Algebraic Effects Meet Hoare Logic in Cubical Agda

DONNACHA OISÍN KIDNEY, Imperial College London, United Kingdom
ZHIXUAN YANG, Imperial College London, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom

This paper presents a novel formalisation of algebraic effects with equations in Cubical Agda. Unlike previous work in the literature that employed *setoids* to deal with equations, the library presented here uses quotient types to faithfully encode the type of terms quotiented by laws. Apart from tools for equational reasoning, the library also provides an *effect-generic Hoare logic* for algebraic effects, which enables reasoning about effectful programs in terms of their pre- and post- conditions. A particularly novel aspect is that equational reasoning and Hoare-style reasoning are related by an elimination principle of Hoare logic.

CCS Concepts: • **Theory of computation** → **Hoare logic**; **Equational logic and rewriting**; **Pre- and post-conditions**; **Program verification**; **Control primitives**.

Additional Key Words and Phrases: algebraic effects, Cubical Agda, Hoare logic, program verification

ACM Reference Format:
Donnacha Oisín Kidney, Zhixuan Yang, and Nicolas Wu. 2024. Algebraic Effects Meet Hoare Logic in Cubical Agda. *Proc. ACM Program. Lang.* 1, POPL (January 2024), 33 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Programs in practice usually produce *computational effects*, such as I/O, mutable memory, nondeterminism, and probabilistic choice. Pioneered by Moggi [1989, 1991], such computational effects can be categorically represented as *monads*. This view is further refined by Plotkin and Power [2002] who pointed out that almost all the monads modelling computational effects can be presented as *algebraic theories* specifying the primitive operations of an effect and its equational laws.

Based on the algebraic view, Plotkin and Pretnar [2009, 2013] proposed a programming language feature, called an *effect handler*, which allows the programmer to introduce and eliminate effects conveniently in a program. Due to their flexibility, effect handlers have been quickly adopted by the programming language community, and implemented in many programming languages, e.g. Brachthäuser et al. [2018]; Ghica et al. [2022]; Leijen [2014, 2017]; Sivaramakrishnan et al. [2021].

Besides these implementations, there are formalisations of algebraic theories [Abel 2021; Gunther et al. 2018] and computational effects [Li and Weirich 2022; Xia et al. 2020; Yoon et al. 2022], which facilitate verification of effectful programs. Notwithstanding, the existing formalisations in the literature fail to provide complete satisfaction for program verification for two reasons:

Setoids. In these formalisations, programs modulo equational laws do not form types but rather *setoids*, which are commonly deemed tedious to work with.

Authors' addresses: Donnacha Oisín Kidney, Imperial College London, London, United Kingdom, o.kidney21@imperial.ac.uk; Zhixuan Yang, Imperial College London, London, United Kingdom, s.yang20@imperial.ac.uk; Nicolas Wu, Imperial College London, London, United Kingdom, n.wu@imperial.ac.uk.

2475-1421/2024/1-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
data Op where          Arity : Op → Type
  `fail : Op           Arity `fail = ⊥
  `⟨|⟩  : Op           Arity `⟨|⟩  = Bool

ℕ𝔻 = Op ◁ Arity
```

```
data Op S where        Arity : Op S → Type
  `get : Op S          Arity `get      = S
  `put : S → Op S      Arity (`put _) = ⊤

𝕊_S = Op S ◁ Arity
```

Fig. 1. Signature of nondeterminism (left) and mutable state (right). The operator ◁ is given in (3).

Hoare Logic. In these formalisations, reasoning about effectful programs in the *equational* style [Gibbons and Hinze 2011; Plotkin and Pretnar 2008] is well supported, but reasoning in the style of Hoare logic [Hoare 1969] does not have good support.

The main contribution of this paper is a new formalisation of algebraic effects which addresses these two shortcomings as follows:

Quotients. Our formalisation of algebraic effects and theories includes equations, but avoids setoids and instead uses *quotients*. This is facilitated by the recent development of Homotopy Type Theory [Univalent Foundations Program 2013] and its computational realisation—Cubical Type Theory [Angiuli et al. 2021; Cohen et al. 2018], which supports quotient types natively. The particular implementation we use is Cubical Agda [Vezzosi et al. 2019].

Hoare Logic. The library contains an *effect-generic Hoare logic*, that is inspired by Schröder and Mossakowski [2003] and Goncharov and Schröder [2013]. The idea is that pre- and post-conditions ϕ and ψ of a Hoare triple $\{\,\phi\,\}\, p\, \{\,\psi\,\}$ are encoded as effectful programs that return a (not necessarily decidable) *proposition*, and the validity of a Hoare triple is encoded as an equality of programs:

$$\begin{aligned} &(\mathrm{do}\ a \leftarrow \phi;\ x \leftarrow p;\ b \leftarrow \psi\ x;\ \mathit{return}\ (x\ ,\ a \mathrel{|\!\!\rightarrow\!\!|} b)) \\ \equiv\ &(\mathrm{do}\ a \leftarrow \phi;\ x \leftarrow p;\ b \leftarrow \psi\ x;\ \mathit{return}\ (x\ ,\ \mathit{True})) \end{aligned}$$

This encoding is explained in full in Section 6. The usual inference rules of Hoare logic are derivable based on this encoding. Moreover, we prove a novel elimination principle for Hoare logic that connects equational reasoning and Hoare-style reasoning pleasantly: whenever the return value of a program p is shown to satisfy a predicate ϕ using Hoare logic, and $\phi\ x$ implies that $f\ x \equiv g\ x$, then $(\mathrm{do}\ x \leftarrow p; f\ x) \equiv (\mathrm{do}\ x \leftarrow p; g\ x)$. This principle is simple to state, but surprisingly difficult to prove constructively. Its proof is a significant novel contribution of this paper, presented in Section 7.

A Quick Taste. To be concrete, we sketch here an example of formalising and verifying a stateful and nondeterministic parser in our framework, which will also be our running example in later sections. The parser will be built in the style of monadic parser combinators [Hutton and Meijer 1998]: the stream of tokens is kept in a mutable state that the parser can inspect and consume, and the parser can make nondeterministic choices to parse the expression in possibly different ways.

To formalise the parser, the first step is to define the algebraic theory of its computational effects. Instead of defining a monolithic theory of all the effectful operations that the parser needs, an appealing feature of algebraic effects is that algebraic theories can be defined modularly and then combined into bigger ones. Wu et al. [2014] showed that parsers can be described by the combination of two separate algebraic theories: mutable state and nondeterminism.

Each algebraic theory consists of (i) a *signature* specifying the name of the operations and their arities and (ii) a collection of *equational laws*. To formalise a signature in our framework, we need to define a type, typically called *Op*, whose elements correspond to the operations, as well as a function, typically called *Arity*, which maps elements of *Op* to a *type*. The signatures of nondeterminism and state are shown in Figure 1. Nondeterminism, $\mathbb{ND}$, has two operations: $\langle|\rangle$ for choices and *fail* for failures. State, $\mathbb{S}_S$, is parameterised by the type of the mutable state S, and it has operations *get* and *put* for reading and writing that state respectively.

To complete the definition of the theory of nondeterminism and state, we also need to specify their equational laws. Our formalisation provides combinators for specifying equations conveniently. For example, associativity of nondeterministic choice is programmed as

$$assoc = \forall^{\mathrm{n}}\ \lambda\ x\ y\ z \longrightarrow (x \langle|\rangle y) \langle|\rangle z \doteq x \langle|\rangle (y \langle|\rangle z)$$

Additionally, the theory of nondeterministic choice has equations stipulating that $\langle|\rangle$ is idempotent and commutative, and that *fail* is an identity element of $\langle|\rangle$. The theory of state has equations (12) stipulating the interaction of reading and writing the state, for example:

$$put\text{-}put\ s = (\mathrm{do}\ put\ s;\ get) \doteq (\mathrm{do}\ put\ s;\ return\ s)$$

With theories $\mathbb{S}$ and $\mathbb{ND}$ in place, we can combine them into the algebraic theory $\mathbb{P}$ for parsing. The combination that is desirable for parsing is *tensoring* [Hyland et al. 2006], $\mathbb{P} = \mathbb{ND} \otimes \mathbb{S}_{String}$, which takes the disjoint union of the operations and equations of $\mathbb{S}$ and $\mathbb{ND}$ and adds new equations saying that every operation in $\mathbb{S}$ commutes with every operation in $\mathbb{ND}$.

The algebraic theory $\mathbb{P}$ then automatically defines a monad *Term* $\mathbb{P}$ of $\mathbb{P}$-computations. The monad is equipped with the operations from $\mathbb{P}$ and satisfies all the equational laws. With this monad, we can write programs with nondeterminism and state.

Now suppose that we are interested in parsing expressions built from a binary operator $\star$ and two constants $\diamond$ and $\spadesuit$. A parser is shown in Figure 2, in which the program *any-char* consumes an arbitrary character from the state, and the program *char c* consumes a character c from the state, and if the next character is not c, the operation *fail* is called. The result of *parse-tree n* is to return a binary tree in the monad *Term*; it simply uses nondeterministic choices to try different ways of parsing the input, and the argument n is fuel which limits the depth of recursion.

To demonstrate the correctness of the parser, we would like to show that for any $t : Tree$, if t is pushed to the input stream, the parser always parses the tree t back:

$$(\mathrm{do}\ push\ (print\ t);\ parse\text{-}tree\ n) \equiv return\ t \tag{1}$$

Here, *push* prepends a string to the input stream ($push\ s = \mathrm{do}\ r \leftarrow get; put\ (s +\!\!+\ r)$). Note that this equality is stronger than merely proving the input-output behaviour of the parser using e.g. Hoare logic, since this equality ensures that each side of the equation can be substituted by the other side in any context without changing the semantics, which can be used in program transformation.

Although it is possible to prove this solely using the equational laws of $\mathbb{P}$, such a proof is miserably tedious because of the many bureaucratic equational rewriting steps needed in the proof. For example, when showing that the first character c printed to the input is consumed by a corresponding *any-char c* by the parser, we will have to invoke the commutativity laws of $\mathbb{P}$ to swap *any-char c* with all operations in between:

$$(\mathrm{do}\ push\text{-}char\ c; \ldots;\ any\text{-}char\ c) = \cdots = (\mathrm{do}\ push\text{-}char\ c;\ any\text{-}char\ c; \ldots)$$

so that *push-char* and *any-char* can cancel out using equations of $\mathbb{P}$. In this example, the operations getting in the way between *push-char* and *any-char* are for printing the rest of the tree t, so we further need an induction on the tree t to do all the swapping. Through such attempts, it does not take too long to realise purely equational reasoning is too low-level for verifying programs with algebraic effects, and higher-level reasoning techniques are needed.

```
any-char = do c :: cs ← get                          char c₁ = do c₂ ← any-char
                 where [] ⟶ fail                               if (does (c₁ ≟ c₂))
              put cs                                              then return ⟨⟩
              return c                                            else fail

data Tree : Type where                parse-tree (suc n) =   (do char '◇'; return ◇)
  ◇ ♠  : Tree                                             ⟨|⟩ (do char '♠'; return ♠)
  _⊛_ : Tree ⟶ Tree ⟶ Tree                                ⟨|⟩ (do char '('; l ← parse-tree n
                                                                   char '*'; r ← parse-tree n
 parse-tree : ℕ ⟶ Term Tree                                        char ')'; return (l ⊛ r))
 parse-tree zero = fail
```

Fig. 2. A simple parser of expressions using nondeterminism and state.

The solution offered by our framework is an *effect-generic Hoare-logic*: for every algebraic theory, there is automatically a notion of Hoare triples satisfying the usual inference rules of Hoare logic. In the parser example, we can prove the following Hoare triple (explained in full later (27)):

$$\forall r \longrightarrow \{\mathit{remaining}\ (\mathit{print}\ t \mathbin{+\!\!+} r)\}\ t' \leftarrow \mathit{parse\text{-}tree}\ n\ \{\mathit{return}\ (t' \equiv t) \land \mathit{remaining}\ r\} \tag{2}$$

where *remaining* s is a predicate saying that the remaining input stream is equal to s.

Hoare triples and program equivalences are related by an "elimination principle" (Section 7) for Hoare triples: let $\phi : A \longrightarrow \Omega$ be a predicate on a type A, and let f and g be two functions $A \longrightarrow \mathit{Term}\, B$ such that for all $x : A$, $\phi\ x$ implies $f\ x \equiv g\ x$, then the following holds:

$$\mathit{H\text{-}elim} : \{\}\ x \leftarrow p\ \{\ \mathit{return}\ (\phi\ x)\ \} \longrightarrow (p \gg\!= f) \equiv (p \gg\!= g)$$

Using *H-elim* one can derive the desired equality (1) from the Hoare triple (2) in a few easy steps. The constructive proof of this elimination principle, one of the major technical contributions of this paper, is highly non-trivial and requires that the arities of the operations of the algebraic theory be *finite*, so the strings in the parser example in fact have to be bounded by a finite length. However, there is also a non-constructive proof of *H-elim* without this restriction.

We would like to emphasise that the role of Hoare logic here is not the verification goal per se, but a high-level proof technique for showing the desired equivalence (1). Such way of using Hoare-style logic has been proposed for specific effects and Hoare-style logics before [Song et al. 2023; Turon et al. 2013], but to our knowledge our framework is the first one to implement this technique for generic algebraic effects in a constructive setting.

Paper Organisation. The contribution of this paper is a library for formalising and verifying effectful programs based on algebraic effects. The library consists of the following components:

- algebraic theories and their free algebras,
- common computational effects and their combinations,
- an equational logic and Hoare logic for reasoning about effectful programs,

which amount to about 9000 lines of code. It is implemented in Cubical Agda 2.6.2.

The rest of this paper presents the main constructions of our formalisation. We will assume basic familiarity with Agda. No prior knowledge about algebraic effects is assumed, so this paper may be read as a tutorial for algebraic effects. The structure of our exposition on algebraic effects in Sections 2–5 is influenced by Bauer [2019]. The material is organised as follows:

- Section 2 defines *signatures*, *algebras*, and *free algebras*.

- Section 3 defines *equations* over algebraic signatures and *algebraic theories*.
- Section 4 defines an *equational logic* for reasoning about terms of an algebraic theory, and uses quotient types to construct term algebras of algebraic theories.
- Section 5 goes into some detail of the parser example that we have seen above to demonstrate how to program with algebraic effects in our formalisation.
- Section 6 shows how Hoare logic can be defined for algebraic effects in a generic way.
- Section 7 shows an elimination principle of Hoare logic which relates equational reasoning and Hoare-style reasoning.
- Section 8 discusses related work and we conclude in Section 9.

2 ALGEBRAIC SIGNATURES AND ALGEBRAS

To warm up, we begin with formalising algebraic theories *without* any equational laws, including their *signatures*, which specify the arities of operations of an algebraic theory; *algebras* of a signature, which are types equipped with operations specified by the signature; and *free algebras*, which are syntactic terms built from variables and operations. These concepts are standard ones from *universal algebra* [Birkhoff 1935; Cohn 1981], except that our formalisation is type-theoretic rather than set-theoretic, and that we will mainly use examples from computational effects.

Foundation. Our formalisation is based on intensional Martin-Löf type theory with universes and inductive types [Martin-Löf 1982], supplemented with *function extensionality* and *effective set quotients*. We use Cubical Agda [Vezzosi et al. 2019] to carry out our formalisation, but we will not rely on features of Cubical Type Theory apart from those mentioned above.

Using Cubical Agda gives our formalisation computational meaning—the formalised proofs are runnable programs, but in principle our formalisation also has semantics in any category that have the aforementioned constructs, in particular, all topoi. For example, by interpreting our formalisation in the presheaf topos over a cartesian closed category $\mathbb{C}$, the formalisation can be read as a formalisation of $\mathbb{C}$-*enriched algebraic theories* [Kelly and Power 1993], but we will not make use of this additional semantic generality in this paper.

2.1 Algebraic Signatures

An *algebraic signature*, or simply a *signature*, is a type *Op* of operation symbols paired with a type family *Arity* giving the arity of each operation:

```
record Signature : Type₁ where
  constructor _◁_
  field Op    : Type
        Arity : Op → Type                                        (3)
```

In the literature, signatures are sometimes also called *containers* [Abbott et al. 2005].

Examples. In our running example of parsers, the *nondeterminism* effect facilitates the expression of computations that can return multiple results. We have seen its signature $\mathbb{ND}$ in Figure 1. The \`*fail* operator represents a program with no results, and \`⟨|⟩ is a binary operator combining the results from two sub-computations. Instead of a natural number, here the *Arity* function returns a type, where the cardinality of *Arity o* corresponds to the arity of the operation *o*. The arity of a nullary operator like \`*fail* is ⊥, the empty type, and a binary operator like \`⟨|⟩ has arity *Bool*. In general, a natural-number arity *n* is represented by *Fin n*:

$$Fin : \mathbb{N} \to Type \qquad Fin\ zero = \bot \qquad Fin\ (suc\ n) = \top \uplus Fin\ n$$

where ⊤ is the singleton type, and ⊎ is disjoint union. As a result *Fin n* has exactly *n* closed elements.

The generalisation of arity from numbers to types allows for infinitary arities. For instance, given any type S, the theory of *mutable S-state* has two operations: $`get$ for reading the state and $`put$ for writing the state. Its signature $\mathbb{S}_S$ is in Figure 1. The arity of $`get$ is type $S$, which may have infinitely many elements. Conceptually, the operation $`get$ takes S-many computations as arguments, and results in a computation that reads the state and then continues as the argument corresponding to the current state. In contrast, $`put\ s$ for each $s : S$ is a unary operation, whose only argument stands for the way to continue after writing s into the state.

However, while it is possible to express infinite arities syntactically, to define the free model over a theory these arities need to preserve quotients (Definition 4.2), a condition we will explain in more detail in Section 4.4. All finite types preserve quotients, but there is strong evidence [Coquand et al. 2017] that infinitary types do not, at least in Cubical Agda.

2.2 Algebras of Signatures

Where a signature $\mathbb{F}$ describes an abstract collection of operations, an $\mathbb{F}$-*algebra* captures the notion of a type supporting or "implementing" those operations. $\mathbb{F}$-algebras are defined in terms of the functor $[\![\ \mathbb{F}\ ]\!] : Type \to Type$, which corresponds to one level of application of an operation in $\mathbb{F}$:

$$[\![\_]\!] : Signature \to Type \to Type \qquad [\![\ Op \lhd Arity\ ]\!]\ X = \Sigma[\ o : Op\ ] \times (Arity\ o \to X)$$

An $\mathbb{F}$-algebra on a type C, referred to as the *carrier* of the algebra, is a function $[\![\ \mathbb{F}\ ]\!]\ C \to C$:

$$\_\text{-Alg}[\_] : Signature \to Type \to Type \qquad \mathbb{F}\text{ -Alg}[\ C\ ] = [\![\ \mathbb{F}\ ]\!]\ C \to C$$

Here $[\![\_]\!]$ and _-Alg[_] are syntactically *mixfix* operators, where the underscores mark the places for arguments. We will use mixfix operators frequently in this paper to aid readability.

Agda uses hierarchical universes to handle large sets, so $Type : Type_1$, $Type_1 : Type_2$, and so on. Agda also supports universe polymorphism, so functions can be generic over what universe level they work with. The technical intricacies of this system are not relevant to understanding our contributions, so we have simplified the presentation of some types, like $[\![\ \mathbb{F}\ ]\!]$, by giving a universe *monomorphic* version in the text, where there is a polymorphic version in the formalisation.

Examples. Lists of A-elements for an arbitrary type A (in this paper, variables A, B, C, always mean arbitrary types) implement nondeterminism by the following algebra:

$$\begin{aligned} &list\text{-}int : \mathbb{ND}\text{ -Alg}[\ List\ A\ ] \\ &list\text{-}int\ (`\langle|\rangle\ , k) = k\ false +\!\!+\ k\ true \\ &list\text{-}int\ (`fail\ , k) = [\,] \end{aligned} \tag{4}$$

The algebra maps each operation in nondeterminism to an operation on lists: the $\langle|\rangle$ operation goes to concatenation of lists, and *fail* constructs an empty list. A signature can have multiple algebras, even with the same carrier. The following are two ways that booleans implement nondeterminism:

$$\begin{aligned} &all\text{-}int : \mathbb{ND}\text{ -Alg}[\ Bool\ ] && any\text{-}int : \mathbb{ND}\text{ -Alg}[\ Bool\ ] \\ &all\text{-}int\ (`fail\ , k) = true && any\text{-}int\ (`fail\ , k) = false \\ &all\text{-}int\ (`\langle|\rangle\ , k) = k\ false\ \&\&\ k\ true && any\text{-}int\ (`\langle|\rangle\ , k) = k\ false\ \|\ k\ true \end{aligned} \tag{5}$$

As another example, the signature $\mathbb{S}$ has an algebra over functions $S \to A \times S$:

$$\begin{aligned} &state\text{-}alg : \mathbb{S}_S\text{ -Alg}[\ (S \to A \times S)\ ] \\ &state\text{-}alg\ (`get\ , k)\ s_1 = k\ s_1\ s_1 \\ &state\text{-}alg\ (`put\ s_2\ , k)\ s_1 = k\ \langle\rangle\ s_2 \end{aligned} \tag{6}$$

This interprets mutable state into the *state monad* [Moggi 1991], a function $S \to A \times S$ taking an initial state as input and produces a value of type A along with the final state. In this algebra, the argument $k : S \longrightarrow (S \longrightarrow A \times S)$ to the operation *get* is thought of as the continuation after reading the state. Therefore the result of *get* continues with s_1 and keeps the state unchanged, where s_1 is the initial state. Similarly, the result of *put* s_2 on $k : \top \longrightarrow (S \longrightarrow A \times S)$ is the computation that ignores its initial state and continues as $k\ \langle\rangle$ with the new state s_2.

2.3 Syntax Trees

Let $\mathbb{F}$ be a signature and v be any type. Syntax trees of terms built out of variables from v and operations from $\mathbb{F}$ are represented by the following *Syntax* type:

$$\begin{aligned}
&\text{data } \mathit{Syntax}\ (\mathbb{F} : \mathit{Signature})\ (v : \mathit{Type}) : \mathit{Type} \text{ where}\\
&\quad \mathit{var} : v \longrightarrow \mathit{Syntax}\ \mathbb{F}\ v\\
&\quad \mathit{op} : [\![\ \mathbb{F}\ ]\!]\ (\mathit{Syntax}\ \mathbb{F}\ v) \longrightarrow \mathit{Syntax}\ \mathbb{F}\ v
\end{aligned} \tag{7}$$

The constructor *var* introduces a variable, and *op* builds a term by applying an operation from $\mathbb{F}$ to some existing terms. The parameter $\mathbb{F}$ will often be omitted: when $\mathbb{F}$ is clear from the context, we will write *Syntax A* rather than *Syntax* $\mathbb{F}$ *A*.

Examples. Consider the following pair of programs:

$$\begin{aligned}
&\mathit{up\text{-}to} : \mathbb{N} \longrightarrow \mathit{Syntax}\ \mathbb{ND}\ \mathbb{N} && \mathit{odds} : \mathbb{N} \longrightarrow \mathit{Syntax}\ \mathbb{ND}\ \mathbb{N}\\
&\mathit{up\text{-}to}\ \mathit{zero} \quad = \mathit{fail} && \mathit{odds}\ n = \text{do } m \leftarrow \mathit{up\text{-}to}\ n\\
&\mathit{up\text{-}to}\ (\mathit{suc}\ n) = \mathit{up\text{-}to}\ n \mathbin{\langle|\rangle} \mathit{return}\ n && \qquad\qquad \mathit{guard}\ (\mathit{odd}\ m)\\
& && \qquad\qquad \mathit{return}\ m
\end{aligned} \tag{8}$$

These programs use the nondeterminism effect. *up-to n* computes all the natural numbers smaller than *n*, and *odds n* computes the *odd* numbers smaller than *n*. The *guard* function takes a condition and causes the computation to fail when the condition is false.

Let us build the value of type *Syntax* $\mathbb{ND}$ $\mathbb{N}$ that represents the syntax tree for the *odds* program. First, the do-notation in (8) can be desugared in the usual way:

$$\mathit{odds}\ n = \mathit{up\text{-}to}\ n \gg\!= \lambda\ m \longrightarrow \mathit{guard}\ (\mathit{odd}\ m) \gg\!= \lambda\ \_ \longrightarrow \mathit{return}\ m$$

The do-bindings have been replaced by the monadic bind, which is defined as follows on *Syntax*:

$$\begin{aligned}
&\_{\gg\!=}\_ : \mathit{Syntax}\ \mathbb{ND}\ A \longrightarrow (A \longrightarrow \mathit{Syntax}\ \mathbb{ND}\ B) \longrightarrow \mathit{Syntax}\ \mathbb{ND}\ B\\
&\mathit{var}\ x \qquad \gg\!= \rho = \rho\ x\\
&\mathit{op}\ (o\ ,\ k) \gg\!= \rho = \mathit{op}\ (o\ ,\ \lambda\ i \longrightarrow k\ i \gg\!= \rho)
\end{aligned}$$

This performs variable substitution. The term $xs \gg\!= \rho$ takes a syntax tree *xs* and returns the tree with all variables replaced according to the continuation ρ.

Next, the operators from the signature must be defined as functions on *Syntax*.

$$\begin{aligned}
&\_\langle|\rangle\_ : \mathit{Syntax}\ \mathbb{ND}\ A \longrightarrow && \mathit{fail} : \mathit{Syntax}\ \mathbb{ND}\ A\\
&\qquad \mathit{Syntax}\ \mathbb{ND}\ A \longrightarrow && \mathit{fail} = \mathit{op}\ (\grave{}\mathit{fail}\ ,\ \mathit{absurd})\\
&\qquad \mathit{Syntax}\ \mathbb{ND}\ A && \\
&x \mathbin{\langle|\rangle} y = \mathit{op}\ (\grave{}\langle|\rangle\ ,\ \lambda\ i \longrightarrow \mathit{if}\ \ i\ \mathit{then}\ y\ \mathit{else}\ x) && \mathit{absurd} : \{A : \mathit{Type}\} \longrightarrow \bot \longrightarrow A
\end{aligned}$$

The definition for $\langle|\rangle$ here enshrines *false* as corresponding to the left branch of $\langle|\rangle$.

Finally, the remaining symbols in the program must be defined in terms of *Syntax*.

$$\begin{aligned}
&\mathit{return} : A \longrightarrow \mathit{Syntax}\ \mathbb{ND}\ A && \mathit{guard} : \mathit{Bool} \longrightarrow \mathit{Syntax}\ \mathbb{ND}\ \top\\
&\mathit{return} = \mathit{var} && \mathit{guard}\ c = \mathit{if}\ \ c\ \mathit{then}\ \mathit{return}\ \langle\rangle\ \mathit{else}\ \mathit{fail}
\end{aligned}$$

2.4 Interpreting the Syntax Tree

Syntax trees of a signature $\mathbb{F}$ can be interpreted into an $\mathbb{F}$-algebra:

$$\begin{aligned}
&\mathit{interp} : \mathbb{F}\text{-Alg}[\ C\ ] \to (v \to C) \to \mathit{Syntax}\ v \to C \\
&\mathit{interp}\ \phi\ \rho\ (\mathit{var}\ x) \qquad\quad = \rho\ x \\
&\mathit{interp}\ \phi\ \rho\ (\mathit{op}\ (O_i\ ,\ k)) = \phi\ (O_i\ ,\ \lambda\ i \to \mathit{interp}\ \phi\ \rho\ (k\ i))
\end{aligned} \tag{9}$$

This function takes an algebra $\phi : \mathbb{F}\text{-Alg}$ and a mapping ρ from the variables v to the algebra C, and interprets syntactic programs with operations of the algebra accordingly.

A more computational reading of this function is that the argument of type *Syntax v* is a program invoking operations from $\mathbb{F}$ and returning a value of type v. The algebra ϕ *handles* the operation calls in the program. The function ρ turns the v-value returned by the program into the type ϕ operates on. Therefore the function *interp* is called a (deep) *effect handler* [Plotkin and Pretnar 2013; Pretnar and Bauer 2014] when it is implemented as a programming language construct.

For an example of interpretation, the syntactic program *odds* of signature $\mathbb{ND}$ can be interpreted using the algebra over lists: *interp list-int* ($\lambda\ v \to [\ v\ ]$) (*odds* 7) computes to [1, 3, 5]. What makes effect handlers a powerful language feature is that different interpretations can be given to the same syntactic program: *interp all-int is-odd* (*odds* 7) computes to *true*.

Another example of an interpretation is the *monadic bind* on *Syntax v*:

$$\begin{aligned}
&\_{\gg\!=}\_ : \mathit{Syntax}\ A \to (A \to \mathit{Syntax}\ B) \to \mathit{Syntax}\ B \\
&\mathit{xs} \gg\!=\ \rho = \mathit{interp}\ \mathit{op}\ \rho\ \mathit{xs}
\end{aligned}$$

This follows from the fact that the *Syntax* type is *itself* an algebra of $\mathbb{F}$, via *op*:

$$\begin{aligned}
&\mathit{syntax\text{-}alg} : \mathbb{F}\text{-Alg}[\ \mathit{Syntax}\ v\ ] \\
&\mathit{syntax\text{-}alg} = \mathit{op}
\end{aligned}$$

There are some useful properties we can prove about interpretation. First, if the algebra supplied is *op*, and the variable assignment is *var*, the interpretation is an identity:

$$(\mathit{xs} : \mathit{Syntax}\ v) \to \mathit{interp}\ \mathit{op}\ \mathit{var}\ \mathit{xs} \equiv \mathit{xs}$$

Moreover, the interpretation supports the *fusion laws*, a powerful tool in reasoning about combinations of interpretations. One such law states that for an $\mathbb{F}$-algebra ϕ, a variable assignment ρ, a continuation k, and a syntax tree *xs*, the following holds:

$$\mathit{interp}\ \phi\ \rho\ (\mathit{xs} \gg\!=\ k) \equiv \mathit{interp}\ \phi\ (\mathit{interp}\ \phi\ \rho \circ k)\ \mathit{xs} \tag{10}$$

This says that an *interp* after a bind can be fused with the continuation supplied to the bind.

2.5 Freeness of Syntax

We saw above that the *Syntax* type is itself an algebra of the signature $\mathbb{F}$ (*syntax-alg*); as it happens it is the *free* algebra. To define freeness formally first we need to define $\mathbb{F}$-*homomorphisms*.

Homomorphisms. An $\mathbb{F}$-homomorphism between two $\mathbb{F}$-algebras is a function h between the two carriers C and $\mathcal{D}$ such that the function commutes with the operations of the signature:

$$\begin{aligned}
&\_{\to_h}\_ : \mathbb{F}\text{-Alg} \to \mathbb{F}\text{-Alg} \to \mathit{Type} \qquad && \mathit{cmap} : (A \to B) \to [\![\ \mathbb{F}\ ]\!]\ A \to [\![\ \mathbb{F}\ ]\!]\ B \\
&(\ C\ ,\ c\ ) \to_h (\ \mathcal{D}\ ,\ d\ ) = && \mathit{cmap}\ f\ (o\ ,\ k) = o\ ,\ \lambda\ i \to f\ (k\ i) \\
&\quad \Sigma[\ h : (C \to \mathcal{D})\ ] \times h \circ c \equiv d \circ \mathit{cmap}\ h &&
\end{aligned}$$

This definition of a homomorphism is the same as the usual definition of, for instance, a monoid homomorphism, stated in slightly more general language. On the signature for monoids, the above code for a homomorphism translates to a function h between two carrier sets for two monoids, along with a pair of proofs that $h\ x \cdot h\ y = h\ (x \cdot y)$ and $h\ \epsilon = \epsilon$.

Freeness. Let v be any type. A *(homotopic-) free* $\mathbb{F}$*-algebra* over v is an algebra $\alpha : \mathbb{F}$ -Alg[A] over some type A with a function $\eta : v \longrightarrow A$ such that for any algebra $\beta : \mathbb{F}$ -Alg[C] and function $\theta : v \longrightarrow C$ there is a unique $\mathbb{F}$-homomorphism $h : A \longrightarrow C$ with $h \circ \eta = \theta$:

$$\begin{tikzcd} v \arrow[r, "\eta"] \arrow[dr, "\theta"'] & A \arrow[d, "h"] \\ & C \end{tikzcd} \qquad \begin{tikzcd} \llbracket \mathbb{F} \rrbracket\ A \arrow[r, "\alpha"] \arrow[d, "cmap\ h"'] & A \arrow[d, dashed, "\exists! h"] \\ \llbracket \mathbb{F} \rrbracket\ C \arrow[r, "\beta"] & C \end{tikzcd} \tag{11}$$

This definition is a little dense: in simpler terms, it amounts to saying that if we inject some variable into the free algebra, and then use the operators of the signature on the free algebra to build a syntax tree, and then interpret that syntax tree into *another* algebra, it's the same as directly using the same operations in the latter algebra.

Theorem 2.1. *Syntax* $\mathbb{F}$ v *with var is a free* $\mathbb{F}$*-algebra over* v.

Proof Sketch. We have already seen that *Syntax* $\mathbb{F}\ v$ is an algebra of $\mathbb{F}$, via *op*. Given another algebra $\phi : \llbracket \mathbb{F} \rrbracket \longrightarrow C$, and a function $f : v \longrightarrow C$, the homomorphism is given by $h = interp\ \phi\ f$. This satisfies the cancellative property by definition: $h \circ \eta = interp\ \phi\ f \circ var = f$. Finally, given any other homomorphism $g : Syntax\ \mathbb{F} \longrightarrow C$ with this cancellative property, it can be shown to be equal to h as functions $Syntax\ \mathbb{F} \longrightarrow C$ by induction on *Syntax* $\mathbb{F}$. To show that they are equal as homomorphisms $(Syntax\ \mathbb{F}, op) \longrightarrow_h (C, \phi)$, we also need to show that the equalities witnessing their commutation with operations are equal. We have formalised this proof of homotopic freeness using a similar proof idea to Awodey et al. [2017] (which proved homotopic initiality). □

3 ALGEBRAIC THEORIES AND MODELS

Algebraic theories are a combination of operations (the signature) and equations (the laws). The previous section dealt only with the signature; this section will introduce laws to the discussion. Mirroring the development in the previous section, we will first define equations and algebraic theories, and then we will define *models*: algebras of the underlying signature of a theory that satisfy all the equations. The free models of algebraic theories, which mirror free algebras of signatures in Section 2, are slightly more involved, and will be treated in the next section.

Equations. The laws for an algebraic theory are a collection of formal *equations* between syntactic terms. Thus the type *Equation* $\mathbb{F}\ v$ is a pair of syntax trees, one for each side of the equation, in the signature $\mathbb{F}$ with free variables v:

```
record Equation 𝔽 v where
  constructor _≐_
  field lhs rhs : Syntax 𝔽 v
```

Here, for instance, is the equation for associativity of ‹|›:

```
assoc : Equation ℕ𝔻 (Fin 3)
assoc = (var 0 ‹|› var 1) ‹|› var 2 ≐ var 0 ‹|› (var 1 ‹|› var 2)
```

This equation has three variables, hence $v = Fin\ 3$. The helper function $\forall^n$ makes the statement of the equation a little more readable:

```
assoc = ∀ⁿ λ x y z ⟶ (x ‹|› y) ‹|› z ≐ x ‹|› (y ‹|› z)
```

Contexts. To state some equations succinctly it is useful to use variables drawn from a *context*: the following statement of the "put-put" law, for instance, uses the variables s_1 and s_2.

$$(\text{do } put\ s_1;\ put\ s_2) \doteq (\text{do } put\ s_2)$$

This law says that a *put* immediately following another *put* overwrites the first one. It would be possible to encode this law as $S \times S$ individual laws—one for each choice of s_1 and s_2—but it is more reasonable to define a law as having a context Γ, which can contain parameters like s_1 and s_2.

```
record Law 𝔽 where                            put-put-law : Law 𝕊_S
  field Γ : Type                             put-put-law .Γ = S × S
        v  : Γ → Type                        put-put-law .v _ = ⊤
        eqn : (γ : Γ) → Equation 𝔽 (v γ)    put-put-law .eqn (s₁ , s₂) =
                                               (do put s₁ ; put s₂) ≐ (do put s₂)
```

A *Law* 𝔽 is a law for signature 𝔽. The law is under context *Γ* and free variables *v*. The free variables can depend on the context: this is necessary, for example, to construct the tensoring law (21).

Theories. Finally, an *algebraic theory* over a signature 𝔽 is a collection of laws over a signature 𝔽:

```
record Theory 𝔽 where
  field Laws : Type
        Eqns  : Laws → Law 𝔽
```

The theory of mutable state, for instance, has three laws:

```
data Laws where put-put put-get get-put : Laws
```

The *Eqns* field then returns the corresponding law for each index:

```
Eqns : Laws → Law 𝕊_S

Eqns put-put .Γ = S × S         Eqns put-get .Γ = S         Eqns get-put .Γ = ⊤
Eqns put-put .v _ = ⊤           Eqns put-get .v _ = S       Eqns get-put .v _ = ⊤

Eqns put-put .eqn (s₁ , s₂) = (do put s₁; put s₂)  ≐ (do put s₂)
Eqns put-get  .eqn s        = (do put s; get)      ≐ (do put s; return s)          (12)
Eqns get-put  .eqn _        = (do s ← get; put s) ≐ (do return ⟨⟩)
```

The equation $(\text{do } s_1 \leftarrow get\,; s_2 \leftarrow get\,; return\,(s_1, s_2)) \doteq (\text{do } s \leftarrow get; return\,(s,\ s))$ is sometimes also required as the "get-get" law of mutable state, but in fact follows from the equations above.

Models. Given an equation $lhs \doteq rhs$ over a signature 𝔽, an 𝔽-algebra $\phi : [\![\,\mathbb{F}\,]\!]\ C \to C$ over some type C is said to *respect* this equation if the following proposition holds:

$$\phi \ \mathsf{Respects}\ (lhs \doteq rhs) = \forall\,(\rho : v \to C) \to interp\ \mathbb{F}\ \phi\ \rho\ lhs \equiv interp\ \mathbb{F}\ \phi\ \rho\ rhs$$

which says that interpreting both sides of the equation using the algebra ϕ under *any* variable assignment ρ results in the same value. Note that _Respects_ is syntactically an infix operator, so it is in the sans serif font to distinguish from an ordinary function.

Moreover, if an algebra respects all the equations in a theory, we say that it *models* the theory:

$$\phi\ \mathsf{Models}\ \mathcal{T} = \forall\ i\ \gamma \to \phi\ \mathsf{Respects}\ \mathcal{T}\ .Eqns\ i\ .eqn\ \gamma$$

A $\mathcal{T}$-model, then, where $\mathcal{T}$ is a theory with signature 𝔽, is an 𝔽-algebra over some carrier type C that respects the laws of $\mathcal{T}$:

$$\mathcal{T}\text{-}\mathsf{Model}[\ C\ ] = \Sigma[\ \phi : \mathbb{F}\text{-}\mathsf{Alg}[\ C\ ]\ ] \times \phi\ \mathsf{Models}\ \mathcal{T}$$

For example, the algebra *state-alg* (6) can be shown to be a model for the theory of mutable state, since *state-alg* respects the state laws (12).

4 FREE MODELS OF ALGEBRAIC THEORIES

The core construction of this paper is the *free model* of an algebraic theory, which is similar to the free algebras of signatures in Section 2.5 but with equations taken into consideration. Free models are important as they give us a formal construction which represents the structure present in *all models* of a theory. Thus they may be seen as encoding the structure in the theory *itself.* In particular, if an algebraic theory describes a computational effect adequately, the free models of the theory may be seen as the *computations* with the effect.

Some algebraic theories have direct descriptions of their free models; for instance, free monoids are precisely lists, and free commutative monoids are $\mathbb{N}$-valued functions with finitely many non-zero values. However, such concrete descriptions do not always exist for algebraic theories. In general, free models are constructed by *quotienting* syntax trees *Syntax* $\mathbb{F}$ (7) with a suitable equivalence relation defined by the equational laws of $\mathcal{T}$.

Section 4.1 will first quickly recap quotient types in Cubical Agda, and Section 4.2 will define the equivalence relation. Finally, the type for the free model itself, *Term*, will be presented in Section 4.3 and its properties expanded upon. Section 4.4 will show that *Term* is the free model for $\mathcal{T}$.

4.1 Quotient Types

Given a type A and a type family $\_\sim\_ : A \to A \to \mathit{Type}$, their *(set) quotient* $A / \_\sim\_$ can be expressed as a *higher inductive type* [Univalent Foundations Program 2013] in Cubical Agda:

$$\begin{array}{l}
\text{data } \_/\_ \; (A : \mathit{Type}) \; (\_\sim\_ : A \to A \to \mathit{Type}) : \mathit{Type} \text{ where} \\
\quad [\_] : A \to A \,/\, \_\sim\_ \\
\quad \mathit{eq/} : (x\ y : A) \to x \sim y \to [\,x\,] \equiv [\,y\,] \\
\quad \mathit{squash/} : \mathit{isSet}\ (A \,/\, \_\sim\_)
\end{array} \tag{13}$$

The first constructor injects every element x of type A into the quotient; the second constructor identifies $[\,x\,]$ and $[\,y\,]$ whenever $x \sim y$ is inhabited; the final constructor makes the type $A / \_\sim\_$ always a *set* in HoTT (also known as an *h-set*):

$$\mathit{isSet}\ B = (x\ y : B) \to (p\ q : x \equiv y) \to p \equiv q \tag{14}$$

An alternative way to phrase (14) is that a type B is a set if for all pairs of elements $x, y : B$, the identity types $x = y$ are *propositions*, which are types whose elements are all identified:

$$\mathit{isProp}\ P = (x\ y : P) \to x \equiv y$$

Quotient types can be consumed by pattern-matching as usual, but in practice it tends to be easier to use the *recursion principle* below, which extends a function $f : A \to B$ to the quotient type provided that B is a set and f respects the relation:

$$\begin{array}{ll}
\mathit{rec/} : \mathit{isSet}\ B & \phi : A \,/\, \_\sim\_ \to B \\
\quad \to\ (f : A \to B) & \phi\ [\,x\,] \qquad\qquad\quad = f\ x \\
\quad \to\ (\forall\ x\ y \to x \sim y \to f\ x \equiv f\ y) & \phi\ (\mathit{eq/}\ x\ y\ x{\sim}y\ i) \quad = \mathit{resp}\ x\ y\ x{\sim}y\ i \\
\quad \to\ A \,/\, \_\sim\_ \to B & \phi\ (\mathit{squash/}\ x\ y\ p\ q\ i\ j) = \\
\mathit{rec/}\ \mathit{set}\ f\ \mathit{resp} = \phi \text{ where} & \quad \mathit{set}\ (\phi\ x)\ (\phi\ y)\ (\mathit{cong}\ \phi\ p)\ (\mathit{cong}\ \phi\ q)\ i\ j
\end{array} \tag{15}$$

Example. One way to construct *unordered lists*, or *bags*, is quotienting the type *List A* of lists by adjacent-element swaps: $\mathit{Bag}\ A = \mathit{List}\ A \,/\, \_\circlearrowleft\_$ where

```
data _↺_ : List A ⟶ List A ⟶ Type where
  swap : x :: y :: ys ↺ y :: x :: ys
  cons  : ∀ x ⟶ xs ↺ ys ⟶ x :: xs ↺ x :: ys
```

The size of bags can be computed using the recursion principle:

```
size : Bag A ⟶ ℕ
size = rec/ isSetNat length ϕ where
  ϕ : (xs ys : List A) ⟶ xs ↺ ys ⟶ length xs ≡ length ys
  ϕ _ _ swap              = refl
  ϕ _ _ (cons _ xs↺ys) = cong suc (ϕ _ _ xs↺ys)
```

The function ϕ witnesses that the *length* function on lists respects element swaps *xs* ↺ *ys*, so *length* : *List A* ⟶ ℕ can be extended to bags.

Remark. In Cubical Agda, not all types are sets, i.e. satisfying the proposition (14). In particular, if we leave out *squash/* in the definition of $A / \_\sim\_$, the reflexive equality *refl* : $[a] \equiv [a]$ for some element $a : A$ and the equality constructed by *eq/ a a r* for any $r : a \sim a$ will be different elements of the equality type $[a] \equiv [a]$, even if A is a set and $\_\sim\_$ is a proposition valued type family. This is certainly very different from quotient sets in the usual set-theoretic maths, and thus we need the additional constructor *squash/* to “force” the quotient type to be a set.

4.2 Program Equivalence

Given an algebraic theory $\mathcal{T}$ over a signature $\mathbb{F}$, the free models are obtained by quotienting syntax trees of $\mathbb{F}$ with what we call the *program equivalence*, which is the *congruence relation* generated by the equational axioms of $\mathcal{T}$. It can be given as an inductive datatype in Cubical Agda:

```
data _~t_ : Syntax A ⟶ Syntax A ⟶ Type where
  eqt    : ∀ i γ ρ ⟶ let lhs ≐ rhs = Eqns i .eqn γ in lhs >>= ρ ~t rhs >>= ρ
  reflt  : x ≡ y ⟶ x ~t y
  symt   : x ~t y ⟶ y ~t x
  transt : x ~t y ⟶ y ~t z ⟶ x ~t z
  congt  : ∀ Oi (kl kr : Arity Oi ⟶ Syntax A) ⟶ (∀ i ⟶ kl i ~t kr i)
           ⟶ op (Oi , kl) ~t op (Oi , kr)
  trunct : (eq1 eq2 : x ~t y) ⟶ eq1 ≡ eq2
```

(1) The first constructor eq_t says if two syntax trees can be obtained by instantiating an equation of $\mathcal{T}$, they are equivalent. Its declaration has three parameters: i, the index of the particular law; γ, the context for that law; and ρ, the variable assignment to the variables in the law.
(2) The next three constructors $refl_t$, sym_t and $trans_t$ extend the relation to an equivalence.
(3) The $cong_t$ constructor makes the relation a *congruence* relation on *op*: an operation acting on equivalent arguments k_l and k_r leads to equivalent result.
(4) The final constructor $trunc_t$ is a higher inductive constructor. It identifies any two elements of the type $x \sim_t y$, thus squashing $x \sim_t y$ from the type of derivation trees of equivalence of x and y to a proof-irrelevant *proposition* of x being equivalent to y.

This datatype encodes precisely Birkhoff [1935]'s inference rules for the *equational logic* of universal algebra, deducing which syntactic terms are regarded as equivalent in an algebraic theory.

4.3 Terms up to Program Equivalence

With the program equivalence $x \sim_t y$ for an algebraic theory $\mathcal{T}$, we can define the type of *$\mathcal{T}$-terms up to program equivalence* as the quotient type:

$$\mathit{Term} : \mathit{Type} \to \mathit{Type} \qquad\qquad \mathit{Term}\ A = \mathit{Syntax}\ A\ /\ \_{\sim_t}\_$$

where the argument A is the type of variables in terms. For clarity, we also define a specialised constructor $[\_]_t : \mathit{Syntax}\ A \to \mathit{Term}\ A$ by $[\_]_t = [\_]$. The rest of this section will be dedicated to proving properties about this type, culminating in a proof that it is indeed the free model and a program equivalence $x \sim_t y$ holds iff the interpretations of x and y are equal in all models.

Interpretation. Like syntax trees, terms can be interpreted into models of $\mathcal{T}$:

$$\mathit{interp}_t : (\phi : \mathbb{F}\text{ -Alg[} C \text{]})\ (\rho : v \to C) \to \phi \text{ Models } \mathcal{T} \to \mathit{isSet}\ C \to \mathit{Term}\ v \to C$$

Given an $\phi : \mathbb{F}$ -Alg that models $\mathcal{T}$, a generator ρ, and a proof that the carrier is a set, this function interprets into the carrier set from *Term*.

The definition of interp_t is as follows:

$$\mathit{interp}_t\ \phi\ \rho\ \mathit{resp}\ \mathit{set} = \mathit{rec/}\ \mathit{set}\ (\mathit{interp}\ \phi\ \rho)\ (\lambda\ \_\ \_ \to \mathit{interp}_t\text{-}\mathit{cong}\ \phi\ \rho\ \mathit{resp}\ \mathit{set})$$

The definition works by extending $\mathit{interp} : \mathit{Syntax}\ A \to C$ to $\mathit{Term}\ A$ using *rec/*. Thus we need the following lemma that *interp* respects the program equivalence $\sim_t$.

Lemma 4.1. *Let $\phi : \mathbb{F}$ -Alg[C] be a model of a theory $\mathcal{T}$: Theory $\mathbb{F}\ \ell$ such that C is a set, and let ρ be a function $v \to C$ for any v. Then $\mathit{interp}\ \phi\ \rho : \mathit{Syntax}\ v \to C$ respects program equivalence $\sim_t$:*

$$\mathit{interp}_t\text{-}\mathit{cong} : \forall\ \{x\ y : \mathit{Syntax}\ A\} \to x \sim_t y \to \mathit{interp}\ \phi\ \rho\ x \equiv \mathit{interp}\ \phi\ \rho\ y$$

Proof. This proof goes by induction on $x \sim_t y$. The cases for the four constructors for congruence follow from the fact that the propositional equality is also a congruence relation:

$$\begin{aligned}
&\mathit{interp}_t\text{-}\mathit{cong}\ (\mathit{refl}_t\ p) &&= \mathit{cong}\ (\mathit{interp}\ \phi\ \rho)\ p\\
&\mathit{interp}_t\text{-}\mathit{cong}\ (\mathit{sym}_t\ p) &&= \mathit{sym}\ (\mathit{interp}_t\text{-}\mathit{cong}\ p)\\
&\mathit{interp}_t\text{-}\mathit{cong}\ (\mathit{trans}_t\ x{\sim}z\ z{\sim}y) &&= \mathit{interp}_t\text{-}\mathit{cong}\ x{\sim}z\ ⨾\ \mathit{interp}_t\text{-}\mathit{cong}\ z{\sim}y\\
&\mathit{interp}_t\text{-}\mathit{cong}\ (\mathit{cong}_t\ O_i\ k\ k'\ p) &&= \mathit{cong}\ \phi\ (\mathit{cong}\ (O_i\ ,\_)\ (\mathit{funExt}\ (\lambda\ i \to \mathit{interp}_t\text{-}\mathit{cong}\ (p\ i))))
\end{aligned}$$

A more interesting case is the constructor $\mathit{eq}_t\ i\,\gamma\ k$, which instantiates the i-th law with a variable substitution $k : v\ \gamma \to \mathit{Syntax}\ A$. We need to show

$$\mathit{interp}\ \phi\ \rho\ (\mathit{interp}\ \mathit{op}\ k\ \mathit{lhs}) \equiv \mathit{interp}\ \phi\ \rho\ (\mathit{interp}\ \mathit{op}\ k\ \mathit{rhs})$$

To show this we apply the fusion law *interp-comp* (10), converting both sides of the goal to the shape $\mathit{interp}\ \phi\ (\mathit{interp}\ \phi\ \rho \circ k)$. Then by the assumption $m : \phi$ Models $\mathcal{T}$, both sides are equal. Using some equational reasoning combinators, the proof goes by

$$\begin{aligned}
&\mathit{interp}_t\text{-}\mathit{cong}\ (\mathit{eq}_t\ i\ \gamma\ k) = \text{let } \mathit{lhs} \doteq \mathit{rhs} = \mathit{Eqns}\ i\ .\mathit{eqn}\ \gamma \text{ in}\\
&\quad \mathit{interp}\ \phi\ \rho\ (\mathit{interp}\ \mathit{op}\ k\ \mathit{lhs}) &&\equiv\langle\ \mathit{interp}\text{-}\mathit{comp}\ \phi\ \rho\ k\ \mathit{lhs}\ \rangle\\
&\quad \mathit{interp}\ \phi\ (\mathit{interp}\ \phi\ \rho \circ k)\ \mathit{lhs} &&\equiv\langle\ m\ i\ \gamma\ (\mathit{interp}\ \phi\ \rho \circ k)\ \rangle\\
&\quad \mathit{interp}\ \phi\ (\mathit{interp}\ \phi\ \rho \circ k)\ \mathit{rhs} &&\equiv\check{\langle}\ \mathit{interp}\text{-}\mathit{comp}\ \phi\ \rho\ k\ \mathit{rhs}\ \rangle\\
&\quad \mathit{interp}\ \phi\ \rho\ (\mathit{interp}\ \mathit{op}\ k\ \mathit{rhs}) &&\blacksquare
\end{aligned}$$

The final case is trunc_t. We need to show that the equality being produced is a proposition. This follows from the assumption of C being a *set*, since equalities on sets are propositions. □

4.4 Term Models

The first step in showing that *Term* is the free model of a theory $\mathcal{T}$ is showing that it is an algebra of the underlying signature $\mathbb{F}$ of $\mathcal{T}$, which amounts to constructing an element

$$op_t : \mathbb{F}\text{-Alg}[\ Term\ A\ ]$$

Perhaps surprisingly, defining this is not entirely straightforward in a constructive setting. The problem becomes clear when the type $\mathbb{F}$ -Alg[*Term A*] is expanded:

$$op_t : (\exists\ o \times (Arity\ o \longrightarrow Syntax\ A\ /\ \_{\sim_t}\_) \longrightarrow Syntax\ A\ /\ \_{\sim_t}\_) \tag{16}$$

The only relevant functions available are the following two:

$$[\_] : Syntax\ A \longrightarrow Syntax\ A\ /\ \_{\sim_t}\_ \qquad op : \exists\ o \times (Arity\ o \longrightarrow Syntax\ A) \longrightarrow Syntax\ A$$

If we were in a classical foundation, we could define op_t by *choosing* a family of representative elements *Arity o* ⟶ *Syntax A* from the input family of quotients *Arity o* ⟶ *Syntax A* / _~_, and then applying *op* and [_]. But in a constructive setting, we cannot make such a choice for all *Arity o* in general. Therefore we need to make an additional assumption on the arities' types. In particular, we need arities to support a *choice* principle: we will call this principle *quotient preserving*.

Definition 4.2 (Quotient Preserving). From a high level, the missing piece in implementing op_t is a way to commute a function arrow *around* a quotient, turning a function into a quotient into a quotiented function. $((A \longrightarrow (B\ /\ R)) \longrightarrow (A \longrightarrow B)\ /\ R')$. First, notice that the inverse always exists:

$$\begin{aligned}&dist : (A \longrightarrow B)\ /\ Pointwise\ R \longrightarrow (A \longrightarrow B\ /\ R)\\ &dist = rec/\ (isSet\Pi\ \lambda\ \_ \longrightarrow squash/)\ (\lambda\ f \longrightarrow [\_] \circ f)\ (\lambda\ \_\ \_ \longrightarrow funExt)\end{aligned} \tag{17}$$

for all A, B : *Type* and $R : B \longrightarrow B \longrightarrow Type$, where *Pointwise* lifts a type family on B to one on $A \longrightarrow B$:

$$Pointwise\ R\ f\ g = \forall\ x \longrightarrow [\ f\ x\ ] \equiv ([\ g\ x\ ] : B\ /\ R)$$

The function needed to implement op_t is actually the *inverse* of *dist*, where A is the arity of the operator in question. Therefore, we say a type A is *quotient preserving* if for all types B and type families $R : B \longrightarrow B \longrightarrow Type$, the function *dist* is an isomorphism:

$$\begin{aligned}&\Sigma[\ trav : ((A \longrightarrow B\ /\ R) \longrightarrow (A \longrightarrow B)\ /\ Pointwise\ R)\ ] \times\\ &\quad (trav \circ dist \equiv id) \times (dist \circ trav \equiv id)\end{aligned} \tag{18}$$

All simple finite types are quotient-preserving: it is not difficult to construct the *trav* function when $A = Bool$, for example. Infinitary types, however, might not preserve quotients without additional axioms. Admitting the axiom of choice allows *all* types to preserve quotients; the axiom of *countable* choice (AC_ω, a weakened form of the axiom of choice that holds in many constructive systems), on the other hand, is precisely equivalent to the condition of $\mathbb{N}$ preserving quotients. The choice of foundation, in other words, influences which arities are permitted in term models. In the most general setting this means arities must be finite, so the parser example Figure 2 can only be applied to strings bounded by some arbitrary length.

Now we assume a theory $\mathcal{T}$ whose arity types *Arity o* all preserve quotients and return to defining the algebra op_t (16). The implementation of op_t uses the recursion principle (15):

$$op_t\ (o\ ,\ k{\sim}) = rec/\ squash/\ (\lambda\ k \longrightarrow [\ op\ (o\ ,\ k)\ ])\ cong\text{-}point\ (trav\ k{\sim})$$

This function takes $k{\sim} : Arity\ o \longrightarrow Syntax\ A\ /\ \_{\sim_t}\_$, and applies *trav* to it, pulling the quotient outside the function arrow. Then, *rec/* is used to apply a lambda function which re-wraps the subtree *k* in *op*, and *cong-point* proves that this re-wrapping respects the quotient.

Finally, for *Term* to be a model of $\mathcal{T}$, op_t must respect the laws of $\mathcal{T}$, but this is straightforward since the eq_t case of $\sim_t$ precisely says that any two programs that are instances of equations of $\mathcal{T}$ are related, and *Term* is a quotient of $\sim_t$.

Since op_t respects the laws of $\mathcal{T}$, *Term* can now implement the monadic bind, via $interp_t$:

$$\begin{aligned}&\_{\gg\!=}\_ : Term\ A \to (A \to Term\ B) \to Term\ B\\ &xs \gg\!= k = interp_t\ op_t\ k\ op_t\text{-}resp\ squash/\ xs\end{aligned}$$

This, along with proofs of the monad laws, shows that *Term* is in fact a monad.

Analogously to the freeness of *Syntax* (Theorem 2.1), the *Term* type is the free *model* for $\mathcal{T}$.

Theorem 4.3. *Let A be a set. The algebra* $(Term\ A, op_t)$ *is the free* $\mathcal{T}$*-model over A.*

The proof of this theorem follows the structure of Theorem 2.1 closely. The theorem immediately implies the following corollary known as *Birkhoff's completeness theorem* [Birkhoff 1935].

Corollary 4.4. *Let* x *and* $y : Syntax\ A$ *be two syntactic trees. The proposition* $x \sim_t y$ *holds if and only if* $interp_t\ \phi\ \rho\ x \equiv interp_t\ \phi\ \rho\ y$ *for all* $\mathcal{T}$*-models* $\phi : [\![\, \mathbb{F} \,]\!]\ C \to C$ *and functions* $\rho : A \to C$.

5 PROGRAMMING WITH EFFECTS

In this section we revisit the parsing example in Figure 2 and flesh out some details that we omitted in the introduction. Recall that we are interested in parsing expressions with a binary operation $\star$ and constants $\diamondsuit$ and $\spadesuit$, and for simplicity, we only consider strings generated by the following function from some tree $t : Tree$, where *Tree* is the AST for expressions defined in Figure 2:

$$\begin{aligned}&print : Tree \to String\\ &print\ \diamondsuit \quad\ \ = \texttt{"}\diamondsuit\texttt{"}\\ &print\ \spadesuit \quad\ \ = \texttt{"}\spadesuit\texttt{"}\\ &print\ (l \circledast r) = \texttt{"("} +\!\!+\ print\ l +\!\!+ \texttt{"}\star\texttt{"} +\!\!+\ print\ r +\!\!+ \texttt{")"}\end{aligned} \tag{19}$$

Combining Signatures. Three operations are used in *parse-tree* in Figure 2: $\langle|\rangle$, *fail*, and *char*. These operations come from a combination, namely the *tensor*, of nondeterminism and mutable state. The signature of a tensor $\mathcal{F} \otimes \mathcal{G}$ is the *coproduct* $\boxplus$ of the signatures of $\mathcal{F}$ and $\mathcal{G}$:

$$\begin{aligned}&\_\boxplus\_ : Signature \to Signature \to Signature\\ &(\mathbb{F} \boxplus \mathbb{G})\ .Op = \mathbb{F}\ .Op \uplus \mathbb{G}\ .Op\\ &(\mathbb{F} \boxplus \mathbb{G})\ .Arity\ (inl\ \ O^f) = \mathbb{F}\ .Arity\ O^f\\ &(\mathbb{F} \boxplus \mathbb{G})\ .Arity\ (inr\ O^g) = \mathbb{G}\ .Arity\ O^g\end{aligned}$$

Syntax trees generated from the signature $\mathbb{P} = \mathbb{ND} \boxplus \mathbb{S}$ have operations from both nondeterminism *and* state. Here, for instance, are the syntactic representations of *fail*, *put*, and *get*:

$$\begin{aligned}&fail : Syntax\ A && put : String \to Syntax\ \top && get : Syntax\ String\\ &fail = op\ (inl\ \grave{}fail\ , absurd) && put\ s = op\ (inr\ (\grave{}put\ s)\ , var) && get = op\ (inr\ \grave{}get\ , var)\end{aligned}$$

The remaining operation to define is *char*. This is a composite operation, defined using operations from both $\mathbb{ND}$ and $\mathbb{S}$. Conceptually, the mutable state effect is being used here to refer to the string being parsed: the *get* operation returns the "rest of the string, yet to be parsed". The parser *eof*, for instance, parses the empty string:

$$\begin{aligned}&eof : Syntax\ \top\\ &eof = \text{do } s \leftarrow get;\ guard\ (null\ s)\end{aligned}$$

where *null* tests if a string is empty, meaning that this parser will *fail* if the remaining string is anything other than the empty string. The parsers *any-char* and *char* from Figure 2 are defined in a similar way. As a reminder, here are their signatures:

$$\mathit{any\text{-}char} : \mathit{Syntax}\ \mathit{Char} \qquad\qquad \mathit{char} : \mathit{Char} \rightarrow \mathit{Syntax}\ \top$$

any-char parses and consumes any single character: it first inspects the remaining string with *get*, *fail*ing if it is empty, then it sets the remaining string to the tail of what it was, finally returning the consumed character. This parser is used by *char* to parse a specific character using an equality test.

To convert a *Syntax* value to a *Term*, it is usually sufficient to wrap it in the set-quotient constructor (13). Sometimes slightly more complex functions, like op_t, are needed.

$$\begin{array}{ll} \mathit{put} : \mathit{String} \rightarrow \mathit{Term}\ \top & \_\langle|\rangle\_ : \mathit{Term}\ A \rightarrow \mathit{Term}\ A \rightarrow \mathit{Term}\ A \\ \mathit{put}\ s = [\ \mathit{op}\ (\mathit{inr}\ (\grave{}\,\mathit{put}\ s)\ ,\ \mathit{var})\ ] & x \langle|\rangle\ y = op_t\ (\mathit{inl}\ \grave{}\langle|\rangle\ ,\ \lambda\ i \rightarrow \mathit{if}\ \ i\ \mathit{then}\ y\ \mathit{else}\ x) \end{array}$$

Combining Laws. Just like the signatures, the *laws* of nondeterminism and state are combined to yield the laws of parsing. Any law that holds in either constituent theory holds in the theory of parsing. We have already seen the state laws (12), the laws of nondeterminism are as follows:

$$\begin{array}{llll} \mathit{Eqns}\ \mathit{id}^l & .\mathit{eqn}\ \_ = \forall^n\ \lambda\ \mathit{xs} & \rightarrow \mathit{fail} \langle|\rangle\ \mathit{xs} & \doteq \mathit{xs} \\ \mathit{Eqns}\ \mathit{assoc} & .\mathit{eqn}\ \_ = \forall^n\ \lambda\ \mathit{xs}\ \mathit{ys}\ \mathit{zs} & \rightarrow (\mathit{xs} \langle|\rangle\ \mathit{ys}) \langle|\rangle\ \mathit{zs} & \doteq \mathit{xs} \langle|\rangle\ (\mathit{ys} \langle|\rangle\ \mathit{zs}) \\ \mathit{Eqns}\ \mathit{comm} & .\mathit{eqn}\ \_ = \forall^n\ \lambda\ \mathit{xs}\ \mathit{ys} & \rightarrow \mathit{xs} \langle|\rangle\ \mathit{ys} & \doteq \mathit{ys} \langle|\rangle\ \mathit{xs} \\ \mathit{Eqns}\ \mathit{idem} & .\mathit{eqn}\ \_ = \forall^n\ \lambda\ \mathit{xs} & \rightarrow \mathit{xs} \langle|\rangle\ \mathit{xs} & \doteq \mathit{xs} \end{array} \tag{20}$$

These are the laws of a semilattice: this means the parser is also a semilattice, under ⟨|⟩ and *fail*.

On the *Term* type, these laws are upgraded to a full equalities.

$$\langle|\rangle\text{-}\mathit{assoc} : (x\ y\ z : \mathit{Term}\ A) \rightarrow (x \langle|\rangle\ y) \langle|\rangle\ z \equiv x \langle|\rangle\ (y \langle|\rangle\ z)$$

However, the simple *sum* of the laws yields an unsatisfactory semantics for the parser. Identities which involve the interaction of the two theories can run into trouble, for instance:

$$\forall\ (c : \mathit{Char})\ (p\ q : \mathit{Term}\ A) \rightarrow (\mathrm{do}\ \mathit{char}\ c;\ p) \langle|\rangle\ (\mathrm{do}\ \mathit{char}\ c;\ q) \equiv (\mathrm{do}\ \mathit{char}\ c;\ p \langle|\rangle\ q)$$

This simple equation says that the nondeterministic choice of parsers that start with the same character is equal to a parser for that character followed by the choice of the rest of the parsers.

Unfortunately, though, this equation does not hold by default: the equation reorders state operations and nondeterminism operations, and it is not correct to assume that this yields an equivalent result. The solution is to have an additional law which insists this is the case. For a pair of signatures $\mathbb{F}$ and $\mathbb{G}$, the *commutativity* law is as follows, for any operators $fs \in \mathbb{F}$ and $gs \in \mathbb{G}$:

$$\begin{array}{c} (\mathrm{do}\ f \leftarrow \mathrm{Op}[\![\ \mathit{inl}\ \mathit{fs}\ \ ]\!]\ ;\ g \leftarrow \mathrm{Op}[\![\ \mathit{inr}\ \mathit{gs}\ ]\!]\ ;\ \mathit{return}\ (f\ ,\ g)) \\ \doteq \\ (\mathrm{do}\ g \leftarrow \mathrm{Op}[\![\ \mathit{inr}\ \mathit{gs}\ ]\!]\ ;\ f \leftarrow \mathrm{Op}[\![\ \mathit{inl}\ \mathit{fs}\ \ ]\!]\ ;\ \mathit{return}\ (f\ ,\ g)) \end{array} \tag{21}$$

$$\begin{array}{l} \mathrm{Op}[\![\_]\!] : \forall\ o \rightarrow \mathit{Syntax}\ (\mathit{Arity}\ o) \\ \mathrm{Op}[\![\ o\ ]\!] = \mathit{op}\ (o\ ,\ \mathit{var}) \end{array}$$

This law states that operations from the constituent theories *commute* around each other. The *tensor* of theories $\mathcal{T}_1$ and $\mathcal{T}_2$, denoted $\mathcal{T}_1 \otimes \mathcal{T}_2$, takes the sum of their signatures and laws with additionally laws of commutativity. The parsing effect is the tensor of nondeterminism and mutable state.

Interpreting into Models. The *Term* monad for the algebraic theory $\mathbb{P}$ is only syntactic terms modulo equations, so programs like *parse-tree* written with *Term* cannot run yet. To actually run them, we need to find more concrete models for the theory $\mathbb{P}$.

We have already seen a model for one of the constituent theories of the parsing effect: state-passing functions (6). In fact, this model is *isomorphic* to the free model of state. The two sides of this isomorphism are given by the following:

```
toState : (S → A × S) → Term A           runState : Term A → (S → A × S)
toState k = do s₁ ← get                   runState = interpₜ state-alg
               let x , s₂ = k s₁                             (λ x s → x , s)
               put s₂                                         runState-resp
               return x                                       ...
```

This isomorphism actually goes further: the *state monad transformer* is in fact isomorphic to tensoring with theory of state [Hyland et al. 2006]. To be precise, for the theory of mutable state $\mathcal{S}$ (over some state type S) and some other theory $\mathcal{T}$, the term monad of their tensor is isomorphic to the state monad transformer:

$$\mathit{Term}\ (\mathcal{S} \otimes \mathcal{T})\ A \Leftrightarrow S \to \mathit{Term}\ \mathcal{T}\ (A \times S) \tag{22}$$

The other effect of parsing is nondeterminism. A suitable model for nondeterminism—considering the laws in (20)—is the *finite set* $\mathcal{K}$, where $\mathcal{K}\ A$ is a finite, enumerable subsets of A. The implementation of this type is not relevant here: see [Frumin et al. 2018] for an explanation of the type, or [Kidney 2020] for an implementation in Cubical Agda. It turns out that this model is also isomorphic to the free model of nondeterminism. Thus by (22), the term monad for parsing is isomorphic to $\mathit{String} \to \mathcal{K}\ (A \times \mathit{String})$, with which we can actually run the parsers.

Recursion. Since elements of the term monad are *inductively* generated trees modulo the program equivalence, programs defined with general recursion cannot always be defined as elements of the term monad, since they may be infinite trees. One way to work around this problem is to use explicit *step indexing* as we did in the parser example (Figure 2): a recursive effectful program is represented as a sequence of finite approximations $p : \mathbb{N} \to \mathit{Term}$, where $p\ 0$ always triggers a failure and $p\ (\mathit{suc}\ n)$ expands the recursion $\mathit{suc}\ n$ times. Consequently, when stating a property about a recursive program, the step indices must be dealt with appropriately.

An alternative approach is to use the *coinductive*-version of *Syntax* to represent programs, which is the approach taken by Xia et al. [2020] in their framework of *interaction trees*; then infinite programs can be directly defined. However, it is not clear in this setting what the appropriate notion of *models* of an equational theory should be, let alone constructing free models.

Therefore the equational approach taken in this paper trades off direct support of recursion for the ability to define program equivalence by a set of equations directly. Consequently, program equalities proved on free models automatically hold in any other semantic models (Corollary 4.4). In contrast, the program equivalence of interaction trees is a bisimilarity induced by a *specific* operational semantics [Xia et al. 2020]. Thus program equalities proven for the bisimilarity do not automatically hold for other models. It is worthwhile to reconcile these two approaches in the future, possibly by using *guarded dependent type theory* [Baunsgaard Kristensen et al. 2022; Bizjak et al. 2016; Sterling and Harper 2018] or *constructive domain theory* [de Jong and Escardó 2021].

6 AN EFFECT-GENERIC HOARE LOGIC

We have presented a full formalisation of algebraic effects which accounts for their equations. We will now demonstrate the power of this formalisation, in allowing for sophisticated reasoning about programs, culminating in a *generic Hoare logic* for algebraic effects. This logic is a constructive version of Schröder and Mossakowski [2003]'s *monad-independent Hoare logic*; what is novel is an elimination principle that connects Hoare-style reasoning and equational reasoning.

The Hoare logic of Schröder and Mossakowski [2003] is flexible: usually, Hoare logic is specific to stateful assertions; the logic we will develop here allows for assertions over *any* effect that can be encoded in our framework. As long as the assertion can be encoded as a term of type *Term Ω* (23) that is *SEF* and *DET* (25) it can be manipulated and expressed using Hoare logic.

Truth Values. In Homotopy Type Theory, truth values are represented by the following type:

$$\begin{aligned}&\text{record } \Omega : \mathit{Type}_1 \text{ where}\\&\quad \text{field } \mathit{ProofOf} : \mathit{Type}\\&\qquad\quad \mathit{IsProp} \quad : \mathit{isProp}\ \mathit{ProofOf}\end{aligned} \tag{23}$$

This is the type of propositions: true and false statements like "7 is a prime number" or "3 is even". A value of type *Ω* is a pair of a type *ProofOf*, representing the type of proofs of the proposition, and a constraint *IsProp* that *ProofOf* is a proposition in the HoTT sense. For example, the *Ω* value corresponding to "three is even" is as follows:

$$\begin{aligned}&\mathit{3\text{-}is\text{-}even} : \Omega\\&\mathit{ProofOf}\ \mathit{3\text{-}is\text{-}even} = 3 \bmod 2 \equiv 0\\&\mathit{IsProp}\quad \mathit{3\text{-}is\text{-}even} = \mathit{isSetNat}\ \_\ \_\end{aligned} \tag{24}$$

The type of proofs of the (false) statement "3 is even" is an equality type $3 \bmod 2 \equiv 0$, and it is a proposition since $\mathbb{N}$ is a set—equalities between any two natural numbers are propositions.

In a classical setting, the booleans are a suitable type for truth values. Constructively, however, *Bool* represents only the *decidable* propositions. The *Ω* type above represents both true and false propositions, even if we do not necessarily know which a particular proposition belongs to.

The presence of the *IsProp* field means that the *ProofOf* type can have at most one inhabitant. This has the corollary that any biconditional propositions are equal:

$$\_\text{iff}\_ : (\mathit{ProofOf}\ X \to \mathit{ProofOf}\ Y) \to (\mathit{ProofOf}\ Y \to \mathit{ProofOf}\ X) \to X \equiv Y$$

This means that all true propositions are equal, as are all false propositions. In fact, there is *no* proposition that is not equal to one of the following values ($\nexists p.\ p \not\equiv \mathit{True} \times p \not\equiv \mathit{False}$):

$$\begin{aligned}&\mathit{False} : \Omega\\&\mathit{ProofOf}\ \mathit{False} = \bot\\&\mathit{IsProp}\quad \mathit{False}\ ()\end{aligned} \qquad\qquad \begin{aligned}&\mathit{True} : \Omega\\&\mathit{ProofOf}\ \mathit{True} = \top\\&\mathit{IsProp}\quad \mathit{True}\ \_\ \_ = \mathit{refl}\end{aligned}$$

For instance, *3-is-even* (24) is equal to *False*:

$$\begin{aligned}&\mathit{3\text{-}is\text{-}not\text{-}even} : \mathit{3\text{-}is\text{-}even} \equiv \mathit{False}\\&\mathit{3\text{-}is\text{-}not\text{-}even} = \mathit{1{\not\equiv}0}\ \text{iff}\ \mathit{absurd}\end{aligned} \qquad \mathit{1{\not\equiv}0} : 1 \equiv 0 \to \bot$$

The usual logical connectives can be defined on *Ω*:

$$\begin{aligned}&\mathit{ProofOf}\ (X\ |{\wedge}|\ Y) = \mathit{ProofOf}\ X \times \mathit{ProofOf}\ Y\\&\mathit{ProofOf}\ (X\ |{\to}|\ Y) = \mathit{ProofOf}\ X \to \mathit{ProofOf}\ Y\\&\mathit{ProofOf}\ (X\ |{\vee}|\ Y) = \|\ \mathit{ProofOf}\ X \uplus \mathit{ProofOf}\ Y\ \|\end{aligned}$$

The odd one out here is $|{\vee}|$: its definition uses propositional truncation, $\|\_\|$, which makes the *ProofOf* type a proposition. Without it, *ProofOf* $(X\ |{\vee}|\ Y)$ could have multiple inhabitants (consider, for instance, $\top \uplus \top$). Propositional truncation is a special case of a quotient type:

$$\begin{array}{l} \|\_\| : \mathit{Type} \to \mathit{Type} \\ \| A \| = A \,/\, \lambda \_\ \_ \to \top \end{array}$$

Propositional truncation is also useful for defining the truth value corresponding to the equality of two arbitrary values of some type A, which is not necessarily a set in the HoTT sense:

$$\begin{array}{l} \_|{\equiv}|\_ : A \to A \to \Omega \\ \mathit{ProofOf}\ (x \mathrel{|{\equiv}|} y) = \| x \equiv y \| \\ \mathit{IsProp}\quad (x \mathrel{|{\equiv}|} y) = \mathit{squash} \end{array}$$

This allows us to talk about equality of values without always having to prove they are a set.

Assertions. It is often convenient to be able to state properties about an effectful program from *within* the program, with an assertion. For instance, with the parser effect, we might want to construct a predicate for parsers that do not consume any input:

$$\begin{array}{l} \mathit{no\text{-}input} : \mathit{Term}\ A \to \mathit{Term}\ \Omega \\ \mathit{no\text{-}input}\ p = \mathrm{do}\ s_1 \leftarrow \mathit{get};\ p;\ s_2 \leftarrow \mathit{get};\ \mathit{return}\ (s_1 \mathrel{|{\equiv}|} s_2) \end{array}$$

This program returns true when the state is the same before and after executing p. However, it constructs a *computation* full of truth values, rather than a truth value itself.

To convert from $\mathit{Term}\ \Omega$ to a proposition we will use $\mathscr{G}$ from Schröder and Mossakowski [2003].

$$\begin{array}{l} \mathscr{G} : \mathit{Term}\ (A \times \Omega) \to \mathit{Type}\ \_ \\ \mathscr{G}\ p = p \equiv (\mathrm{do}\ x\ ,\ \_ \leftarrow p;\ \mathit{return}\ (x\ ,\ \mathit{True})) \end{array}$$

The proposition $\mathscr{G}\ t$ corresponds to the statement that every truth value in t is *True*. t here is allowed to have a return value, as well: this can be useful for chaining together predicates. The assertion *no-input*, then, can be encoded as:

$$\mathit{no\text{-}input}\ p = \mathscr{G}\ \mathrm{do}\ s_1 \leftarrow \mathit{get};\ x \leftarrow p;\ s_2 \leftarrow \mathit{get};\ \mathit{return}\ (x\ ,\ s_1 \mathrel{|{\equiv}|} s_2)$$

Pure Assertions. Usually, it is sensible to insist that assertions themselves do not *produce* effects. They can of course depend on effects: for instance, an assertion in the parsing effect probably shouldn't modify the string being parsed, but it should be able to *observe* the string. The following pair of conditions capture this notion of semi-purity.

$$\begin{array}{l} \mathit{SEF}\ \ p = (\mathrm{do}\ p;\ \mathit{return}\ \langle\rangle) \equiv \mathit{return}\ \langle\rangle \\ \mathit{DET}\ p = (\mathrm{do}\ x \leftarrow p;\ y \leftarrow p;\ \mathit{return}\ (x\ ,\ y)) \equiv (\mathrm{do}\ x \leftarrow p;\ p;\ \mathit{return}\ (x\ ,\ x)) \end{array} \tag{25}$$

SEF (side-effect free) encodes that a program doesn't modify the ambient environment with some effect. This outlaws, for instance, programs like *put x*. *DET* (deterministic) encodes that the assertion itself can't be nondeterministic; this outlaws things like $\mathit{True} \mathbin{\langle|\rangle} \mathit{False}$. A "pure" assertion is an assertion that is both *SEF* and *DET*.

Note that these notions of semi-purity instantiate to *different* notions of semi-purity depending on the underlying monad. In the case of state, *SEF* encodes a term that does not modify the state; on a monad for exceptions, *SEF t* is a statement that t does not throw an exception.

Pure assertions are still quite expressive because they can return any semantic propositions. For example, for the effect of state with the state type $\mathit{Loc} \to \mathit{Maybe}\ (\mathit{Loc} \uplus \mathit{Val})$ modelling a finite heap of some location type *Loc* and each allocated location stores either a pointer to another location or a value of type *Val*, *separating conjunction* in separation logic [Reynolds 2002a] can be expressed as

$$\mathrm{do}\ \mathit{mem} \leftarrow \mathit{get};\ \ \mathit{return}\ (\exists s_1.\ \exists s_2.\ (\mathit{is\text{-}disj\text{-}union}\ \mathit{mem}\ s_1\ s_2) \mathrel{|{\wedge}|} p\ s_1 \mathrel{|{\wedge}|} q\ s_2)$$

which returns the proposition that the current allocated memory cells are the disjoint union of two heaps satisfying predicates p and q respectively.

Hoare Triples. We now have all of the components necessary to define Hoare triples.

Definition 6.1 (Hoare Triple). A Hoare triple $\{\phi\}\ x \leftarrow p\ \{\psi\ x\}$ is a predicate over a program p : *Term A*, and a pair of pure assertions ϕ : *Term Ω* and $\psi : A \rightarrow$ *Term Ω*. The predicate holds when ϕ "implies" ψ, when ϕ, p, and ψ are executed in sequence:

$$\mathit{Hoare}\ \phi\ p\ \psi = \mathscr{G}\ \text{do}\ a \leftarrow \phi;\ x \leftarrow p;\ b \leftarrow \psi\ x;\ \mathit{return}\ (x\ ,\ a\ |{\rightarrow}|\ b)$$

We do not require the pre- and post-conditions be *SEF* or *DET* in the definition of Hoare triples, but instead take them as arguments to the inference rules or connectives where needed. For instance, the conjunction rule (26) requires *SEF* ϕ, *DET* ϕ, and *SEF* ψ.

As a small example, the *no-input* property can be encoded as a Hoare triple as follows:

$$\begin{array}{ll} \mathit{remaining} : \mathit{String} \rightarrow \mathit{Term}\ \Omega & \mathit{no\text{-}input} : \mathit{Term}\ A \rightarrow \mathit{Type} \\ \mathit{remaining}\ s_1 = \text{do}\ s_2 \leftarrow \mathit{get} & \mathit{no\text{-}input}\ p = \\ \qquad\qquad\qquad\quad \mathit{return}\ (s_1\ |{\equiv}|\ s_2) & \quad \forall\ s \rightarrow \{\ \mathit{remaining}\ s\ \}\ \_ \leftarrow p\ \{\ \mathit{remaining}\ s\ \} \end{array}$$

The combinator *remaining* asserts that the remaining string is equal to a supplied string. *no-input p* states that, for all strings s, if s remains before t's execution, then it will remain after t's execution.

Connectives on Assertions. Assertions can be combined using the standard logical connectives, giving a calculus for Hoare logic. We have, for instance, conjunction between two Hoare clauses:

$$(\{\ \phi\ \}\ x \leftarrow p\ \{\ \psi\ x\ \}) \rightarrow (\{\ \phi\ \}\ x \leftarrow p\ \{\ \chi\ x\ \}) \rightarrow (\{\ \phi\ \}\ x \leftarrow p\ \{\ \psi\ x\ \langle\wedge\rangle\ \chi\ x\ \}) \tag{26}$$

This says that if two assertions ψ and χ hold after the precondition ϕ and the program p, then their conjunction also holds. Inference rules for other logical connectives exist as well.

Using Hoare Logic. Now that the fundamental components of Hoare logic have been defined, let's use it to verify the (partial) correctness of a simple program: the tree parser in Figure 2. We will not present the full proof here, just some highlights to demonstrate the power of the framework.

First, let's construct a predicate for a parser returning a particular value.

$$s \in p \mapsto v = \forall\ r \rightarrow \{\ \mathit{remaining}\ (s +\!\!+\ r)\ \}\ v' \leftarrow p\ \{\ \mathit{return}\ (v\ |{\equiv}|\ v')\ \langle\wedge\rangle\ \mathit{remaining}\ r\ \} \tag{27}$$

The predicate $s \in p \mapsto v$ states that the parser p parses the string s and returns the value v. It does so by asserting that the string s is the prefix of the input (*remaining*$(s +\!\!+ r)$), running the parser p, and asserting that the value returned is equal to v and the rest of the string has not been altered.

The parser we are interested in verifying is the tree parser Figure 2:

$$\mathit{round\text{-}trip} : \forall\ n\ t \rightarrow \mathit{print}\ t \in \mathit{parse\text{-}tree}\ n \mapsto t \tag{28}$$

This simple property asserts that, for any tree t, parsing the printed representation of t returns something equal to t. Note that this is a specification of *partial* correctness: as is the norm for Hoare logic, this specification only applies if the underlying program terminates successfully. Put another way, when the parser fails, *any* property holds.

$$(\psi : \mathit{Term}\ \Omega) \rightarrow \{\}\ \mathit{fail}\ \{\ \psi\ \} \tag{29}$$

This proof is built from small components, by pattern-matching on n and t. For instance, in the first case, when $n = 0$, the proof goal is:

$$\{\ \mathit{remaining}\ (\mathit{print}\ t +\!\!+\ s)\ \}\ v \leftarrow \mathit{parse\text{-}tree}\ \mathit{zero}\ \{\ \mathit{return}\ (t\ |{\equiv}|\ v)\ \langle\wedge\rangle\ \mathit{remaining}\ s\ \}$$

Since *parse-tree zero* = *fail*, this goal follows from (29).

We will look at one other case. When $t = \spadesuit$ and $n > 0$, the parser should re-parse the result of printing the tree $\spadesuit$. In Figure 2, notice that when $n > 0$, the *parse-tree* function produces three alternative parsers, combined using $\langle|\rangle$. Parsers that use $\langle|\rangle$ can be verified with the following lemma:

$$(\{\ \phi\ \}\ x \leftarrow p\ \{\ \psi\ x\ \}) \rightarrow (\{\ \phi\ \}\ x \leftarrow q\ \{\ \psi\ x\ \}) \rightarrow (\{\ \phi\ \}\ x \leftarrow p \langle|\rangle q\ \{\ \psi\ x\ \})$$

If a Hoare triple holds on p and q, then it holds on $p \langle|\rangle q$. So, to verify this case of the proof we need to verify each of the alternative parsers.

Let's look at the two alternatives which parse $\spadesuit$ and $\diamond$. The proof obligations are:

$$\{\ \mathit{remaining}\ (\texttt{'}\spadesuit\texttt{'} :: s)\ \}\ v \leftarrow (\text{do } \mathit{char}\ \texttt{'}\spadesuit\texttt{'};\ \mathit{return}\ \spadesuit)\ \{\ \mathit{return}\ (\spadesuit\ |{\equiv}|\ v)\ \langle\wedge\rangle\ \mathit{remaining}\ s\ \}$$
$$\{\ \mathit{remaining}\ (\texttt{'}\spadesuit\texttt{'} :: s)\ \}\ v \leftarrow (\text{do } \mathit{char}\ \texttt{'}\diamond\texttt{'};\ \mathit{return}\ \diamond)\ \{\ \mathit{return}\ (\spadesuit\ |{\equiv}|\ v)\ \langle\wedge\rangle\ \mathit{remaining}\ s\ \}$$

Both of these alternatives can be verified using the following lemma:

$$\forall\ c_1\ c_2\ s \rightarrow \{\ \mathit{remaining}\ (c_1 :: s)\ \}\ \mathit{char}\ c_2\ \{\ \mathit{return}\ (c_1\ |{\equiv}|\ c_2)\ \langle\wedge\rangle\ \mathit{remaining}\ s\ \} \tag{30}$$

This states that, if the remaining string starts with a character c_1, and *char* c_2 succeeds, then $c_1 \equiv c_2$ and the tail of the string remains. Notice that the postcondition here can be false: c_1 need not equal c_2. The lemma is still valid, however, because if $c_1 \not\equiv c_2$ then *char* c_2 fails, so (29) applies.

Returning to the two alternatives, in both branches `'♠'` prefixes the remaining string. This means the first alternative succeeds, since it parses the character `'♠'`, and returns the leaf $\spadesuit$.

The second alternative, however, returns $\diamond$, which makes the postcondition of the obligation $\mathit{return}\ (\spadesuit\ |{\equiv}|\ \diamond)\ \langle\wedge\rangle\ \mathit{remaining}\ s$, which is clearly false. Luckily, applying (30) gives the following:

$$\{\ \mathit{remaining}\ (\texttt{'}\spadesuit\texttt{'} :: s)\ \}\ \mathit{char}\ \texttt{'}\diamond\texttt{'}\ \{\ \mathit{return}\ (\texttt{'}\spadesuit\texttt{'}\ |{\equiv}|\ \texttt{'}\diamond\texttt{'})\ \langle\wedge\rangle\ \mathit{remaining}\ s\ \}$$

The postcondition here contains `'♠'` $|{\equiv}|$ `'◇'`, itself a false assertion, from which we can derive any other assertion, including the obligation for this case.

7 RELATING HOARE-STYLE AND EQUATIONAL REASONING

Apart from being generic for the underlying computational effect, another appealing feature of the Hoare logic in Section 6 is that it is defined solely in terms of equations of programs. Therefore the two common ways for reasoning about programs—*equational reasoning*, which is traditionally more popular in functional programming, and *Hoare-style reasoning*, which is traditionally more popular in imperative programming—pleasantly meld together: a Hoare triple $\{\ \phi\ \}\ x \leftarrow p\ \{\ \psi\ x\ \}$ is valid if and only if the following program equivalence holds:

$$\begin{aligned} &(\text{do } a \leftarrow \phi;\ x \leftarrow p;\ b \leftarrow \psi\ x;\ \mathit{return}\ (x\ ,\ a\ |{\rightarrow}|\ b)) \\ \equiv\ &(\text{do } a \leftarrow \phi;\ x \leftarrow p;\ b \leftarrow \psi\ x;\ \mathit{return}\ (x\ ,\ \mathit{True})) \end{aligned} \tag{31}$$

However, one might wonder if equation (31) is really "useful" for program reasoning, or is it just an arbitrary equality that encodes Hoare logic. For a minimal example, if we can use Hoare logic to show $\{\}\ x \leftarrow p\ \{\ \mathit{return}\ (x \equiv 0)\ \}$, does (31) imply that $(\text{do } x \leftarrow p\,;\ \mathit{put}\ x) \equiv (\text{do } p\,;\ \mathit{put}\ 0)$? To address this question, in this section we show the following "elimination principle" for Hoare logic that allows useful program equivalences to be extracted from valid Hoare triples.

Theorem 7.1 (*H-elim*). *Given a theory with finite (Definition 7.2) variables and arities, let* $\phi : A \rightarrow \Omega$ *be a predicate on some type* A*, and let* f *and* g *be two functions* $A \rightarrow \mathit{Term}\ B$ *such that for all* $x : A$*,* $\mathit{ProofOf}\ \ (\phi\ x)$ *implies* $f\ x \equiv g\ x$*. Then the following implication holds:*

$$\mathit{H\text{-}elim} : \{\}\ x \leftarrow p\ \{\ \mathit{return}\ (\phi\ x)\ \} \rightarrow (p \gg\!= f) \equiv (p \gg\!= g)$$

At first glance, it might not be obvious why this theorem is difficult to prove, or what its significance is. It seems related to a simple congruence law, like:

$$\forall\, p, f, g.\ (\forall x.\ f\ x \equiv g\ x) \rightarrow p \gg\!= f \equiv p \gg\!= g \tag{32}$$

Indeed, this law holds straightforwardly: by function extensionality, since f and g are equal at every input $(\forall x.\ f\ x \equiv g\ x)$ then they are equal as functions $(f \equiv g)$, and by congruence $f \equiv g$ implies $p \gg\!= f \equiv p \gg\!= g$.

In the case of *H-elim*, however, the continuations f and g are *not* equal everywhere: they are only equal where the proposition ϕ holds on their input. This makes the job of *H-elim* more difficult: it must take a proof $\{\}\ x \leftarrow p\ \{\mathit{return}\ (\phi\ x)\}$ that ϕ holds on the result of some term p, and use that to prove that f and g are equal when *executed after* p.

In the following, we first show that when the *law of excluded middle* (LEM) is assumed, *H-elim* can be proved fairly easily by a case split of whether ϕ holds for each return value of p.

Proof (with LEM). By definition the Hoare triple $\{\}\ x \leftarrow p\ \{\mathit{return}\ (\phi\ x)\}$ is the equality

$$(\text{do } x \leftarrow p; \mathit{return}\ (x, \phi\ x)) \equiv (\text{do } x \leftarrow p; \mathit{return}\ (x, \mathit{True})) \tag{33}$$

One formulation of LEM is $\Omega \equiv \mathit{Bool}$, allowing us to change the type of the predicate to $\phi : A \rightarrow \mathit{Bool}$. We then construct a continuation

$$k = \lambda(x, b) \rightarrow \mathit{if}\ b\ \mathit{then}\ g\ x\ \mathit{else}\ f\ x$$

Applying $\gg\!= k$ to each side of (33) yields

$$(\text{do } x \leftarrow p; \mathit{if}\ \phi\ x\ \mathit{then}\ g\ x\ \mathit{else}\ f\ x) \equiv (\text{do } x \leftarrow p; g\ x) \equiv p \gg\!= g$$

Recall that the goal is $p \gg\!= f \equiv p \gg\!= g$. The remaining obligation, therefore, is:

$$f\ x \equiv \mathit{if}\ \phi\ x\ \mathit{then}\ g\ x\ \mathit{else}\ f\ x$$

By cases on $\phi\ x$, when $\phi\ x = \mathit{False}$, $f\ x$ is returned and $f\ x \equiv f\ x$. When $\phi\ x = \mathit{True}$, then $g\ x$ is returned, but recall that $\mathit{ProofOf}\ (\phi\ x)$ implies $f\ x \equiv g\ x$, so the obligation is discharged. □

Of course relying on LEM means that constructivity is sacrificed—there may be no way to demonstrate a concrete derivation of a program equality produced by *H-elim*. If a fully constructive formalisation is desired, we have to take another route. Rather than inspecting the return value of ϕ we will inspect the program equivalence (31) underlying the given Hoare triple derived using the rules of $\sim_t$ (Section 4.2); this will give us the concrete structure allowing us to map a value $x : A$ back to its position in its term, and from this derive the proof of $\phi\ x$. Finally, this derivation will be modified to be the required program equivalence $p \gg\!= f \equiv p \gg\!= g$.

In order to map a value back to its position we will need to be able to search terms exhaustively for particular variables. This is only possible when the arity types of the algebraic theory are *finite*. The formal definition of finiteness we use here is *Bishop finiteness* [Bishop and Bridges 1985; Frumin et al. 2018; Kidney 2020; Yorgey 2014].

Definition 7.2 (Bishop Finiteness). A type A is bishop finite, $\mathscr{E}\ A$, if it is merely isomorphic to some finite prefix of the natural numbers.

$$\mathscr{E}\ A = \exists\ n \times \|\ \mathit{Fin}\ n \simeq A\ \|$$

Bishop finite types are *searchable*, meaning that for any decidable predicate P on a Bishop-finite type A we can decide if the predicate holds on *any* value of the type.

$$\mathscr{E}\ A \rightarrow (P? : \forall\ (x : A) \rightarrow \mathit{Dec}\ (P\ x)) \rightarrow \mathit{Dec}\ \|\ \exists\ x \times P\ x\ \|$$

This last predicate can be lifted and applied to syntax trees, meaning that any syntax tree with bishop-finite arities can itself be searched with a decidable predicate.

To modify the program equality encoding the Hoare triple to the goal $p \ggg f \equiv p \ggg g$, we will crucially rely on the fact that we can track the *provenance* in a derivation $d : s \sim_t t$. By provenance, what we mean is the correspondence between the leaf nodes of s and t if we view $d : s \sim_t t$ as a process rewriting s to t. For example, suppose $d : t_1 \sim_t t_2$ uses an equation $x = x \otimes y$ to rewrite a term t_1 to another term t_2 in one step, then the left subtree of t_2 is necessarily t_1. Moreover, no matter how we modify the right subtree of t_2, the equivalence $t_1 \sim_t t_2$ still holds, since y can be taken to be anything when rewriting with the equation $x = x \otimes y$.

More precisely, the provenance of a derivation $d : s \sim_t t$ for $s, t : Syntax\ A$ is a pair of terms $s_p, t_p : Syntax\ I_p$ for a *discrete* type I_p, i.e. having *decidable equality* (while the type A may not have decidable equality) such that s_p and t_p are equivalent under $\sim_t$, and that there exists a function $k_p : I_p \longrightarrow A$ which mapped to s_p and $t_p : Syntax\ I_p$ resulting in exactly s and $t : Syntax\ A$ (that is to say, s and t have the same shape as s_p and t_p respectively). Provenance information is encoded in these data because the decidable equality of I_p allows us to test if a leaf node of s_p and a leaf node of t_p is the same; if they are different, then we know that the corresponding leaf nodes at the same position of s and t can be different for the derivation $s \sim_t t$ to go through.

Formally, the data for provenance are collected as the following record:

```
record Provenance (s t : Syntax A) : Type where
  field
    Iₚ  : Type a                      kₚ  : Iₚ → A
    I≟ : Discrete Iₚ                  s-k : ∀ i → lookup s i ≡ kₚ (sₚ i)
    sₚ : Index s → Iₚ                 t-k : ∀ i → lookup t i ≡ kₚ (tₚ i)
    tₚ : Index t → Iₚ                 eqvₚ : fill _ sₚ ~ₜ fill _ tₚ
```

In this record, the two terms s_p and t_p are represented as two functions $s_p : Index\ s \longrightarrow I_p$ and $t_p : Index\ t \longrightarrow I_p$ from indices into each respective tree, since they necessarily have the same shape as s and t, differing only in leaf nodes. Terms can be reconstructed from such functions using *fill*.

At first, it may seem that there is an alternative design for the *Provenance* structure that is simpler and more intuitive. Instead of tracking every variable in the tree, we could instead track subtrees. In concrete terms, this would change the type of the k_p field from $I_p \longrightarrow A$ to $I_p \longrightarrow Syntax\ A$, where k_p returns the *subtree* at a particular index. This approach seems more natural in the context of equations like $x \otimes y = x$, where clearly the succinct way to represent this rewriting step is as the removal of the whole subtree y. Our approach would instead represent this step as the field s_p having a larger image than t_p, which is much more indirect.

As is often the case with mechanised proofs, what is simple for humans is complex for computers. In this case, using indices that point to every value in the tree allows for a much more *uniform* representation that would not be possible with a subtree-based representation. In particular, this representation lets us construct an actual *signature* that corresponds to syntax trees, where the interpretation of that signature is isomorphic to *Syntax*.

```
Position : Syntax A → Type                   ⋆ : Signature
Position (var _) = ⊤                         ⋆ = Syntax ⊤ ◁ Position
Position (op (Oᵢ , k)) =
  Σ[ i : Arity Oᵢ ] × Position (k i)         Indexed : Syntax A ⇔ ⟦ ⋆ ⟧ A
```

This representation means that the fields s_p and t_p can be easily converted to and from *Syntax* types, via the *Indexed* isomorphism. This isomorphism is witnessed by the *lookup* and *fill* functions.

The key lemma for showing Theorem 7.1 is the following.

Lemma 7.3. *Given a pair of syntax trees s and t with $s \sim_t t$, in a theory with finite arities and variables, the type $\|Provenance\ s\ t\|$ is inhabited.*

Proof. The proof proceeds via induction on the relation $s \sim_t t$.

$\underline{refl_t}$ Given $s : Syntax\ A$, we want to construct an element of $Provenance\ s\ s$. The index for this case is $I_p = Index\ s$. In other words an index into the original tree. And we let $k_p = lookup\ s$. The other fields follow trivially since the copies of the trees are exact copies of s.

$\underline{sym_t}$ The case for symmetry is even simpler than $refl_t$: given a $Provenance\ s\ t$, we need to produce a $Provenance\ t\ s$. Since the datatype is itself symmetric, we can simply switch around all the chiral fields, i.e. $s_p\ (sym_t\ p) = t_p\ p$; $t_p\ (sym_t\ p) = s_p\ p$.

$\underline{cong_t}$ Given an operator O_i, a pair of continuations $k_l, k_r : Arity\ O_i \to Syntax\ A$, and provenance $\mathcal{D}$ at every point of $Arity\ O_i$, the task for the case of congruence is to construct an element of $Provenance\ (op\ (O_i, k_l))\ (op\ (O_i, k_r))$. The index that we will construct for this case is:

$$I_p = \Sigma[\ i : Arity\ O_i\ ] \times \mathcal{D}.\ I_p\ i$$

Again, the rest of the coherence proofs follow in a relatively trivial way.

$\underline{eq_t}$ The eq_t case is the most important of the proof: in the relation, this case is what can allow the shape of the equivalent trees to change, and it is where variables can be introduced or removed. Given an equation with left-hand-side lhs, right-hand-side rhs, and continuation k, this case constructs $Provenance\ (lhs \ggg k)\ (rhs \ggg k)$. The index here is a variable in the equations paired with an index into the continuation applied at that variable:

$$I_p = \Sigma[\ v : Eqns\ i\ .v\ \Gamma\ ] \times Index\ (k\ v)$$

The rest of the fields follow by applying the continuation and looking up in the result.

$\underline{trans_t}$ Perhaps surprisingly, this is the most difficult case of the proof. Given $s, t, u : Syntax\ A$, $lhs : Provenance\ s\ t$, and $rhs : Provenance\ t\ u$, the task is to construct an element of $Provenance\ s\ u$. We need to pick an index here that can be remapped to every variable in either of the trees. We will use a (set) *pushout*: a pushout of two functions $f : A \to B$ and $g : A \to C$ is a disjoint union of B and C, with the added quotient that alternate injections $inl\ x$ and $inr\ y$ are equal if there is some i that points to both ($f\ i \equiv x \land g\ i \equiv y$):

```
data $Pushout\ (f : A \to B)\ (g : A \to C) : Type$ where
  $inl : B \to Pushout\ f\ g$
  $inr : C \to Pushout\ f\ g$
  $push : \forall\ i \to inl\ (f\ i) \equiv inr\ (g\ i)$
  $trunc : isSet\ (Pushout\ f\ g)$
```

Note that the *trunc* constructor makes this a *set* pushout, rather than a fully-general pushout type.

The new index is the following pushout, i.e. a sum of the indices from both sides, with the quotient that if they point to the same variable in the *middle* (u) they are equal:

$$I_p = Pushout\ \{A = Index\ u\}\ \{B = lhs.\ I_p\}\ \{C = rhs.\ I_p\}\ lhs.\ t_p\ rhs.\ s_p \tag{34}$$

To reconstruct the original variable we will pattern match on the pushout, and apply k_p from the left-hand-side or right-hand-side respectively:

$$\begin{aligned} &k_p : I_p \to A && k_p\ (push\ i\ j) = (sym\ (lhs.\ t\text{-}k\ i) ⨾ rhs.\ s\text{-}k\ i)\ j \\ &k_p\ (inl\ x) = lhs.\ k_p\ x && k_p\ (trunc\ x\ y\ p\ q\ i\ j) = \\ &k_p\ (inr\ x) = rhs.\ k_p\ x && \quad setA\ (k_p\ x)\ (k_p\ y)\ (cong\ k_p\ p)\ (cong\ k_p\ q)\ i\ j \end{aligned}$$

This function is then proved coherent by applying the combination of t-k and s-k from the left-hand-side and right-hand-side respectively; these proofs show that an index pointing into the same place in the middle tree u is equal.

The trees s_p and t_p are given by:

$$\begin{array}{ll} s_p : \mathit{Index}\ s \longrightarrow I_p & t_p : \mathit{Index}\ t \longrightarrow I_p \\ s_p = \mathit{inl} \circ \mathit{lhs}.\ s_p & t_p = \mathit{inr} \circ \mathit{rhs}.\ t_p \end{array}$$

By the definition of the quotient, we have *fill* (*shape u*) $s_p \equiv$ *fill* (*shape u*) t_p, which satisfies the eqv_p coherence property. What remains is to show the index type (34) has decidable equality, i.e. being discrete, and it is a consequence of the following lemma. □

Lemma 7.4. *A Pushout* $(f : A \longrightarrow B)$ $(g : A \longrightarrow C)$ *is discrete if* B *and* C *are discrete and* A *is finite.*

Proof. Given two values x, y : *Pushout f g*, we need to decide if they are equal. This is complicated by the presence of the pushout paths available: $\mathit{inl}\ x \equiv \mathit{inl}\ x$ obviously holds, as does $\mathit{inr}\ x \equiv \mathit{inr}\ x$, but pushout values can be equal in other ways. For instance, $\mathit{inl}\ (f\ i) \equiv \mathit{inr}\ (g\ i)$ and, by extension, $\mathit{inl}\ x \equiv \mathit{inl}\ x'$ if there is an i and j such that $x \equiv f\ i \wedge g\ i \equiv g\ j \wedge f\ j \equiv x'$. In fact, there can be a chain of As representing a path between two pushout variables. Deciding if two pushout values are equal amounts to determining if such a chain exists. Since A is finite, we can enumerate all possible chains, and check each for coherence, yielding a terminating algorithm.

First, we characterise the type of paths in a pushout:

$$\begin{array}{l} x \approx^* y = \exists\ p \times \mathit{Push}\ p\ x\ y \\ \mathit{Push} : \mathit{List}\ A \longrightarrow B \uplus C \longrightarrow B \uplus C \longrightarrow \mathit{Type} \\ \mathit{Push}\ []\qquad x \qquad\ \ y = x \equiv y \\ \mathit{Push}\ (i :: \mathit{is})\ (\mathit{inl}\ x)\ \ y = x \equiv f\ \ i \times \mathit{Push}\ \mathit{is}\ (\mathit{inr}\ (g\ \ i))\ y \\ \mathit{Push}\ (i :: \mathit{is})\ (\mathit{inr}\ x)\ y = x \equiv g\ \ i \times \mathit{Push}\ \mathit{is}\ (\mathit{inl}\ (f\ \ i))\ y \end{array}$$

A path is represented by a (possibly empty) list of As; the *Push* predicate returns if a list forms a coherent path between two sums.

Every path between two sums is represented by at least one such chain: we can prove this by showing the relation $\approx$ * is reflexive, transitive, and symmetric, and supports the *push* relation.

Notice, however, that a chain can have *loops*, where the same A is passed through multiple times. A loopless path is one with no duplicate As:

$$\mathit{Loopless} = \mathit{NoDup} \circ \mathit{fst}$$

Any such chain can be shortened yielding an equivalent path simply by cutting out the loop:

$$\mathit{deloop} : \forall\ \{x\ y\} \longrightarrow x \approx^* y \longrightarrow \Sigma[\ p : x \approx^* y\ ] \times \mathit{Loopless}\ p$$

As a result, any path between two sums must be represented by at least one *loopless* chain.

Finally, we know that there are only finitely many lists of As without duplicates (if A is finite). As a result, there are only finitely many "loopless" paths between two sums; since B and C are discrete we can test all of these for coherence, meaning we can exhaustively search for all such paths, yielding an algorithm for decidable equality. □

Finally, we are ready to prove *H-elim* (Theorem 7.1) assuming that all arity types of the algebraic theory are finite (Definition 7.2) but not LEM.

Proof Sketch. The first step of this proof is to dispense with the *Term* and equality, and replace them with *Syntax* trees and $\sim_t$. The two are equivalent by the effectiveness of quotients, although we had to be careful to preserve coherence when translating between the two representations. Concretely, this means that our goal has changed to the following for some $[s] = p$:

$$(\phi?\ \langle\$\rangle\ s \sim_t \phi T\ \langle\$\rangle\ s)\ \ \longrightarrow\ \ (s \gg\!= f \sim_t s \gg\!= g)$$

where $f\ \langle\$\rangle\ s$ = do $x \leftarrow s$; *return* $(f\ s)$, and $\phi? = \lambda x \longrightarrow (x\ , \phi\ x)$, and $\phi T = \lambda x \longrightarrow (x\ , \mathit{True})$.

First, we will apply Lemma 7.3 to build *Provenance* $(\phi? \,\langle\$\rangle\, s)\ (\phi T \,\langle\$\rangle\, s)$, from which we will extract eqv_p. We will then apply a continuation k' to each side of eqv_p, and finally we will show that the result is equal to $s \ggg f \sim_t s \ggg g$. The continuation k' is

$$k'\ i = \mathsf{let}\ v = k_p\ i\,.fst\ \mathsf{in}\ \mathit{if}\ \ does\,(\in s?\ i)\ then\,f\ \ v\ else\ g\ v$$

This takes an index $i : I_p$ and finds the variable it points to with k_p, and then if the index is in the left-hand-side tree it applies f, otherwise it applies g.

The expression $does\ (\in s?\ i)$ returns a Boolean indicating whether or not the index i is present in the tree s. To search a tree for a particular value the tree must be *finite*; this is the case since all trees are well founded and we have assumed that all the arities are finite.

Next we will prove this continuation is coherent. For the left-hand-side we have:

$$\in s_p \Rightarrow f : (i : Index\ (\phi? \,\langle\$\rangle\, s)) \to k'\ (s_p\ i) \equiv f\ (fst\ (k_p\ (s_p\ i)))$$

This says that, for a position in the left-hand-side tree, applying k' to the index at that position in the copy is the same as applying f to the variable at that position. Similarly for the right-hand-side,

$$\in t_p \Rightarrow g : (i : Index\ (\phi T \,\langle\$\rangle\, s)) \to k'\ (t_p\ i) \equiv g\ (fst\ (k_p\ (t_p\ i)))$$

We will not go into the details of these respective proofs (they involve k and the fields $s\text{-}k$ and $t\text{-}k$). However, they can be used to construct the following proofs:

$$ts_1 : f \,\langle\$\rangle\, s \equiv k' \,\langle\$\rangle\, fill\ \_\ s_p \qquad\qquad ts_2 : g \,\langle\$\rangle\, s \equiv k' \,\langle\$\rangle\, fill\ \_\ t_p$$

These combined with eqv_p give the final goal $s \ggg f \sim_t s \ggg g$. □

Remark. The constructive proof of *H-elim* above requires that all arities be finite. This means that, for instance, the constructive proof of *H-elim* applies to the parser example of this paper only when strings stored in the mutable state are finitely bounded. This is a reasonable constraint: after all, computers in the real world have finite memory. However, note that the nonconstructive proof of *H-elim* does not need finiteness, so constructivity can be traded off for infinite arities.

Applying *H-elim* to the Parser. Finally we demonstrate how to apply Theorem 7.1 to the parser example (Figure 2), to derive the program equality (1) from the round-trip lemma (28). First, recall the definition of $s \in p \mapsto v$, given in (27). Expanding some definitions yields the following:

$$\begin{aligned}&\forall r \to\\ &\quad \{r' \leftarrow get; return\ (r'\ |{\equiv}|\ s +\!\!+\ r)\}\ v' \leftarrow p; r'' \leftarrow get\ \{return\ ((r\ |{\equiv}|\ r'')\ |{\wedge}|\ (v\ |{\equiv}|\ v'))\}\end{aligned}$$

Using some of the standard Hoare combinators we convert the above to:

$$\{\}\ r \leftarrow get; put\ (s +\!\!+\ r); v' \leftarrow p; r' \leftarrow get\ \{return\ ((r\ |{\equiv}|\ r')\ |{\wedge}|\ (v\ |{\equiv}|\ v'))\}$$

This is essentially a rephrasing of (27): "when $s \in p \mapsto v$ holds, prepending s to the input stream and then running the parser p returns the value v and leaves the input stream in its original state".

From here, we can apply *H-elim* to turn the equalities in the postcondition to program equalities:

$$\begin{aligned}&(\mathsf{do}\ s \leftarrow get;\ put\ (print\ t +\!\!+\ s);\ t' \leftarrow parse\text{-}tree\ n;\ s' \leftarrow get;\ put\ s';\ return\ t')\\ &\quad\equiv\\ &(\mathsf{do}\ s \leftarrow get;\ put\ (print\ t +\!\!+\ s);\ \_ \leftarrow parse\text{-}tree\ n;\ s' \leftarrow get;\ put\ s\ ;\ return\ t)\end{aligned}$$

This equality states that the value returned by *parse-tree* will indeed be the correct tree. By the state and monad laws, the equation above simplifies to

$$(\mathsf{do}\ push\ (print\ t); parse\text{-}tree\ n) \equiv (\mathsf{do}\ push\ (print\ t); parse\text{-}tree\ n; put\ s; return\ t) \qquad (35)$$

This equation and the goal (1) already have the same left-hand side. To simplify the right-hand side, we use the following lemma.

Lemma 7.5. *For the effect of parsing, if a term p : Term A is* total*, in the sense that there exists some syntax tree s : Syntax A such that* $[\ s\ ] = p$ *and s does not contain the operation fail in its all subtrees,*

$$(\text{do } p;\ \ put\ s) = put\ s$$

Using this lemma, if we assume that the program do $\{push\ (print\ t); parse\text{-}tree\ n\}$ is total, which is in fact true for sufficiently large step-index n, then (35) immediately implies our goal:

$$(\text{do } push\ (print\ t);\ parse\text{-}tree\ n) \equiv return\ t$$

8 RELATED WORK

Algebraic Effects. This paper is basically a formalisation of universal algebra [Birkhoff 1935; Cohn 1981], whose connection to computational effects was first developed by Plotkin and Power [2002, 2004] and Hyland et al. [2007, 2006]. Plotkin and Pretnar [2008] also introduced an equational first-order logic tailored for reasoning about programs using algebraic effects. In contrast, our formalisation is embedded in Cubical Agda. The expressiveness of Cubical Agda makes it possible to implement all the logic connectives and inference rules in [Plotkin and Pretnar 2008] as just functions over the datatype *Term* and the universe Ω of propositions. For example, their *principle of induction over computations* is subsumed by the usual induction principles of *Term* and *Syntax*.

Plotkin and Pretnar [2009, 2013] internalised homomorphisms out of the free model of an algebraic theory as a programming language feature—*effect handlers*. Since then, many researchers have developed the theory and implementation of effect handlers, to name a few, CPS translations [Schuster et al. 2022, 2020], effect systems [Bauer and Pretnar 2013; Leijen 2014], parametricity [Biernacki et al. 2017; Zhang and Myers 2019], type dependency [Ahman 2017]. Our paper is not about effect handlers qua language construct, but future work may use our library to build upon.

Formalisations. Our paper is certainly not the first formalisation of universal algebras in type theory. Previous formalisations include the work by Gunther et al. [2018], Abel [2021], and DeMeo and Carette [2022] in Agda and Capretta [1999]'s formalisation in Coq. These formalisations use setoids to deal with quotients, while our formalisation uses higher inductive types, which is typically regarded as easier to work with. A more important difference of our formalisation from these works is that we focus on the applications of universal algebras in computational effects.

There is a very rich literature on frameworks of formalising effects in proof assistants, especially Coq, that we cannot provide a full survey here. Xia et al. [2020] used *interaction trees*, also known as *free completely iterative algebras* [Aczel et al. 2003] in category theory, which is a coinductive version of the *Syntax* monad to formalise effectful programs with recursion in Coq. In comparison, in our library recursive programs have to be formalised with a fuel argument. However, the strength of our library is that it avoids setoids, and it provides a generic Hoare logic for reasoning about effects more intuitively. Li and Weirich [2022] introduced and formalised *Tlön embeddings* and *program adverbs* which are a generalisation of free monads that encompass non-monadic notions of computational effects such as applicative functors. Saito and Affeldt [2022] formalised in Coq a hierarchy of monad classes and many applications of monadic equational reasoning.

Fiore et al. [2022] show that *quotient-inductive types* can be constructed in the type theory of toposes with natural numbers and universes satisfying the "Weakly Initial Set of Covers" axiom. In this setting, it would be unnecessary to require the arities type in *Term* to be quotient preserving. However, sets in HoTT do not form a topos but only a ΠW-pretopos so their construction does not apply. It is worth investigating whether their construction can be adapted to HoTT.

The agda-unimath library [Rijke et al. 2021] recently added a formalisation of universal algebra in HoTT independently of ours (and we would like to thank the reviewers for bringing it to our attention). Presently it is a relatively small formalisation and features a proof that the quotient of a model by a congruence relation is again a model, whereas our formalisation aims to be a full-fledged framework for verifying programs with algebraic effects.

Generic Hoare Logics. The Hoare logic in our library is based on Schröder and Mossakowski [2003]'s *monad-independent Hoare logic* formalised in the HasCasl specification language [Schröder and Mossakowski 2002]. Our new development is the fully constructive proof of the elimination principle of Hoare logic that we saw in Section 7.

A variation of the idea of encoding Hoare logic as program equations is proposed by Goncharov and Schröder [2013]. Under the assumption that there is a certain kind of CPO structure on the Kleisli arrows of a monad T, they devise a *relatively complete* Hoare logic for T-computations. In this Hoare logic, assertions are encoded as programs returning a unit value instead of a proposition; if the assertion is not met, the corresponding program should fail. The advantage of this framework is that this construction can be carried out on categories that do not have a universe of propositions.

Hasuo [2015] generalised Goncharov and Schröder [2013]'s construction by allowing weakest precondition transformation to be specified by any Eilenberg-Moore algebras satisfying certain conditions. This flexibility makes this framework expressive enough to cover both total correctness and partial correctness for various kinds of effects.

Aguirre and Katsumata [2020]'s fibrational framework seems to be the most general form of generic Hoare logic so far. In this framework, assertions no longer need to be programs, as long as there is a way to do weakest precondition transformation along Kleisli arrows—captured as the categorical concept *Grothendieck fibrations*. For example, "assertions" of the effect of probabilistic choice can be the remaining running time of a program, which is just a number, and the weakest precondition transformation computes the expectation of the running time.

Logic and Refinement. The idea of unifying Hoare-style reasoning and (in)equational reasoning has been proposed before, especially for separation logic [Frumin et al. 2021; Gäher et al. 2022; Liang and Feng 2016; Song et al. 2023; Turon et al. 2013]. Turon et al. [2013] introduce an expressive concurrent separation logic, *CaReSL*, with which program refinement can be encoded as certain Hoare triples. In contrast, our framework works with program equivalence rather than program refinement, and more importantly we take program equivalence as the primitive concept and Hoare logic as the derived concept. In this aspect, our framework is closer to Song et al. [2023]'s *conditional contextual refinement*, which takes program refinement as the basic concept and on which a notion $S \vdash e_1 \sqsubseteq e_2$ of a program e_1 refining e_2 under the pre-condition S is defined, where S is a separation logic formula. Such conditional refinement can be used to define Hoare triples. The main difference between Song et al. [2023]'s framework and ours is that Song et al. [2023] have fixed the effects (IO, mutable state, nondeterminism, function calls), and use a specific trace semantics of these effects to define the basic notion of program refinements, whereas our framework is completely generic to the effects, whose semantics is specified by an algebraic theory.

Dijkstra monads [Maillard et al. 2019; Silver and Zdancewic 2021; Swamy et al. 2016, 2013] are another approach to specifying and verifying monadic computations, using a separate monadic construction—the Dijkstra monad itself—to build and compose specifications over some base monad. Specification using Dijkstra monads allows the specification of arbitrary computations of type $M\ A$, but expects the specifier to define a bespoke "specification monad" for the particular effect in question. Our work, on the other hand, requires computations to be defined as algebraic effects, but provides a generic specification framework automatically. That said, it is possible to define a Dijkstra monad over an arbitrary *Term* monad, which would be worth exploring in future work.

9 CONCLUSION

This paper presents a formalisation of algebraic effects in Cubical Agda that is then extended with a Hoare logic. We hope that this library can be a useful building block for future programming language designers seeking to formalise the semantics of their languages with effects in Agda, as well as programmers who would like to verify their effectful programs.

Future work abounds. One direction is to improve the support for recursive programs. A promising approach is to adopt Clocked Cubical Type Theory [Baunsgaard Kristensen et al. 2022], which provides guarded recursion and coinductive counterparts of higher-inductive types.

Another direction is to support more general forms of algebraic effects such as *parameterised algebraic effects* [Staton 2013, 2015] for dynamically created instances of effects, *scoped effects* for operations that delimit scopes [Piróg et al. 2018; Wu et al. 2014; Yang et al. 2022], *latent effects* for effects which incorporate advanced control flow mechanisms, [van den Berg et al. 2021], and their generalisations into monoids with operations [Yang and Wu 2023].

Also, it is worthwhile to extend the effect-generic Hoare logic. Currently the logic is only for partial correctness, but it should be relatively easy to adapt the Hoare logic to account for total correctness. Also, we would like to investigate *separation logic* [Reynolds 2002b] in the effect-generic setting and look for an axiomatisation of algebraic effects that admit reasoning by separation logic.

Finally, there is still a lot of room for improvement regarding the user experience of program verification with our library. The speed of type checking in Agda 2.6.2 can sometimes be quite slow when the definitions get complicated, which can probably be mitigated using *controlled unfolding* [Gratzer et al. 2022] that is already available in Agda 2.6.4. Moreover, many simple equational rewriting steps are still needed when verifying a program in our framework. In the future we plan to resolve this by incorporating solvers for decidable algebraic theories into our framework.

With these future prospects in mind, we anticipate that our work can be the starting point of a powerful and easy to use program verification framework that reaps the rewards of the advancements in modern type theory and denotational semantics.

REFERENCES

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science* 342, 1 (Sept. 2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Andreas Abel. 2021. Birkhoff's Completeness Theorem for Multi-Sorted Algebras Formalized in Agda. https://doi.org/10.48550/arXiv.2111.07936 arXiv:2111.07936 [cs]

Peter Aczel, Jiří Adámek, Stefan Milius, and Jiří Velebil. 2003. Infinite Trees and Completely Iterative Theories: A Coalgebraic View. *Theoretical Computer Science* 300, 1 (May 2003), 1–45. https://doi.org/10.1016/S0304-3975(02)00728-4

Alejandro Aguirre and Shin-ya Katsumata. 2020. Weakest Preconditions in Fibrations. *Electronic Notes in Theoretical Computer Science* 352 (Oct. 2020), 5–27. https://doi.org/10.1016/j.entcs.2020.09.002

Danel Ahman. 2017. Handling Fibred Algebraic Effects. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 7 (Dec. 2017), 7:1–7:29 pages. https://doi.org/10.1145/3158095

Carlo Angiuli, Guillaume Brunerie, Thierry Coquand, Robert Harper, Kuen-Bang Hou (Favonia), and Daniel R. Licata. 2021. Syntax and Models of Cartesian Cubical Type Theory. *Mathematical Structures in Computer Science* 31, 4 (April 2021), 424–468. https://doi.org/10.1017/S0960129521000347

Steve Awodey, Nicola Gambino, and Kristina Sojakova. 2017. Homotopy-Initial Algebras in Type Theory. *J. ACM* 63, 6 (Jan. 2017), 51:1–51:45. https://doi.org/10.1145/3006383

Andrej Bauer. 2019. What Is Algebraic about Algebraic Effects and Handlers? *arXiv:1807.05923 [cs]* (March 2019). arXiv:1807.05923 [cs] http://arxiv.org/abs/1807.05923

Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science (Lecture Notes in Computer Science)*, Reiko Heckel and Stefan Milius (Eds.). Springer, Berlin, Heidelberg, 1–16. https://doi.org/10.1007/978-3-642-40206-7_1

Magnus Baunsgaard Kristensen, Rasmus Ejlers Mogelberg, and Andrea Vezzosi. 2022. Greatest HITs: Higher Inductive Types in Coinductive Definitions via Induction under Clocks. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 42. https:

//doi.org/10.1145/3531130.3533359

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2017. Handle with Care: Relational Interpretation of Algebraic Effects and Handlers. *Proceedings of the ACM on Programming Languages* 2, POPL (Dec. 2017), 8:1–8:30. https://doi.org/10.1145/3158096

Garrett Birkhoff. 1935. On the Structure of Abstract Algebras. *Mathematical Proceedings of the Cambridge Philosophical Society* 31, 4 (Oct. 1935), 433–454. https://doi.org/10.1017/S0305004100013463

Errett Bishop and Douglas Bridges. 1985. *Constructive Analysis.* Grundlehren Der Mathematischen Wissenschaften, Vol. 279. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-61667-9

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35.

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2018. Effect Handlers for the Masses. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 111 (Oct. 2018). https://doi.org/10.1145/3276481

Venanzio Capretta. 1999. Universal Algebra in Type Theory. In *Theorem Proving in Higher Order Logics*, Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 131–148.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *Leibniz International Proceedings in Informatics, LIPIcs* 69 (2018), 51–534. https://doi.org/10.4230/LIPIcs.TYPES.2015.5 arXiv:1611.02108

P.M. Cohn. 1981. *Universal Algebra.* Springer Netherlands.

Thierry Coquand, Bassel Mannaa, and Fabian Ruch. 2017. Stack Semantics of Type Theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS).* 1–11. https://doi.org/10.1109/LICS.2017.8005130

Tom de Jong and Martín Hötzel Escardó. 2021. Domain Theory in Constructive and Predicative Univalent Foundations. In *29th EACSL Annual Conference on Computer Science Logic (CSL 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 183)*, Christel Baier and Jean Goubault-Larrecq (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 28:1–28:18. https://doi.org/10.4230/LIPIcs.CSL.2021.28

William DeMeo and Jacques Carette. 2022. A Machine-Checked Proof of Birkhoff's Variety Theorem in Martin-Löf Type Theory. In *27th International Conference on Types for Proofs and Programs (TYPES 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 239)*, Henning Basold, Jesper Cockx, and Silvia Ghilezan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:21. https://doi.org/10.4230/LIPIcs.TYPES.2021.4

Marcelo P. Fiore, Andrew M. Pitts, and S. C. Steenkamp. 2022. Quotients, Inductive Types, and Quotient Inductive Types. *Logical Methods in Computer Science* Volume 18, Issue 2, arXiv:2101.02994 (June 2022). https://doi.org/10.46298/lmcs-18(2:15)2022 arXiv:2101.02994 [cs]

Dan Frumin, Herman Geuvers, Léon Gondelman, and Niels van der Weide. 2018. Finite Sets in Homotopy Type Theory. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2018).* ACM, New York, NY, USA, 201–214. https://doi.org/10.1145/3167085

Dan Frumin, Robbert Krebbers, and Lars Birkedal. 2021. ReLoC Reloaded: A Mechanized Relational Logic for Fine-Grained Concurrency and Logical Atomicity. *Logical Methods in Computer Science* Volume 17, Issue 3 (July 2021). https://doi.org/10.46298/lmcs-17(3:9)2021

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: A Separation Logic Framework for Verifying Concurrent Program Optimizations. *Proc. ACM Program. Lang.* 6, POPL, Article 28 (Jan. 2022). https://doi.org/10.1145/3498689

Dan Ghica, Sam Lindley, Marcos Maroñas Bravo, and Maciej Piróg. 2022. High-Level Effect Handlers in C++. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 183 (Oct. 2022). https://doi.org/10.1145/3563445

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 (ICFP '11).* Association for Computing Machinery, New York, NY, USA, 2–14. https://doi.org/10.1145/2034773.2034777

Sergey Goncharov and Lutz Schröder. 2013. A Relatively Complete Generic Hoare Logic for Order-Enriched Effects. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science.* 273–282. https://doi.org/10.1109/LICS.2013.33

Daniel Gratzer, Jonathan Sterling, Carlo Angiuli, Thierry Coquand, and Lars Birkedal. 2022. Controlling unfolding in type theory. https://doi.org/arXiv.2210.05420 arXiv:2210.05420 [cs.LO]

Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. 2018. Formalization of Universal Algebra in Agda. *Electronic Notes in Theoretical Computer Science* 338 (Oct. 2018), 147–166. https://doi.org/10.1016/j.entcs.2018.10.010

Ichiro Hasuo. 2015. Generic Weakest Precondition Semantics from Monads Enriched with Order. *Theoretical Computer Science* 604, April 2014 (2015), 2–29. https://doi.org/10.1016/j.tcs.2015.03.047

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Communications of The Acm* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Graham Hutton and Erik Meijer. 1998. Monadic Parsing in Haskell. *Journal of Functional Programming* 8, 4 (July 1998), 437–444. https://doi.org/10.1017/S0956796898003050

Martin Hyland, Paul Blain Levy, Gordon Plotkin, and John Power. 2007. Combining Algebraic Effects with Continuations. *Theoretical Computer Science* 375, 1 (May 2007), 20–40. https://doi.org/10.1016/j.tcs.2006.12.026

Martin Hyland, Gordon Plotkin, and John Power. 2006. Combining Effects: Sum and Tensor. *Theoretical Computer Science* 357, 1-3 (July 2006), 70–99. https://doi.org/10.1016/j.tcs.2006.03.013

G.M. Kelly and A.J. Power. 1993. Adjunctions Whose Counits Are Coequalizers, and Presentations of Finitary Enriched Monads. *Journal of Pure and Applied Algebra* 89, 1-2 (Oct. 1993), 163–179. https://doi.org/10.1016/0022-4049(93)90092-8

Donnacha Oisín Kidney. 2020. *Finiteness in Cubical Type Theory.* MRes Thesis. University College Cork, Cork, Ireland. https://cora.ucc.ie/handle/10468/11338

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (June 2014), 100–126. https://doi.org/10.4204/eptcs.153.8

Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

Yao Li and Stephanie Weirich. 2022. Program Adverbs and Tlön Embeddings. *Proc. ACM Program. Lang.* 6, ICFP, Article 101 (Aug. 2022). https://doi.org/10.1145/3547632

Hongjin Liang and Xinyu Feng. 2016. A Program Logic for Concurrent Objects under Fair Scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16).* Association for Computing Machinery, New York, NY, USA, 385–399. https://doi.org/10.1145/2837614.2837635

Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. 2019. Dijkstra Monads for All. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 104 (July 2019), 29 pages. https://doi.org/10.1145/3341708 arXiv:1903.01237

Per Martin-Löf. 1982. Constructive Mathematics and Computer Programming. In *Studies in Logic and the Foundations of Mathematics.* Vol. 104. North-Holland Publishing Company, 153–175. https://doi.org/10.1016/S0049-237X(09)70189-2

Eugenio Moggi. 1989. *An Abstract View of Programming Languages.* Technical Report ECS-LFCS-90-113. University of Edinburgh. 51 pages. http://www.lfcs.inf.ed.ac.uk/reports/90/ECS-LFCS-90-113/

Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (July 1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

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 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.).

Gordon Plotkin and John Power. 2002. Notions of Computation Determine Monads. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Mogens Nielsen and Uffe Engberg (Eds.). Springer, Berlin, Heidelberg, 342–356. https://doi.org/10.1007/3-540-45931-6_24

Gordon Plotkin and John Power. 2004. Computational Effects and Operations: An Overview. *Electronic Notes in Theoretical Computer Science* 73 (Oct. 2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008

Gordon Plotkin and Matija Pretnar. 2008. A Logic for Algebraic Effects. In *2008 23rd Annual IEEE Symposium on Logic in Computer Science.* IEEE, Pittsburgh, PA, USA, 118–129. https://doi.org/10.1109/LICS.2008.45

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

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* 9, 4 (Dec. 2013), 23. https://doi.org/10.2168/LMCS-9(4:23)2013 arXiv:1312.1399

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

J.C. Reynolds. 2002a. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* 55–74. https://doi.org/10.1109/LICS.2002.1029817

J.C. Reynolds. 2002b. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science.* 55–74. https://doi.org/10.1109/LICS.2002.1029817

Egbert Rijke, Elisabeth Bonnevier, Jonathan Prieto-Cubides, Fredrik Bakke, et al. 2021. The Agda-Unimath Library. https://github.com/UniMath/agda-unimath/

Ayumu Saito and Reynald Affeldt. 2022. Towards a Practical Library for Monadic Equational Reasoning in Coq. In *Mathematics of Program Construction*, Ekaterina Komendantskaya (Ed.). Springer International Publishing, Cham, 151–177.

Lutz Schröder and Till Mossakowski. 2002. HasCasl: Towards Integrated Specification and Development of Functional Programs. In *Algebraic Methodology and Software Technology*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Hélène Kirchner, and Christophe Ringeissen (Eds.). Vol. 2422. Springer Berlin Heidelberg, Berlin, Heidelberg, 99–116.

https://doi.org/10.1007/3-540-45719-4_8

Lutz Schröder and Till Mossakowski. 2003. Monad-Independent Hoare Logic in HASCASL. In *Fundamental Approaches to Software Engineering*, Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Mauro Pezzè (Eds.). Vol. 2621. Springer Berlin Heidelberg, Berlin, Heidelberg, 261–277. https://doi.org/10.1007/3-540-36578-8_19

Philipp Schuster, Jonathan Immanuel Brachthäuser, Marius Müller, and Klaus Ostermann. 2022. A Typed Continuation-Passing Translation for Lexical Effect Handlers. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 566–579. https://doi.org/10.1145/3519939.3523710

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proceedings of the ACM on Programming Languages* 4, ICFP, Article 93 (Aug. 2020), 93:1–93:28 pages. https://doi.org/10.1145/3408975

Lucas Silver and Steve Zdancewic. 2021. Dijkstra Monads Forever: Termination-Sensitive Specifications for Interaction Trees. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan. 2021), 26:1–26:28. https://doi.org/10.1145/3434307

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. https://doi.org/10.1145/3453483.3454039

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 39:1121–39:1151. https://doi.org/10.1145/3571232

Sam Staton. 2013. Instances of Computational Effects: An Algebraic Perspective. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. 519–519. https://doi.org/10.1109/LICS.2013.58

Sam Staton. 2015. Algebraic Effects, Linearity, and Quantum Programming Languages. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 395–406. https://doi.org/10.1145/2676726.2676999

Jonathan Sterling and Robert Harper. 2018. Guarded Computational Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. Association for Computing Machinery, New York, NY, USA, 879–888. https://doi.org/10.1145/3209108.3209153

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 2016. Dependent Types and Multi-Monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. Association for Computing Machinery, New York, NY, USA, 256–270. https://doi.org/10.1145/2837614.2837655

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-Order Programs with the Dijkstra Monad. *ACM SIGPLAN Notices* 48, 6 (June 2013), 387–398. https://doi.org/10.1145/2499370.2491978

Aaron Turon, Derek Dreyer, and Lars Birkedal. 2013. Unifying Refinement and Hoare-Style Reasoning in a Logic for Higher-Order Concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 377–390. https://doi.org/10.1145/2500365.2500600

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

Birthe van den Berg, Tom Schrijvers, Casper Bach Poulsen, and Nicolas Wu. 2021. Latent Effects for Reusable Language Components. In *Programming Languages and Systems: 19th Asian Symposium, APLAS 2021, Chicago, IL, USA, October 17–18, 2021, Proceedings* (Chicago, IL, USA). Springer-Verlag, Berlin, Heidelberg, 182–201. https://doi.org/10.1007/978-3-030-89051-3_11

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. *Proc. ACM Program. Lang.* 3, ICFP (July 2019), 87:1–87:29. https://doi.org/10.1145/3341691

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2633357.2633358

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction Trees: Representing Recursive and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–32. https://doi.org/10.1145/3371119

Zhixuan Yang, Marco Paviotti, Nicolas Wu, Birthe van den Berg, and Tom Schrijvers. 2022. Structured Handling of Scoped Effects. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Vol. 13240. Springer International Publishing, Cham, 462–491. https://doi.org/10.1007/978-3-030-99336-8_17

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

Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal Reasoning about Layered Monadic Interpreters. *Proceedings of the ACM on Programming Languages* 6, ICFP (Aug. 2022), 99:254–99:282. https://doi.org/10.1145/3547630

Brent A. Yorgey. 2014. *Combinatorial Species and Labelled Structures.* Ph. D. Dissertation. University of Pennsylvania, Pennsylvania. https://repository.upenn.edu/dissertations/AAI3668177

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-Safe Effect Handlers via Tunneling. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 29. https://doi.org/10.1145/3290318