ H\ A \qquad\qquad \Downarrow : \mathit{co}\ H\ A \to \mathit{tm}\ (\mathit{th}\ H\ A)$$

Thunks can be packed into a raw monad:

$$\begin{aligned} &\mathit{th\text{-}mnd} : \mathit{RawHFunctor} \to \mathit{RawMonad} \\ &\mathit{th\text{-}mnd}\ H\ .0 \quad\ = \mathit{th}\ H \\ &\mathit{th\text{-}mnd}\ H\ .\mathit{ret} \quad = \lambda A\ x.\ \Downarrow (\mathit{val}\ x) \\ &\mathit{th\text{-}mnd}\ H\ .\mathit{bind} = \lambda A\ B\ m\ k.\ \Downarrow (\mathit{let\text{-}in}\ (\Uparrow m)\ (\lambda a.\ \Uparrow (k\ a))) \end{aligned}$$

Following from equations ($\mathrm{F}^{\mathrm{ha}}_{\omega}$-3), *th-mnd* satisfies the monad laws too.

Levy et al. [2003] presented the FGCBV calculus using *effectful functions*:

$$\begin{aligned} \_ \Rightarrow [\_]\_ &: \mathit{el}\ \mathit{ty} \to \mathit{RawHFunctor} \to \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty} \\ \mathit{ef\text{-}iso} &: \{A, H, B\} \to \mathit{tm}\ (A \Rightarrow [H]\ B) \cong (\mathit{tm}\ A \to \mathit{co}\ H\ B) \end{aligned}$$

But since we already have pure functions in the language, it is sufficient to have the thunk type, and define effectful functions as $(A \Rightarrow [H]\ B) \coloneqq (A \Rightarrow_t \mathit{th}\ H\ B)$.

2.3.3 Operations. Effectful operations that computations can perform are introduced by

$$\mathit{op} : \{H, A, B\} \to \mathit{tm}\ (H\ (\mathit{th}\ H)\ A) \to (\mathit{tm}\ A \to \mathit{co}\ H\ B) \to \mathit{co}\ H\ B \tag{$\mathrm{F}^{\mathrm{ha}}_{\omega}$-5}$$

The first argument $o : \mathit{tm}\ (H\ (\mathit{th}\ H)\ A)$ is the input to an operation call, such as some parameters or computations that the operation call acts on. The second argument $k : \mathit{tm}\ A \to \mathit{co}\ H\ B$ of *op* is the 'continuation' of the computation after this operation call, where the argument $\mathit{tm}\ A$ of k is the result of the operation call. The result $\mathit{op}\ o\ k$ is understood as the computation that first makes an operation call with input o, which returns an A-value, and then continues as k.

Example 2.5. The higher-order functor H_{exc} for the effects of exception throwing and catching is

$$\begin{aligned} &H_{exc}\ .0 : \mathit{el}\ ((\mathit{ty} \Rightarrow_k \mathit{ty}) \Rightarrow_k (\mathit{ty} \Rightarrow_k \mathit{ty})) \\ &H_{exc}\ .0 = \mathit{abs}_k\ (\lambda F.\ \mathit{abs}_k\ (\lambda A.\ \mathit{unit}\ +\ (\mathit{app}_k\ F\ A) \times (\mathit{app}_k\ F\ A))) \end{aligned}$$

with the evident functorial action on F and A, where abs_k and app_k denote the two directions of the isomorphism $\Rightarrow_k$-*iso* (F_{ω}-3), and $(\times), (+) : \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty}$ are the binary product and

sum types that are definable using Church encodings (alternatively they can be added directly into $\mathrm{F}^{\mathrm{ha}}_{\omega}$). For the computation judgement $co\ H_{exc}$, the operations of throwing and catching are

$$\begin{array}{ll} throw : \{A\} \to co\ H_{exc}\ A & catch : \{A\} \to co\ H_{exc}\ A \to co\ H_{exc}\ A \to co\ H_{exc}\ A \\ throw = op\ (inl\ *)\ val & catch\ p\ h = op\ (inr\ (\Downarrow p,\ \Downarrow h))\ val \end{array}$$

The interaction of operation calls and sequential composition of computations is the following, which is similar to the condition for *algebraic operations* of Plotkin and Power [2001]:

$$\begin{array}{l} let\text{-}op : \{H, A, B, C\} \to (p : tm\ (H\ (th\ H)\ A)) \\ \qquad \to (k : tm\ A \to co\ H\ B) \to (k' : tm\ B \to co\ H\ C) \\ \qquad \to let\text{-}in\ (op\ p\ k)\ k' = op\ p\ (\lambda a.\ let\text{-}in\ (k\ a)\ k') \end{array} \tag{$\mathrm{F}^{\mathrm{ha}}_{\omega}$-6}$$

The equation ($\mathrm{F}^{\mathrm{ha}}_{\omega}$-6) implies that every operation call $op\ o\ k$ is equal to $let\text{-}in\ (op\ o\ val)\ k$, we could have alternatively defined ($\mathrm{F}^{\mathrm{ha}}_{\omega}$-5) as $op' : \{H, A, B\} \to tm\ (H\ (th\ H)\ A) \to co\ H\ A$ without the k argument, which is the formulation (1) that we used in the introduction. This does not make a big technical difference and we choose the formulation ($\mathrm{F}^{\mathrm{ha}}_{\omega}$-5) as it is closer to the rule presented by Plotkin and Pretnar [2009, 2013] for ordinary algebraic effects.

2.3.4 Evaluation. Now we axiomatise that computations $co\ H\ A$ can be *evaluated*, or *handled*, by any raw monads supporting the operations from H (for which we wrote HANDLE p WITH h in Section 1). We define the following structure for monads supporting operations from H:

$$\begin{array}{l} \textsc{record}\ MonadAlg\ (H : RawHFunctor) : \mathbb{J}\ \textsc{where} \\ \quad \textsc{include}\ RawMonad\ \textsc{as}\ M \\ \quad malg : tm\ (trans\ (H\ 0)\ 0) \end{array}$$

where by 'INCLUDE *RawMonad* AS *M*', we mean that *MonadAlg* has all the fields of the record *RawMonad* from Figure 1 – namely, *0*, *ret*, and *bind*. Moreover, for every $m : MonadAlg\ H$, there is a projection $m.M : RawMonad$. We then add to $\mathrm{F}^{\mathrm{ha}}_{\omega}$ the following declaration that evaluates a computation with effect H with any monad that supports the effect:

$$eval : \{H\} \to (m : MonadAlg\ H) \to (A : el\ ty) \to co\ H\ A \to tm\ (m\ A) \tag{$\mathrm{F}^{\mathrm{ha}}_{\omega}$-7}$$

The last piece of the signature of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is the computation rules for *eval*, which are similar to the operational semantics of handlers in conventional algebraic effects [Plotkin and Pretnar 2009, 2013]:

- When the computation is a value, it is handled by the *ret* of the monad,

$$\begin{array}{l} eval\text{-}val : \{H, A\} \to (m : MonadAlg\ H) \to (a : tm\ A) \\ \qquad \to eval\ m\ A\ (val\ a) = m\ .ret\ A\ a \end{array} \tag{$\mathrm{F}^{\mathrm{ha}}_{\omega}$-8}$$

- When the computation is an operation call, it is handled by the corresponding operation on the monad, with all subterms recursively evaluated:

$$\begin{array}{l} eval\text{-}op : \{H, A, B\} \to (m : MonadAlg\ H) \\ \quad \to (p : tm\ (H\ (th\ H)\ A)) \to (k : tm\ A \to co\ H\ B) \\ \quad \to \textsc{let}\ malg = m\ .malg\ A \\ \qquad\qquad T \quad = fct\text{-}of\text{-}mnd\ (th\text{-}mnd\ H) \\ \qquad\qquad M \quad = fct\text{-}of\text{-}mnd\ (m\ .M) \\ \qquad \textsc{in}\ eval\ m\ B\ (op\ p\ k) \\ \qquad\quad = m\ .bind\ A\ B\ (malg\ (H\ .hmap\ T\ M\ (\lambda\alpha\ c.\ eval\ m\ \alpha\ (\Uparrow c))\ A\ p)) \\ \qquad\qquad (\lambda a.\ eval\ m\ B\ (k\ a)) \end{array} \tag{$\mathrm{F}^{\mathrm{ha}}_{\omega}$-9}$$

where *fct-of-mnd* is the canonical functor structure of a monad:

$$fct\text{-}of\text{-}mnd : RawMonad \to RawFunctor$$

$$\mathit{fct\text{-}of\text{-}mnd}\ m\ .0 = m\ .0$$
$$\mathit{fct\text{-}of\text{-}mnd}\ m\ .\mathit{fmap}\ \alpha\ \beta\ f\ \ ma = m\ .\mathit{bind}\ \alpha\ \beta\ \ ma\ (\lambda a.\ m\ .\mathit{ret}\ \_\ (f\ a))$$

This completes the signature of $\mathrm{F}^{\mathrm{ha}}_{\omega}$. The full signature of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is collected in Appendix A.

Remark 2.6. We do not include in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ the equation asserting that *eval* also commutes with *let-in*:

$$\mathit{eval\text{-}let} : \{H, A, B\} \to (m : \mathit{MonadAlg}\ H) \to (c : \mathit{co}\ H\ A) \to (f : \mathit{tm}\ A \to \mathit{co}\ H\ B)$$
$$\to \mathit{eval}\ m\ B\ (\mathit{let\text{-}in}\ c\ f) = m\ .\mathit{bind}\ A\ B\ (\mathit{eval}\ m\ A\ c)\ (\lambda a.\ \mathit{eval}\ m\ B\ (f\ a))$$

This is because we have chosen to work with *raw* monads that may not validate the monad laws, whereas computations *co H A* are axiomatised to always satisfy these laws ($\mathrm{F}^{\mathrm{ha}}_{\omega}$-3). Consequently, we can freely re-associate let-bindings in computations but not in raw monads, so having *eval-let* would result in inconsistency. Although *eval-let* is left out, later we will prove the *canonicity* of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ – evaluating *closed* elements of computations never get stuck. This is intuitively because in the empty context, every computation is always equal to a computation without *let-in* because of the equations *let-val*, *let-assoc* and *let-op*.

Remark 2.7. We did not include in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ any built-in support for *type-and-effect systems* that track the effect operations that a computation may perform [Bauer and Pretnar 2014; Kammar and Plotkin 2012; Lucassen and Gifford 1988] or support for *modular handlers* [Yang and Wu 2021, 2023] that organise handlers in a composable way, because both of them can be *derived concepts* in $\mathrm{F}^{\mathrm{ha}}_{\omega}$, provided that we extend $\mathrm{F}^{\mathrm{ha}}_{\omega}$ with some standard type/kind connectives such as products and lists. For example, an effect row of *algebraic operations* can be given as a type-level list $[\,ty\ \times_k\ ty\,]$, where each element (P, A) of the list determines an operation receiving an argument of type P and returning a value of type A. Every list $[\,ty\ \times_k\ ty\,]$ then determines a *RawHFunctor* that can be supplied to the computation judgement *co*. In this way, effect tracking is a *user-level library* rather than a built-in feature of $\mathrm{F}^{\mathrm{ha}}_{\omega}$, and effect polymorphism is just a special case of the (higher-kinded) polymorphism that $\mathrm{F}^{\mathrm{ha}}_{\omega}$ already has. Details of how this is done can be found in Appendix A.2.

Remark 2.8. We will not discuss type checking for $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in this paper, as $\mathrm{F}^{\mathrm{ha}}_{\omega}$ adds little type-level complexity to F_{ω}, and we expect the existing algorithms for type-checking polymorphic calculi [Dunfield and Krishnaswami 2013; Jones et al. 2007; Leijen 2008] can be extended to work with $\mathrm{F}^{\mathrm{ha}}_{\omega}$.

2.4 An Extension of General Recursion

The *eval* construct of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is a form of *structural recursion*, and it is also possible to extend $\mathrm{F}^{\mathrm{ha}}_{\omega}$ with *general recursion*. We will refer to this extension as $\mathrm{rF}^{\mathrm{ha}}_{\omega}$. The signature of $\mathrm{rF}^{\mathrm{ha}}_{\omega}$ extends that of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ with a new family of judgements *pco* for *partial computations* that has the same signature as *co*:

$$\mathit{pco} : (H : \mathit{RawHFunctor}) \to (A : \mathit{el}\ \mathit{ty}) \to \mathbb{J}. \qquad (\mathrm{rF}^{\mathrm{ha}}_{\omega}\text{-}1)$$

The original computation judgement *co* is still kept in the language and is used for total computations as usual. Most accompanying rules for *co* in Section 2.3 are inherited by *pco*: *val*, *let-in*, *th*, *op*, and all their associated equations. We shall refer to the copy of them for *pco* by same names as before, except for thunks of partial computations, which we call $\mathit{pth} : \mathit{RawHFunctor} \to \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty}$.

The new rules for *pco* are as expected a fixed-point combinator:

$$Y : \{H, A\} \to (\mathit{pth}\ H\ A \to \mathit{pco}\ H\ A) \to \mathit{pco}\ H\ A \qquad (\mathrm{rF}^{\mathrm{ha}}_{\omega}\text{-}2)$$
$$Y\text{-}eq : \{H, A, f\} \to Y\ f = f\ (\Downarrow Y\ f)$$

Another difference between *pco* and *co* is the their elimination rule: *eval* allows computations *co H A* to be evaluated into any raw monad T equipped with an H-operation, but naturally, *pco* shall only be evaluated into monads T that 'support recursion'. In the current call-by-value setting,

the only thing that supports recursion is *pco*, so we will require that the raw monad T send every type $A : \mathit{el}\ \mathit{ty}$ to thunks of partial computations $\mathit{pth}\ H\ (F\ A)$ for some $H : \mathit{RawHFunctor}$ and type constructor $F : \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty}$:

$$\begin{aligned}&\textsc{record}\ \mathit{MonadAlgRec}\ (H : \mathit{RawHFunctor}) : \mathbb{J}\ \textsc{where}\\ &\quad \textsc{include}\ \mathit{MonadAlg}\ H\ \textsc{as}\ T\\ &\quad H\ : \mathit{RawHFunctor}\\ &\quad F\ \ : \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty}\\ &\quad \mathit{eq} : (A : \mathit{el}\ \mathit{ty}) \to T\ A = \mathit{pth}\ H\ (F\ A)\end{aligned}$$

Here we have formulated the requirement *eq* using the equality type of LccLF, and in an implementation of the type checker for $\mathrm{rF}_\omega^{\mathrm{ha}}$, the equation *eq* may be mechanically checked since the kind language of $\mathrm{rF}_\omega^{\mathrm{ha}}$ is normalising. The language $\mathrm{rF}_\omega^{\mathrm{ha}}$ then has the following declaration:

$$\mathit{eval} : \{H\} \to (T : \mathit{MonadAlgRec}\ H) \to (A : \mathit{el}\ \mathit{ty}) \to \mathit{pco}\ H\ A \to \mathit{tm}\ (T\ A) \qquad (\mathrm{rF}_\omega^{\mathrm{ha}}\text{-}3)$$

together with equations *eval-val* and *eval-op* similar to the those of *co* ($\mathrm{F}_\omega^{\mathrm{ha}}$-8, $\mathrm{F}_\omega^{\mathrm{ha}}$-9) with *co* replaced by *pco*, *th* replaced by *pth*, and *MonadAlg* replaced by *MonadAlgRec*.

For discussing its meta-theoretic properties, it is convenient to include in $\mathrm{rF}_\omega^{\mathrm{ha}}$ the empty type:

$$\mathit{empty} : \mathit{el}\ \mathit{ty} \qquad \mathit{absurd} : (A : \mathit{el}\ \mathit{ty}) \to \mathit{tm}\ \mathit{empty} \to \mathit{tm}\ A \qquad (\mathrm{rF}_\omega^{\mathrm{ha}}\text{-}4)$$

so that we have a judgement *pco VoidH* of partial computations without any other effects, where *VoidH* is the constant raw higher-order functor: $\mathit{VoidH}\ \_\ \_ = \mathit{empty}$.

Remark 2.9. Due to space limit, in this paper we cannot demonstrate some programming examples of $\mathrm{F}_\omega^{\mathrm{ha}}$, and we refer readers interested in concrete examples to the previous work on higher-order effect handlers [Bosman et al. 2024; van den Berg and Schrijvers 2024; van den Berg et al. 2021; Wu et al. 2014; Yang et al. 2022], whose examples can be adapted to $\mathrm{F}_\omega^{\mathrm{ha}}$/$\mathrm{rF}_\omega^{\mathrm{ha}}$ easily.

2.5 Semantic Models and the Category of Judgements

We have presented the calculus $\mathrm{F}_\omega^{\mathrm{ha}}$ using LccLF, which we hope to be an example demonstrating the compactness and precision of using a type-theoretic LF to present programming languages. However, syntactic nicety is not the only advantage of using LFs. A bigger advantage is that an LF can provide useful general results for object languages defined in it. For every signature, LccLF provides (1) a notion of semantic *models*, (2) a category of *judgements*, and (3) an equivalence between models and functors out of the category of judgements. These results are established by Yang [2025] in detail, and below we record the special cases for the signature $\mathrm{F}_\omega^{\mathrm{ha}}$.

2.5.1 Semantic Models. To define the concept of models of $\mathrm{F}_\omega^{\mathrm{ha}}$ in a category, we first need an auxiliary concept of categories that can interpret the whole logical framework LccLF.

Definition 2.10. An *LF-category* is a category $\mathscr{C}$ together with an interpretation of the type formers of LccLF (Language 2.1), i.e. the unit type 1, Σ-types, restricted Π-types, and a universe $\mathbb{J}$ closed under 1, Σ, Π, and extensional equality types $a = b$.

We will say 'an LF-category $\langle \mathscr{C}, U \rangle$' where U is the interpretation of the universe. Strictly speaking, the interpretation of other type formers is also part of the structure, but they are determined uniquely (up to isomorphisms) by their respective universal properties.

For example, the category of Set can be made an LF-category, with $\mathbb{J}$ being interpreted as some set-theoretic universe U. A trivial choice of U is just the one-element set $\{1\}$, which is closed under $1, \Sigma, \Pi, =$. More generally, the presheaf category $\mathrm{Pr}\,\mathscr{C}$ over a (small) locally cartesian closed category (LCCC) $\mathscr{C}$ can be made an LF-category $\langle \mathrm{Pr}\,\mathscr{C}, U_{\mathscr{C}} \rangle$ where the universe $U_{\mathscr{C}}$ classifies exactly (the Yoneda embedding of) the objects and morphisms of $\mathscr{C}$; see Yang [2025, §IV] for details.

Definition 2.11. A *model* M of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in a (small) LCCC $\mathscr{C}$ is a morphism $M : 1 \to [\![\mathrm{F}^{\mathrm{ha}}_{\omega}]\!]_{\mathscr{C}}$ in the presheaf category $\mathrm{Pr}\,\mathscr{C}$, where the object $[\![\mathrm{F}^{\mathrm{ha}}_{\omega}]\!]_{\mathscr{C}}$ is the interpretation of the record type that has all the declarations of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ (F_{ω}-1 to F_{ω}-5, $\mathrm{F}^{\mathrm{ha}}_{\omega}$-1 to $\mathrm{F}^{\mathrm{ha}}_{\omega}$-9) as fields, with $\mathbb{J}$ interpreted as $U_{\mathscr{C}}$.

$$\begin{array}{l}\textsc{record}\ \mathrm{F}^{\mathrm{ha}}_{\omega}\ \textsc{where}\\ \quad ki : \mathbb{J};\quad el : ki \to \mathbb{J};\quad ty : ki;\quad \ldots\end{array} \tag{7}$$

Yang [2025] also defined a notion of *model isomorphisms*, so we have a groupoid $\mathrm{F}^{\mathrm{ha}}_{\omega}$-$\textsc{Mod}(\mathscr{C})$ of $\mathrm{F}^{\mathrm{ha}}_{\omega}$-models and isomorphisms in an LCCC $\mathscr{C}$. In this paper, model isomorphisms will not play an important role so we omit their definition here.

Remark 2.12. Definitions 2.10 and 2.11 may appear as rather opaque to the reader, but we need not worry about them too much. For our purposes in this paper, what we need know about them is that, for every LCCC $\mathscr{C}$, we can define a model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in $\mathscr{C}$ by defining a closed element of the record type $\mathrm{F}^{\mathrm{ha}}_{\omega}$ (7) in an internal language of $\mathrm{Pr}\,\mathscr{C}$ with $\mathbb{J}$ replaced by the universe $U_{\mathscr{C}}$.

In fact, our semantic domain $\mathscr{C}$ usually already has a universe U that can interpret $\mathbb{J}$. In this case, we do not have to move to the bigger category $\mathrm{Pr}\,\mathscr{C}$. To construct a model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in $\mathscr{C}$, it is sufficient to construct a closed element of the record $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in an internal language of $\mathscr{C}$ itself, with $\mathbb{J}$ replaced by the universe U. We will see several examples of this in the following sections.

2.5.2 Category of Judgements. In categorical logic, we usually have the category of *types* or *contexts* that organises the syntactic entities of a language as a category. For languages defined using LccLF, the category containing the syntactic entities is the *category of judgements*.

Definition 2.13. The *category of judgements* $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$ for the language $\mathrm{F}^{\mathrm{ha}}_{\omega}$ has (1) LccLF-terms $\mathrm{F}^{\mathrm{ha}}_{\omega} \vdash A : \mathbb{J}$ as objects, and (2) terms $\mathrm{F}^{\mathrm{ha}}_{\omega} \vdash f : A \to B$ as morphisms. Identities and composition are the evident identity function and function composition in LccLF.

The category of judgements is locally cartesian closed. For every object $A : \mathbb{J}$ of $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$ (here we omit the context '$\mathrm{F}^{\mathrm{ha}}_{\omega} \vdash$'), the terminal object of the slice category $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}/A$ is $(\lambda x.\ x) : A \to A$. Given two objects $f : B \to A$ and $g : C \to A$ in $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}/A$, their product is $\lambda p.\ f\ (\pi_1\ p) : P \to A$ where $P \coloneqq \Sigma(b : B).\ \Sigma(c : C).\ f\ b = g\ c$, and their exponential is $\pi_1 : E \to A$ where

$$E \coloneqq \Sigma(a : A).\ B_a \to C_a \qquad B_a \coloneqq \Sigma(b : B).\ f\ b = a \qquad C_a \coloneqq \Sigma(c : C).\ g\ c = a$$

Among all LCCCs, the category $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$ has the following universal property:

Theorem 2.14 (Yang [2025]). *Let $\mathscr{C}$ be an LCCC. The groupoid $\mathrm{LCCC}_{\cong}(\textsc{Jdg}\,F^{ha}_{\omega}, \mathscr{C})$ of LCC-functors and natural isomorphisms is equivalent to the groupoid of F^{ha}_{ω}-models in $\mathscr{C}$.*

The practical relevance of this theorem is two-folds. Firstly, it gives us a functor $[\![-]\!]_M : \textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega} \to \mathscr{C}$ after we define a model M of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in $\mathscr{C}$. This functor assigns a meaning of every $\mathrm{F}^{\mathrm{ha}}_{\omega}$-judgement and derivation (not just the generating ones declared in the signature $\mathrm{F}^{\mathrm{ha}}_{\omega}$) in the category $\mathscr{C}$. Secondly, it provides a connection between $\mathrm{F}^{\mathrm{ha}}_{\omega}$ as a *syntactic signature* and as a *category* $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$. This connection enables us to use categorical tools to study the theory $\mathrm{F}^{\mathrm{ha}}_{\omega}$. For example, in Section 4, we will use a categorical tool known as *Artin gluing* to prove properties of $\mathrm{F}^{\mathrm{ha}}_{\omega}$.

3 The Realizability Model

In this section, we present a model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in the category $\textsc{Asm}(\mathbb{A})$ of *assemblies* over an arbitrary *partial combinatory algebra* (PCA) $\mathbb{A}$. This model serves two purposes: (1) it shows that $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is consistent in the sense that *tt* and *ff* are not equal terms of *bool* in the equational theory of $\mathrm{F}^{\mathrm{ha}}_{\omega}$, and (2) it provides a way to extract executable programs, e.g. terms of λ-calculus, from well typed terms of $\mathrm{F}^{\mathrm{ha}}_{\omega}$, giving us a way to run terms of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ without an explicit operational semantics.

3.1 Assemblies and Their Language

We will only use the category of assemblies as a black box via a type-theoretic internal language Language 3.1, so in principle the reader does not even need to know what an assembly is to read this section and treat the model presented in this section as a syntactic translation from $\mathrm{F}_\omega^{\mathrm{ha}}$ to another type theory. We refer readers who are interested in how assemblies work 'under the hood' to tutorials on realizability by Bauer [2022], de Jong [2024], Streicher [2017], and the comprehensive book account by van Oosten [2008].

A partial combinatory algebra $\langle \mathbb{A}, \cdot \rangle$ is an abstraction for an untyped model of computation: $\mathbb{A}$ is a set whose elements serve the dual purpose of programs and data, and $\cdot : \mathbb{A} \times \mathbb{A} \rightharpoonup \mathbb{A}$ is a partial binary operation subject to certain conditions. The intuition for $n \cdot m$ is applying the program n to the input data m. Notable examples of PCAs include (1) β-equivalence classes of closed λ-terms together with λ-term application and (2) the set of natural numbers together with $n \cdot m$ being the result of running the n-th Turing machine with input m (if the n-th Turing machine does not halt on m, $n \cdot m$ is undefined). The second example is called *Kleene's first algebra* $\mathbb{K}$.

For every PCA $\langle \mathbb{A}, \cdot \rangle$, the category $\textsc{Asm}(\mathbb{A})$ of assemblies over $\mathbb{A}$ (also known as *ω-sets*) is roughly a category of 'computable sets and functions': an object $\langle X, |-|_X \rangle$ of $\textsc{Asm}(\mathbb{A})$ is a set X with a function $|-|_X : X \to \mathscr{P}(\mathbb{A})$ mapping every every $x \in X$ to a *non-empty* subset $|x|_X$ of $\mathbb{A}$. The intuition is that $|x|_X$ is the set of $\mathbb{A}$-elements that *encode* or *realize* $x \in X$. A morphism $f : \langle X, |-|_X \rangle \to \langle Y, |-|_Y \rangle$ in $\textsc{Asm}(\mathbb{A})$ is a set-theoretic function $f : X \to Y$ such that there exists an element $r \in \mathbb{A}$ and for all $x \in X$, $n \in |x|_X$, $r \cdot n$ is defined and $r \cdot n \in |f\ x|_Y$.

The category $\textsc{Asm}(\mathbb{A})$ of assemblies has many pleasant properties, making it a standard tool for interpreting programming languages, especially those with impredicative polymorphism. We will access the nice structure of $\textsc{Asm}(\mathbb{A})$ via the following type-theoretic internal language.

Language 3.1. The language AsmTT is a dependent type theory with the following type formers:

- dependent function types (Π-types), dependent pair types (Σ-types), extensional equality types, and inductive types (e.g. the unit type, the empty type, the natural number type);
- three universes $P : V_1 : V_2$, each closed under the aforementioned type formers;
- P is *impredicative* in the sense that for all types A (not necessarily in P) and type families $B : A \to P$, the dependent function type $(x : A) \to B\ x$ is in P.

The language AsmTT can be interpreted in the category $\textsc{Asm}(\mathbb{A})$ for every non-trivial PCA $\mathbb{A}$. Interpretations of similar type theories in the category of assemblies can be found in Hofmann [1997, §3.4] and Luo [1994, Chapter 6]. Non-triviality of $\mathbb{A}$ is needed here for the impredicative universe P to have types with more than one elements (if $\mathbb{A}$ is the trivial one-element PCA, $\textsc{Asm}(\mathbb{A})$ degenerates to the category of sets and P is the set $\{\bot, \top\}$ of classical propositions).

3.2 The Realizability Model of $\mathrm{F}_\omega^{\mathrm{ha}}$

As mentioned in Remark 2.12, to define a model of $\mathrm{F}_\omega^{\mathrm{ha}}$ in $\textsc{Asm}(\mathbb{A})$, it is sufficient to construct an element of the record type $[\![\mathrm{F}_\omega^{\mathrm{ha}}]\!]_{V_2}$ in AsmTT that contains all declarations of $\mathrm{F}_\omega^{\mathrm{ha}}$ with $\mathbb{J}$ replaced by the universe V_2. In the following, we construct a such model $R : [\![\mathrm{F}_\omega^{\mathrm{ha}}]\!]_{V_2}$.

The model of the F_ω-fragment is standard. Kinds are interpreted as the predicative universe V_1 and the base kind $ty : ki$ is interpreted as the impredicative universe P:

$$R.ki : V_2 \qquad R.el : R.ki \to V_2 \qquad R.ty : R.ki \qquad R.tm : R.el\ R.ty \to V_2$$
$$R.ki = V_1 \qquad R.el\ k = k \qquad R.ty = P \qquad R.tm\ A = A$$

Function kinds $R.\_\Rightarrow_k\_ : R.ki \to R.ki \to R.ki$ are interpreted by function types in V_1, and $R.\Rightarrow_k\text{-}iso$ is the identity isomorphism. The unit, Boolean, function types of ty in $\mathrm{F}_\omega^{\mathrm{ha}}$ are interpreted as the

corresponding type formers in the universe P. The impredicative polymorphic function type $\bar{\forall}$ is interpreted as dependent function type:

$$\begin{aligned} &R.\bar{\forall} : (k : R.ki) \to (R.el\ k \to R.el\ R.ty) \to R.el\ R.ty \\ &R.\bar{\forall}\ k\ A = (\alpha : k) \to A\ \alpha \end{aligned}$$

This is well typed because $R.ty$, i.e. P, is an impredicative universe.

The model of the computation judgement $co\ H\ A$ is less obvious because of the mismatch between computations and raw monads in $\mathrm{F}^{\mathrm{ha}}_{\omega}$: computations satisfy the monadic laws strictly (*let-val*, *val-let*, *let-assoc* from $\mathrm{F}^{\mathrm{ha}}_{\omega}$-3), while raw monads do not. Consequently, we cannot model $co\ H$ as the initial *raw monad* equipped with H-operations because it then would not satisfy the monadic laws. Conversely, we cannot model it as the initial *monad* equipped with H-operations either because then it cannot be evaluated into raw monads. Our solution is to model computations by a combination of impredicative encoding and continuation-passing transformation:

$$\begin{aligned} &R.co : R.RawHFunctor \to R.el\ R.ty \to P \\ &R.co\ H\ A = (T : R.MonadAlg\ H) \to (B : P) \to (A \to T\ B) \to T\ B \end{aligned} \tag{8}$$

Thunking $th\ H\ A$ can be modelled as the identity, because in the model R, computations and values live in the same universe P. The computation formers and *eval* are defined as follows:

$$\begin{aligned} &R.val : \{H, A\} \to A \to R.co\ H\ A \\ &R.val\ a = \lambda T\ B\ (r : A \to T\ B).\ r\ a \end{aligned}$$

$$\begin{aligned} &R.let\text{-}in : \{H, A, B\} \to R.co\ H\ A \to (A \to R.co\ H\ B) \to R.co\ H\ B \\ &R.let\text{-}in\ \{A, B\}\ c\ k = \lambda T\ C\ (r : B \to T\ C).\ c\ T\ C\ (\lambda a.\ k\ a\ T\ C\ r) \end{aligned}$$

The model of *eval* directly follows from the definition of $R.co$:

$$\begin{aligned} &R.eval : \{H\} \to (T : R.MonadAlg\ H) \to (A : P) \to R.co\ H\ A \to T\ A \\ &R.eval\ \{H\}\ T\ A\ c = c\ T\ A\ (T.ret) \end{aligned}$$

It can be checked the definitions for computations above collectively validate all the equational laws of $\mathrm{F}^{\mathrm{ha}}_{\omega}$. Detailed calculations can be found in Appendix B

This completes our definition of the model $R : [\![\mathrm{F}^{\mathrm{ha}}_{\omega}]\!]_{V_2}$ in $\textsc{Asm}(\mathbb{A})$. An immediate consequence is the *consistency* of the equational theory of System $\mathrm{F}^{\mathrm{ha}}_{\omega}$.

Theorem 3.2. *The equational theory of System F^{ha}_{ω} is* consistent*, in the sense that the closed terms tt and ff : bool are not judgementally equal.*

Proof. Let $\mathbb{A}$ be any non-trivial PCA, such as Kleene's first algebra. The interpretation of tt and ff : *bool* in the realizability model R are different morphisms $1 \to 1 + 1$ in $\textsc{Asm}(\mathbb{A})$. Because interpretation respects judgemental equalities, tt and ff must not be judgementally equal. □

The realizability model also gives us a way to do *program extraction*:

Theorem 3.3. *For every closed term t : tm bool of F^{ha}_{ω}, there exists a λ-term $|t|$ that normalises to a Church Boolean value. Moreover, if $t = t'$, $|t|$ and $|t'|$ normalises to the same value.*

Proof. Terms of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ are interpreted as morphisms in the category $\textsc{Asm}(\mathbb{A})$ of assemblies, which are realized by elements of the underlying PCA $\mathbb{A}$, in particular, the PCA Λ of λ-terms. Therefore every closed term t : *tm bool* can be interpreted as a morphism $1 \to 1 + 1$ in $\textsc{Asm}(\Lambda)$, which by the definition of Asm is realised by a λ-term. □

The interpretation of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in the realizability model is clearly constructive, so based on the realizability model we can implement a compiler that takes in well typed $\mathrm{F}^{\mathrm{ha}}_{\omega}$-terms and outputs λ-terms following the definition of the model R (by simply erasing the type information in R). In

the next section, we will further see that the realizability model is in fact *adequate* with respect to the equational theory of $\mathrm{F}_{\omega}^{\mathrm{ha}}$: if the interpretation of a closed Boolean term $p : tm\ bool$ in the realizability model is true (resp. false), then $p = tt$ (resp. $p = ff$) in the equational theory of $\mathrm{F}_{\omega}^{\mathrm{ha}}$.

Modelling General Recursion. The realizability model of $\mathrm{F}_{\omega}^{\mathrm{ha}}$ can be extended to a model of $\mathrm{rF}_{\omega}^{\mathrm{ha}}$ from Section 2.4 using *synthetic domain theory* [Hyland 1991; Phoa 1991; Rosolini 1986]. We will not go into this here for the lack of space, but interested readers can see how it is done in Appendix E, which also provides a mini introduction to synthetic domain theory. Consequences of this model are analogues of Theorem 3.2 and Theorem 3.3.

Theorem 3.4. *The equational theory of rF_{ω}^{ha} is consistent: for every $H : RawHFunctor$, $(val\ tt)$ and $(val\ ff) : pco\ H\ bool$ are not judgemental equal in rF_{ω}^{ha}.*

Theorem 3.5. *For every closed term $t : pco\ VoidH\ bool$ of F_{ω}^{ha}, there exists a λ-term $|t|$ that either diverges or normalises to a Church Boolean value. Moreover, if $t = t'$, then $|t|$ and $|t'|$ are Kleene equal.*

4 The Synthetic Logical Relation Model

The equational theory of $\mathrm{F}_{\omega}^{\mathrm{ha}}$ provides the programmer with a set of reasoning principles to understand the behaviour of $\mathrm{F}_{\omega}^{\mathrm{ha}}$-programs, and also provides compiler writers with a set of program transformations for optimisation. One natural question is – *how complete is the equational theory*? We answer this question by proving the following theorem in this section.

Theorem 4.1 (Canonicity). *For every closed Boolean term $b : bool$ of System F_{ω}^{ha}, either $b = tt$ or $b = ff$ holds (but not both) in the equational theory of F_{ω}^{ha}.*

The common proof strategy for showing meta-theoretic properties of programming languages such as canonicity is the method of *logical relations* [Plotkin 1973, 1980], also known as *the computability method* or the *reducibility method* in the literature [Girard 1972; Martin-Löf 1975a,b; Statman 1985; Tait 1967]. The high-level idea is to construct a model of a language L such that each judgement J of L is interpreted as a set of J-derivations satisfying certain properties or equipped with certain data. Inference rules of L are then shown to preserve those associated properties or data. Categorically, such a logical-relation model lives in the *glued category* of the category of types/judgements of L and the category of sets (or some other presheaf topos where the meta-theoretic information naturally lives) [Altenkirch et al. 1995; Fiore 2022; Freyd 1978].

A recent development of the method of logical relations is Sterling's *synthetic Tait computability* (STC) [Sterling 2021, 2022; Sterling and Harper 2021], whose idea is to (1) embed the category of types/judgements in the (presheaf) topos over it by Yoneda embedding, (2) glue this (presheaf) topos containing information of the object language with the (presheaf) topos where the meta-theoretic information lives, which always results in a new (presheaf) topos, and then (3) use a type-theoretic language to describe the constructions in the glued topos. Passing to the presheaf category in the first step is needed so that the resulting glued topos is a topos, where we have a very rich internal language to describe the construction of the logical relation model. The advantage of this approach is that after the internal language is set up, many tedious aspects in a typical logical-relation proof is taken care of automatically, turning a logical-relation proof into a guided programming puzzle.

4.1 The Language of STC for $\mathrm{F}_{\omega}^{\mathrm{ha}}$

To apply the method of STC to prove canonicity of $\mathrm{F}_{\omega}^{\mathrm{ha}}$ (Theorem 4.1), we first recall that in Section 2.5.2 we defined a category of judgements $\textsc{Jdg}\,\mathrm{F}_{\omega}^{\mathrm{ha}}$ for $\mathrm{F}_{\omega}^{\mathrm{ha}}$, whose objects are terms $J : \mathbb{J}$ and morphisms are functions $f : J \to J'$ under the signature $\mathrm{F}_{\omega}^{\mathrm{ha}}$ in LccLF.

Definition 4.2. The *glued category* $\textsc{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}}$ of $\textsc{Pr}\,(\,\textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}})$ and $\textsc{Set}$ along the global section functor $\textsc{Hom}(1,-) : \textsc{Pr}\,(\,\textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}}) \to \textsc{Set}$ is exactly the *comma category* $\textsc{Set} \downarrow \textsc{Hom}(1,-)$, whose objects are tuples $\langle A \in \textsc{Pr}\,(\,\textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}}),\ P \in \textsc{Set},\ p : P \to \textsc{Hom}(1,A)\rangle$, and morphisms $\langle A,P,p\rangle \to \langle B,Q,q\rangle$ are pairs $\langle f : A \to B, g : P \to Q\rangle$ making the following diagram in $\textsc{Set}$ commute:

$$\begin{array}{ccc} P & \xrightarrow{\quad g \quad} & Q \\ {\scriptstyle p}\downarrow & & \downarrow{\scriptstyle q} \\ \textsc{Hom}(1,A) & \xrightarrow[\textsc{Hom}(1,f)]{} & \textsc{Hom}(1,B) \end{array}$$

Usually, the presheaf A of an object $\langle A,P,p\rangle \in \textsc{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}}$ will just be the Yoneda embedding of some $\mathrm{F}_\omega^{\mathrm{ha}}$-judgement J. In this case $\textsc{Hom}(1,A)$ is the set of closed derivations of the judgement J, and $p : P \to \textsc{Hom}(1,A)$ is understood as a *proof-relevant predicate* over J-derivations. For every $a \in \textsc{Hom}(1,A)$, the set $\{e \in P \mid p(e) = a\}$ is the set of proofs that a satisfies the predicate $p : P \to \textsc{Hom}(1,A)$. A morphism $\langle f,g\rangle : \langle A,P,p\rangle \to \langle B,Q,q\rangle$ in $\textsc{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}}$ is then a derivation from A to B that preserves the associated predicates P and Q on A and B.

The presheaf category $\textsc{Pr}\,(\,\textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}})$ contains a *syntactic model* M of $\mathrm{F}_\omega^{\mathrm{ha}}$, where every judgement J is interpreted as its Yoneda embedding. A (proof-relevant) logical-relation model of $\mathrm{F}_\omega^{\mathrm{ha}}$ is then a model M^* of $\mathrm{F}_\omega^{\mathrm{ha}}$ in the category $\textsc{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}}$ such that M^* under the first projection $\textsc{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}} \to \textsc{Pr}\,(\,\textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}})$ is exactly the syntactic model M. We will construct our logical-relation model M^* via an internal language of $\textsc{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}}$. In this following, we first introduce this language; for a proper introduction, we refer the reader to the exposition by Huang [2023] and Sterling [2022]. The most comprehensive account of STC so far is still Sterling's [2021] thesis.

Language 4.3. The language StcTT of *synthetic Tait computability* for $\mathrm{F}_\omega^{\mathrm{ha}}$ is a dependent type theory with the structure of an elementary topos (Axiom 4.4), universes (Axiom 4.5), a distinguished proposition $\mathfrak{ob}$ and a model of $\mathrm{F}_\omega^{\mathrm{ha}}$ under $\mathfrak{ob}$ (Axiom 4.6), and glue types (Axiom 4.6).

First of all, StcTT has the type formers that axiomatise the structure of an elementary topos.

Axiom 4.4 (StcTT-ETopos). StcTT has the following type formers:

- unit type 1, function types $A \to B$, Σ-types $\Sigma(a : A).\ B$, extensional identity types $a = b$;
- a universe Ω such that (1) it *classifies all propositional types*: if a type P satisfies $(a,b : P) \to a = b$, then there is $\ulcorner P \urcorner : \Omega$ with an isomorphism $\phi_P : \ulcorner P \urcorner \cong P$; (2) it is *univalent*: if $A, B : \Omega$ and $A \cong B$, then $A = B$; and (3) it is *proof irrelevant*: if $A : \Omega$ and $p, q : A$, then $p = q$.

The theory of *elementary toposes* [Borceux 1994; Mac Lane and Moerdijk 1994] tells us that a great deal of well behaved logical structures can be defined from the type formers in Axiom 4.4, including Π-types $(a : A) \to B$, (intuitionistic) logical connectives ($\wedge$, $\vee$, $\to$, $\top$, $\bot$, $\exists$, $\forall$) on Ω, the empty type 0, coproduct types $A + B$, quotient types A/R.

Given a type A and $P : A \to \Omega$, we define $\{x : A \mid P(x)\} \coloneqq \Sigma(a : A).\ P(a)$ and treat $\{x : A \mid P(x)\}$ as a subtype of A, eliding the proof of the proposition $P(x)$ and the pairing/projections:

$$\frac{a : A \qquad \_ : P(a)}{a : \{x : A \mid P(x)\}} \qquad \frac{a : \{x : A \mid P(x)\}}{a : A} \qquad \frac{a : \{x : A \mid P(x)\}}{\_ : P(a)}$$

Of course, when using this notation, we must ensure that we only use $a : A$ as an element of $\{x : A \mid P(x)\}$ when a (necessarily unique) element of $P(a)$ is available. Such an informal abuse of notation can be formally justified by Luo et al. [2013]'s *coercive subtyping*.

An important special case of subtypes are *extension types*: given a type A, $\phi : \Omega$ and $a : \phi \to A$, we define $\{A \mid \phi \hookrightarrow a\} \coloneqq \{x : A \mid (p : \phi) \to x = a(p)\}$ for the type of A-elements that are

strictly equal to the partial element a when ϕ holds. Similarly, if a partial element of A is given as an implicit function $a : \{\phi\} \to A$, we also write $\{A \mid \phi \hookrightarrow a\}$ for $\{x : A \mid \{\phi\} \to x = a\}$.

Similar to the realizability model, we will again need three universes to model $\mathrm{F}_\omega^{\mathrm{ha}}$, which exist provided that the ambient set theory has enough universes [Gratzer et al. 2022].

Axiom 4.5 (StcTT-Universe). StcTT has three cumulative predicative universes $U_0 : U_1 : U_2$, each closed under Π-types, Σ-types, extensional equality types, and inductive types. Moreover, the universe of propositions is in U_0, i.e. $\Omega : U_0$.

The next ingredient of StcTT, perhaps the most important one, is the following axiom.

Axiom 4.6 (StcTT-Obj). StcTT has $\mathfrak{ob} : \Omega$ and $M : \{\mathfrak{ob}\} \to [\![\mathrm{F}_\omega^{\mathrm{ha}}]\!]_{U_0}$.

The category $\mathrm{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}}$ embeds the 'object space' $\mathrm{Pr}\,(\mathrm{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}})$ and the 'meta space' Set as full subcategories. In the language StcTT, the proposition $\mathfrak{ob}$ serves the purpose of accessing the object space and the meta space: $\mathfrak{ob}$ is interpreted as an object $[\![\mathfrak{ob}]\!]$ in $\mathrm{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}}$ such that the exponential functor $(-)^{[\![\mathfrak{ob}]\!]}$ sends every object $\langle B, Q, q\rangle \in \mathrm{Gl}\,\mathrm{F}_\omega^{\mathrm{ha}}$ to $\langle B, \mathrm{Hom}(1, B), id\rangle$. In other words, the function type $\mathfrak{ob} \to A$ in StcTT erases the meta-space information in A.

For every type A, we will write $\bigcirc A := \{\mathfrak{ob}\} \to A$, and if the function $\eta^\circ_A := (\lambda a.\ \lambda\{z : \mathfrak{ob}\}.\ a) : A \to \bigcirc A$ is an isomorphism, we say that the type A is $\bigcirc$-*modal*. These are types containing no meta-space information and are essentially objects of $\mathrm{Pr}\,(\mathrm{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}})$. Axiom 4.6 asserts that there is a $\mathrm{F}_\omega^{\mathrm{ha}}$-model M in the object space; this is interpreted as the syntactic model of $\mathrm{F}_\omega^{\mathrm{ha}}$ in $\mathrm{Pr}\,(\mathrm{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}})$.

Remark 4.7. When working in StcTT, the reader should pay special attention to whether the proposition $\mathfrak{ob}$ is assumed in the current context, because we will have types A and B that may look different but are *judgementally equal* when $\mathfrak{ob}$ is assumed. In this way, $\mathfrak{ob}$ is more like a modality for 'entering the object space', instead of a tradititional mathematical proposition about which we care whether it is true or false. The syntax of StcTT is chosen to highlight when $\mathfrak{ob}$ is assumed; for example, in the type $\{\ ?1\ \mid \mathfrak{ob} \hookrightarrow\ ?2\ \}$ the hole $?2$ has $\mathfrak{ob}$ assumed. Moreover, when $\mathfrak{ob}$ is assumed, the type of implicit functions $\{\mathfrak{ob}\} \to A$ and the type A can be used interchangeably, since the proposition $\mathfrak{ob}$ has at most one element and it is assumed in the context.

We have also a modality $\bullet$ for erasing the object-space information in types, which may be expressed as a quotient inductive type in StcTT:

$$
\begin{aligned}
&\textsc{data}\ \bullet A\ \textsc{where}\\
&\quad \eta^\bullet_A : A \to \bullet A\\
&\quad pt : \{\mathfrak{ob}\} \to \bullet A\\
&\quad eq : \{\mathfrak{ob}\} \to (a : A) \to \eta^\bullet_A\ a = pt
\end{aligned}
$$

This quotient inductive type $\bullet A$ can be explicitly constructed using quotient and coproduct types, in the same way as constructing pushouts in Set. A type A is called $\bullet$-*modal* if the function $\eta^\bullet_A : A \to \bullet A$ is an isomorphism. In this case, we write $\epsilon^\bullet_A : \bullet A \to A$ for the inverse of $\eta^\bullet_A : A \to \bullet A$. The following lemma says that $\bullet$-modal types have no object-level information.

Lemma 4.8. *A type A in StcTT is $\bullet$-modal iff $\bigcirc A$ is isomorphic to the unit type* 1.

Proof. Assume A is $\bullet$-modal. We define a function $f : 1 \to \bigcirc A$ by $f\ * = \lambda\{z : \mathfrak{ob}\}.\ \epsilon^\bullet_A\ (pt\ \{z\})$. The function f and $\lambda a.\ * : \bigcirc A \to 1$ form an isomorphism. To see $f \cdot (\lambda a.\ *) = id$, for every $a : \bigcirc A$, we have $f((\lambda a.\ *)\ a) = f\ * = \lambda\{z : \mathfrak{ob}\}.\ \epsilon^\bullet_A\ (pt\ \{z\})$, which is equal to $\lambda\{z : \mathfrak{ob}\}.\ \epsilon^\bullet_A\ (\eta^\bullet_A\ a)$ by $eq\ a : \eta^\bullet_A\ a = pt$, and $\epsilon^\bullet_A\ (\eta^\bullet_A\ a) = a$ because $\epsilon^\bullet_A$ is the inverse of $\eta^\bullet_A$, so we have $f((\lambda a.\ *)\ a) = \lambda\{z : \mathfrak{ob}\}.\ a = a$. In the other direction, $(\lambda a.\ *) \cdot f = id$ is trivial.

Conversely, assume $\bigcirc A \cong 1$. Let a be the unique element of $\bigcirc A$. We can define $\epsilon^\bullet_A : \bullet A \to A$ by $\epsilon^\bullet_A\ (\eta^\bullet_A\ a') = a'$ and $\epsilon^\bullet_A\ pt = a$. The function $\epsilon^\bullet_A$ is a mutual inverse of $\eta^\bullet_A$ following the defining property of the quotient inductive type $\bullet A$. □

Given an object-space type A and a meta-space type family B indexed by A, we can 'glue' them together by $\Sigma(a : A).\ B\ a$. Under $\mathfrak{ob}$, we have $\Sigma(a : A).\ B\ a \cong \Sigma(a : A).1 \cong A$. The following, and the final, piece of StcTT allows us to do better, giving us a type $(a : A) \ltimes B\ a \cong \Sigma(a : A).\ B\ a$ such that under $\mathfrak{ob}$, $(a : A) \ltimes B\ a$ is *judgementally equal* in StcTT, not just isomorphic, to A.

Axiom 4.9 (StcTT-Glue). StcTT has *strict glue types* in its universes U_i:

$$\frac{A : \bigcirc U_i \qquad B : (\{\mathfrak{ob}\} \to A) \to \{X : U_i \mid \bullet\text{-}modal\ X\}}{(a : A) \ltimes B\ a : \{U_i \mid \mathfrak{ob} \hookrightarrow A\}}$$

and isomorphisms between $(a : A) \ltimes B\ a$ and $\Sigma\ (a : \{\mathfrak{ob}\} \to A).\ B\ a$

$$glue : \{\ \Sigma(a : \{\mathfrak{ob}\} \to A).\ B\ a \ \cong\ (a : A) \ltimes B\ a\ \mid \mathfrak{ob} \hookrightarrow \pi_1\text{-}iso\}$$

where $\pi_1\text{-}iso : \{\mathfrak{ob}\} \to \Sigma(a : \{\mathfrak{ob}\} \to A).\ B\ a \cong A$ is the isomorphism that has the projection $\pi_1 : (\Sigma(a : A).\ B\ a) \to A$ as its forward direction (under $\mathfrak{ob}$, $B\ a \cong 1$ so π_1 is an isomorphism).

For all $a : \{\mathfrak{ob}\} \to A$ and $b : B\ a$, we will use the following notation in place of $glue.fwd\ (a, b)$ to signal that the value is strictly equal to a when $\mathfrak{ob}$ is assumed:

$$[\mathfrak{ob} \hookrightarrow a \mid b]\ :=\ glue.fwd\ (a, b) : (a : A) \ltimes B\ a.$$

Given an element $g : (a : A) \ltimes B\ a$, when $\mathfrak{ob}$ holds $(a : A) \ltimes B\ a$ is equal to A, so we can directly use g as an element of A. To access the second component of a glued element conveniently, we define

$$\begin{aligned} &unglue\ : (g : (a : A) \ltimes B\ a) \to B\ (\lambda\{\_ : \mathfrak{ob}\}.\ g) \\ &unglue\ g = \pi_2\ (glue.bwd\ g) \end{aligned}$$

4.2 Constructing the Logical Relation Model

Now we construct our logical-relation model to prove Theorem 4.1. Our goal is to define

$$M^* : \{[\![\mathrm{F}^{\mathrm{ha}}_\omega]\!]_{U_2} \mid \mathfrak{ob} \hookrightarrow M\}$$

such that $M^*.bool$ encodes the property that we need. Due to space constraint, this section can only be a digest of the full proof, focusing on the novel part pertaining to the computation judgements of $\mathrm{F}^{\mathrm{ha}}_\omega$. Appendix C contains a full proof that explains all the definitions slowly.

Notation 4.10. For every declaration dec in the signature of $\mathrm{F}^{\mathrm{ha}}_\omega$, we will write dec^* for $M^*.dec$ and just dec for $M.dec$. For example, $ki^* : \{U_2 \mid \mathfrak{ob} \hookrightarrow ki\}$ means $M^*.ki : \{U_2 \mid \mathfrak{ob} \hookrightarrow ki\}$.

4.2.1 Kinds and Types. Every $\mathrm{F}^{\mathrm{ha}}_\omega$-type A is interpreted as a *proof-irrelevant* predicate A^* over closed terms of A. Every kind k is interpreted as a *proof-relevant* predicate k^* over closed elements of k. In particular, for the base kind ty and every closed $\mathrm{F}^{\mathrm{ha}}_\omega$-type A, $ty^*(A)$ is the set of all proof-irrelevant predicates over A-terms. The idea here is essentially the same as Girard's [1989] technique of *reducibility candidates* in his proof of normalisation for System F, except that here (1) we employ *proof-relevant* predicates to deal with higher kinds, and (2) the object language is formulated as an equational theory rather than a reduction system.

In the language StcTT, the idea above for kinds is precisely expressed as follows:

$$\begin{aligned} &ki^* : \{U_2 \mid \mathfrak{ob} \hookrightarrow ki\} && el^* : \{ki^* \to U_1 \mid \mathfrak{ob} \hookrightarrow el\} \\ &ki^* = (\alpha : ki) \ltimes \{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \alpha\} && el^*\ g = unglue\ g \end{aligned}$$

Define the universe of meta-space propositions by $\Omega^\bullet := \{p : \Omega \mid \bullet\text{-}modal\ p\}$, which corresponds to simply the propositions in the ambient set theory. The interpretation of types and terms are

$$ty^* : \{ki^* \mid \mathfrak{ob} \hookrightarrow ty\} \qquad tm^* : \{el^*\ ty^* \to U_0 \mid \mathfrak{ob} \hookrightarrow tm\}$$
$$ty^* = [\mathfrak{ob} \hookrightarrow ty \mid (A : el\ ty) \ltimes (\{\mathfrak{ob}\}\ tm\ A) \to \Omega^\bullet] \qquad tm^*\ [\mathfrak{ob} \hookrightarrow A \mid P] = (t : tm\ A) \ltimes P\ t$$

The interpretation for the base type *bool* is the predicate $P_{can}(b)$ that the closed term b is either equal to $tt : bool$ or $ff : bool$ as follows. The predicate P_{can} is the only place in the logical-relation model M^* that is specific to canonicity, so we can in fact replace $P_{can}(b)$ with any other predicates that hold for tt and ff and get other logical-relation models.

$$bool^* : \{el^*\ ty^* \mid \mathfrak{ob} \hookrightarrow bool\} \qquad tt^* : \{tm^*\ bool^* \mid \mathfrak{ob} \hookrightarrow tt\}$$
$$bool^* = [\mathfrak{ob} \hookrightarrow bool \mid P_{can}] \qquad tt^* = [\mathfrak{ob} \hookrightarrow tt \mid \eta^\bullet\ (\mathrm{inl}\ refl)]$$
$$P_{can} : (\{\mathfrak{ob}\} \to tm\ bool) \to \Omega^\bullet \qquad ff^* : \{tm^*\ bool^* \mid \mathfrak{ob} \hookrightarrow ff\}$$
$$P_{can}\ b = \bullet(\{\mathfrak{ob}\} \to (b = tt \vee b = ff)) \qquad ff^* = [\mathfrak{ob} \hookrightarrow ff \mid \eta^\bullet\ (\mathrm{inr}\ refl)]$$

The other type/kind formers are interpreted in the standard way, which is explained in great detail in Appendix C. Here, let us only show the interpretation for the polymorphic function type:

$$\bar{\forall}^* : \{(k^* : ki^*) \to (el^*\ k^* \to el^*\ ty^*) \to el^*\ ty^* \mid \mathfrak{ob} \hookrightarrow \bar{\forall}\}$$
$$\bar{\forall}^*\ k^*\ F = [\mathfrak{ob} \hookrightarrow \bar{\forall}\ k^*\ F \mid \lambda t.\ \forall(\alpha^* : el^*\ k^*).\ unglue\ (F\ \alpha^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ (t\ \alpha^*))]$$

The logical predicate here on closed terms $t : \{\mathfrak{ob}\} \to tm\ (\bar{\forall}\ k\ F)$ of the polymorphic function type is a universal quantification $\forall(\alpha^* : el^*\ k^*)$ over all the 'logical predicate candidates' of the kind k^*, and we demand the result $t\ \alpha^*$ of applying the polymorphic function t to α^* to satisfy the logical predicate $F\ \alpha^*$. It is explained in detail in Appendix C.2 how this definition type checks.

With the F_ω-fragment of M^* defined, we automatically have the interpretation for the judgements that are derived in F_ω (Figure 1), such as *RawHFuncor* and *RawMonad*. We will also denote their interpretation by superscripting an asterisk. For example, for the judgement $tyco = (ty \Rightarrow_t ty)$ from Figure 1, its interpretation $tyco^*$ is then $ty^* \Rightarrow_{k^*} ty^*$.

4.2.2 Computations. What remains is to define the computation fragment of $\mathrm{F}^{\mathrm{ha}}_\omega$ (Section 2.3) for our logical-relation model M^*. Because in $\mathrm{F}^{\mathrm{ha}}_\omega$ the only way to 'observe' a computation is to evaluate it with a raw monad using *eval* ($\mathrm{F}^{\mathrm{ha}}_\omega$-7), a natural attempt to define the logical predicate $P_{co}(c)$ for computations is to assert that the computation $c : co\ H\ A$ evaluated with every semantic raw-monad m^* (i.e. a syntactic raw-monad m with a proof of m satisfying the proof-relevant logical predicate) yields a term that satisfies the logical predicate $m^*.0\ A^*$:

$$P^{wrong}_{co} : \{H^*, A^*\} \to (\{\mathfrak{ob}\} \to co\ H^*\ A^*) \to \Omega^\bullet$$
$$P^{wrong}_{co}\ c = \forall(m^* : MonadAlg^*\ H^*).\ unglue\ (m^*.0\ A^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ eval\ m^*\ A^*\ c)$$

However, this definition will not work later when showing that the term constructor *let-in* ($\mathrm{F}^{\mathrm{ha}}_\omega$-2) satisfies its logical predicate. That is, we need to prove

$$\forall(m^* : MonadAlg^*\ H^*).\ unglue\ (m^*.0\ B^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ \ eval\ m^*\ B^*\ (let\text{-}in\ c\ f)\ ),$$

but as mentioned in Remark 2.6, $\mathrm{F}^{\mathrm{ha}}_\omega$ does not have the law saying that *eval* commutes with *let-in*, so we have no way to further simplify the shaded part above to make use the assumptions that c and f satisfy their logical predicates.

The way to fix this problem is to use the idea of $\top\top$-*lifting* [Lindley and Stark 2005]: we strengthen P^{wrong}_{co} above to quantify over all 'good continuations' k of the computation c, and we demand *eval* m (*let-in* c k) to satisfy its logical predicate. Here a continuation k is 'good' if k followed by *eval* sends input satisfying its logical predicate to output satisfying its logical predicate, which can

be succinctly expressed by a function $k^* : tm^*\ A^* \to tm^*\ (m^*.0\ R^*)$. Precisely, the type of 'good' continuations accepting A^*-values is the following record:

$$\begin{aligned}
&\textsc{record}\ Con\ (H^* : RawHFunctor^*)\ (A^* : el^*\ ty^*) : U_1\ \textsc{where}\\
&\quad m^* : MonadAlg^*\ H^*\\
&\quad R^* : el^*\ ty^*\\
&\quad k : \{\mathfrak{ob}\} \to A^* \to co\ H^*\ R^*\\
&\quad k^* : \{tm^*\ A^* \to tm^*\ (m^*.0\ R^*) \mid \mathfrak{ob} \hookrightarrow \lambda a.\ eval\ m^*\ R^*\ (k\ a)\}
\end{aligned}$$

and the correct definition of P_{co} and the interpretation of computation judgements co^* is

$$\begin{aligned}
&P_{co} : \{H^*, A^*\} \to (\{\mathfrak{ob}\} \to co\ H^*\ A^*) \to \Omega^\bullet\\
&P_{co}\ c = \forall(K : Con\ H^*\ A^*).\ unglue\ (K.m^*.0\ \ K.R^*)\ (\lambda\{_ : \mathfrak{ob}\}.\\
&\qquad\qquad eval\ K.m^*\ K.R^*\ (let\text{-}in\ c\ K.k))\\
&co^* : \{HFunctor^* \to el^*\ ty^* \to U_0 \mid \mathfrak{ob} \hookrightarrow co\}\\
&co^*\ H^*\ A^* = (c : co\ H^*\ A^*) \ltimes P_{co}\ c
\end{aligned}$$

Based on this definition of co^*, the interpretation of all constructs of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ pertaining to computations can be defined and are shown in detail in Appendix C.4. Here we only show the case for the term former *let-in*, which was previously problematic:

$$\begin{aligned}
&let\text{-}in^* : \{\{H^*, A^*, B^*\} \to co^*\ H^*\ A^* \to (co^*\ H^*\ A^* \to co^*\ H^*\ B^*)\\
&\qquad\qquad \to co^*\ H^*\ B \mid \mathfrak{ob} \hookrightarrow let\text{-}in\}\\
&let\text{-}in^*\ c\ f = [\mathfrak{ob} \hookrightarrow let\text{-}in\ c\ f \mid \lambda(K : Con\ H^*\ A^*).\ unglue\ c\ K']
\end{aligned}$$

where each field of $K' : Con\ H^*\ B^*$ is defined as follows:

$$\begin{aligned}
&K'.m^* = K.m^* \qquad K'.k = \lambda\{_ : \mathfrak{ob}\}\ a.\ let\text{-}in\ (f\ a)\ K.k\\
&K'.R^* = K.R^* \qquad K'.k^* = \lambda a.\ [\mathfrak{ob} \hookrightarrow eval\ K'.m^*\ K'.R^*\ (let\text{-}in\ (f\ a)\ K.k) \mid unglue\ (f\ a)\ K]
\end{aligned}$$

The definition of $K'.k^*$ type checks because $f\ a : co^*\ H^*\ B^*$, so $unglue\ (f\ a) : P_{co}\ (f\ a)$, so by the definition of P_{co}, the type of $unglue\ (f\ a)\ K$ is

$$unglue\ (K.m^*.0\ \ K.R^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ eval\ K.m^*\ K.R^*\ (let\text{-}in\ (f\ a)\ K.k))$$

which is indeed the type of proofs that the syntactic component of $K'.k^*$ satisfies the logical predicate of the type $k.m^*.0\ K.R^*$.

To summarise, we have established the 'fundamental lemma' for the logical relation of $\mathrm{F}^{\mathrm{ha}}_{\omega}$.

Lemma 4.11 (Fundamental). *In the language* StcTT, *given any* $P : (\{\mathfrak{ob}\} \to M.tm\ M.bool) \to \Omega^\bullet$ *with* $t : P(M.tt)$ *and* $f : P(M.ff)$, *there is an* $M^* : \{[\![F^{ha}_{\omega}]\!]_{U_2} \mid \mathfrak{ob} \hookrightarrow M\}$ *such that*

$$M^*.tm\ M^*.bool = (b : M.tm\ M.bool) \ltimes P\ b.$$

4.3 External Closed Term Canonicity

Finally, we can now prove canonicity (Theorem 4.1) using our logical-relation modal M^*.

Proof of Theorem 4.1. The denotation of the StcTT-type $tm^*\ bool^*$ in $\mathrm{GL}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$ is the following object (strictly speaking the denotation could be an object only *isomorphic* to this object, but pretending they are strictly equal will not cause problems in this proof):

$$B^* := \langle \mathrm{Y}(tm\ bool) \in \mathrm{Pr}\,(\mathrm{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}),\ \{t : 1 \to \mathrm{Y}(tm\ bool)) \mid (t = \mathrm{Y}\ tt) \vee (t = \mathrm{Y}\ ff)\},\ j\rangle$$

where j is the inclusion function into $\{t : 1 \to \mathrm{Y}(tm\ bool))\}$. For every closed term $b : tm\ bool$ in $\mathrm{F}^{\mathrm{ha}}_{\omega}$, its interpretation in the model M^* is a morphism $1 \to B^*$ in $\mathrm{G}_{\mathrm{L}}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$ as follows:

$$\begin{array}{ccc} \{*\} & \longrightarrow & \{t \mid (t = \mathrm{Y}\ tt) \vee (t = \mathrm{Y}\ ff)\} \\ \downarrow & & \downarrow{\scriptstyle j} \\ \{*\} & \xrightarrow[\lambda *.\ \mathrm{Y}b]{} & \{t : 1 \to \mathrm{Y}(tm\ bool)\} \end{array}$$

The commutativity of this diagram entails $\mathrm{Y}b = \mathrm{Y}tt$ or $\mathrm{Y}b = \mathrm{Y}ff$, so $b = tt$ or $b = ff$ since Yoneda embedding is fully faithful. Moreover, $b = tt$ and $b = ff$ cannot be true at the same time because tt and ff have different interpretations in the realizability model in Section 3.2. □

Corollary 4.12. An immediate consequence of canonicity of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is that the realizability model in Section 3 is *adequate* in the sense that if two closed Boolean terms b_1 and b_2 have the same denotation in the realizability model, they must be judgementally equal. This is because canonicity says that both b_1 and b_2 are either equal to tt or ff, which have different interpretations in the realizability model, so $b_1 = b_2$ must be true if their realizability interpretation is the same.

Remark 4.13. Apart from canonicity, Lemma 4.11 can be also used to show other *parametricity* results about terms of $\mathrm{F}^{\mathrm{ha}}_{\omega}$; for example, for a closed term $t : tm\ (\bar{\forall}\ ty\ (\lambda\alpha.\ \alpha \Rightarrow_t \alpha))$, t applied to every closed type A and closed term $a : tm\ A$ is equal a. Even more pleasantly, we can obtain a binary (or n-ary) logical-relation model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ from the seemingly unary logical-predicate model Lemma 4.11 by interpreting StcTT in a different glued topos, without modifying the definition of M^* at all. These results are elaborated in Appendix D.

Remark 4.14. We based $\mathrm{F}^{\mathrm{ha}}_{\omega}$ on fine-grain call-by-value (FGCBV) rather than call-by-push-value (CBPV) since the theory of higher-order algebraic effects [Yang and Wu 2023] is based on monads rather than adjunctions, but CBPV is also possible and we sketch the judgements for a CBPV variant of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ without stack judgements here. Again, starting with F_{ω}, instead of $co : RawHFunctor \to el\ ty \to \mathbb{J}$, we add to F_{ω} a new kind for *computation types* and a judgement for *computation terms*:

$$cty : RawHFunctor \to ki \qquad\qquad ctm : \{H\} \to el\ (cty\ H) \to \mathbb{J}$$

We then add two new type formers for value-returning computations (called *returners* by Levy [2003]) and thunk values:

$$F : (H : RawHFunctor) \to el\ ty \to el\ (cty\ H) \qquad\qquad U : \{H\} \to el\ (cty\ H) \to el\ ty$$

We have value returning and sequential composition as usual:

$$\begin{aligned} val\ &: \{H, A\} \to tm\ A \to ctm\ (F\ H\ A) \\ let\text{-}in &: \{H, A, X\} \to ctm\ (F\ H\ A) \to (tm\ A \to ctm\ X) \to ctm\ X \end{aligned}$$

Note that the second argument of *let-in* can be an arbitrary computation type $X : el\ (cty\ H)$ rather than just value-returning computations $F\ H\ A : el\ (cty\ H)$. Terms of a thunk type are in bijection with the terms of the computation type:

$$U\text{-}iso : \{H\}\ \{X : el\ (cty\ H)\} \to tm\ (U\ X) \cong ctm\ X$$

The judgement $co\ H\ A$ in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ then corresponds to $ctm\ (F\ H\ A)$ in CBPV. What we have in CBPV but not in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ are *function computations* from a value type A to a computation type X:

$$\begin{aligned} \_\Rightarrow_c\_\ &: \{H\} \to (A : el\ ty) \to (X : el\ (cty\ H)) \to el\ (cty\ H) \\ \Rightarrow_c\text{-}iso &: \{H, A, X\} \to ctm\ (A \Rightarrow_c X) \cong (tm\ A \to ctm\ X) \end{aligned}$$

Finally, we have H-operations and evaluation by raw monads with H-operations:

$$
\begin{aligned}
&\mathit{th} : \mathit{RawHFunctor} \to \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty}\\
&\mathit{th}\ H\ A = U\ (F\ H\ A)\\
&\mathit{op} : \{H, A, X\} \to \mathit{tm}\ (H\ (\mathit{th}\ H)\ A) \to (\mathit{tm}\ A \to \mathit{ctm}\ X) \to \mathit{ctm}\ X\\
&\mathit{eval} : \{H, A\} \to (m : \mathit{MonadAlg}\ H) \to \mathit{ctm}\ (F\ H\ A) \to \mathit{tm}\ (m\ .0\ A)
\end{aligned}
$$

We see no obvious difficulties in adapting our results for $\mathrm{F}^{\mathrm{ha}}_{\omega}$ to this CBPV variant by interpreting F and U using the Eilenberg-Moore adjunction of the monads that we used to model $\mathrm{F}^{\mathrm{ha}}_{\omega}$.

5 Related Work

The most related work is the line of research on (higher-order) algebraic effects and handlers, and we have discussed the position of this paper within this line of research in Section 1. In this section, we discuss some more aspects of related work that were not discussed in Section 1.

• In the context of handlers of algebraic effects, the paper by Wu et al. [2014] seems to be the first to consider higher-order operations. Although the examples of (higher-order) operations considered in this paper are all what are later called scoped operations, the framework in this paper is actually designed for general higher-order effects that can be given as higher-order functors, similar to Yang and Wu [2023], van den Berg and Schrijvers [2024], and the present paper here, except that Wu et al. [2014] demand the signature (higher-order) functors to come with a *weave* operation, which is used for modularly combining handlers of different effects. This design of *weave* seems inherently tied to effects similar to mutable state and has not been further developed since then.

• The paper by Wu et al. [2014] is a practically-minded paper presented in Haskell, and the underlying mathematics for higher-order effects was not clear at its time. Therefore in the following years, several authors studied the categorical foundation (and practical applications) of several *special cases* of Wu et al.'s [2014] framework, including *scoped effects* by Piróg et al. [2018] and Yang et al. [2022], *latent effects* by van den Berg et al. [2021], *heafty algebras* by Bach Poulsen and van der Rest [2023]. All these families can be implemented in our calculus $\mathrm{F}^{\mathrm{ha}}_{\omega}$.

• After this trend of diversification, the families of higher-order algebraic effects are re-unified by Yang and Wu [2023] and van den Berg and Schrijvers [2024]. Yang and Wu [2023] presented (1) a general categorical framework for defining higher-order algebraic effects (with equations) as algebraic theories of operations on monoids, and (2) constructions for combining handlers in a modular way. In contrast, the present paper studies a *programming language* $\mathrm{F}^{\mathrm{ha}}_{\omega}$ for (equation-less) higher-order algebraic effects. Yang and Wu's [2023] constructions of modular handlers can be readily used in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ but they are not baked in the language. Another difference is that Yang and Wu [2023] worked at the level of abstraction of *monoids in monoidal categories*, encompassing not just monads but also applicative functors, graded monads, etc.

van den Berg and Schrijvers's [2024] work is also a general framework for (equation-less) higher-order algebraic effects (following approach ii), presented as a Haskell library. Their paper provides a plethora of interesting concrete examples of higher-order effects and handlers, which we did not explore in this paper but in principle can be programmed in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ too.

• The papers discussed above are all about category theory or programming libraries for higher-order effects. To our knowledge, the only account of standalone programming languages for higher-order effect handlers so far is Bosman et al.'s [2024] work on λ_{sc}, *calculus for scoped effects and handlers*. The key differences between their λ_{sc} and our $\mathrm{F}^{\mathrm{ha}}_{\omega}$ are that (1) λ_{sc} is designed for algebraic and scoped operations, while $\mathrm{F}^{\mathrm{ha}}_{\omega}$ supports arbitrary higher-order operations that can be given as higher-order functor; (2) λ_{sc} uses operations with an continuation argument together with mere type constructors as handlers (approach (ii) in Section 1), while we uses raw monads as handlers; (3) λ_{sc} has a baked-in type-and-effect system, while $\mathrm{F}^{\mathrm{ha}}_{\omega}$ supports it as a user-level construct

(Remark 2.7). Also, the work on λ_{sc} focused more on the user-facing aspects of the language, such as type inference, whereas we have focused on the meta-theoretic properties of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ – an equational theory validated by the 'compiler' (the realizability model), canonicity, and parametricity.

6 Future Prospects

In this paper, we defined System $\mathrm{F}^{\mathrm{ha}}_{\omega}$, an extension of System F_{ω} with (equation-less) higher-order algebraic effects. We gave a denotational model of it using realizability and proved the canonicity of closed terms using synthetic Tait computability. A further extension with general recursion was introduced and was modelled using synthetic domain theory. Future work abound:

(1) We should be able to prove normalisation of open $\mathrm{F}^{\mathrm{ha}}_{\omega}$-terms following the lines of Sterling [2021]. A subtlety is that we will need in StcTT an *impredicative* universe U_0 that contains the normalisation model and the syntactic model $M : \{\mathfrak{ob}\} \to [\![\mathrm{F}^{\mathrm{ha}}_{\omega}]\!]_{U_0}$, which means that in the first place the category of judgements of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ has to be constructed in some impredicative universe of the ambient meta-theory, and this should be workable since we did not rely on anything classical.

(2) The language $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is a core calculus. Although important features such as effect systems and modular models may be implemented as libraries, for practical use they should be supported in a more seamless way, such as by elaboration or by directly baking into the language.

(3) Only monadic computations are considered in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ for simplicity. Generalising $\mathrm{F}^{\mathrm{ha}}_{\omega}$ from monads to arbitrary user-defined monoidal structures should be very useful. There does not seem to be much theoretical obstacle, but designing a user-friendly syntax may be challenging.

(4) Efficiency of implementations is also an interesting aspect. Note that a continuation-passing style translation for $\mathrm{F}^{\mathrm{ha}}_{\omega}$-computations can be readily extracted from the realizability model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$, but it should be possible to further optimise out all the overhead of effect handlers for statically known computations and handlers by *meta-programming*.

(5) The biggest limitation of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is probably that equational axioms on effects are not formally supported, since we did not include dependent types, in particular equality/identity types. Adding (intensional) identity types to $\mathrm{F}^{\mathrm{ha}}_{\omega}$ is straightforward:

$$
\begin{aligned}
&\mathit{Id} \quad : \{A : \mathit{el}\ \mathit{ty}\} \to (a, b : \mathit{tm}\ A) \to \mathit{el}\ \mathit{ty}\\
&\mathit{refl} : \{A : \mathit{el}\ \mathit{ty}\} \to (a : \mathit{tm}\ A) \to \mathit{el}\ (\mathit{Id}\ a\ a)\\
&\mathit{J} \quad : \{A : \mathit{el}\ \mathit{ty}\}\ \{C : (a, b : \mathit{tm}\ A) \to \mathit{Id}\ a\ b \to \mathit{el}\ \mathit{ty}\}\\
&\qquad \to ((x : \mathit{tm}\ A) \to C\ x\ x\ (\mathit{refl}\ x)) \to \{a, b : \mathit{tm}\ A\} \to (p : \mathit{Id}\ a\ b) \to C\ a\ b\ p
\end{aligned}
$$

and Σ and Π types are no more difficult. With *Id* we can then define in $\mathrm{F}^{\mathrm{ha}}_{\omega}$ *law-abiding* functors, monads, higher-order functors, equational systems, etc. These allow us to ensure that a user-defined monad to be used with *eval* must satisfy the equations associated to the operations. However, what is challenging is adding equalities from the algebraic theory of effectful operations to the computation judgements without breaking the canonicity of the type theory. This difficulty is the same as that of adding *quotient types* to types theories without breaking canonicity, which is possible in *observational type theory* [Pujet and Tabareau 2022] and *cubical type theory* [Coquand et al. 2018]. Apart from the difficulty with quotients, having general recursion, impredicative polymorphism, and dependent types all together is not straightforward either because the category of well complete objects that we used in Section 2.4 as predomains is unlikely locally cartesian closed, so we need to find another category of predomains.

References

Agda Developers. 2025. *Agda.* https://agda.readthedocs.io/

Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. 1995. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, David Pitt, David E. Rydeheard, and Peter Johnstone (Eds.). Springer Berlin Heidelberg, 182–199.

Steve Awodey, Jonas Frey, and Sam Speight. 2018. Impredicative Encodings of (Higher) Inductive Types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, 76–85. doi:10.1145/3209108.3209130

Casper Bach Poulsen and Cas van der Rest. 2023. Hefty Algebras: Modular Elaboration of Higher-Order Algebraic Effects. *Proc. ACM Program. Lang.* 7, POPL (2023). doi:10.1145/3571255

Andrej Bauer. 2022. Notes on Realizability. https://github.com/andrejbauer/notes-on-realizability Lecture notes for the Midlands Graduate School 2022 lecture series on realizability.

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* Volume 10, Issue 4 (2014). doi:10.2168/LMCS-10(4:9)2014

J.A. Bergstra and J.W. Klop. 1985. Algebra of communicating processes with abstraction. *Theoretical Computer Science* 37 (1985), 77–121. doi:10.1016/0304-3975(85)90088-x

Francis Borceux. 1994. *Handbook of Categorical Algebra: Volume 3, Sheaf Theory*. Vol. 3. Cambridge University Press.

Roger Bosman, Birthe van den Berg, Wenhao Tang, and Tom Schrijvers. 2024. A Calculus for Scoped Effects & Handlers. *Logical Methods in Computer Science* Volume 20, Issue 4, Article 17 (2024). doi:10.46298/lmcs-20(4:17)2024

Thierry Coquand, Carl Gunter, and Glynn Winskel. 1989. Domain theoretic models of polymorphism. *Information and Computation* 81, 2 (1989), 123–167. doi:10.1016/0890-5401(89)90068-0

Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. On Higher Inductive Types in Cubical Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 255–264. doi:10.1145/3209108.3209197

Roy L. Crole. 1994. *Categories for Types*. Cambridge University Press.

Tom de Jong. 2024. Categorical Realizability. https://github.com/andrejbauer/notes-on-realizability Lecture notes for the course on Categorical Realizability at the Midlands Graduate School 2024.

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 429–442. doi:10.1145/2500365.2500582

Marcelo Fiore. 2022. Semantic Analysis of Normalisation by Evaluation for Typed Lambda Calculus. arXiv:2207.08777 [cs.LO] doi:10.48550/arXiv.2207.08777.

Marcelo Fiore and Gordon Plotkin. 1997. An extension of models of Axiomatic Domain Theory to models of Synthetic Domain Theory. In *Computer Science Logic*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Dirk Dalen, and Marc Bezem (Eds.). Vol. 1258. Springer Berlin Heidelberg, 129–149. doi:10.1007/3-540-63172-0_36

Marcelo Fiore and Giuseppe Rosolini. 1997. Two models of synthetic domain theory. *Journal of Pure and Applied Algebra* 116, 1–3 (1997), 151–162. doi:10.1016/S0022-4049(96)00164-8

Peter Freyd. 1978. On proving that 1 is an indecomposable projective in various free categories. *Manuscript* (1978).

Dan Frumin, Amin Timany, and Lars Birkedal. 2024. Modular Denotational Semantics for Effects with Guarded Interaction Trees. *Proc. ACM Program. Lang.* 8, POPL (2024). doi:10.1145/3632854

Jeremy Gibbons and Ralf Hinze. 2011. Just do it: simple monadic equational reasoning. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 2–14. doi:10.1145/2034773.2034777

Jean-Yves Girard. 1972. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État. Université Paris VII.

Jean-Yves Girard. 1986. The System F of variable types, fifteen years later. *Theoretical Computer Science* 45 (1986), 159–192. doi:10.1016/0304-3975(86)90044-7

Jean-Yves Girard. 1989. *Proofs and types*. Cambridge University Press.

Daniel Gratzer, Michael Shulman, and Jonathan Sterling. 2022. Strict universes for Grothendieck topoi. arXiv:2202.12012 https://arxiv.org/abs/2202.12012

Harrison Grodin, Yue Niu, Jonathan Sterling, and Robert Harper. 2024. Decalf: A Directed, Effectful Cost-Aware Logical Framework. *Proceedings of the ACM on Programming Languages* 8, POPL (2024), 273–301. doi:10.1145/3632852

Robert Harper. 2016. *Practical Foundations for Programming Languages* (2nd ed.). Cambridge University Press, Cambridge.

Robert Harper, Furio Honsell, and Gordon Plotkin. 1993. A framework for defining logics. *J. ACM* 40, 1 (1993), 143–184. doi:10.1145/138027.138060

Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In *Semantics and Logics of Computation (Publications of the Newton Institute)*, Andrew M. Pitts and P.Editors Dybjer (Eds.). Cambridge University Press, 79–130. doi:10.1017/CBO9780511526619.004

Xu Huang. 2023. Synthetic Tait Computability the Hard Way. arXiv:2310.02051 [cs.LO] https://arxiv.org/abs/2310.02051

J.M.E. Hyland. 1991. First steps in synthetic domain theory. In *Category Theory*, Aurelio Carboni, Maria Cristina Pedicchio, and Guiseppe Rosolini (Eds.). Vol. 1488. Springer Berlin Heidelberg, 131–156. doi:10.1007/BFb0084217

Mamuka Jibladze. 1997. A presentation of the initial lift-algebra. *Journal of Pure and Applied Algebra* 116, 1–3 (1997), 185–198. doi:10.1016/S0022-4049(96)00108-9

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (Jan. 2007), 1–82. doi:10.1017/s0956796806006034

Ohad Kammar and Gordon Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 349–360. doi:10.1145/2103656.2103698

Shin-ya Katsumata, Tetsuya Sato, and Tarmo Uustalu. 2018. Codensity Lifting of Monads and its Dual. *Logical Methods in Computer Science* Volume 14, Issue 4 (2018). doi:10.23638/LMCS-14(4:6)2018

Shin-ya Katsumata. 2005. *A Semantic Formulation of* ⊤⊤*-Lifting and Logical Predicates for Computational Metalanguage*. Lecture Notes in Computer Science, Vol. 3634. Springer Berlin Heidelberg, 87–102. doi:10.1007/11538363_8

Oleg Kiselyov, Shin-Cheng Mu, and Amr Sabry. 2021. Not by equations alone. *Journal of Functional Programming* 31 (2021). doi:10.1017/S0956796820000271

Søren Bøgh Lassen. 1998. *Relational Reasoning about Functions and Nondeterminism*. Ph. D. Dissertation. Aarhus University. https://www.brics.dk/DS/98/2/ Series: BRICS Dissertation Series.

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. Association for Computing Machinery, New York, NY, USA, 283–294. doi:10.1145/1411204.1411245

Paul Blain Levy. 2003. *Call-By-Push-Value*. Springer Netherlands. doi:10.1007/978-94-007-0954-6

Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. doi:10.1016/S0890-5401(03)00088-9

Sam Lindley and Ian Stark. 2005. Reducibility and ⊤⊤-Lifting for Computation Types. In *Typed Lambda Calculi and Applications (Lecture Notes in Computer Science, Vol. 3461)*, Paweł Urzyczyn (Ed.). Springer Berlin Heidelberg, 262–277. doi:10.1007/11417170_20

John Longley and Alex Simpson. 1997. A uniform approach to domain theory in realizability models. *Mathematical Structures in Computer Science* 7, 5 (1997), 469–505. doi:10.1017/S0960129597002387

John R. Longley. 1995. *Realizability Toposes and Language Semantics*. Ph. D. Dissertation. University of Edinburgh. https://era.ed.ac.uk/handle/1842/402

J. M. Lucassen and D. K. Gifford. 1988. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, 47–57. doi:10.1145/73560.73564

Zhaohui Luo. 1994. *Computation and reasoning: a type theory for computer science*. Oxford University Press.

Zhaohui Luo, Sergei Soloviev, and Tao Xue. 2013. Coercive subtyping: Theory and implementation. *Information and Computation* 223 (2013), 18–42. doi:10.1016/j.ic.2012.10.020

Saunders Mac Lane and Ieke Moerdijk. 1994. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer New York. doi:10.1007/978-1-4612-0927-0

Per Martin-Löf. 1975a. About models for intuitionistic type theories and the notion of definitional equality. In *Studies in Logic and the Foundations of Mathematics*. Vol. 82. Elsevier, 81–109.

Per Martin-Löf. 1975b. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium 73 Proceedings of the Logic Colloquium*, H. E. Rose and J. C. Shepherdson (Eds.). Elsevier, 73–118.

Per Martin-Löf. 1987. The Logic of Judgements. https://raw.githubusercontent.com/michaelt/martin-lof/master/pdfs/The-logic-of-judgements-typeset-1987.pdf Talk at Workshop on General Logic, Laboratory for Foundations of Computer Science, University of Edinburgh, 23-27 February 1987.

Cristina Matache, Sam Lindley, Sean Moss, Sam Staton, Nicolas Wu, and Zhixuan Yang. 2025. Scoped Effects, Scoped Operations, and Parameterized Algebraic Theories. *ACM Trans. Program. Lang. Syst.* 47, 2, Article 8 (2025), 33 pages. doi:10.1145/3731678

Yue Niu, Jonathan Sterling, Harrison Grodin, and Robert Harper. 2022. A cost-aware logical framework. *Proc. ACM Program. Lang.* 6, POPL, Article 9 (2022), 31 pages. doi:10.1145/3498670

Wesley Phoa. 1991. *Domain Theory in Realizability Toposes*. Ph. D. Dissertation. University of Edinburgh. https://www.lfcs.inf.ed.ac.uk/reports/91/ECS-LFCS-91-171/

Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. Association for Computing Machinery, 809–818. doi:10.1145/3209108.3209166

Gordon Plotkin. 1973. *Lambda Definability and Logical Relations.* Memorandum SAI-RM-4. University of Edinburgh. https://homepages.inf.ed.ac.uk/gdp/publications/logical_relations_1973.pdf

Gordon Plotkin. 1977. LCF considered as a programming language. *Theoretical Computer Science* 5, 3 (1977), 223–255. doi:10.1016/0304-3975(77)90044-5

Gordon Plotkin. 1980. Lambda-Definability in the Full Type Hierarchy. In *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, J. P. Seldin and J. R. Hindley (Eds.). Academic Press, 363–373. https://homepages.inf.ed.ac.uk/gdp/publications/Lambda_Definability.pdf

Gordon Plotkin and John Power. 2001. Semantics for algebraic operations. *Electronic Notes in Theoretical Computer Science* 45 (2001), 332–345. doi:10.1016/S1571-0661(04)80970-8

Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures, 5th International Conference (FOSSACS 2002)*, Mogens Nielsen and Uffe Engberg (Eds.). Springer, 342–356. doi:10.1007/3-540-45931-6_24

Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (2003), 69–94. doi:10.1023/A:1023064908962

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, 80–94. doi:10.1007/978-3-642-00590-9_7

Gordon Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (2013). doi:10.2168/lmcs-9(4:23)2013

Loïc Pujet and Nicolas Tabareau. 2022. Observational equality: now for good. *Proc. ACM Program. Lang.* 6, POPL, Article 32 (2022), 27 pages. doi:10.1145/3498693

Bernhard Reus. 1996. *Program verification in synthetic domain theory.* Ph. D. Dissertation. Ludwig Maximilian University of Munich. https://www2.mathematik.tu-darmstadt.de/~streicher/THESES/reus.pdf

Bernhard Reus. 1999. Formalizing Synthetic Domain Theory. *Journal of Automated Reasoning* 23, 3 (1999), 411–444. doi:10.1023/A:1006258506401

Bernhard Reus and Thomas Streicher. 1999. General synthetic domain theory – a logical approach. *Mathematical Structures in Computer Science* 9, 2 (1999), 177–223. doi:10.1017/S096012959900273X

John C. Reynolds. 1983. Types, Abstraction and Parametric Polymorphism. In *IFIP Congress*.

Giuseppe Rosolini. 1986. *Continuity and effectiveness in topoi.* Ph. D. Dissertation. University of Oxford.

Dana S. Scott. 1993. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science* 121, 1 (1993), 411–440. doi:10.1016/0304-3975(93)90095-B

Alex Simpson. 2004. Computational adequacy for recursive types in models of intuitionistic set theory. *Annals of Pure and Applied Logic* 130, 1-3 (2004), 207–275. doi:10.1016/j.apal.2003.12.005

Alex K. Simpson. 1999. Computational Adequacy in an Elementary Topos. In *Computer Science Logic*. Vol. 1584. Springer Berlin Heidelberg, 323–342. doi:10.1007/10703163_22 Series Title: Lecture Notes in Computer Science.

R. Statman. 1985. Logical relations and the typed λ-calculus. *Information and Control* 65, 2 (1985), 85–97. doi:10.1016/S0019-9958(85)80001-2

Jonathan Sterling. 2021. *First Steps in Synthetic Tait Computability: The Objective Metatheory of Cubical Type Theory.* Ph. D. Dissertation. Carnegie Mellon University. doi:10.5281/zenodo.6990769 Version 1.1, revised May 2022.

Jonathan Sterling. 2022. Naïve logical relations in synthetic Tait computability. (June 2022). Unpublished manuscript.

Jonathan Sterling. 2023. Adequacy of sheaf semantics of noninterference. https://www.jonmsterling.com/jms-005Z.xml Erratum.

Jonathan Sterling and Carlo Angiuli. 2021. Normalization for Cubical Type Theory. *Proceedings - Symposium on Logic in Computer Science* 2021-June (2021), 1–22. doi:10.1109/LICS52264.2021.9470719 arXiv: 2101.11479.

Jonathan Sterling and Robert Harper. 2021. Logical Relations as Types: Proof-Relevant Parametricity for Program Modules. *J. ACM* 68, 6, Article 41 (2021). doi:10.1145/3474834

Jonathan Sterling and Robert Harper. 2022. Sheaf Semantics of Termination-Insensitive Noninterference. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 228)*, Amy P. Felty (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 5:1–5:19. doi:10.4230/LIPIcs.FSCD.2022.5

Thomas Streicher. 2006. *Domain-theoretic foundations of functional programming.* World Scientific Publishing Company.

Thomas Streicher. 2017. Realizability. https://www2.mathematik.tu-darmstadt.de/~streicher/REAL/REAL.pdf Lecture notes.

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. http://www.jstor.org/stable/2271658

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics.* https://homotopytypetheory.org/book.

Birthe van den Berg and Tom Schrijvers. 2024. A framework for higher-order effects & handlers. *Sci. Comput. Program.* 234, C (2024), 32 pages. doi:10.1016/j.scico.2024.103086

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent Effects for Reusable Language Components. In *Programming Languages and Systems*, Hakjoo Oh (Ed.). Springer International Publishing, Cham, 182–201.

Jaap van Oosten. 2008. *Realizability: an introduction to its categorical side* (1st ed.). Elsevier, Oxford.

Philip Wadler. 1989. Theorems for free!. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture - FPCA '89*, Vol. 19. ACM Press, 347–359. doi:10.1145/99370.99404

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell - Haskell '14* (2014), 1–12. doi:10.1145/2633357.2633358

Zhixuan Yang. 2025. Revisiting the Logical Framework for Locally Cartesian Closed Categories. (2025). https://yangzhixuan.github.io/pdf/lcclf.pdf Manuscript.

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. 2022. *Structured Handling of Scoped Effects*. Springer International Publishing, 462–491. doi:10.1007/978-3-030-99336-8_17

Zhixuan Yang and Nicolas Wu. 2021. Reasoning about Effect Interaction by Fusion. *Proc. ACM Program. Lang.* 5, ICFP, Article 73 (2021), 29 pages. doi:10.1145/3473578

Zhixuan Yang and Nicolas Wu. 2023. Modular Models of Monoids with Operations. *Proc. ACM Program. Lang.* 7, ICFP, Article 208 (2023), 38 pages. doi:10.1145/3607850

A Complete Signatures of the Languages

This appendix collects the full signatures of the languages in the paper.

A.1 Signature of System $\mathrm{F}_{\omega}^{\mathrm{ha}}$

The following is the signature of System $\mathrm{F}_{\omega}^{\mathrm{ha}}$ from Section 2 for easy reference.

- Kinds

$$
\begin{aligned}
&ki && : \mathbb{J} \\
&el && : ki \to \mathbb{J} \\
&ty && : ki \\
&\_\Rightarrow_k\_ && : ki \to ki \to ki
\end{aligned}
$$

- Elements of the kind of types

$$
\begin{aligned}
&unit && : el\ ty \\
&bool && : el\ ty \\
&\_\Rightarrow_t\_ && : el\ ty \to el\ ty \to el\ ty \\
&\bar{\forall} && : (k : ki) \to (el\ k \to el\ ty) \to el\ ty
\end{aligned}
$$

- Elements of function kinds

$$
\begin{aligned}
&\textsc{record}\ A \cong B : \mathbb{J}\ \textsc{where} \\
&\quad fwd\ : A \to B \\
&\quad bwd : B \to A \\
&\quad \_ \quad : (a : A) \to bwd\ (fwd\ a) = a \\
&\quad \_ \quad : (b : B) \to fwd\ (bwd\ b) = b \\
&\Rightarrow_k\text{-}iso : \{A, B : ki\} \to el\ (A \Rightarrow_k B) \cong (el\ A \to el\ B)
\end{aligned}
$$

- Terms

$$
\begin{aligned}
&tm && : el\ ty \to \mathbb{J} \\
&unit\text{-}iso && : tm\ unit \cong 1 \\
&\Rightarrow_t\text{-}iso && : \{A, B : el\ ty\} \to tm\ (A \Rightarrow_t B) \cong (tm\ A \to tm\ B) \\
&\bar{\forall}\text{-}iso && : \{k : \_\}\ \{A : \_\} \to tm\ (\bar{\forall}\ k\ A) \cong ((\alpha : el\ k) \to tm\ (A\ \alpha))
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{tt} && : \mathit{tm}\ \mathit{bool}\\
&\mathit{ff} && : \mathit{tm}\ \mathit{bool}
\end{aligned}
$$

- Functors

$$
\begin{aligned}
&\mathit{tyco} : \mathit{ki}\\
&\mathit{tyco} = (\mathit{ty} \Rightarrow_k \mathit{ty})\\
&\mathit{fmap\text{-}ty} : (F : \mathit{el}\ \mathit{tyco}) \to \mathit{el}\ \mathit{ty}\\
&\mathit{fmap\text{-}ty}\ F = \bar{\forall}\ \mathit{ty}\ (\lambda\alpha.\ \bar{\forall}\ \mathit{ty}\ (\lambda\beta.\ (\alpha \Rightarrow_t \beta) \Rightarrow_t (F\ \alpha \Rightarrow_t F\ \beta)))\\
&\textsc{record}\ \mathit{RawFunctor} : \mathbb{J}\ \textsc{where}\\
&\quad 0 : \mathit{el}\ \mathit{tyco}\\
&\quad \mathit{fmap} : \mathit{tm}\ (\mathit{fmap\text{-}ty}\ 0)
\end{aligned}
$$

- Monads

$$
\begin{aligned}
&\textsc{record}\ \mathit{RawMonad} : \mathbb{J}\ \textsc{where}\\
&\quad 0 \quad : \mathit{el}\ \mathit{tyco}\\
&\quad \mathit{ret} \quad : \mathit{tm}\ (\bar{\forall}\ \mathit{ty}\ (\lambda\alpha.\ \alpha \Rightarrow_t 0\ \alpha))\\
&\quad \mathit{bind} : \mathit{tm}\ (\bar{\forall}\ \mathit{ty}\ (\lambda\alpha.\ \bar{\forall}\ \mathit{ty}\ (\lambda\beta.\ 0\ \alpha \Rightarrow_t (\alpha \Rightarrow_t 0\ \beta) \Rightarrow_t 0\ \beta)))
\end{aligned}
$$

- Higher-order functors

$$
\begin{aligned}
&\mathit{htyco} : \mathit{ki}\\
&\mathit{htyco} = \mathit{tyco} \Rightarrow_k \mathit{tyco}\\
&\mathit{trans} : (F, G : \mathit{el}\ \mathit{tyco}) \to \mathit{el}\ \mathit{ty}\\
&\mathit{trans}\ F\ G = \bar{\forall}\ \mathit{ty}\ (\lambda\alpha.\ F\ \alpha \Rightarrow_t G\ \alpha)\\
&\textsc{record}\ \mathit{RawHFunctor} : \mathbb{J}\ \textsc{where}\\
&\quad 0 \quad : \mathit{el}\ \mathit{htyco}\\
&\quad \mathit{hfmap} : (F : \mathit{RawFunctor}) \to \mathit{tm}\ (\mathit{fmap\text{-}ty}\ (0\ (F\ .0)))\\
&\quad \mathit{hmap} \ : (F, G : \mathit{RawFunctor}) \to \mathit{tm}\ (\mathit{trans}\ (F\ .0)\ (G\ .0))\\
&\qquad\quad \to \mathit{tm}\ (\mathit{trans}\ (0\ (F\ .0))\ (0\ (G\ .0)))
\end{aligned}
$$

- Computations

$$
\begin{aligned}
&\mathit{co} \quad : (H : \mathit{RawHFunctor}) \to (A : \mathit{el}\ \mathit{ty}) \to \mathbb{J}\\
&\mathit{val} \quad : \{H, A\} \to \mathit{tm}\ A \to \mathit{co}\ H\ A\\
&\mathit{let\text{-}in} : \{H, A, B\} \to \mathit{co}\ H\ A \to (\mathit{tm}\ A \to \mathit{co}\ H\ B) \to \mathit{co}\ H\ B
\end{aligned}
$$

- Laws of computations

$$
\begin{aligned}
&\mathit{val\text{-}let} \quad : \{H, A, B\} \to (a : \mathit{tm}\ A) \to (k : \mathit{tm}\ A \to \mathit{co}\ H\ B)\\
&\qquad\quad \to \mathit{let\text{-}in}\ (\mathit{val}\ a)\ k = k\ a\\
&\mathit{let\text{-}val} \quad : \{H, A\} \to (m : \mathit{co}\ H\ A) \to \mathit{let\text{-}in}\ m\ \mathit{val} = m\\
&\mathit{let\text{-}assoc} : \{H, A, B, C\} \to (m_1 : \mathit{co}\ H\ A)\\
&\qquad\quad \to (m_2 : \mathit{tm}\ A \to \mathit{co}\ H\ B) \to (m_3 : \mathit{tm}\ B \to \mathit{co}\ H\ C)\\
&\qquad\quad \to \mathit{let\text{-}in}\ (\mathit{let\text{-}in}\ m_1\ m_2)\ m_3 = \mathit{let\text{-}in}\ m_1\ (\lambda a.\ \mathit{let\text{-}in}\ (m_2\ a)\ m_3)
\end{aligned}
$$

- Thunks

$$
\begin{aligned}
&\mathit{th} : \mathit{RawHFunctor} \to \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty}\\
&\mathit{th\text{-}iso} : \{H, A\} \to \mathit{tm}\ (\mathit{th}\ H\ A) \cong \mathit{co}\ H\ A\\
&\Uparrow : \{H, A\} \to \mathit{tm}\ (\mathit{th}\ H\ A) \to \mathit{co}\ H\ A
\end{aligned}
$$

```
⇑ = th-iso .fwd
⇓ : {H, A} → co H A → tm (th H A)
⇓ = th-iso .bwd

th-mnd : RawHFunctor → RawMonad
th-mnd H .0    = th H
th-mnd H .ret  = λA x. ⇓ (val x)
th-mnd H .bind = λA B m k. ⇓ (let-in (force m) (λa. ⇑ (k a)))
```

- Operations

```
op : {H, A, B} → tm (H (th H) A) → (tm A → co H B) → co H B

let-op : {H, A, B, C} → (p : tm (H (th H) A))
       → (k : tm A → co H B) → (k′ : tm B → co H C)
       → let-in (op p k) k′ = op p (λa. let-in (k a) k′)
```

- Monads with algebras

```
RECORD MonadAlg (H : RawHFunctor) : 𝕁 WHERE
  INCLUDE RawMonad AS M
  malg : tm (trans (H .0 0) 0)

th-alg : (H : RawHFunctor) → MonadAlg H
th-alg H .M    = th-mnd H
th-alg H .malg = λα o. ⇓ (op o val)
```

- Evaluation of computations

```
eval : {H} → (m : MonadAlg H) → (A : el ty) → co H A → tm (m .0 A)

eval-val : {H, A} → (m : MonadAlg H) → (a : tm A)
         → eval m A (val a) = m .ret A a

eval-op : {H, A, B} → (m : MonadAlg H)
        → (p : tm (H (th H) A)) → (k : tm A → co H B)
        → LET bind = m .bind A B
              malg = m .malg A
              T    = fct-of-mnd (th-mnd H)
              M    = fct-of-mnd (m .M)
          IN eval m B (op p k)
           = bind (malg (H .hmap T M (λα c. eval m α (⇑ c)) A p))
                  (λa. eval m B (k a))

fct-of-mnd : RawMonad → RawFunctor
fct-of-mnd m .0 = m .0
fct-of-mnd m .fmap α β f ma = m .bind α β ma (λa. m .ret _ (f a))
```

A.2 Effect Families in F^{ha}_{ω}

We did not include in System F^{ha}_{ω} any judgements for *modular handlers* [Yang and Wu 2021, 2023] or *effect systems* [Bauer and Pretnar 2014; Kammar and Plotkin 2012; Lucassen and Gifford 1988] that track the effect operations that a computation may perform, because both of them can be *derived concepts* in F^{ha}_{ω}.

Firstly, the judgement for *effect families* [Yang and Wu 2023] is the following record in $\mathrm{F}^{\mathrm{ha}}_{\omega}$:

$$\begin{aligned}&\textsc{record}\ \mathit{Fam} : \mathbb{J}\ \textsc{where}\\ &\quad \mathit{eff}\ \ : \mathit{ki}\\ &\quad \mathit{sig}\ \ : \mathit{el}\ \mathit{eff} \to \mathit{RawHFunctor}\\ &\quad \mathit{add} : \mathit{el}\ \mathit{eff} \to \mathit{el}\ \mathit{eff} \to \mathit{el}\ \mathit{eff}\end{aligned}$$

The elements of the kind $\mathit{eff} : \mathit{ki}$ are effect signatures in this family, each of them determining a higher-order functor via *sig*. Additionally, there is a way *add* to combine two effects in a family.

Then we have the following definitions for monads and computations for an effect e in a family F, which can be viewed as a generic effect system parameterised by an effect family F:

$$\begin{aligned}&\mathit{MonadEff} : (F : \mathit{Fam}) \to (e : \mathit{el}\ (F\ .\mathit{eff})) \to \mathbb{J}\\ &\mathit{MonadEff}\ F\ e = \mathit{MonadAlg}\ (F\ .\mathit{sig}\ e)\\ &\mathit{co}[\_\ni\_] : (F : \mathit{Fam}) \to (e : \mathit{el}\ (F\ .\mathit{eff})) \to \mathit{el}\ \mathit{ty} \to \mathbb{J}\\ &\mathit{co}[F \ni e] = \mathit{co}\ (F\ .\mathit{sig}\ e)\end{aligned}$$

A *modular handler* processing the effect e in a family F and outputting the effect o is the structure

$$\begin{aligned}&\textsc{record}\ \mathit{Hdl}\ (F : \mathit{Fam})\ (e\ o : \mathit{el}\ (F\ .\mathit{eff})) : \mathbb{J}\ \textsc{where}\\ &\quad \mathit{alg} : (\mu : \mathit{el}\ (F\ .\mathit{eff})) \to \mathit{MonadEff}\ F\ (F\ .\mathit{add}\ o\ \mu)\\ &\qquad \to \mathit{MonadEff}\ F\ (F\ .\mathit{add}\ e\ \mu)\\ &\quad \mathit{res}\ \ : \mathit{el}\ (\mathit{ty} \Rightarrow_k \mathit{ty})\\ &\quad \mathit{run} : (\mu : \mathit{el}\ (F\ .\mathit{eff})) \to (\mathit{Mo} : \mathit{MonadEff}\ F\ (F\ .\mathit{add}\ o\ \mu))\\ &\qquad \to \mathit{tm}\ (\mathit{trans}\ (\mathit{alg}\ \mu\ \mathit{Mo})\ (\lambda A.\ \mathit{Mo}\ (\mathit{res}\ A)))\end{aligned}$$

Modular handlers as such can be applied to computations $\mathit{co}[F \ni (F\ .\mathit{add}\ e\ \mu)]\ A$, for all 'ambient' effects μ, removing the effect e and generating the effect o, yielding computations $\mathit{co}[F \ni (F\ .\mathit{add}\ o\ \mu)]\ (h\ .\mathit{res}\ A)$:

$$\begin{aligned}&\mathit{handle} : \{F, e, o, \mu, A\} \to (h : \mathit{Hdl}\ F\ e\ o) \to \mathit{co}[F \ni (F\ .\mathit{add}\ e\ \mu)]\ A\\ &\qquad \to \mathit{co}[F \ni (F\ .\mathit{add}\ o\ \mu)]\ (h\ .\mathit{res}\ A)\\ &\mathit{handle}\ h\ c = \Uparrow (h\ .\mathit{run}\ \mu\ T\ A\ c')\ \textsc{where}\\ &\quad T : \mathit{MonadEff}\ F\ (F\ .\mathit{add}\ o\ \mu)\\ &\quad T = \mathit{th\text{-}alg}\ (F\ .\mathit{sig}\ (F\ .\mathit{add}\ o\ \mu))\\ &\quad c' : \mathit{tm}\ (h\ .\mathit{alg}\ \mu\ T\ A)\\ &\quad c' = \mathit{eval}\ (h\ .\mathit{alg}\ \mu\ (\mathit{th\text{-}alg}\ (F\ .\mathit{sig}\ (F\ .\mathit{add}\ o\ \mu))))\ \_\ c\end{aligned}$$

The complete definition of the effect family *algFam*, together with the needed standard type connectives, is collected below.

- Kind-level and type-level products

$$\begin{aligned}&\_ \times_k \_ : \mathit{ki} \to \mathit{ki} \to \mathit{ki}\\ &\times_k\text{-}\mathit{iso} : \{k\ k' : \mathit{ki}\} \to \mathit{el}\ (k \times_k k') = \Sigma\ (\mathit{el}\ k)\ (\lambda\_.\ \mathit{el}\ k')\\ &\_ \times_t \_ : \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty} \to \mathit{el}\ \mathit{ty}\\ &\times_t\text{-}\mathit{iso} : \{A\ B : \mathit{el}\ \mathit{ty}\} \to \mathit{tm}\ (A \times_t B) = \Sigma\ (\mathit{tm}\ A)\ (\lambda\_.\ \mathit{tm}\ B)\end{aligned}$$

- The empty type

$$\mathit{empty} : \mathit{el}\ \mathit{ty}$$

$absurd : (A : el\ ty) \to tm\ empty \to tm\ A$
$absurd\text{-}uniq : \{A : el\ ty\} \to (f : tm\ empty \to tm\ A) \to f = absurd\ A$

- Coproducts
  We only need type-level coproducts, but for generality we define coproducts parameterised by judgements $U : \mathbb{J}$ and $T : U \to \mathbb{J}$:

  RECORD $coprod\_intro\ (U : \mathbb{J})\ (T : U \to \mathbb{J}) : \mathbb{J}$ WHERE
  $\_+\_ : U \to U \to U$
  $inl\ : \{a, b\} \to T\ a \to T\ (a + b)$
  $inr\ : \{a, b\} \to T\ b \to T\ (a + b)$

  RECORD $coprod\_elim\ (U : \mathbb{J})\ (T : U \to \mathbb{J})\ (V : \mathbb{J})\ (S : U \to \mathbb{J})$
  $(intr : coprod\_intro\ U\ T) : \mathbb{J}$ WHERE
  OPEN $coprod\_intro\ intr$

  $case : \{a, b, c\} \to (T\ a \to S\ c) \to (T\ b \to S\ c) \to (T\ (a + b) \to S\ c)$

  $case\_\beta\_l : \{a, b, c\} \to (l : T\ a \to S\ c) \to (r : T\ b \to S\ c) \to (x : T\ a)$
  $\to case\ l\ r\ (inl\ x) = l\ x$

  $case\_\beta\_r : \{a, b, c\} \to (l : T\ a \to S\ c) \to (r : T\ b \to S\ c) \to (x : T\ b)$
  $\to case\ l\ r\ (inr\ x) = r\ x$

  $case\_\eta : \{a, b, c\} \to (f : T\ (a + b) \to S\ c)$
  $\to case\ (\lambda x.\ f\ (inl\ x))\ (\lambda x.\ f\ (inr\ x)) = f$

  RECORD $coprod\ (U : \mathbb{J})\ (T : U \to \mathbb{J}) : \mathbb{J}$ WHERE
  $cpintr : coprod\_intro\ U\ T$
  $cpelim : coprod\_elim\ U\ T\ U\ T\ cpintr$

  We then instantiate with $U = el\ ty$ and $T = tm$ to get type-level coproducts:

  $coprodTy : coprod\ (el\ ty)\ tm$

  In this way, if kind-level coproducts are also needed, they can be easily added by a declaration $coprodKi : coprod\ ki\ el$.
- Kind-level lists with elimination to ML-style signatures
  We first define the judgements of lists parameterised by the universe (U, T) that the lists live in and the universe (V, S) that the lists can eliminate into:

  RECORD $ListAlg\ \{U : \mathbb{J}\}\ \{V : \mathbb{J}\}\ (T : U \to \mathbb{J})\ (S : V \to \mathbb{J})$
  $(k : U)\ (a : V) : \mathbb{J}$
  WHERE
  $fst : S\ a$
  $snd : T\ k \to S\ a \to S\ a$

  RECORD $ListHom\ \{U : \mathbb{J}\}\ \{V : \mathbb{J}\}\ \{W : \mathbb{J}\}$
  $\{T : U \to \mathbb{J}\}\ \{S : V \to \mathbb{J}\}\ \{R : W \to \mathbb{J}\}$
  $\{k : U\}\ \{a : V\}\ \{b : W\}$
  $(alga : ListAlg\ T\ S\ k\ a)\ (algb : ListAlg\ T\ R\ k\ b) : \mathbb{J}$
  WHERE
  $f : S\ a \to R\ b$
  $homnil : f\ (alga\ .fst) = algb\ .fst$

```
  homcons : (x : T k) → (a : S a) → f (alga .snd x a) = algb .snd x (f a)
record ListIntro (U : 𝕁) (T : U → 𝕁) : 𝕁 where
  listc : U → U
  listcalg : {k : U} → ListAlg T T k (listc k)
  nil : (k : U) → T (listc k)
  nil k = listcalg .fst
  cons : {k : U} → T k → T (listc k) → T (listc k)
  cons x xs = listcalg .snd x xs
record ListElim (U : 𝕁) (T : U → 𝕁) (V : 𝕁) (S : V → 𝕁)
  (intr : ListIntro U T) : 𝕁
  where
  open ListIntro intr
  fold : {k : U} → {a : V} → (alga : ListAlg T S k a)
       → T (listc k) → S a
  foldβnil : {k, a} → (alga : ListAlg T S k a)
           → fold alga (nil k) = alga .fst
  foldβcons : {k, a} → (alga : ListAlg T S k a)
            → (x : T k) (xs : T (listc k))
            → fold alga (cons x xs) = alga .snd x (fold alga xs)
  foldη : {k, a} → (alga : ListAlg T S k a)
        → (h : ListHom listcalg alga)
        → fold alga = h .f
record List (U : 𝕁) (T : U → 𝕁) (V : 𝕁) (S : V → 𝕁) : 𝕁 where
  intr : ListIntro U T
  elim : ListElim U T V S intr
```

We have kind-level lists with declarations

```
ListKi : List ki el ki el
```

We additionally have elimination of kind-level lists to ML-style signatures

```
si : 𝕁
si = Σ ki (λk. (el k → el ty))
mo : si → 𝕁
mo (k, t) = Σ (el k) (λα. tm (t α))
ListKiTyElim : ListElim ki el si mo (ListKi .intr)
```

In the following we will rename the components of lists as follows:

```
open List ListKi renaming
  (listc      ↦ list_k; nil          ↦ nil_k;   cons ↦ cons_k;
   fold       ↦ fold_k; foldβnil     ↦ foldkβni;
   foldβcons  ↦ fold_kβcons; foldη   ↦ fold_kη)
open ListElim ListKiTyElim renaming
```

$$(fold \mapsto fold_{kt}; \quad fold\beta nil \mapsto fold_{kt}\beta nil;\\ fold\beta cons \mapsto fold_{kt}\beta cons; fold\eta \mapsto fold_{kt}\eta)$$

- The constantly empty higher-order functor

```
VoidH : RawHFunctor
VoidH = RECORD {0 = voidH0
               ; hfmap = hfmapVoid
               ; hmap = hmapVoid}
  WHERE
  voidH0 : el htyco
  voidH0 _ _ = empty
  hfmapVoid : (F : RawFunctor) → tm (fmap-ty (voidH₀ (0 F)))
  hfmapVoid F α β f x = x
  hmapVoid : (F G : RawFunctor) → tm (trans (0 F) (0 G))
     → tm (nat − ty (voidH₀ (0 F)) (voidH₀ (0 G)))
  hmapVoid F G _ α x = x
```

- Coproduct of higher-order functors

```
coprodHF : RawHFunctor → RawHFunctor → RawHFunctor
coprodHF H₁ H₂ .0 = λF A. (H₁ .0 F A) + (H₂ .0 F A)
coprodHF H₁ H₂ .hfmap = λF α β f x.
  case {c = H₁ .0 (0 F) β + H₂ .0 (0 F) β}
       (λl. inl (H₁ .hfmap F α β f l))
       (λr. inr (H₂ .hfmap F α β f r))
       x
coprodHF H₁ H₂ .hmap = λF G s α x.
  case {c = H₁ .0 (0 G) α + H₂ .0 (0 G) α}
       (λl. inl (H₁ .hmap F G s α l))
       (λr. inr (H₂ .hmap F G s α r))
       x
```

- Higher-order functor for an algebraic operation

```
AlgOpHFun : el ty → el ty → RawHFunctor
AlgOpHFun P A =
  RECORD {0 = λ_ X. P ×ₜ (A ⇒ₜ X)
         ; hfmap = λF α β f (p, k). (p, (λx. f (k x)))
         ; hmap = λF G s α pk. pk}
```

- The ML-style signature corresponding of functors

```
FctSig : si
FctSig = tyco, fmap-ty
FctToMod : RawFunctor → mo FctSig
FctToMod F = (0 F), (fmap F)
```

$$
\begin{array}{l}
\mathit{FctFromMod} : \mathit{mo}\ \mathit{FctSig} \to \mathit{RawFunctor} \\
\mathit{FctFromMod}\ (F, \mathit{fmap}) = \textsc{record}\ \{0 = F; \mathit{fmap} = \mathit{fmap}\}
\end{array}
$$

- The ML-style signature corresponding of higher-order functors

$$
\begin{array}{l}
\mathit{HFctSig} : \mathit{si} \\
\mathit{HFctSig} = \mathit{htyco}, (\lambda H.\ \mathit{hfmapTy}\ H\ \times_t\ \mathit{hmapTy}\ H)\ \textsc{where} \\
\quad \mathit{hfmapTy} : (H : \mathit{el}\ \mathit{htyco}) \to \mathit{el}\ \mathit{ty} \\
\quad \mathit{hfmapTy}\ H = \bar{\forall}\ \mathit{tyco}\ (\lambda F.\ \mathit{fmap\text{-}ty}\ F \Rightarrow_t \mathit{fmap\text{-}ty}\ (H\ F)) \\
\quad \mathit{hmapTy} : (H : \mathit{el}\ \mathit{htyco}) \to \mathit{el}\ \mathit{ty} \\
\quad \mathit{hmapTy}\ H = \bar{\forall}\ \mathit{tyco}\ (\lambda F.\ \mathit{fmap\text{-}ty}\ F \Rightarrow_t \\
\qquad \bar{\forall}\ \mathit{tyco}\ (\lambda G.\ \mathit{fmap\text{-}ty}\ G \Rightarrow_t \\
\qquad\qquad (\mathit{trans}\ F\ G \Rightarrow_t \mathit{nat\_ty}\ (H\ F)\ (H\ G)))) \\
\mathit{HFctToMod} : \mathit{RawHFunctor} \to \mathit{mo}\ \mathit{HFctSig} \\
\mathit{HFctToMod}\ H = (H\ .0), \\
\quad ((\lambda F\ \mathit{fmap}.\ H\ .\mathit{hfmap}\ (\mathit{FctFromMod}\ (F, \mathit{fmap}))) \\
\quad , \lambda F\ \mathit{fmap}_1\ G\ \mathit{fmap}_2. \\
\qquad H\ .\mathit{hmap}\ (\mathit{FctFromMod}\ (F, \mathit{fmap}_1)) \\
\qquad\quad (\mathit{FctFromMod}\ (G, \mathit{fmap}_2))) \\
\mathit{HFctFromMod} : \mathit{mo}\ \mathit{HFctSig} \to \mathit{RawHFunctor} \\
\mathit{HFctFromMod}\ (H, (\mathit{hfmap}, \mathit{hmap})) = \\
\quad \textsc{record}\ \{0 = H \\
\qquad\quad ; \mathit{hfmap} = \lambda F.\ \mathit{hfmap}\ (F\ .0)\ (F\ .\mathit{fmap}) \\
\qquad\quad ; \mathit{hmap} = \lambda F\ G.\ \mathit{hmap}\ (F\ .0)\ (F\ .\mathit{fmap})\ (G\ .0)\ (G\ .\mathit{fmap})\}
\end{array}
$$

- The family of algebraic operations

$$
\begin{array}{l}
\mathit{AlgSig} : \mathit{el}\ (\mathit{list}_k\ (\mathit{ty} \Rightarrow_k \mathit{ty})) \to \mathit{RawHFunctor} \\
\mathit{AlgSig}\ \mathit{es} = \mathit{HFctFromMod} \\
\quad (\mathit{fold}_{kt}\ \{a = \mathit{HFctSig}\} \\
\qquad (\textsc{record}\ \{\mathit{fst} = \mathit{HFctToMod}\ \mathit{VoidH} \\
\qquad\qquad ; \mathit{snd} = \lambda (P, A)\ H. \\
\qquad\qquad\quad \mathit{HFctToMod}\ (\mathit{coprodHF}\ (\mathit{AlgOpHFun}\ P\ A) \\
\qquad\qquad\qquad\qquad (\mathit{HFctFromMod}\ H))\}) \\
\qquad \mathit{es}) \\
\mathit{ListApp}_k : \{k : \mathit{ki}\} \to \mathit{el}\ (\mathit{list}_k\ k) \to \mathit{el}\ (\mathit{list}_k\ k) \to \mathit{el}\ (\mathit{list}_k\ k) \\
\mathit{ListApp}_k\ \{k\}\ x\ y = \\
\quad \mathit{fold}_k\ \{a = \mathit{list}_k\ k\} \\
\qquad (\textsc{record}\ \{\mathit{fst} = y; \mathit{snd} = \mathit{cons}_k\})\ x \\
\mathit{algFam} : \mathit{Fam} \\
\mathit{algFam} = \textsc{record}\ \{\mathit{eff} = \mathit{list}_k\ (\mathit{ty}\ \times_k\ \mathit{ty}) \\
\qquad\qquad\qquad\quad ; \mathit{sig} = \mathit{AlgSig} \\
\qquad\qquad\qquad\quad ; \mathit{add} = \mathit{ListApp}_k\}
\end{array}
$$

B Equations of Computations for the Realizability Model

In Section 3.2 we defined a model of $\mathrm{F}_\omega^{\mathrm{ha}}$ in the language of assemblies (Language 3.1). This appendix shows that the equational laws of $\mathrm{F}_\omega^{\mathrm{ha}}$ are validated by the definitions in Section 3.2.

The monadic laws of computations ($\mathrm{F}_\omega^{\mathrm{ha}}$-3) are satisfied:

- For *val-let*, given any $a : R.tm\ A$ and $k : R.tm\ A \to R.co\ H\ B$,

$$\begin{aligned}
& R.let\text{-}in\ (R.val\ a)\ k \\
= \quad & \{\text{by definition of } R.let\text{-}in\} \\
& \lambda T\ C\ r.\ R.val\ a\ T\ C\ (\lambda a.\ k\ a\ T\ C\ r) \\
= \quad & \{\text{by definition of } R.val\} \\
& \lambda T\ C\ r.\ k\ a\ T\ C\ r \\
= \quad & \{\eta\text{-rule for functions}\} \\
& k\ a
\end{aligned}$$

- The case for *let-val* is very similar. Given any $c : R.co\ H\ A$,

$$\begin{aligned}
& R.let\text{-}in\ c\ R.val \\
= \quad & \{\text{by definition of } R.let\text{-}in\} \\
& \lambda T\ C\ r.\ c\ T\ C\ (\lambda a.\ R.val\ a\ T\ C\ r) \\
= \quad & \{\text{by definition of } R.val\} \\
& \lambda T\ C\ r.\ c\ T\ C\ (\lambda a.\ r\ a) \\
= \quad & \{\eta\text{-rule for functions}\} \\
& c
\end{aligned}$$

- For *let-assoc*, given any c_1, c_2, and c_3, we have

$$\begin{aligned}
& R.let\text{-}in\ (R.let\text{-}in\ c_1\ c_2)\ c_3 \\
= {} & \lambda T\ C\ r.\ (R.let\text{-}in\ c_1\ c_2)\ T\ C\ (\lambda b.\ c_3\ b\ T\ C\ r) \\
= {} & \lambda T\ C\ r.\ c_1\ T\ C\ (\lambda a.\ c_2\ a\ T\ C\ (\lambda b.\ c_3\ b\ T\ C\ r)) \\
= {} & \lambda T\ C\ r.\ c_1\ T\ C\ (\lambda a.\ R.let\text{-}in\ (c_2\ a)\ c_3) \\
= {} & R.let\text{-}in\ c_1\ (\lambda a.\ R.let\text{-}in\ (c_2\ a)\ c_3)
\end{aligned}$$

Now we check that the equation *eval-val* ($\mathrm{F}_\omega^{\mathrm{ha}}$-8) is satisfied: for all $H : R.RawHFunctor$, $A : R.el\ R.ty$, $T : R.MonadAlg\ H$ and $a : A$,

$$\begin{aligned}
& R.eval\ T\ A\ (R.val\ a) \\
= \quad & \{\text{by definition of } R.eval\} \\
& R.val\ a\ T\ A\ T.ret \\
= \quad & \{\text{by definition of } R.val\} \\
& (\lambda T\ B\ r.\ r\ a)\ T\ A\ T.ret \\
= {} & T.ret\ a
\end{aligned}$$

The model of operations is defined as follows:

$$
\begin{aligned}
&R.op : \{H, A, B\} \to H\ (th\ H)\ A \to (A \to R.co\ H\ B) \to R.co\ H\ B \\
&R.op\ o\ k = \lambda T\ C\ r. \\
&\quad T.bind\ A\ C \\
&\qquad (T.malg\ A\ (H.hmap\ (th\ H)\ T\ (R.eval\ T)\ A\ o)) \\
&\qquad (\lambda a.\ k\ a\ T\ C\ r)
\end{aligned}
$$

It remains to check that the equations *let-op* ($\mathrm{F}^{\mathrm{ha}}_{\omega}$-6) and *eval-op* ($\mathrm{F}^{\mathrm{ha}}_{\omega}$-9) are satisfied.

For *let-op*, given arbitrary $o : H\ (th\ H)\ A$, $k : A \to co\ H\ B$, $k' : B \to co\ H\ C$,

$$
\begin{aligned}
&\quad let\text{-}in\ (op\ o\ k)\ k' \\
&= \quad \{\text{by definition of } R.let\text{-}in\} \\
&\quad \lambda T\ C\ r.\ (op\ o\ k)\ T\ C\ (\lambda b.\ k'\ b\ T\ C\ r) \\
&= \quad \left\{ \begin{array}{l} \text{by definition of } R.op \text{ and let } o' \text{ be} \\ \quad T.malg\ \_\ (H.hmap\ \_\ \_\ (R.eval\ T)\ \_\ o) \end{array} \right\} \\
&\quad T.bind\ \_\ \_\ o'\ (\lambda a.\ k\ a\ T\ \_\ (\lambda b.\ k'\ b\ T\ \_\ r)) \\
&= \quad \{\text{by definition of } R.let\text{-}in\ (k\ a)\ k'\} \\
&\quad op\ o\ (\lambda a.\ let\text{-}in\ (k\ a)\ k')
\end{aligned}
\tag{9}
$$

For *eval-op*, given any $T : MonadAlg\ H$, $o : H\ (th\ H)\ A$ and $k : A \to co\ H\ B$,

$$
\begin{aligned}
&\quad eval\ T\ (op\ o\ k) \\
&= \quad \{\text{by definition of } R.eval\} \\
&\quad (op\ o\ k)\ T\ \_\ T.ret \\
&= \quad \{\text{by definition of } R.op \text{ and let } o' \text{ be the same as in (9)}\} \\
&\quad T.bind\ \_\ \_\ o'\ (\lambda a.\ k\ a\ T\ \_\ T.ret) \\
&= \quad \{\text{by definition of } R.eval\ T\ \_\ k\ a\} \\
&\quad T.bind\ \_\ \_\ o'\ (\lambda a.\ eval\ T\ \_\ (k\ a))
\end{aligned}
$$

C The Synthetic Logical Relation Model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$

In this appendix, we define the logical relation model of $\mathrm{F}^{\mathrm{ha}}_{\omega}$ in detail. Let us start two useful lemmas that we did not include in the main text.

The first of them says that we can not only glue but also tear types apart. Given any type $A : U$ in StcTT, we can tear it to an object-space fragment A° and a meta-space fragment $A^\bullet$:

$$
\begin{aligned}
&A^\circ : \bigcirc U && A^\bullet : (\{\mathfrak{ob}\} \to A) \to U^\bullet \\
&A^\circ = \eta^\circ_U\ A && A^\bullet = \lambda o.\ \{A \mid \mathfrak{ob} \hookrightarrow o\}
\end{aligned}
$$

where $U^\bullet := \{A : U \mid \bullet\text{-}modal\ A\}$ is the subuniverse of $\bullet$-modal types. The type $\{A \mid \mathfrak{ob} \hookrightarrow a\}$ is $\bullet$-modal because it is a singleton under $\mathfrak{ob}$ (Lemma 4.8).

Lemma C.1. *For every type $A : U$ in StcTT, there is an isomorphism $A \cong (o : A^\circ) \ltimes A^\bullet\ o$.*

Proof. The two directions of the isomorphism are

$$
\begin{aligned}
&fwd : A \to (o : A^\circ) \ltimes A^\bullet\ o && bwd : ((o : A^\circ) \ltimes A^\bullet\ o) \to A \\
&fwd\ a = [\mathfrak{ob} \hookrightarrow \lambda\{\_ : \mathfrak{ob}\}.\ a \mid a] && bwd\ [\mathfrak{ob} \hookrightarrow o \mid c] = c
\end{aligned}
$$

These two functions are indeed mutual inverses: for all $a : A$,

$$bwd\ (fwd\ a) = bwd\ [\mathfrak{ob} \hookrightarrow \lambda\{\_ : \mathfrak{ob}\}.\ a \mid a] = a;$$

for all $[\mathfrak{ob} \hookrightarrow o \mid c]$, by definition $fwd\ (bwd\ [\mathfrak{ob} \hookrightarrow o \mid c]) = [\mathfrak{ob} \hookrightarrow c \mid c]$, but c has type $A^\bullet := \{c \mid \mathfrak{ob} \hookrightarrow o\}$, so $[\mathfrak{ob} \hookrightarrow c \mid c] = [\mathfrak{ob} \hookrightarrow o \mid c]$. □

Since every type of StcTT is isomorphic to a glue type, we can characterise function types of StcTT more extrinsically, which explicitises the idea that a map between logical predicates sends (proofs for) related input to (proofs for) related output.

Lemma C.2. *For all universes U of StcTT, there is an isomorphism $\ltimes$-fun-iso:*

$$\big((a : A) \ltimes P\ a\big) \to \big((b : B) \ltimes Q\ b\big) \ \cong\ (f : A \to B) \ltimes \big((a : \{\mathfrak{ob}\} \to A) \to P\ a \to Q\ (f\ a)\big)$$

for all $A, B : \bigcirc U$, $P : (\{\mathfrak{ob}\} \to A) \to U^\bullet$, and $Q : (\{\mathfrak{ob}\} \to B) \to U^\bullet$, where $U^\bullet$ is the subuniverse $\{A : U \mid \bullet\text{-modal } A\}$ of $\bullet$-modal types.

Proof. The two directions of the isomorphism are

$$\begin{aligned} &fwd\ g = [\mathfrak{ob} \hookrightarrow \lambda a.\ g\ a \mid \lambda a\ p.\ unglue\ (g\ [\mathfrak{ob} \hookrightarrow a \mid p])] \\ &bwd\ [\mathfrak{ob} \hookrightarrow f \mid h]\ [\mathfrak{ob} \hookrightarrow a \mid p] = [\mathfrak{ob} \hookrightarrow f\ a \mid h\ a\ p] \end{aligned}$$

It is routine calculation to check that these two directions are mutual inverses.

$$\begin{aligned} &bwd\ (fwd\ g) \\ &= bwd\ ([\mathfrak{ob} \hookrightarrow \lambda a.\ g\ a \mid \lambda a\ p.\ unglue\ (g\ [\mathfrak{ob} \hookrightarrow a \mid p])]) \\ &= \lambda[\mathfrak{ob} \hookrightarrow a \mid p].\ [\mathfrak{ob} \hookrightarrow g\ a \mid unglue\ (g\ [\mathfrak{ob} \hookrightarrow a \mid p])] \\ &= \lambda a^*.\ g\ a^* \\ &= g \end{aligned}$$

$$\begin{aligned} &fwd\ (bwd\ [\mathfrak{ob} \hookrightarrow f \mid h]) \\ &= fwd\ (\lambda[\mathfrak{ob} \hookrightarrow a \mid p].\ [\mathfrak{ob} \hookrightarrow f\ a \mid h\ a\ p]) \\ &= [\mathfrak{ob} \hookrightarrow \lambda a.\ f\ a \mid \lambda a\ p.\ unglue\ [\mathfrak{ob} \hookrightarrow f\ a \mid h\ a\ p]] \\ &= [\mathfrak{ob} \hookrightarrow f \mid h] \end{aligned}$$

□

Now we come back to define the logical relation model:

$$M^* : \{[\![\mathrm{F}^{\mathrm{ha}}_\omega]\!]_{U_2} \mid \mathfrak{ob} \hookrightarrow M\} \tag{10}$$

Notation C.3. In the rest of this section, for every declaration *dec* in the signature of $\mathrm{F}^{\mathrm{ha}}_\omega$, we will write dec^* for $M^*.dec$ and just *dec* for $M.dec$. For example, $ki^* : \{U_2 \mid \mathfrak{ob} \hookrightarrow ki\}$ means $M^*.ki : \{U_2 \mid \mathfrak{ob} \hookrightarrow ki\}$.

The logical predicate model of the judgement of kinds (F_ω-1) is

$$\begin{aligned} &ki^* : \{U_2 \mid \mathfrak{ob} \hookrightarrow ki\} \\ &ki^* = (\alpha : ki) \ltimes \{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \alpha\} \end{aligned} \tag{11}$$

This uses the glue type (4.9) correctly because the generic model M (4.6) has type $\{\mathfrak{ob}\} \to [\![\mathrm{F}^{\mathrm{ha}}_\omega]\!]_{U_0}$, so the type of ki, or more explicitly $\lambda\{z : \mathfrak{ob}\}.\ (M\ \{z\}).ki$, is $\{\mathfrak{ob}\} \to U_0$, i.e. $\bigcirc U_0$. The type $\{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \alpha\}$ is $\bullet$-modal because when $\mathfrak{ob}$ holds, all elements of $\{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \alpha\}$ are equal to $el\ \alpha$, so the type $\{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \alpha\}$ has exactly one element so isomorphic to the unit 1. By Lemma 4.8, the type $\{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \alpha\}$ is $\bullet$-modal.

More intuitively, the definition (11) is the *proof-relevant* logical predicate for kinds. A proof for a kind $\alpha : ki$ satisfying the predicate is a type $A : U_1$ that restrict to $el\ \alpha$ in the object space. Such a type A is a 'candidate' for the logical predicate for the kind α. This is the same idea as *reducibility candidates* in Girard's proof of strong normalisation of System F [Girard 1989].

In accordance, the corresponding $M^*.el$ is as follows:

$$\begin{aligned} &el^* : \{ki^* \to U_1 \mid \mathfrak{ob} \hookrightarrow el\} \\ &el^*\ g = unglue\ g \end{aligned} \tag{12}$$

Let us more carefully examine how this definition type checks: the argument g has type $ki^* = (\alpha : ki) \ltimes \{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \alpha\}$. Therefore $unglue\ g$ has type $\{U_1 \mid \mathfrak{ob} \hookrightarrow el\ g\}$ (note that under $\mathfrak{ob}$, the type of g is strictly equal to the type ki, thus it makes sense to write $el\ g$ in a context where $\mathfrak{ob}$ holds). Thus el^* is indeed a function $ki^* \to U_1$ that strictly restricts to el under $\mathfrak{ob}$.

For kind-level functions, we need to define

$$\_\Rightarrow_k\_^* : \{ki^* \to ki^* \to ki^* \mid \mathfrak{ob} \hookrightarrow \_\Rightarrow_k\_\}$$

Let us derive the definition step-by-step. Our goal is to fill the hole $?0$ in

$$[\mathfrak{ob} \hookrightarrow \alpha \mid A] \Rightarrow_k^* [\mathfrak{ob} \hookrightarrow \beta \mid B] = ?0 : \{ki^* \mid \mathfrak{ob} \hookrightarrow \alpha \Rightarrow_k \beta\},$$

where the variables in context have the following types

$$\alpha, \beta : \{\mathfrak{ob}\} \to ki \qquad A : \{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \alpha\} \qquad B : \{U_1 \mid \mathfrak{ob} \hookrightarrow el\ \beta\}. \tag{13}$$

Since ki^* is a glue type (11), we can use the term former of glue types:

$$[\mathfrak{ob} \hookrightarrow \alpha \mid A] \Rightarrow_k^* [\mathfrak{ob} \hookrightarrow \beta \mid B] = [\mathfrak{ob} \hookrightarrow ?1 \mid ?2]$$

Since $?0$ must restrict to $\alpha \Rightarrow_k \beta$ under $\mathfrak{ob}$, $?1$ has to be $\alpha \Rightarrow_k \beta$:

$$[\mathfrak{ob} \hookrightarrow \alpha \mid A] \Rightarrow_k^* [\mathfrak{ob} \hookrightarrow \beta \mid B] = [\mathfrak{ob} \hookrightarrow \alpha \Rightarrow_k \beta \mid ?2]$$

The hole $?2$ now has type $\{U_1 \mid \mathfrak{ob} \hookrightarrow el\ (\alpha \Rightarrow_k \beta)\}$; in other words, $?2$ is a type in U_1 such that it restricts to $el\ (\alpha \Rightarrow_k \beta)$ when $\mathfrak{ob}$ holds. We again use the glue type to satisfy the restriction:

$$?2 \coloneqq (f : el\ (\alpha \Rightarrow_k \beta)) \ltimes ?3.$$

Conceptually, $?2$ is the logical predicate for the function kind $\alpha \Rightarrow_k \beta$. Readers experienced with traditional logical relations might expect $?3$ to be the proposition asserting that f sends input $a : el\ \alpha$ satisfying the logical predicate A to output $f\ a : el\ \beta$ satisfying logical predicate B. However, here the predicates A and B are proof-relevant, so the correct definition of $?3$ should be the *type* of functions sending proofs for $a : el\ \alpha$ satisfying A to proofs for $f\ a : el\ \beta$ satisfying B. This can be concisely expressed in StcTT as

$$?3 \coloneqq \{A \to B \mid \mathfrak{ob} \hookrightarrow \Rightarrow_k\text{-}iso.fwd\ f\}$$

where $\Rightarrow_k\text{-}iso$ is the isomorphism in $\mathrm{F}_\omega^{\mathrm{ha}}$ specifying function kinds:

$$\Rightarrow_k\text{-}iso : el\ (\alpha \Rightarrow_k \beta) \cong (el\ \alpha \to el\ \beta).$$

The function type $A \to B$ in StcTT is translated to exponentials in the glued topos $\mathscr{G}$, which takes care of 'sending related input to related output' by construction.

For the record, we have completed our initial goal $\_\Rightarrow_k\_^*$:

$$\begin{aligned} &\_\Rightarrow_k\_^* : \{ki^* \to ki^* \to ki^* \mid \mathfrak{ob} \hookrightarrow \_\Rightarrow_k\_\} \\ &[\mathfrak{ob} \hookrightarrow \alpha \mid A] \Rightarrow_k^* [\mathfrak{ob} \hookrightarrow \beta \mid B] = [\mathfrak{ob} \hookrightarrow \alpha \Rightarrow_k \beta \mid F] \end{aligned} \tag{14}$$

where F is the logical predicate for the function kind $\alpha \Rightarrow_k \beta$:

$$F := (f : el\ (\alpha \Rightarrow_k \beta)) \ltimes \{A \to B \mid \mathfrak{ob} \hookrightarrow \Rightarrow_k\text{-}iso.fwd\ f\}. \tag{15}$$

We also need to exhibit the isomorphism $\Rightarrow_k\text{-}iso$ ($\mathrm{F}_\omega - 3$) for M^*:

$$\Rightarrow_k\text{-}iso^* : \{\alpha^*, \beta^* : ki^*\} \to \{el^*\ (\alpha^* \Rightarrow^*_k \beta^*) \cong (el^*\ \alpha^* \to el^*\ \beta^*) \mid \mathfrak{ob} \hookrightarrow \Rightarrow_k\text{-}iso\}.$$

Again by pattern matching the input α^* and β^* as $[\mathfrak{ob} \hookrightarrow \alpha \mid A]$ and $[\mathfrak{ob} \hookrightarrow \beta \mid B]$ as in (13), after expanding out the definition of el^*, what we need to construct is an isomorphism $F \cong A \to B$ that restricts to $\Rightarrow_k\text{-}iso$ under $\mathfrak{ob}$, where F is defined as in (15). We let the two directions of this isomorphism be

$$fwd\ [\mathfrak{ob} \hookrightarrow f \mid g] = g \qquad\qquad bwd\ h = [\mathfrak{ob} \hookrightarrow \Rightarrow_k\text{-}iso.bwd\ h \mid h]$$

where $h : A \to B$, $f : \{\mathfrak{ob}\} \to el\ (\alpha \Rightarrow_k \beta)$, and

$$g : \{A \to B \mid \mathfrak{ob} \hookrightarrow \Rightarrow_k\text{-}iso.fwd\ f\}.$$

These two functions are mutual inverses because

$$fwd\ (bwd\ h) = fwd\ ([\mathfrak{ob} \hookrightarrow \Rightarrow_k\text{-}iso.bwd\ h \mid h]) = h$$

and from the other direction,

$$bwd\ (\ fwd\ [\mathfrak{ob} \hookrightarrow f \mid g]\ ) = bwd\ g = [\mathfrak{ob} \hookrightarrow \Rightarrow_k\text{-}iso.bwd\ g \mid g];$$

now by the type of g, $g = (\Rightarrow_k\text{-}iso.fwd\ f)$ under $\mathfrak{ob}$, so the above further equals

$$[\mathfrak{ob} \hookrightarrow \Rightarrow_k\text{-}iso.bwd\ (\Rightarrow_k\text{-}iso.fwd\ f) \mid g] = [\mathfrak{ob} \hookrightarrow f \mid g].$$

The definition (15) of the logical predicate F for function kinds may look complicated at first, but it has a very intuitive explanation: F is basically the same as the type $A \to B$, except that its component in the object space, which is equal to $el\ \alpha \to el\ \beta$, is swapped for $el\ (\alpha \Rightarrow_k \beta)$ along the isomorphism $\Rightarrow_k\text{-}iso$, just like in the old days when a component of a personal computer can be replaced by a compatible part. This will be a recurring construction in the future, so for every universe U we define

$$\begin{aligned}
&realign : (A : U) \to (B : \{\mathfrak{ob}\} \to U) \to (\{\mathfrak{ob}\} \to B \cong A) \to \{U \mid \mathfrak{ob} \hookrightarrow B\}\\
&realign\ A\ B\ \phi = (b : B) \ltimes \{A \mid \mathfrak{ob} \hookrightarrow \phi.fwd\ b\}
\end{aligned}$$

$$\begin{aligned}
&realign\text{-}iso : (A : U) \to (B : \{\mathfrak{ob}\} \to U) \to (\phi : \{\mathfrak{ob}\} \to B \cong A)\\
&\qquad \to \{realign\ A\ B\ \phi \cong A \mid \mathfrak{ob} \hookrightarrow \phi\}\\
&(realign\text{-}iso\ A\ B\ \phi).fwd\ [\mathfrak{ob} \hookrightarrow b \mid a] = a\\
&(realign\text{-}iso\ A\ B\ \phi).bwd\ a = [\mathfrak{ob} \hookrightarrow \phi.bwd\ a \mid a]
\end{aligned}$$

This construction is called *realignment* [Sterling 2021, §3.3] on the universe U. In fact, realignment and strict glue types (Axiom 4.9) are inter-definable: if we take *realign* and *realign-iso* as axioms, we can define strict glue types $(a : A) \ltimes B$ by realigning the dependent pair type $\Sigma(a : A).\ B$.

Using realignment, the definition (14) can be succinctly expressed as

$$[\mathfrak{ob} \hookrightarrow \alpha \mid A] \Rightarrow^*_k [\mathfrak{ob} \hookrightarrow \beta \mid B] = [\mathfrak{ob} \hookrightarrow \alpha \Rightarrow_k \beta \mid realign\ (A \to B) \Rightarrow_k\text{-}iso]$$

and $\Rightarrow_k\text{-}iso^*$ is simply $realign\text{-}iso\ (A \to B) \Rightarrow_k\text{-}iso$.

We move on to the logical predicates for types and terms. Similar to function kinds, ty^* is ty glued together with some additional data:

$$\begin{aligned}
&ty^* : \{ki^* \mid \mathfrak{ob} \hookrightarrow ty\}\\
&ty^* = [\mathfrak{ob} \hookrightarrow ty \mid\ ?0 : \{U_1 \mid \mathfrak{ob} \hookrightarrow el\ ty\}\ ]
\end{aligned}$$

Since $?0$ is a type in U_1 that is equal to $el\ ty$ under $\mathfrak{ob}$, it can be a glue type:

$$ty^* = [\mathfrak{ob} \hookrightarrow ty \mid (A : el\ ty) \ltimes ?1\ ] \tag{16}$$

which means that an element of the kind ty in the model M^* is a syntactic type A together with the data $?1$. It is natural to expect that the data $?1$ associated to a type A is a (candidate of) logical predicate for the type A, which is just any type that restricts to $tm\ A$ under $\mathfrak{ob}$:

$$ty^* = [\mathfrak{ob} \hookrightarrow ty \mid (A : el\ ty) \ltimes \{U_0 \mid \mathfrak{ob} \hookrightarrow tm\ A\}\ ] \tag{$*$}$$

mimicking the kind structure (11) that we have seen earlier. However, this definition will not work when we come to *impredicative* polymorphic types $\forall\alpha.A$ later, because U_0 is not impredicative in the sense of being closed under Π-types $\Pi\ A\ B$ for *arbitrary* types A that are not necessarily in U_0.

In every topos, we do have an impredicative universe – the universe Ω of *propositions*. Unfortunately, this universe is 'too small' for interpreting $\mathrm{F}^{\mathrm{ha}}_{\omega}$-types. If we have an element $A^* : \{\Omega \mid \mathfrak{ob} \hookrightarrow tm\ A\}$ for some object-space type $A : el\ ty$, when $\mathfrak{ob}$ holds, A^* is equal to $tm\ A$, but A^* is in the universe Ω, so we have $\{\mathfrak{ob}\} \to (a, b : tm\ A) \to a = b$, which means that the object-space type A has at most one element, and this is clearly not true in general.

To find a way out, let us recall how traditional logical predicates/relations of System F work in, for example, Girard's [1989] normalisation proof. For every type A of System F, its logical predicate is a proof-irrelevant predicate on the set of terms of A, or equivalently, a function from terms of A to the set of classical propositions. Moreover, the logical predicate $P(t)$ of the impredicative polymorphic type $\forall\alpha.\ A$ is defined by 'for all types X and all candidate logical predicates Q over terms of X, the term $t\ [X]$ is related by the logical predicate of A with α replaced by (X, Q)'. This works because classical propositions are impredicative, so we can quantify over all X and Q.

Mimicking the traditional approach, we first define a universe of *meta-space propositions* (which are just classical propositions $\{\top, \bot\}$ when StcTT is interpreted in the Artin gluing of the syntactic category and the category of sets):

$$\Omega^\bullet := \{p : \Omega \mid \bullet\text{-}modal\ p\}.$$

The universe $\Omega^\bullet$ inherits all the connectives that Ω has, including impredicative quantification. For example, if A is an arbitrary type and $B : A \to \Omega^\bullet$, the type $\forall(x : A).B\ x$ is in Ω, and when $\mathfrak{ob}$ holds, $B\ x \cong 1$ because a type is $\bullet$-modal iff it is isomorphic to 1 under $\mathfrak{ob}$ (Lemma 4.8), so $\forall(x : A).B\ x = \forall(x : A).1 \cong 1$.

Using $\Omega^\bullet$, we fill out the hole $?1$ in ty^* (16) by

$$\begin{aligned} &ty^* : \{ki^* \mid \mathfrak{ob} \hookrightarrow ty\} \\ &ty^* = [\mathfrak{ob} \hookrightarrow ty \mid (A : el\ ty) \ltimes (\{\mathfrak{ob}\} \to tm\ A) \to \Omega^\bullet\ ] \end{aligned} \tag{17}$$

That is to say, the candidate of a logical predicate for a type A is given as a meta-space predicate $P : (\{\mathfrak{ob}\} \to tm\ A) \to \Omega^\bullet$.

Then tm^* glues terms $tm\ A$ of an object-space type A with the predicate P:

$$\begin{aligned} &tm^* : \{el^*\ ty^* \to U_0 \mid \mathfrak{ob} \hookrightarrow tm\} \\ &tm^*\ [\mathfrak{ob} \hookrightarrow A \mid P] = (t : tm\ A) \ltimes P\ t \end{aligned} \tag{18}$$

That is to say, in the model M^*, a term of the semantic type $[\mathfrak{ob} \hookrightarrow A \mid P] : ty^*$ is a term t of the syntactic type A that satisfies the meta-space predicate P.

Notation C.4. For every $A^* : el^*\ ty^*$, we define

$$\begin{aligned} &pre\ A^* : (\{\mathfrak{ob}\} \to tm\ A^*) \to \Omega^\bullet \\ &pre\ A^* = unglue\ A^* \end{aligned}$$

to remind us that ungluing a semantic type gives its underlying logical predicate. Similarly, for every $a^* : tm^*\ A^*$, we define

$$\begin{aligned} &prf\ a^* : pre\ A^*\ (\lambda\{\_ : \mathfrak{ob}\}.\ a^*) \\ &prf\ a^* = unglue\ a^* \end{aligned}$$

to remind us that ungluing a semantic term is the proof that the underlying syntactic term satisfies the corresponding logical predicate.

Remark C.5. For every $A^* : el^*\ ty^*$, the type $tm^*\ A^*$ satisfies the property that for every $a : \{\mathfrak{ob}\} \to A^*$, there is at most one element $a^* : tm^*\ A^*$ that restricts to a under $\mathfrak{ob}$, because the meta-space component of $tm^*\ A^*$ is a (fiberwise) meta-space proposition. Based on this observation, there is a more intrinsic alternative definition of ty^*: for every universe U of StcTT, we can define its *proof-irrelevant* subuniverse U^{ir} to be

$$U^{\mathrm{ir}} := \{A : U \mid \forall(a : \{\mathfrak{ob}\} \to A).\ (x, y : \{A \mid \mathfrak{ob} \hookrightarrow a\}) \to (x = y)\}.$$

Then we can define ty^* and tm^* as simply

$$\begin{aligned} ty^* &= [\mathfrak{ob} \hookrightarrow ty \mid (A : el\ ty) \ltimes \{U_0^{\mathrm{ir}} \mid \mathfrak{ob} \hookrightarrow tm\ A\}\ ] \\ tm^*\ A^* &= unglue\ A^* \end{aligned}$$

which directly mirrors the definition of ki^* (11) and el^* (12).

This alternative definition is in a suitable sense equivalent to the one above (17, 18) because for every $A : \{\mathfrak{ob}\} \to U$, we have an equivalence

$$\{U_0^{\mathrm{ir}} \mid \mathfrak{ob} \hookrightarrow A\} \ \cong\ ((\{\mathfrak{ob}\} \to\ A) \to \Omega^\bullet).$$

when treating them as categories (in fact, preorders) suitably. We choose to work with ty^* (17) in terms of $\Omega^\bullet$-valued predicates because it is slightly more convenient for logical predicates on computation judgements later.

C.1 Base Types

Since in the theory of $\mathrm{F}_\omega^{\mathrm{ha}}$, the unit type is specified to be isomorphic to meta-level unit type (F_ω-4), we have no choice for the logical predicate for the logical predicate of the unit type (of $\mathrm{F}_\omega^{\mathrm{ha}}$) other than the always true predicate:

$$\begin{aligned} &unit^* : \{el^*\ ty^* \mid \mathfrak{ob} \hookrightarrow unit\} \\ &unit^* = [\mathfrak{ob} \hookrightarrow unit \mid \lambda(\_ : \{\mathfrak{ob}\} \to tm\ unit).\ 1] \end{aligned}$$

Recall that $tm^*\ unit^*$ computes to $(t : tm\ unit) \ltimes 1$, we define

$$\begin{aligned} &unit\text{-}iso^* : tm^*\ unit^* \cong 1 \\ &unit\text{-}iso^*.fwd\ \_ = * \\ &unit\text{-}iso^*.bwd\ \_ = [\mathfrak{ob} \hookrightarrow unit\text{-}iso.bwd \mid *] \end{aligned}$$

This is an isomorphism because $tm\ unit \cong 1$ by *unit-iso*.

The other base type is the weak Boolean type *bool*. It is also the type that canonicity is about, so its logical predicate is specific to canonicity:

$$\begin{aligned} &bool^* : \{el^*\ ty^* \mid \mathfrak{ob} \hookrightarrow bool\} \\ &bool^* = [\mathfrak{ob} \hookrightarrow bool \mid P_{can}] \\ \\ &P_{can} : (\{\mathfrak{ob}\} \to tm\ bool) \to \Omega^\bullet \\ &P_{can}\ b = \bullet(\{\mathfrak{ob}\} \to (b = tt \vee b = ff)) \end{aligned} \tag{19}$$

The closed modality ● is needed here to erase the object-space component of the proposition $\{\mathfrak{ob}\} \to (b = tt \vee b = ff)$, turning it ●-modal. We also need to define the two terms of the weak Boolean types, i.e. showing that the two terms ff and tt satisfy the logical predicate of $bool$:

$$\begin{aligned} &tt^* : \{tm^*\ bool^* \mid \mathfrak{ob} \hookrightarrow tt\} \\ &tt^* = [\mathfrak{ob} \hookrightarrow tt \mid \eta^\bullet\ (\mathrm{inl}\ refl)] \end{aligned}$$

$$\begin{aligned} &ff^* : \{tm^*\ bool^* \mid \mathfrak{ob} \hookrightarrow ff\} \\ &ff^* = [\mathfrak{ob} \hookrightarrow ff \mid \eta^\bullet\ (\mathrm{inr}\ refl)] \end{aligned}$$

In the construction of M^*, the only things that are specific to canonicity are P_{can}, tt^* and ff^*. They can be changed to anything else without affecting other parts of M^* (although there seemingly are not many interesting choices of P_{can}).

C.2 Function Types

A function $t : tm\ (A \Rightarrow_t B)$ is related by the logical predicate for the type $A \Rightarrow_t B$ if it maps input a satisfying the logical predicate for A to output $t\ a$ satisfying the logical predicate for B:

$$\begin{aligned} &\_\Rightarrow_t\_^* : \{el^*\ ty^* \to el^*\ ty^* \to el^*\ ty^* \mid \mathfrak{ob} \hookrightarrow \_\Rightarrow_t\_\} \\ &[\mathfrak{ob} \hookrightarrow A \mid P] \Rightarrow_t^* [\mathfrak{ob} \hookrightarrow B \mid Q] = [\mathfrak{ob} \hookrightarrow A \Rightarrow_t B \mid P_{\Rightarrow_t}] \\ &P_{\Rightarrow_t} := \lambda t.\ \forall (a : \{\mathfrak{ob}\} \to A).\ P\ a \to Q\ (\lambda\{\_ : \mathfrak{ob}\}.\ t\ a) \end{aligned} \tag{20}$$

Note that in the expression $t\ a$, we have elided the isomorphism $\Rightarrow_t\text{-}iso$ (F_ω-4) between $tm\ (A \Rightarrow_t B)$ and $tm\ A \to tm\ B$.

We also need to define an isomorphism, for all $A^*, B^* : el^*\ ty^*$,

$$\Rightarrow_t\text{-}iso^* : tm^*\ (A^* \Rightarrow_t^* B^*) \cong (tm^*\ A^*) \to (tm^*\ B^*).$$

Letting $A^* = [\mathfrak{ob} \hookrightarrow A \mid P]$ and $B^* = [\mathfrak{ob} \hookrightarrow B \mid Q]$, we compute as follows:

$$\begin{aligned} & tm^*\ ([\mathfrak{ob} \hookrightarrow A \mid P] \Rightarrow_t^* [\mathfrak{ob} \hookrightarrow B \mid Q]) \\ =\ & (t : tm\ (A \Rightarrow_t B)) \ltimes (a : \{\mathfrak{ob}\} \to tm\ A) \to P\ a \to Q\ (t\ a) \\ \cong\ & \quad \{\text{by } \Rightarrow_t\text{-}iso\ (\mathrm{F}_\omega\text{-}4)\} \\ & (t : tm\ A \to tm\ B) \ltimes (a : \{\mathfrak{ob}\} \to tm\ A) \to P\ a \to Q\ (t\ a) \\ \cong\ & \quad \{\text{by } \ltimes\text{-}fun\text{-}iso \text{ from Lemma C.2}\} \\ & \big((a : tm\ A) \ltimes P\ a\big) \to \big((b : tm\ B) \ltimes Q\ b\big) \\ =\ & (tm^*\ [\mathfrak{ob} \hookrightarrow A \mid P]) \to (tm^*\ [\mathfrak{ob} \hookrightarrow B \mid Q]) \end{aligned}$$

The logical predicate for polymorphic functions is

$$\begin{aligned} &\bar\forall^* : \{(k^* : ki^*) \to (el^*\ k^* \to el^*\ ty^*) \to el^*\ ty^* \mid \mathfrak{ob} \hookrightarrow \bar\forall\} \\ &\bar\forall^*\ k^*\ F = [\mathfrak{ob} \hookrightarrow \bar\forall\ k^*\ F \mid \lambda t.\ \forall(\alpha^* : el^*\ k^*).\ pre\ (F\ \alpha^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ (t\ \alpha^*))] \end{aligned} \tag{21}$$

Let us check the type of this definition step-by-step. The object-space component $\bar\forall\ k^*\ F$ is well typed because under $\mathfrak{ob}$, ki^* equals ki, so $k^* : ki$ under $\mathfrak{ob}$, and similarly $F : el\ k \to el\ ty$ under $\mathfrak{ob}$, thus $\bar\forall\ k^*\ F : el\ ty$ as expected.

The meta-space component $\lambda t. \cdots$ should be an $\Omega^\bullet$-valued predicate on $t : \{\mathfrak{ob}\} \to tm\ (\bar\forall\ k^*\ F)$. We have $F\ \alpha^* : el^*\ ty^*$; this type computes to

$$(A : el\ ty) \ltimes (\{\mathfrak{ob}\} \to\ tm\ A) \to \Omega^\bullet$$

by definitions (12, 17). Thus $pre\ (F\ \alpha^*)$ has type $(\{\mathfrak{ob}\} \to\ tm\ (F\ \alpha^*)) \to \Omega^\bullet$. On the other hand, t has type $\{\mathfrak{ob}\} \to tm\ (\bar\forall\ k^*\ F)$, which is isomorphic to $\{\mathfrak{ob}\} \to (\alpha^* : el\ k^*) \to tm\ (F\ \alpha^*)$

via $\bar{\forall}$-*iso* (F_ω-4), which we elided in the definition above. The implicit function $\lambda\{\_ : \mathfrak{ob}\}.\ (t\ \alpha^*)$ then has type $\{\mathfrak{ob}\} \to tm\ (F\ \alpha^*)$, and therefore it can be supplied as an argument to $pre\ (F\ \alpha^*)$, yielding a proposition in $\Omega^\bullet$. The quantification $\forall(\alpha^* : el^*\ k^*)$ is allowed because $\Omega^\bullet$ is closed under impredicative universal quantification.

We also need to define an isomorphism for $k^* : ki^*$ and $F : el^*\ k^* \to el^*\ ty^*$

$$\bar{\forall}\text{-}iso^* : tm^*\ (\bar{\forall}^*\ k^*\ F) \cong ((\alpha^* : el^*\ k^*) \to tm^*\ (F\ \alpha^*)).$$

This is very similar to $\Rightarrow_t$-iso^* in (20), and we give a direct definition here:

$$fwd\ [\mathfrak{ob} \hookrightarrow t \mid p] = \lambda\alpha^*.\ [\mathfrak{ob} \hookrightarrow p\ \alpha^* \mid \bar{\forall}\text{-}iso.fwd\ t\ \alpha^*]$$

$$bwd\ h = [\mathfrak{ob} \hookrightarrow \bar{\forall}\text{-}iso.bwd\ h \mid \lambda\alpha^*.\ prf\ (h\ \alpha^*)]$$

By this point we have completed the definition of the logical predicates for the F_ω-fragment of F^{ha}_ω, so the derived concepts in Figure 1 such as raw functors can be interpreted in M^* as well. For example, we have

$$\begin{aligned}
&tyco^* : ki^*\\
&tyco^* = ty^* \Rightarrow^*_k ty^*\\
&fmap\text{-}ty^* : (F : el^*\ tyco^*) \to el^*\ ty^*\\
&fmap\text{-}ty^*\ F = \bar{\forall}^*\ ty^*\ (\lambda\alpha.\ \bar{\forall}^*\ ty^*\ (\lambda\beta.\ (\alpha \Rightarrow^*_t \beta) \Rightarrow^*_t (F\ \alpha \Rightarrow^*_t F\ \beta)))\\
&\textsc{record}\ RawFunctor^* : U_1\ \textsc{where}\\
&\quad 0 : el^*\ tyco^*\\
&\quad fmap : tm^*\ fmap\text{-}ty^*\ 0
\end{aligned}$$

which is simply the same as the definition of *RawFunctor* in Figure 1 except that all the judgements of F_ω such as *ki* and *ty* are replaced by their corresponding interpretation in M^*. Interpretations of other derived concepts in Figure 1 such as $RawMonad^*$ and $RawHFunctor^*$ can be obtained in this way as well.

C.3 Computation Judgements

What remains is the logical predicates for computation judgements of F^{ha}_ω. Recall that given H : *RawHFunctor* and A : *el ty*, the judgements *co H A* in F^{ha}_ω roughly axiomatise a monad equipped with H-operations – more precisely, *co H A* axiomatises the Kleisli category of this monad since *co H A* is not an F^{ha}_ω-type but a separate judgement.

(*First attempt*) Since our logical predicates live in an impredicative universe, a natural idea is to define the logical predicate for *co* by an impredicative encoding of the initial monad equipped with H-operations:

$$\begin{aligned}
&co^* : \{HFunctor^* \to el^*\ ty^* \to U_0 \mid \mathfrak{ob} \hookrightarrow co\}\\
&co^*\ H^*\ A^* = (c : co\ H^*\ A^*) \ltimes P_{co}\ c\\
&P_{co} : \{H^*, A^*\} \to (\{\mathfrak{ob}\} \to co\ H^*\ A^*) \to \Omega^\bullet\\
&P_{co}\ c = \forall(m : MonadAlg^*\ H^*).\ pre\ (m.0\ A^*)\ (eval\ m\ A^*\ c)
\end{aligned} \tag{22}$$

The function P_{co} type-checks as follows: the type of $m.0$ is $el^*\ tyco^*$, and

$$\begin{aligned}
&el^*\ tyco^*\\
=\ &\{\text{by definition}\}\\
&el^*\ (ty^* \Rightarrow^*_k ty^*)\\
\cong\ &\{\text{by the axiom } (F_\omega\text{-3})\}
\end{aligned}$$

$$el^*\ ty^* \to el^*\ ty^*$$

Therefore the type of $m.0\ A^*$ is $el^*\ ty^*$, which is the glue type

$$(A : el\ ty) \ltimes (\{\mathfrak{ob}\} \to\ tm\ A) \to \Omega^\bullet$$

by (12, 17). Then $pre\ (m.0\ A^*)$ has type $(\{\mathfrak{ob}\} \to\ tm\ (m.0\ A^*)) \to \Omega^\bullet$, i.e. it is a meta-space predicate on terms of type $\{\mathfrak{ob}\} \to m.0\ A^*$. The term $eval\ m\ A^*\ c$, which in fact is the implicit function $\lambda\{\_ : \mathfrak{ob}\}.\ eval\ m\ A^*\ c$, has precisely the type $\{\mathfrak{ob}\} \to\ tm\ (m.0\ A^*)$. Finally, since $\Omega^\bullet$ is closed under universal quantification $\forall(m : MonadAlg^*\ H^*), P_{co}\ c$ has type $\Omega^\bullet$, i.e. $\{\Omega \mid \mathfrak{ob} \hookrightarrow 1\}$.

Remark C.6. Usually, the impredicative encoding of a datatype needs to be 'refined' by some additional equalities to have the correct universal property [Awodey et al. 2018]. For example, the impredicative encoding of the coproduct type $A + B$ in an impredicative universe U is

$$\begin{aligned} A + B &= \textstyle\sum(\alpha : (X : U) \to (A \to X) \to (B \to X) \to X)\ N\ \alpha \\ N\ \alpha &= (X, Y : U) \to (f : X \to Y) \to (h : A \to X) \to (k : B \to X) \\ &\quad \to f\ (\alpha\ X\ h\ k) = \alpha\ Y\ (f \circ h)(f \circ k) \end{aligned}$$

Without imposing N on α, the impredicative encoding would not satisfy the η-rule of the coproduct type. However, our logical predicates land in a universe *propositions*, where two elements of the same type are automatically equal, so this refinement is unnecessary.

However, as we commented in Remark 2.6, evaluating the sequential composition $let\text{-}in\ c\ f$ is *not* compositional because of the discrepancy between computations *co* and raw monads: *co* satisfies the monadic laws while raw monads do not. For this reason, with the above definition of P_{co}, we will have problems with showing the term constructor *let-in* satisfies its logical predicate:

$$\begin{aligned} &let\text{-}in^* : \{\{H^*, A^*, B^*\} \to co^*\ H^*\ A^* \to (tm^*\ A^* \to co^*\ H^*\ B^*) \\ &\qquad\qquad \to co^*\ H^*\ B^* \mid \mathfrak{ob} \hookrightarrow let\text{-}in\} \\ &let\text{-}in^*\ c\ f = [\mathfrak{ob} \hookrightarrow let\text{-}in\ c\ f \mid \lambda m.\ \texttt{?1}\ ] \end{aligned}$$

where the hole ?1 has type $P_{co}\ (let\text{-}in\ c\ f)$, that is, by the definition of P_{co} above,

$$\forall(m : MonadAlg^*\ H^*).\ pre\ (m.0\ B^*)\ (eval\ m\ B^*\ (let\text{-}in\ c\ f)).$$

Since we do not have the equation *eval-let* in Remark 2.6 to simplify the computation $eval\ m\ B^*\ (let\text{-}in\ c\ f)$, we have no way to fill in the hole ?1 using c and f.

To fix this problem, we strengthen $P_{co}\ c$ above to take into account all possible *continuations* after the computation c, which is essentially the idea of $\top\top$*-lifting* [Katsumata et al. 2018; Katsumata 2005; Lindley and Stark 2005]. We first define a type of continuations accepting A^*-values:

$$\begin{aligned} &\textsc{record}\ Con\ (H^* : RawHFunctor^*)\ (A^* : el^*\ ty^*) : U_1\ \textsc{where} \\ &\quad m^* : MonadAlg^*\ H^* \\ &\quad R^* : el^*\ ty^* \\ &\quad k : \{\mathfrak{ob}\} \to A^* \to co\ H^*\ R^* \\ &\quad k^* : \{tm^*\ A^* \to tm^*\ (m^*.0\ R^*) \mid \mathfrak{ob} \hookrightarrow \lambda a.\ eval\ m^*\ R^*\ (k\ a)\} \end{aligned}$$

and the strengthened definition of P_{co} is

$$\begin{aligned} &P_{co} : \{H^*, A^*\} \to (\{\mathfrak{ob}\} \to co\ H^*\ A^*) \to \Omega^\bullet \\ &P_{co}\ c = \forall(K : Con\ H^*\ A^*).\ pre\ (K.m^*.0\ \ K.R^*)\ (\lambda\{\_ : \mathfrak{ob}\}. \\ &\qquad\qquad eval\ K.m^*\ K.R^*\ (let\text{-}in\ c\ K.k)) \end{aligned} \tag{23}$$

Compared to the earlier version of P_{co} (22), the new version asserts that the computation c extended with an arbitrary 'good' continuation k and evaluated into a raw monad results in a value satisfying its logical predicate. Here a continuation k is 'good' if k followed by *eval* sends input satisfying

its logical predicate to output satisfying its logical predicate, which is succinctly expressed by a function $k^* : tm^*\ A^* \to tm^*\ (m^*.0\ R^*)$, c.f. Lemma C.2.

Remark C.7. The new definition of P_{co} is similar to the model of computations in the realizability model ((8)), except that here we only consider Kleisli morphisms $k^* : A^* \to m^*.0\ R^*$ whose object-space component factors through some $k : \{\mathfrak{ob}\} \to A^* \to co\ H^*\ R^*$. The author currently does not know if there could be a conceptual explanation of such a modified codensity transformation.

The logical predicate for thunks is the same as that for computations, modulo the isomorphism $th\text{-}iso : \{H, A\} \to tm\ (th\ H\ A) \cong co\ H\ A$ from ($\mathrm{F}^{\mathrm{ha}}_{\omega}$-4):

$$\begin{aligned} &th^* : \{RawHFunctor^* \to el^*\ ty^* \to el^*\ ty^* \mid \mathfrak{ob} \hookrightarrow \mathbb{T}\} \\ &th^*\ H^*\ A^* = [\mathfrak{ob} \hookrightarrow \mathbb{T}\ H^*\ A \mid \lambda t.\ P_{co}\ (\lambda\{\_ : \mathfrak{ob}\}.\ \Uparrow t)] \end{aligned}$$

where $\Uparrow$ is the forward direction of the isomorphism $th\text{-}iso$. The isomorphism $th\text{-}iso^* : tm^*\ (\mathbb{T}\ H^*\ A^*) \cong co^*\ H^*\ A^*$ is also straightforward:

$$fwd\ [\mathfrak{ob} \hookrightarrow t \mid p] = [\mathfrak{ob} \hookrightarrow \Uparrow t \mid p], \qquad bwd\ [\mathfrak{ob} \hookrightarrow c \mid p] = [\mathfrak{ob} \hookrightarrow \Downarrow c \mid p].$$

C.4 Computation Terms

Finally, we need to prove that the constructors *val*, *let-in*, *op* and the eliminator *eval* of computations satisfy the logical predicates. We start with *val*:

$$\begin{aligned} &val^* : \{\{H^*, A^*\} \to tm^*\ A^* \to co^*\ H^*\ A^* \mid \mathfrak{ob} \hookrightarrow val\} \\ &val^*\ \{H^*\}\ \{A^*\}\ a = [\mathfrak{ob} \hookrightarrow val\ a \mid \ ?1\ ] \end{aligned}$$

where the hole $?1$ has type $P_{co}\ (val\ a)$, that is, by definition (23),

$$\begin{aligned} &\forall(K : Con\ H^*\ A^*).\ pre\ (K.m^*.0\ \ K.R^*) \\ &\qquad (\lambda\{\_ : \mathfrak{ob}\}.\ \ eval\ K.m^*\ K.R^*\ (let\text{-}in\ (val\ a)\ K.k)) \\ =\quad &\{\text{by axiom } let\text{-}val\ (\mathrm{F}^{\mathrm{ha}}_{\omega}\text{-}3)\} \\ &\forall(K : Con\ H^*\ A^*).\ pre\ (K.m^*.0\ \ K.R^*) \\ &\qquad (\lambda\{\_ : \mathfrak{ob}\}.\ \ eval\ K.m^*\ K.R^*\ (K.k\ a)) \end{aligned}$$

We put $?1 = \lambda K.\ prf\ (K.k^*\ a)$, which is well typed because $K.k^*$ has type

$$k^* : \{tm^*\ A^* \to tm^*\ (m^*.0\ R^*) \mid \mathfrak{ob} \hookrightarrow \lambda a.\ eval\ m^*\ R^*\ (k\ a)\} \tag{24}$$

so $K.k^*\ a$ has type $tm^*\ (K.m^*.0\ K.R^*)$, and $prf\ (K.k^*\ a)$ has type

$$\begin{aligned} &pre\ (K.m^*.0\ K.R^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ (K.k^*\ a)) \\ =\quad &\{\text{by the restriction of } k^* \text{ under } \mathfrak{ob} \text{ in (24)}\} \\ &pre\ (K.m^*.0\ K.R^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ \lambda a.\ eval\ K.m^*\ K.R^*\ (K.k\ a)) \end{aligned}$$

which is the desired type of $?1$.

The case for *let-in* is similar:

$$\begin{aligned} &let\text{-}in^* : \{\{H^*, A^*, B^*\} \to co^*\ H^*\ A^* \to (co^*\ H^*\ A^* \to co^*\ H^*\ B^*) \\ &\qquad\qquad \to co^*\ H^*\ B \mid \mathfrak{ob} \hookrightarrow let\text{-}in\} \\ &let\text{-}in^*\ c\ f = [\mathfrak{ob} \hookrightarrow let\text{-}in\ c\ f \mid \lambda(K : Con\ H^*\ A^*).\ unglue\ c\ K'] \end{aligned}$$

where each field of $K' : Con\ H^*\ B^*$ is defined as follows:

$$\begin{aligned} K'.m^* &= K.m^* \\ K'.R^* &= K.R^* \end{aligned}$$

$$K'.k = \lambda\{\_ : \mathfrak{ob}\}\ a.\ \mathit{let\text{-}in}\ (f\ a)\ K.k$$
$$K'.k^* = \lambda a.\ [\mathfrak{ob} \hookrightarrow \mathit{eval}\ K'.m^*\ K'.R^*\ (\mathit{let\text{-}in}\ (f\ a)\ K.k) \mid \mathit{unglue}\ (f\ a)\ K]$$

The last line type checks because $f\ a : \mathit{co}^*\ H^*\ B^*$, so $\mathit{unglue}\ (f\ a) : P_{co}\ (f\ a)$, so by definition (23), the type of $\mathit{unglue}\ (f\ a)\ K$ is

$$\mathit{pre}\ (K.m^*.0\ \ K.R^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ \mathit{eval}\ K.m^*\ K.R^*\ (\mathit{let\text{-}in}\ (f\ a)\ K.k))$$

which is indeed the type of proofs that the syntactic component of $K'.k^*$ satisfies the logical predicate of the type $k.m^*.0\ K.R^*$.

The case for *op* is slightly more involved, so let us first show that *eval* satisfies the corresponding logical predicate:

$$\begin{aligned}\mathit{eval}^* : \{\{H^*\} &\to (m^* : \mathit{MonadAlg}^*\ H^*) \to (A^* : \_)\\ &\to \mathit{co}^*\ H^*\ A^* \to \mathit{tm}^*\ (m^*.0\ A^*) \mid \mathfrak{ob} \hookrightarrow \mathit{eval}\}\\ \mathit{eval}^*\ m^*\ A^*\ c^* &= [\mathfrak{ob} \hookrightarrow \mathit{eval}\ m^*\ A^*\ c \mid \mathit{unglue}\ c^*\ K]\end{aligned}$$

where the continuation $K : \mathit{Con}\ H^*\ A^*$ is defined by

$$K.m^* = m^* \qquad K.R^* = A^*$$
$$K.k = \lambda\{\_ : \mathfrak{ob}\}.\ \mathit{val} \qquad K.k^* = m^*.\mathit{ret}$$

The definition of $K.k^*$ is well typed because the expected type of $K.k^*$ is

$$\begin{aligned}&\{\mathit{tm}^*\ A^* \to \mathit{tm}^*\ (m^*.0\ R^*) \mid \mathfrak{ob} \hookrightarrow \lambda a.\ \mathit{eval}\ m^*\ R^*\ (k\ a)\}\\ =\ &\{\text{by the definition of } K.k \text{ above}\}\\ &\{\mathit{tm}^*\ A^* \to \mathit{tm}^*\ (m^*.0\ A^*) \mid \mathfrak{ob} \hookrightarrow \lambda a.\ \mathit{eval}\ m^*\ R^*\ (\mathit{val}\ a)\}\\ =\ &\{\text{by axiom } \mathit{eval\text{-}val}\ (\mathrm{F}^{\mathrm{ha}}_{\omega}\text{-}8)\}\\ &\{\mathit{tm}^*\ A^* \to \mathit{tm}^*\ (m^*.0\ A^*) \mid \mathfrak{ob} \hookrightarrow \lambda a.\ m^*.\mathit{ret}\ R^*\ a\}\end{aligned}$$

Coming back to *op*, we start with some obvious steps and a hole:

$$\begin{aligned}\mathit{op}^* : \{\{H^*, A^*, B^*\} &\to \mathit{tm}^*\ (H^*.0\ (\mathbb{T}^*\ H^*)\ A^*)\\ &\to (\mathit{tm}^*\ A^* \to \mathit{co}^*\ H^*\ B^*) \to \mathit{co}^*\ H^*\ B^* \mid \mathfrak{ob} \hookrightarrow \mathit{op}\}\\ \mathit{op}^*\ o\ k &= [\mathfrak{ob} \hookrightarrow \mathit{op}\ o\ k \mid \lambda(K : \mathit{Con}\ H^*\ B^*).\ \texttt{?1}\ ]\end{aligned}$$

where the hole `?1` has type

$$\begin{aligned}&\mathit{pre}\ (K.m^*.0\ \ K.R^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ \mathit{eval}\ K.m^*\ K.R^*\ (\mathit{let\text{-}in}\ (\mathit{op}\ o\ k)\ K.k))\\ =\ &\{\text{by axiom } \mathit{let\text{-}op}\ (\mathrm{F}^{\mathrm{ha}}_{\omega}\text{-}6)\}\\ &\mathit{pre}\ (K.m^*.0\ \ K.R^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ \mathit{eval}\ K.m^*\ K.R^*\ (\mathit{op}\ o\ (\lambda a.\ \mathit{let\text{-}in}\ (k\ a)\ K.k)))\\ =\ &\{\text{by axiom } \mathit{eval\text{-}op}\ (\mathrm{F}^{\mathrm{ha}}_{\omega}\text{-}9)\}\\ &\mathit{pre}\ (K.m^*.0\ \ K.R^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ K.m^*.\mathit{bind}\ o'\ (\lambda a.\ \mathit{eval}\ \_\ \_\ (\mathit{let\text{-}in}\ (k\ a)\ K.k)))\end{aligned}$$

where $o' : \mathit{tm}^*\ (K.m^*.0\ A^*)$ is the result of evaluating the operand o inside the higher-order functor H and then applying the operation on the monad $K.m^*$:

$$o' := K.m^*.\mathit{malg}\ \_\ (H^*.\mathit{hmap}\ \_\ \_\ e\ \_\ o),$$

and $e : \mathit{tm}^*\ (\mathit{trans}^*\ (M^*.\mathbb{T}\ H^*)\ K.m^*.0)$ is eval^* specialised to $K.m^*$:

$$e\ A^*\ c = \mathit{eval}^*\ K.m^*\ A^*\ (\Uparrow c).$$

Now coming back to the hole ?1, using k we can define

$$f : (a : tm^*\ A^*) \to tm^*\ (K.m^*.0\ \ K.R^*)$$
$$f = [\mathfrak{ob} \hookrightarrow eval\ \_\ \_\ (let\text{-}in\ (k\ a)\ K.k) \mid unglue\ (k\ a)\ K]$$

and finally we can put ?1 $= prf\ (K.m^*.bind\ o'\ f)$.

The last bit of our construction of the glued model M^* is showing that it satisfies the equational axioms of $\mathrm{F}_\omega^{\mathrm{ha}}$ pertaining to computations, but this is easy because our interpretation of computations and terms in M^* is proof *irrelevant*. For every universe U of StcTT, there is a subuniverse

$$U^{\mathrm{ir}} = \{A : U \mid \forall(a : \{\mathfrak{ob}\} \to A).\ (x, y : \{A \mid \mathfrak{ob} \hookrightarrow a\}) \to x = y\}$$

which classifies *proof-irrelevant* logical predicates in U, in the sense that partial elements $a : \{\mathfrak{ob}\} \to A$ of a type $A : U^{\mathrm{ir}}$ have unique total extensions (if exist).

Lemma C.8. *For all $A^* : el^*\ ty^*$ and $H^* : RawHFunctor^*$, the types*

$$tm^*\ A^* : U_0 \quad \text{and} \quad co^*\ H^*\ A^* : U_0$$

are classified by the subuniverse U_0^{ir}.

Proof. Let A^* be $[\mathfrak{ob} \hookrightarrow A \mid P]$ where $A : \{\mathfrak{ob}\} \to el\ ty$ is an object-space type and $P : (\{\mathfrak{ob}\} \to tm\ A) \to \Omega^\bullet$ is a meta-space predicate. By definition (18), $tm^*\ A^*$ is the glue type $(a : tm\ A) \ltimes P\ a$. Thus a partial element a of $tm^*\ A^*$ is exactly an element $a : \{\mathfrak{ob}\} \to tm\ A$. Given two elements $x, y : \{tm^*\ A^* \mid \mathfrak{ob} \hookrightarrow a\}$, $unglue\ x = unglue\ y$ since they are elements of the propositional type $P\ a$, so $x = [\mathfrak{ob} \hookrightarrow a \mid unglue\ x] = [\mathfrak{ob} \hookrightarrow a \mid unglue\ y] = y$. The case for co^* is similar. □

Corollary C.9. The glued model M^* satisfies the equational axioms *val-let*, *let-val*, *let-assoc* ($\mathrm{F}_\omega^{\mathrm{ha}}$-3), *let-op* ($\mathrm{F}_\omega^{\mathrm{ha}}$-6), *eval-val* ($\mathrm{F}_\omega^{\mathrm{ha}}$-8), *eval-op* ($\mathrm{F}_\omega^{\mathrm{ha}}$-9) of $\mathrm{F}_\omega^{\mathrm{ha}}$.

Proof. Taking *val-let* for example, we need to show

$$val\text{-}let^* : \{H^*, A^*, B^*\} \to (a : tm^*\ A^*) \to (k : tm^*\ A \to co^*\ H^*\ B^*)$$
$$\to let\text{-}in^*\ (val^*\ H^*\ a)\ k = k\ a$$

Since the type $co^*\ H^*\ B^*$ is in the universe U_0^{ir}, it is sufficient to show that $let\text{-}in^*\ (val^*\ H^*\ a)\ k$ and $k\ a$ are equal under $\mathfrak{ob}$,

$$\begin{aligned} & let\text{-}in^*\ (val^*\ H^*\ a)\ k \\ = \quad & \{\text{under } \mathfrak{ob},\ let\text{-}in^* = let\text{-}in \text{ and } val^* = val\} \\ & let\text{-}in\ (val\ H^*\ a)\ k \\ = \quad & \{M \text{ satisfies } val\text{-}let\} \\ & k\ a \end{aligned}$$

The case for other equational axioms are similar. □

We have completed the construction of M^* and thus proved Lemma 4.11.

D Parametricity and Free Theorems

An appealing aspect of the synthetic fundamental lemma (4.11) is that it is proved solely in the language StcTT, thus applicable to any category $\mathscr{G}$ that models StcTT. As an instance, we can deduce the *abstraction theorem* [Reynolds 1983], also known as *parametricity* [Wadler 1989], for System $\mathrm{F}_\omega^{\mathrm{ha}}$.

Let $M : \mathrm{Jdg}\ \mathrm{F}_\omega^{\mathrm{ha}} \to \mathscr{C}$ be any model of $\mathrm{F}_\omega^{\mathrm{ha}}$ in a small LCCC $\mathscr{C}$. We can interpret StcTT in the Artin gluing $\mathscr{G}_\mathscr{C}$ of $\mathrm{Pr}\,\mathscr{C}$ and Set along the global section functor $\mathrm{Hom}_{\mathrm{Pr}\,\mathscr{C}}(1, -) : \mathrm{Pr}\,\mathscr{C} \to \mathrm{Set}$, with the

object-space model M of StcTT interpreted as the given functor $M : \textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega} \to \mathscr{C}$ composed with Yoneda embedding $\mathrm{Y} : \mathscr{C} \to \textsc{Pr}\,\mathscr{C}$.

We have a functor $\overline{M^*} : \textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega} \to \mathscr{G}_{\mathscr{C}}$ by instantiating the fundamental lemma (4.11) with P_{can} as in (19). For every $A : 1 \to el\ ty \in \textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$, we let P_A be $\overline{M^*}\ (tm\ A) \in \mathscr{G}$ viewed as a predicate (in the ambient meta-theory) on the set $\mathscr{C}(1, M\ (tm\ A))$. Similarly, for every $K : 1 \to ki \in \textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$, we let P_K be $\overline{M^*}\ (el\ K) \in \mathscr{G}$ viewed as a family of sets indexed by the set $\mathscr{C}(1, M\ (el\ K))$.

Theorem D.1 (Unary Parametricity). *For every $A : 1 \to el\ ty$ and $t : 1 \to tm\ A$ in $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$, $P_A(M\ t)$ holds. Moreover, for every $K : 1 \to ki$ and $t : 1 \to el\ K$, there is an element $t^* \in P_K(M\ t)$.*

Proof. Given $t : 1 \to tm\ A \in \textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$, it is mapped by the logical predicate model $\overline{M^*}$ to a morphism $1 \to M^*(tm\ A)$ in $\mathscr{G}_{\mathscr{C}}$, which amounts to a commutative square:

$$\begin{array}{ccc} \{*\} & \xrightarrow{t^*} & \{t : 1 \to M(tm\ A) \mid P_A(t)\} \\ \downarrow{\scriptstyle !} & & \downarrow{\scriptstyle \subseteq} \\ \{*\} & \xrightarrow[\lambda *.\ Mt]{} & \mathrm{Hom}_{\textsc{Pr}\,\mathscr{C}}(1, M(tm\ A)) \end{array}$$

The commutativity of the square means that Mt satisfies P_A.

The statement for $t : 1 \to el\ K$ is essentially the same, with the element $t^* \in P_K(t)$ given by the top arrow of the diagram. □

Example D.2. Parametricity are useful for deriving 'free theorems' of programming languages [Wadler 1989]. As a 'hello world'-application, we can use parametricity to deduce that for every closed $\mathrm{F}^{\mathrm{ha}}_{\omega}$ term $t : tm\ (\bar{\forall}\ ty\ (\lambda\alpha.\ \alpha \Rightarrow_t \alpha))$, t applied to every closed type A and closed term $a : tm\ A$ is equal a.

First of all, internal to StcTT, we prove the following statement:

$$\begin{aligned} &lem : (t^* : tm^*\ (\bar{\forall}^*\ ty^*\ (\lambda\alpha.\ \alpha \Rightarrow^*_t \alpha))) \\ &\quad \to (A : \{\mathfrak{ob}\} \to el\ ty) \to (a : \{\mathfrak{ob}\} \to tm\ A) \\ &\quad \to \bullet(\{\mathfrak{ob}\} \to t^*\ A\ a = a) \\ &lem\ t^*\ A\ a\ =\ ?0 \end{aligned}$$

Recall that $prf\ t^*$ is the proof that the object-space component of t^* satisfies its logical predicate. Expanding definitions (18, C.4, 21), we have

$$prf\ t^* : \forall(\alpha^* : el^*\ ty^*).\ pre\ (\alpha^* \Rightarrow^*_t \alpha^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ (t^*\ \alpha^*)).$$

To use $prf\ t^*$, we define a predicate $A^* : \{el^*\ ty^* \mid \mathfrak{ob} \hookrightarrow A\}$ by

$$A^* := [\mathfrak{ob} \hookrightarrow A \mid \lambda x.\ \bullet(\{\mathfrak{ob}\} \to x = a)]$$

for which only the element $a : \mathfrak{ob} \to tm\ A$ is satisfied. Now we have

$$prf\ t^*\ A^* : pre\ (A^* \Rightarrow^*_t A^*)\ (\lambda\{\_ : \mathfrak{ob}\}.\ (t^*\ A)).$$

Expanding the definition of $\Rightarrow^*_t$ from (20), we have

$$prf\ t^*\ A^* : \forall(x : \{\mathfrak{ob}\} \to A).\ \bullet(\{\mathfrak{ob}\} \to x = a) \to \bullet(\{\mathfrak{ob}\} \to t^*\ A\ x = a).$$

The element a is always equal to itself, so we can complete the hole:

$$?0\ = prf\ t^*\ A^*\ a\ (\eta^{\bullet}\ refl_a).$$

Now we interpret *lem* in the glued topos $\textsc{Gl}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$. Evaluating the interpretation of *lem* at t, A, and a, we get a global section of the interpretation of $\bullet(t\ A\ a = a)$, which implies $t\ A\ a$ and a are equal morphisms $1 \to tm\ A$ in $\textsc{Jdg}\,\mathrm{F}^{\mathrm{ha}}_{\omega}$.

It is also possible to extend the unary parametricity result above to the binary (or n-ary) case. Following Sterling and Harper [2021], given two models $M_L : \textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}} \to \mathscr{C}$ and $M_R : \textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}} \to \mathscr{D}$, we consider the Artin gluing $\mathscr{G}_{\mathscr{C}\mathscr{D}}$ of the product category $\textsc{Pr}\,\mathscr{C} \times \textsc{Pr}\,\mathscr{D}$ and the category of sets along the functor

$$\langle A, B\rangle \mapsto \mathscr{C}(1, A) \times \mathscr{D}(1, B).$$

The category $\mathscr{G}_{\mathscr{C}\mathscr{D}}$ is equivalent to the presheaf topos over $(\mathscr{C} + \mathscr{D})_\top$, and every object in the category $\mathscr{G}_{\mathscr{C}\mathscr{D}}$ is a tuple

$$\langle A \in \textsc{Pr}\,\mathscr{C},\ B \in \textsc{Pr}\,\mathscr{D},\ P \in \textsc{Set},\ l : P \to \textsc{Hom}(1, A),\ r : P \to \textsc{Hom}(1, B)\rangle,$$

i.e. a proof-relevant binary relation (also known as a span) over global elements of the presheaves A and B. The category $\mathscr{G}_{\mathscr{C}\mathscr{D}}$ has two subterminal objects

$$\mathfrak{ob}_L \coloneqq \langle 1_{\textsc{Pr}\,\mathscr{C}}, 0, \emptyset, !, !\rangle \qquad \text{and} \qquad \mathfrak{ob}_R \coloneqq \langle 0, 1_{\textsc{Pr}\,\mathscr{D}}, \emptyset, !, !\rangle,$$

which determine two open subtoposes that are equivalent to $\textsc{Pr}\,\mathscr{C}$ and $\textsc{Pr}\,\mathscr{D}$ respectively. The disjunction of $\mathfrak{ob}_L$ and $\mathfrak{ob}_R$ is another subterminal object

$$\mathfrak{ob} \coloneqq \langle 1_{\textsc{Pr}\,\mathscr{C}}, 1_{\textsc{Pr}\,\mathscr{D}}, \emptyset, !, !\rangle,$$

whose corresponding open subtopos is equivalent to $\textsc{Pr}\,(\mathscr{C} + \mathscr{D})$.

The type theory StcTT can be interpreted in $\mathscr{G}_{\mathscr{C}\mathscr{D}}$ as usual, with $\mathfrak{ob} : \Omega$ interpreted as the subterminal object $\mathfrak{ob}$ above. Moreover, we can extend StcTT with the following new constants with the evident interpretation in $\mathscr{G}_{\mathscr{C}\mathscr{D}}$:

$$\mathfrak{ob}_L : \Omega \qquad \mathfrak{ob}_R : \Omega \qquad \_ : \mathfrak{ob}_L \vee \mathfrak{ob}_R = \mathfrak{ob} \qquad \_ : \mathfrak{ob}_L \wedge \mathfrak{ob}_R = 0$$

$$M_L : \{\mathfrak{ob}_L\} \to [\![\mathrm{F}_\omega^{\mathrm{ha}}]\!]_{U_0} \qquad M_R : \{\mathfrak{ob}_R\} \to [\![\mathrm{F}_\omega^{\mathrm{ha}}]\!]_{U_0}$$

$$\_ : M = \lambda\{z : \mathfrak{ob}\}.\ \textsc{case}\ z\ \textsc{of}\ \{\mathrm{inl}\ (\_ : \mathfrak{ob}_L) \mapsto M_L;\ \mathrm{inr}\ (\_ : \mathfrak{ob}_R) \mapsto M_R\}$$

We refer to the extended language by 2-TTstc.

The synthetic fundamental lemma (4.11) holds in 2-TTstc without needing any modification, since 2-TTstc only adds new axioms to StcTT. However, in 2-TTstc an $\mathfrak{ob}$-partial element $\{\mathfrak{ob}\} \to A$ of some type A is now equal to an element of type $\{\mathfrak{ob}_L \vee \mathfrak{ob}_R\} \to A$, which are equivalently two partial elements $\{\mathfrak{ob}_L\} \to A$ and $\{\mathfrak{ob}_R\} \to A$. Therefore the unary logical predicates in the proof of Lemma 4.11 can be now read as binary logical relations.

Specially, we can set both M_L and M_R to be $\mathrm{Id} : \textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}} \to \textsc{Jdg}\,\mathrm{F}_\omega^{\mathrm{ha}}$, and we obtain the binary version of parametricity of closed $\mathrm{F}_\omega^{\mathrm{ha}}$-terms (Theorem D.1) by instantiating the fundamental lemma with the logical relation P for *bool* to be equality (this relation cannot be internally defined in 2-TTstc though, since this relation only makes sense when $M_L = M_R$).

E The Realizability Model for General Recursion

Simply typed λ-calculi with general recursion famously can be modelled by variations of *complete partial orders* from classical domain theory [Plotkin 1977; Scott 1993; Streicher 2006]. However, the language $\mathrm{rF}_\omega^{\mathrm{ha}}$ has impredicative polymorphism, which is very tricky to model using classical domain theory, although not impossible [Coquand et al. 1989; Crole 1994].

Alongside a few other reasons, the difficulty of modelling polymorphism in classical domain theory motivated the development of *synthetic domain theory* (SDT) [Hyland 1991; Phoa 1991; Rosolini 1986]. The idea of SDT is to axiomatise 'domains', in the general sense of objects that provide meaning to (recursive) programs, as special 'sets' satisfying certain properties in the logic of toposes or constructive set theory, so that every function between those special sets is automatically a 'continuous map' between domains. In this way, one can give denotational semantics to recursive programs in a naive set-theoretic way.

The exact axiomatisation of SDT varies across authors, but there are mainly two kinds of models: realizability toposes [Longley and Simpson 1997; Phoa 1991] and Grothendieck toposes [Fiore and Plotkin 1997; Fiore and Rosolini 1997]. Since we are already modelling $\mathrm{F}_\omega^{\mathrm{ha}}$ using a realizability model in Section 3, we will stick with the realizability model, following the ideas of SDT concretely in this model (as opposed to using SDT as an axiomatic language).

The rest of this section is a short introduction to SDT based on Longley and Simpson's [1997] approach using *well complete objects*, adapted to a type-theoretic language. See also Longley's [1995] thesis, the more general treatment by Simpson [2004, 1999], and the type-theoretic formalisation of SDT using *well complete* Σ*-spaces* by Reus [1996, 1999] and Reus and Streicher [1999]. With the machinery of SDT in this section, the interpretation of $\mathrm{rF}_\omega^{\mathrm{ha}}$ will be almost trivial and will be presented in the next section.

Before going into SDT, let us quickly recall a typical setup of interpreting recursion in classical domain theory, which we are going to mirror in the SDT.

A *predomain* (or precisely, an ω-cpo, in this setup) is a partially ordered set $\langle A, \sqsubseteq \rangle$ that has suprema $\sqcup_i a_i$ for all ω-chains $a_0 \sqsubseteq a_1 \sqsubseteq a_2 \sqsubseteq a_3 \sqsubseteq \cdots$ in A; a predomain need not have a bottom element. Morphisms between predomains are monotone functions preserving those suprema of ω-chains.

A *(Scott-) open set* of a predomain A is a subset $O \subseteq A$ that is (1) upward closed: for all $x, y \in A$, if $x \in O$ and $x \sqsubseteq y$ then $y \in O$, and (2) continuous: for all ω-chains a_i in A, $\sqcup_i a_i \in O$ iff there exists some n such that $a_n \in O$. Open sets of a predomain A are in bijection with morphisms $A \to \mathfrak{S}$, where $\mathfrak{S}$ is the two-element predomain $\{\bot \sqsubseteq \top\}$, sometimes called the *Sierpiński space* ($\mathfrak{S}$ is the Fraktur letter for S). Namely, every open set $O \subseteq A$ corresponds to the morphism $\chi : A \to \mathfrak{S}$ where $\chi(a) = \top$ if $a \in O$ and $\chi(a) = \bot$ if $a \notin O$.

The *lifting monad* LA on predomains adjoins a new *bottom element* $\bot$ to A, with a monad structure similar to that of the monad $1 + -$ on sets. Kleisli morphisms of predomains $f : \Gamma \to LA$ are in bijection with *partial* morphisms $\langle O, \bar{f} \rangle : \Gamma \rightharpoonup A$, each consisting of an open set $O \subseteq \Gamma$ and a (total) morphism $\bar{f} : O \to A$.

A *domain* D is a predomain with bottom element $\bot_D$, which is the same as an Eilenberg-Moore algebra of the lifting monad L. Every endo-morphism $f : D \to D$ on domains then has a least fixed point by taking the supremum of the chain $\bot_D \sqsubseteq f(\bot_D) \sqsubseteq f(f(\bot_D)) \sqsubseteq \cdots$ in D.

Contexts Γ and types σ of a call-by-value programming language with recursion are then interpreted as predomains $[\![\Gamma]\!], [\![\sigma]\!]$. Terms $\Gamma \vdash t : \sigma$ are interpreted as morphisms $[\![\Gamma]\!] \to L[\![\sigma]\!]$, i.e. partial morphisms between predomains.

Recall that the internal language of assemblies $\textsc{Asm}(\mathbb{A})$ over a partial combinator algebra $\mathbb{A}$, which we used to construct a model of $\mathrm{F}_\omega^{\mathrm{ha}}$ in Section 3.2, is an extensional MLTT with three cumulative universes $P : V_1 : V_2$ such that

- each closed under the unit type, Σ, Π, and inductive types (W-types);
- for all types A and $a, b : A$, the equality type $a = b$ is in the universe P;
- for all types A and P-valued type families $B : A \to P$, $\Pi\ A\ B$ is in P.

The interpretation of P is the assembly of modest sets (i.e. PERs), and V_i is the assembly of U_i-small assemblies, for universe of sets U_i in the meta-theory. Details of the interpretation can be found in Reus's thesis [Reus 1996, §8].

In the following, we will further fix the PCA $\mathbb{A}$ to be *Kleene's first algebra* $\mathbb{K}$ [van Oosten 2008], whose elements are natural numbers (which intuitively play dual roles as both *data* and *computation* via Gödel codes of Turing machines), and partial application $n\ m$ is defined to be $\phi_n(m)$, the possibly divergent result of running the Turing machine coded by n with input m. We will write $n\ m \uparrow$ and $n\ m \downarrow$ to mean that the partial application diverges and converges respectively.

Specialising $\mathbb{A}$ to $\mathbb{K}$ is only for providing more intuition, and interested readers can consult Longley and Simpson [1997] to see how it can be done more generally with an arbitrary PCA equipped with a notion of *divergence*.

A type A is said to be a *proposition* if the type $\mathit{is\text{-}prop}\ A := (a, b : A) \to x = y$ is inhabited [Univalent Foundations Program 2013]. The subuniverse $P_{-1} \subseteq P$ of *propositional modest sets* is then defined by

$$\begin{aligned} &P_{-1} : V_1 \\ &P_{-1} = \Sigma(A : P).\ \mathit{is\text{-}prop}\ A \end{aligned}$$

whose elements are decoded as types by first projection π_1, which we will left as implicit, as if P_{-1} is a Russell-style universe.

It might be useful to see an external description for the universe P_{-1}, in the sense of universes in categories. The semantics of the universe (P_{-1}, π_1) is (isomorphic to) an assembly morphism $i : \tilde{P}_{-1} \to P_{-1}$, where P_{-1} has an underlying set containing all *sub-singleton modest sets* A, and $r \Vdash_{P_{-1}} A$ holds for all r and A. The assembly $\tilde{P}_{-1}$ has an underlying set containing all modest sets A with exactly one element $a \in |A|$, and $r \Vdash_{\tilde{P}_{-1}} A$ if and only if $r \Vdash_A a$. The morphism $i : \tilde{P}_{-1} \to P_{-1}$ is the inclusion morphism.

From the above explicit description, we can see that the universe P_{-1} satisfies Voevodsky's *propositional resizing axiom*: in the language of $\textsc{Asm}(\mathbb{K})$, for every propositional type A, there is some $\lceil A \rceil : P_{-1}$ isomorphic to A, since a sub-singleton assembly A is necessarily a modest set.

The universe P_{-1} is similar to the universe Ω of propositions in elementary toposes as axiomatised in 4.4, and we can define logical connectives $\top$, $\bot$, $\wedge$, $\vee$, $\forall$, and $\exists$ on P_{-1} in exactly the same way as we do in elementary toposes.

The crucial difference between P_{-1} and the universe Ω in elementary toposes is that P_{-1} is *not* univalent: given $p, q : P_{-1}$ with $p \cong q$, it is not always the case that $p = q$. Indeed, two singleton modest sets $\langle \{*\}, \Vdash_p \rangle$ and $\langle \{*\}, \Vdash_q \rangle$ can have different realizing relations even when their underlying sets are exactly the same.

This seemingly insignificant flaw of P_{-1} has an impact bigger than one may expect on doing mathematics internal to $\textsc{Asm}(\mathbb{K})$; For one thing, we cannot construct *quotient types* using P_{-1} in the way how it is usually done in elementary toposes.

We can switch to the realizability topos to use the better-behaved universe Ω, but we will stay in category of assemblies, as it turns out to be good enough for carrying out our development, and more importantly, the simplicity of $\textsc{Asm}(\mathbb{K})$ allows us to give simple external descriptions of many constructions of SDT, which I found essential when learning SDT for the first time.

The universe P_{-1} has a subuniverse of *semi-decidable* propositions:

$$P^s_{-1} := \{p : P_{-1} \mid \exists f : \mathbb{N} \to 2.\ p \cong (\exists n : \mathbb{N}.\ f\ n = 0)\}. \tag{25}$$

Roughly speaking, a proposition in P^s_{-1} is determined by the (semi-decidable) property of the existence of zero points for a computable function $f : \mathbb{N} \to 2$.

A simple external description of P^s_{-1} is available: the assembly P^s_{-1} is isomorphic, *up to bi-implication* in P_{-1}, to the assembly $\mathfrak{S} := \langle \{\bot, \top\}, \Vdash_{\mathfrak{S}} \rangle$ where

$$r \Vdash_{\mathfrak{S}} \bot \quad \text{iff} \quad r\ 0 \uparrow \qquad\qquad \text{and} \qquad\qquad r \Vdash_{\mathfrak{S}} \top \quad \text{iff} \quad r\ 0 \downarrow .$$

In sketch, the direction $\mathfrak{S} \to P^s_{-1}$ sends $\bot$ and $\top$ to the empty and terminal assemblies respectively, and is realized by the Turing machine accepting r and returning the computable function $f : \mathbb{N} \to 2$ that accepts n and returns 0 if and only if running the Turing machine r halts in n steps. The other direction $P^s_{-1} \to \mathfrak{S}$ sends sends an assembly to $\top$ iff it is non-empty, and this is realized by the Turing machine accepting the code for $f : \mathbb{N} \to 2$ (and $p \cong \exists n.\ f\ n = 0$) and returns the machine r that searches for a zero point of f iteratively.

The type $\mathfrak{S}$ can be viewed as a universe directly: every element $p : \mathfrak{S}$ is decoded as the equality type $p = \top$. Although P^s_{-1} and $\mathfrak{S}$ are equivalent universes, we will prefer using the universe $\mathfrak{S}$ over P^s_{-1} because $\mathfrak{S}$ is univalent:

$$(p, q : \mathfrak{S}) \to (p \cong q) \to (p = q).$$

The universe $\mathfrak{S}$ is closed under truth $\top : \mathfrak{S}$ and dependent conjunction $\Sigma(p : \mathfrak{S}).\ q(p) : \mathfrak{S}$ for all $p : \mathfrak{S}$ and $q : p \to \mathfrak{S}$. Therefore, it is a *dominance* [Rosolini 1986], which is the fundamental notion in general SDT [Hyland 1991]. In the present situation, the dominance $\mathfrak{S}$ is moreover closed under falsity $\bot$ and countable disjunction $\exists n : \mathbb{N}.\ p(n)$ for $p : \mathbb{N} \to \mathfrak{S}$.

As suggested by the notation, the universe $\mathfrak{S}$ of semi-decidable propositions will play the role of the Sierpiński space $\{\bot \sqsubseteq \top\}$ in classical domain theory. In the internal language, a (Scott-) open of a type A is defined as a function $O : A \to \mathfrak{S}$, giving rise to a subtype $\{a : A \mid O\ a = \top\}$, which we shall usually just write as O when no confusion. An ($\mathfrak{S}$-) partial function $\Gamma \rightharpoonup A$ is again an open set O of Γ with a function $O \to A$. Externally, an open set of an assembly $\langle |A|, \Vdash_A \rangle$ is a subset $O \subseteq A$ such that there is a Turing machine r satisfying that whenever $n \Vdash_A a$, then $r\ n \downarrow$ iff $a \in O$.

Analogous to the lifting monad in classical domain theory, we have a lifting monad

$$\begin{aligned} &L : P \to P \\ &L\ A = \Sigma(p : \mathfrak{S}).\ (\{p\} \to A) \end{aligned}$$

on modest sets in the internal language of $\textsc{Asm}(\mathbb{K})$. We can actually define L on all types but we shall only need it on P. The monad structure for L is

$$\begin{aligned} &\eta : A \to L\ A && \mu : (A \to L\ B) \to L\ B \\ &\eta\ a = (\top, a) && \mu\ (p, a)\ k = (\Sigma(\_ : p).\ \pi_1\ (k\ a),\ \pi_2\ (k\ a)) \end{aligned}$$

An isomorphic external description of the monad L is that it sends every modest set $\langle |A|,\ \Vdash_A \rangle$ to the modest set $\langle 1 + |A|,\ \Vdash_{LA} \rangle$ where

$$\begin{aligned} r \Vdash_{LA} \text{inl}\ * \quad &\text{iff} \quad r\ 0 \uparrow, \\ r \Vdash_{LA} \text{inr}\ a \quad &\text{iff} \quad r\ 0 \downarrow \ \wedge\ r\ 0 \Vdash_A a. \end{aligned}$$

That is to say, if a Turing machine r diverges on the input 0 then it realizes the 'bottom element' inl $*$, and otherwise r realizes inr a, for elements $a \in |A|$ that are realized by $r\ 0$. The input 0 here is completely arbitrary, and the definition will be isomorphic if 0 is replaced by any other fixed number or r itself. The monad structure on L is the same as the one on $1 + - : \textsc{Set} \to \textsc{Set}$; see Longley and Simpson [1997, §4] for details.

Note that LA is not the same as the coproduct $1 + A$ in $\textsc{Asm}(\mathbb{K})$. The latter has the same underlying set $1 + |A|$, but the existence predicate of $1 + A$ is

$$\begin{aligned} r \Vdash_{1+A} \text{inl}\ * \quad &\text{iff} \quad \pi_1\ r = 0 \\ r \Vdash_{1+A} \text{inr}\ a \quad &\text{iff} \quad \pi_1\ r \neq 0 \wedge \pi_2\ r \Vdash_A a, \end{aligned}$$

where π_1, π_2, and $\langle -, - \rangle$ are some Turing machines implementing projections and pairing of natural numbers $\mathbb{N} \times \mathbb{N} \cong \mathbb{N}$. The crucial difference between LA and $1 + A$ is that morphisms of assemblies $f : X \to 1 + A$ must be realised by Turing machines that can *decide* whether $f(x)$ is inr a given a realizer of x, while morphisms $f : X \to LA$ need only be realised by Turing machines that *semi-decide* whether $f(x)$ is inr a. Thus LA is the right choice of the lifting monad capturing the idea of 'possibly divergent elements of A'.

As an *endofunctor* on the universe P, L has both a final coalgebra $\langle \bar{\omega} : P, \sigma : \bar{\omega} \to L\bar{\omega} \rangle$ and an initial algebra $\langle \omega : P, \tau : L\omega \to \omega \rangle$. The following formulae of $\bar{\omega}$ and ω are due to Jibladze [1997]:

$$\bar{\omega} = \{f : \mathbb{N} \to \mathfrak{S} \mid \forall n.\ f\ (n + 1) \to f\ n\}$$

$$\omega = \{f : \bar{\omega} \mid \forall p : P_{-1}.\ \big(\forall (n : \mathbb{N}).\ (f\ n \to p) \to p\big) \to p\}$$

which in fact works for any dominance in any elementary topos with a natural number object (with P_{-1} in the formula of ω replaced by Ω).

Again, we have simple external descriptions for $\bar{\omega}$ and ω in the case of $\textsc{Asm}(\mathbb{K})$. The carrier $\bar{\omega}$ is isomorphic to the assembly $\langle \mathbb{N} \cup \{\infty\}, \Vdash_{\bar{\omega}} \rangle$ where

$$\begin{aligned} r \Vdash_{\bar{\omega}} n \quad &\text{iff} \quad \forall k \in \mathbb{N}.\ (k < n) \leftrightarrow (r\ k\downarrow) \\ r \Vdash_{\bar{\omega}} \infty \quad &\text{iff} \quad \forall k \in \mathbb{N}.\ r\ k\downarrow \end{aligned}$$

with $\sigma : \bar{\omega} \to L\bar{\omega}$ given by $\sigma(0) = \text{inl}\ *$, $\sigma(n+1) = \text{inr}\ n$, and $\sigma(\infty) = \text{inr}\ \infty$. The type ω is given by the assembly $\langle \mathbb{N}, \Vdash_{\omega} \rangle$ that restricts $\bar{\omega}$ to the sub-underlying set $\mathbb{N}$. The algebra $\tau : L\omega \to \omega$ is simply $\tau(\text{inl}\ *) = 0$ and $\tau(\text{inr}\ n) = n + 1$.

From the explicit description we see that the assembly ω is a non-standard representation of natural numbers as an assembly, different from the standard representation $\langle \mathbb{N}, \{(n, n) \mid n \in \mathbb{N}\} \rangle$ that satisfies the universal property of a natural number object in $\textsc{Asm}(\mathbb{K})$. In ω, every natural number $n \in \mathbb{N}$ is represented as a Turing machine that halts exactly for inputs k smaller than n. Since Turing machines are unable to tell whether other machine halts, assembly maps $\omega \to A$ are constrained to be 'continuous' in a sense.

Let $\kappa : \omega \to \bar{\omega}$ be the canonical inclusion morphism (given as the unique algebra homomorphism from the initial algebra $\tau : L\omega \to \omega$ to the L-algebra $\sigma^{-1} : L\bar{\omega} \to \bar{\omega}$). The morphism κ plays an important role in synthetic domain theory: a morphism $c : \omega \to X$ of assemblies will play the role of an ω-chain of elements $c_0 \sqsubseteq c_1 \sqsubseteq \cdots$ in a partial order X. Similarly, a morphism $c^* : \bar{\omega} \to X$ is analogous to a chain c_i together with its supremum $\sqcup_i c_i$.

Definition E.1. A modest set $X : P$ is called *complete* if the function

$$(- \cdot \kappa) : (\bar{\omega} \to X) \to (\omega \to X)$$

is an isomorphism, i.e. the following proposition holds

$$\mathit{complete}\ X := \exists \overline{(-)} : (\omega \to X) \to (\bar{\omega} \to X).\ (\forall c.\ \bar{c} \cdot \kappa = c) \wedge (\forall d.\ \overline{d \cdot \kappa} = d)$$

internally in $\textsc{Asm}(\mathbb{K})$. A modest set $X : P$ is called *well complete* if LX is complete.

Well complete types will be our synthetic version of predomains:

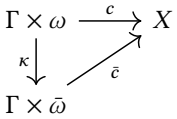
$$\begin{aligned} &\mathit{PDom} : V_1 \\ &\mathit{PDom} = \Sigma(A : P).\ \mathit{complete}\ (L\ A) \end{aligned}$$

They are intuitively modest sets in which an ω-chain of partial elements has a unique (partial) supremum. Their crucial difference from predomains in classical domain theory is that they are just 'sets' satisfying a property, rather than sets carrying additional data (the partial order).

There are some nuances in the external meaning of (well) completeness. Firstly, we notice that $\mathit{complete} : P \to P_{-1}$ in Definition E.1 is a proper *realizability predicate* in the sense that $\mathit{complete}\ X$ for a modest set $X : P$ has non-trivial realizers. Namely, $\mathit{complete}\ X$ is realized by machines sending realizers of $\omega \to X$ to realizers of $\bar{\omega} \to X$, in a way that is an inverse to $- \cdot \kappa$. Secondly, $\mathit{complete}\ X : P_{-1}$ makes sense in an arbitrary context Γ in the internal language of $\textsc{Asm}(\mathbb{K})$. Therefore, externally X is not one modest set but a family of modest sets $\Gamma \to P$ indexed by an assembly Γ of the context.

If we forget about realizers of completeness and consider only *global* elements $X : P$, then $\mathit{complete}\ X$ has a realizer if and only if the morphism $X^{\kappa} : X^{\bar{\omega}} \to X^{\omega}$ in $\textsc{Asm}(\mathbb{K})$ is an isomorphism. The latter condition is precisely the definition of completeness for an object in $\textsc{Asm}(\mathbb{A})$ by Longley

and Simpson [1997]. This further means that for all assemblies Γ and $c : \Gamma \times \omega \to X$, there is a unique $\bar{c} : \Gamma \times \bar{\omega} \to X$ making the following diagram commute:



To see this, by Yoneda embedding, X is complete if and only if

$$(X^{\kappa} \cdot -) : \textsc{Hom}(\Gamma, X^{\bar{\omega}}) \to \textsc{Hom}(\Gamma, X^{\omega})$$

is an isomorphism, natural in Γ. By adjointness, this is equivalent to asking

$$(- \cdot \Gamma \times \kappa) : \textsc{Hom}(\Gamma \times \bar{\omega}, X) \to \textsc{Hom}(\Gamma \times \omega, X)$$

to be a natural isomorphism. A natural transformation is a natural isomorphism iff every component of it is an isomorphism, so X is complete if and only if for every $c : \Gamma \times \omega \to X$, there is a unique $\bar{c} : \Gamma \times \bar{\omega} \to X$ such that $\bar{c} \cdot \Gamma \times \kappa = c$.

Theorem E.2. *The subuniverse $PDom \subseteq P$ is closed under liftings L, the unit type, Σ-types, Π-types, equality types, coproducts, the natural number type $\mathbb{N}$ in P. Moreover, predomains are also complete (i.e. well completeness implies completeness).*

Proof. This is essentially shown by Longley and Simpson [1997, §7] aside from the difference between Longley and Simpson's external definition of completeness and our internal definition. Longley and Simpson defined completeness of an assembly X as a proposition in the ambient logic ($X^{\kappa} : X^{\bar{\omega}} \to X^{\omega}$ being an isomorphism), while our definition is in internal in the language of $\textsc{Asm}(\mathbb{K})$, which has non-trivial realizers (Turing machines accepting code of $c : \omega \to X$ and outputting code of $\bar{\omega} \to X$). Therefore, we have to check that the proofs of the closure properties by Longley and Simpson are realizable. For example, to show $L : P \to P$ restricts to $L : PDom \to PDom$, we need to check that there is a Turing machine sending realizers of X being well complete to realizers of LX being well complete. This is indeed the case by observing that the proofs by Longley and Simpson can be carried internally in the language of $\textsc{Asm}(\mathbb{K})$. □

Definition E.3. Mirroring the setup of classical domain theory, the universe of *domains* is defined as the type of Eilenberg-Moore algebras of the monad $L : PDom \to PDom$:

$$\begin{aligned}
&\textsc{record}\ Dom : V_1\ \textsc{where}\\
&\quad A : PDom\\
&\quad \alpha : L\ A \to A\\
&\quad _ : (x : A) \to \alpha\ (\eta^L\ x) = x\\
&\quad _ : (x : L\ (L\ A)) \to \alpha\ (\mu^L\ x) = \alpha\ (L\ \alpha\ x)
\end{aligned}$$

As usual, given $D : Dom$, we usually write the type $\pi_1\ (D.A)$ as just D.

The crucial property of domains $D : Dom$ is that they admit fixed points for all endofunctions:

$$fix : \{D : Dom\} \to (f : D \to D) \to D$$

which is defined as follows: first we define a function $\alpha_f : L\ D \to D$ by $\alpha_f = \alpha \cdot Lf$. By the initiality of $\tau : L\omega \to \omega$, we have a homomorphism $c : \omega \to D$. Then using the completeness of D, we have $\bar{c} : \bar{\omega} \to D$, and we let $fix\ f := \bar{c}\ \infty$. It is then the case that $f\ (fix\ f) = fix\ f$ [Reus and Streicher 1999, Theorem 7.3].

We note that by Longley and Simpson [1997, Theorem 5.6], Eilenberg-Moore L-algebras on a predomain are unique if exist, so it makes sense to say 'a predomain is a domain' as a proposition.

E.1 The Interpretation of $\mathrm{rF}_\omega^{\mathrm{ha}}$

Now we are ready to construct a (V_2-small) model of the signature $\mathrm{rF}_\omega^{\mathrm{ha}}$ (Section 2.4) in $\textsc{Asm}(\mathbb{K})$. Our goal is to define an element M of the record type $[\![\mathrm{rF}_\omega^{\mathrm{ha}}]\!]_{V_2}$ containing all the declarations of $\mathrm{rF}_\omega^{\mathrm{ha}}$ with $\mathbb{J}$ replaced by the universe V_2.

The non-recursive fragment of $\mathrm{rF}_\omega^{\mathrm{ha}}$ will be interpreted in almost the same way as $\mathrm{F}_\omega^{\mathrm{ha}}$ in Section 3.2, except that all the occurrences of the universe $P : V_1$ will be replaced by the subuniverse $PDom : V_1$. For example, $M.ty$ is now $PDom$ instead of P, and the computation judgement $M.co$ is now

$$\begin{aligned}&M.co : M.RawHFunctor \to M.el\ M.ty \to PDom\\ &M.co\ H\ A = (T : M.MonadAlg\ H) \to (B : PDom) \to (A \to T\ B) \to T\ B\end{aligned}$$

The constructions in Section 3.2 still work because by Theorem E.2, the universe $PDom$ is closed under the type formers that we used to interpret $\mathrm{F}_\omega^{\mathrm{ha}}$, in particular, impredicative Π-types.

The empty type from ($\mathrm{rF}_\omega^{\mathrm{ha}}$-4) is as expected interpreted as the empty modest set 0, which is trivially well complete.

The interesting thing is modelling partial computations pco (($\mathrm{rF}_\omega^{\mathrm{ha}}$-1)). In Section 2.4, we had a type $MonadAlgRec$ of monads supporting recursion (and some effectful operations). However, we cannot simply define $M.pco$ by replacing $MonadAlg$ in the definition of $M.co$ above with $MonadAlgRec$, because the definition of $MonadAlgRec$ depends on $M.pth$ and thus $M.pco$.

The type $MonadAlgRec$ ensures that a monad T in $\mathrm{rF}_\omega^{\mathrm{ha}}$ supports recursion by requiring the monad T to be partial thunks *syntactically*. What we need here is a semantic counterpart of monad supporting recursion:

$$\begin{aligned}&\textsc{record}\ MonadAlgL\ (H : RawHFunctor) : V_2\ \textsc{where}\\ &\quad \textsc{include}\ MonadAlg\ H\ \textsc{as}\ T\\ &\quad dom : (A : PDom) \to \{D : Dom \mid D.A = T\ A\}\end{aligned}$$

which requires that $T\ A : PDom$ is a domain for all $A : PDom$.

The model of partial computations is then

$$\begin{aligned}&M.pco : M.RawHFunctor \to M.el\ M.ty \to PDom\\ &M.pco\ H\ A = (T : MonadAlgL\ H) \to (B : PDom) \to (A \to T\ B) \to T\ B\end{aligned}$$

The models of the declarations *val*, *let-in*, *pth* and *op* are the same as those for *co* in Section 3, which we shall not repeat here.

The model for the fixed-point combinator has type

$$M.Y : \{H, A\} \to (M.pth\ H\ A \to M.pco\ H\ A) \to M.pco\ H\ A$$

In the present model, $pth\ H\ A$ is simply equal to $pco\ H\ A$, so by appendix E, it is sufficient to show that $M.pco\ H\ A$ is a domain. We define the algebra by

$$\begin{aligned}&\alpha : \{H, A\} \to L\ (M.pco\ H\ A) \to M.pco\ H\ A\\ &\alpha\ (p, c) = \lambda T\ B\ k.\ \beta_{TB}\ (p, c\ T\ B\ k)\end{aligned}$$

where $\beta_{TB} : L\ (T\ B) \to T\ B$ is $\beta_{TB} \coloneqq (T.dom\ B).\alpha$. The L-algebra α is a product of a family of L-algebras, so it is easy to check that α satisfies the laws:

$$\begin{aligned}&\alpha\ (\top,\ c)\\ &= \lambda T\ B\ k.\ \beta_{TB}\ (\top,\ c\ T\ b\ k)\\ &= \quad \{\beta_{TB} \text{ is an Eilenberg-Moore algebra}\}\\ &\quad \lambda T\ B\ k.\ c\ T\ b\ k\\ &= c\end{aligned}$$

and similarly for all $(p, (q, c)) : L\ (L\ (M.pco\ H\ A))$,

$$\begin{aligned}
&\alpha\ (L\ \alpha\ (p, (q, c))) \\
=\ &\alpha\ (p, \lambda T\ B\ k.\ \beta_{TB}\ (q, c\ T\ B\ k)) \\
=\ &\lambda T\ B\ k.\ \beta_{TB}\ (p, \beta_{TB}\ (q, c\ T\ B\ k)) \\
=\ &\quad \{\beta_{TB} \text{ is an Eilenberg-Moore algebra}\} \\
&\lambda T\ B\ k.\ \beta_{TB}\ (\mu^L\ (p, (q, c\ T\ B\ k))) \\
=\ &\lambda T\ B\ k.\ \beta_{TB}\ (\Sigma(_ : p).\ q,\ c\ T\ B\ k) \\
=\ &\alpha\ (\mu^L\ (p, (q, c)))
\end{aligned}$$

We have shown that $pco\ H\ A$ is a domain, so we can use *fix* (appendix E) to define

$$M.Y\ f = fix\ f.$$

Finally, we need to give an interpretation of *eval* ($\mathrm{rF}_\omega^{\mathrm{ha}}$-3):

$$\begin{aligned}
M.eval : \{H\} \to (T : M.MonadAlgRec\ H) \to (A : M.el\ ty) \\
\to M.pco\ H\ A \to M.tm\ (T\ A)
\end{aligned}$$

Note that the type of T is *MonadAlgRec* rather than *MonadAlgL*. By the definition of *MonadAlgRec* in Section 2.4, there exists some $F : M.el\ M.ty \to M.el\ M.ty$ such that the monad T maps every $A : PDom$ to $M.pth\ H\ (F\ A)$. By the discussion above in Appendix E.1, $M.pth\ H\ (F\ A)$, which is just $M.pco\ H\ (F\ A)$, is always a domain. Therefore we have a conversion function

$$\sigma : (T : M.MonadAlgRec\ H) \to \{T' : MonadAlgL\ H \mid T.T = T'.T\},$$

and we define the model of *eval* to be

$$M.eval\ T\ A\ c = c\ (\sigma\ T)\ A\ T.ret$$

This completes the definition of the model $M : [\![\mathrm{rF}_\omega^{\mathrm{ha}}]\!]_{V_2}$.

The realizability model M of $\mathrm{rF}_\omega^{\mathrm{ha}}$ gives a way to compute recursive programs written in $\mathrm{rF}_\omega^{\mathrm{ha}}$ by program extraction. Since $L\ A$ is a domain, the monad $L : PDom \to PDom$ can be extended to

$$L' : \{L' : M.MonadAlgL\ VoidH \mid L'.T = L\}$$

Therefore we have a function

$$\begin{aligned}
&toL : \{A : PDom\} \to M.pco\ VoidH\ A \to L\ A \\
&toL\ c = c\ L'\ A\ \eta^L
\end{aligned}$$

Every closed program $p : pco\ VoidH\ bool$ is interpreted as a global element of $M.pco\ VoidH\ 2$ in $\textsc{Asm}(\mathbb{K})$. Composing it with *toL*, we then have a global element of the modest set $L\ 2$. The realizer of this element is then a possibly divergent Turing machine r that yields a Boolean value if it halts.

We conjecture that the realizability model of $\mathrm{rF}_\omega^{\mathrm{ha}}$ in this section is *adequate*.

Conjecture E.4. *For all closed program $c : pco\ VoidH\ bool$ in rF_ω^{ha}, if the morphism $toL \cdot [\![c]\!] : 1 \to L\ 2$ in $\textsc{Asm}(\mathbb{K})$ is $\mathrm{inr}\ tt$ (or $\mathrm{inr}\ ff$), then $c = val\ tt$ (or $c = val\ ff$) in the theory of rF_ω^{ha}.*

This implies that if $toL \cdot [\![c]\!] = \mathrm{inl}\ *$, then c does not equal to $val\ tt$ or $val\ ff$, otherwise $toL \cdot [\![c]\!]$ would not be $\mathrm{inl}\ *$.

We expect adequacy can be proved using synthetic Tait computability (STC) *internally* in the effective topos Eff, in which we glue the (internal) category P with the category of P-valued presheaves over the category of judgements of $\mathrm{rF}_\omega^{\mathrm{ha}}$ (constructed internally in Eff). Such an internal STC argument has been attempted by Sterling and Harper [2022] to prove adequacy for a language with *security levels* and *general recursion* but without impredicative polymorphism, whose

denotational semantics is a sheaf-model of SDT, although there is a currently unfixed problem in their proof [Sterling 2023].