Types with Extra Structure: Predicates, Equations, Composition

Brandon Hewer

A thesis presented for the degree of
Doctor of Philosophy

School of Computer Science
University of Nottingham

April 2024

Abstract

Intuitionistic type theory was first introduced by Martin-Lof [1984] as a foundation for constructive mathematics and also serves as a dependently typed programming language. Dependent types provide us with a framework to reason about and guide the construction of programs by specifying both their structure and properties in a manner that can be automatically verified by a type-checker.

A ubiquitous pattern that arises in the formulation of dependent type abstractions involves equipping an underlying type, which captures the general form of a program, with extra structure that captures the program's properties. Two such type abstractions include subtypes in which a type is equipped with a *predicate* over its values and quotient types in which a type is equipped with *equations* over its values. While subtypes have found much practical use in general purpose programming, quotient types have not seen many applications outside of proof assistants. Two key obstacles to the wider adoption of quotient types include an absence of practical demonstrations of their applications to general purpose programming and the significant burden of proof-obligations that arises from their use.

In this thesis, we introduce three new applications of type theoretic concepts that involve equipping types with extra structure. Firstly, we introduce a new practical application for higher-inductive types whereby they are used to encode subtypes in a manner that grants fine-grained control over the reduction behaviour of terms. Our second key contribution is the extension of a liquid type system to include a class of quotient types for which the necessary proof-obligations are decidable by an SMT-solver. This work is accompanied by a practical demonstration in the form of Quotient Haskell, which was developed as an extension to the liquid type system of Liquid Haskell. Finally, we present a constructive theory of operads, which were first introduced by Peter May to describe composable algebraic structures in symmetric monoidal structures. Intuitively, an operad can be understood as a finite family of types equipped with a well-behaved notion of *composition*. We demonstrate how an internalisation of the theory of operads in homotopy type theory gives rise to a generic framework for capturing and reasoning about collections of operations.

Acknowledgements

I would first like to thank my supervisor, Graham Hutton, whose encouragement and enthusiasm over the past four years have made my PhD journey both memorable and enjoyable. I am grateful not only for his guidance in the development of this thesis but also for the great many professional lessons I have learned from him. His insights on technical writing, effective communication of ideas, and consideration for the practical implications of my work are lessons I hope to carry with me throughout my future career.

I would also like to thank my examiners, Martín Escardó and Ulrik Buchholtz for a pleasant and engaging viva. I am grateful for all of their useful suggestions and comments.

I was fortunate to be surrounded by wonderful staff and peers at the University of Nottingham and would like to thank every member of the FP Lab for the insightful discussions and talks over the past few years. I am particularly grateful to my fellow PhD students, Filipo Sestini and Joshua Chen, with whom I spent countless hours discussing and exploring ideas in type theory. Special thanks also go to Nicolai Kraus and Thorsten Altenkirch for their invaluable teaching and support in type theory for both myself and many others in the FP Lab.

I am grateful to everyone in the programming languages and type theory research communities whom I've had the pleasure of meeting and discussing with at various events over the past few years. In particular, I want to thank Ambrus Kaposi and Patrick Bahr for their valuable comments on areas of work that are included in this thesis.

Finally, I would like to thank my family for all of their support, not only over the duration of my PhD but through each major step in my life. Thank you to my parents, Daegon and Jane, my sister, Chloe, and the newest members of our family, my nephews, Ares and Asher.

Contents

Chapter 1

Introduction

Two fundamental principles of programming are clarity and correctness. That is, programmers will often strive to ensure their programs have a clear, communicable purpose while accurately serving their intended function. To assist with these principles, many programming languages come equipped with type systems of varying complexity.

Types capture collections of terms that share a distinguished property. In turn, a type system is a logical framework that determines precisely how types are assigned to terms. For example, the type systems of many prominent programming languages include a 'type of integers' that ranges over arithmetic expressions built from integral variables and constants. In the case of integers, the property that we often wish to capture is that upon running our program every term that is typed as an integer is evaluated to an integral constant. Type systems are present in many prominent programming languages including the simply typed C, the object-oriented Java, the functional language Haskell, and the dependently typed Agda.

A key property of many type systems is decidability of type-checking. Put simply, it is desirable for a system of typing rules to be automatically checked within a compiler or similar automated reasoning tool. Indeed, for the general purpose programmer this is precisely how a type system can be seen to address the issue of program correctness. With a sufficiently expressive type system and decidable type-checking, a wide range of program properties can be verified by automated tools. A prominent example of such a type system is Per Martin Löf's intensional dependent type theory [Martin-Löf, 1975], of which several dependently typed languages such as Agda [Norell, 2007] are based upon.

For the purposes of both formalising constructive mathematics and capturing a more expansive collection of program properties, type theorists have developed increasingly expressive type systems. Examples include the Calculus of Inductive Constructions implemented by Coq [Coquand and Paulin, 1990; Bertot and Castéran, 2013] and the Quantitative Type Theory implemented by Idris [Atkey, 2018; Brady, 2013]. Included amongst these examples are type systems that implement ideas from Homotopy Type Theory which itself builds upon ideas introduced in Voevodsky's Univalent Foundations of Mathematics [UniMath, 2021]. Homotopy Type Theory can be informally understood as a version of intuitionistic type theory in which the equality between two types is internally characterised by the type of equivalences between them.

Unfortunately, the increased expressivity of type systems has frequently come at a significant cost in both the performance of type-checking and the complexity of use for practical programming. This is especially notable in existing implementations of Homotopy Type Theory such as Cubical Agda [Vezzosi et al., 2019]. In particular, the interval bounds checking of Cubical Agda can notoriously result in unreasonable compile times without employing specialized techniques. As a consequence of these drawbacks, dependent types along with a variety of useful concepts in type theory are rarely used for general purpose programming. Instead, features of type systems such as coinduction and quotient inductive types are instead primarily used for the formalisation of mathematics within proof assistants.

This thesis contains three related bodies of work that focus on applying key ideas from homotopy type theory to practical programming. In Chapter 2 we introduce a new practical application for higher-inductive types, in which they are used to encode subtypes in a manner that grants fine-grained control over the reduction behaviour of the subtyping condition. In Chapter 3 we show how a liquid type system can be extended to include a class of quotient inductive types with implicit subtyping and for which the necessary proof-obligations are decidable by an SMT-solver. Finally, in Chapter 4 we present a constructive theory of operads internal to homotopy type theory, that presents a generic framework for reasoning about collections of operations.

1.1 Chapter summaries

This thesis is organised in three parts alongside a conclusion, with each part comprising its own chapter. The contents of Chapters 2, 3 and 4 each constitute a complete article and each includes a review of the relevant literature. The publications corresponding to each chapter are listed in Section 1.3. The chapters of this thesis are organised as follows:

Chapter 2 presents an encoding for subtypes and subtyping relations that gives fine grained control over the reduction behaviour of functions defined on them. The general form of this technique is presented in the meta-theory of Cubical Agda by exploiting its support for higher-inductive types.

Chapter 3 presents a refinement type system with support for quotient types for which the corresponding respectfulness theorems can be decided by an SMT-solver. This type system is demonstrated in practice by *Quotient Haskell* which is an extension of Liquid Haskell with support for quotient types.

Chapter 4 presents a constructive theory of operads in the internal language of homotopy type theory. This internal theory of operads gives rise to a generic framework for reasoning about distinguised collections of operations and their algebraic properties. These ideas are demonstrated and formalised in the meta-theory of Cubical Agda.

Chapter 5 presents a high level conclusion and summary of the contributions of this thesis. Moreover, individual conclusions are provided in each of the chapters detailed above.

1.2 Contributions

A more detailed overview of the contributions of this thesis can be organised by chapter. In particular, in Chapter 2 we make the following contributions:

- We introduce and formalise a general technique for translating a subtype with given operations into two representations that are isomorphic to the original subtype, but avoid the need to compute the proof of the subtyping condition for the operations;

- We discuss the differences between the two representations, compare their advantages and disadvantages, and provide practical examples of each;

- We describe a generalisation of our method for $\sum$-types in which the second component is an arbitrary type, rather than just a proposition.

In Chapter 3 we make the following contributions:

- We present an approach to extending a liquid type system with quotient inductive types whose corresponding respectfulness theorems can be automatically checked by an SMT solver.

- We discuss the key ideas through examples such as mobiles, the Boom hierarchy and rational numbers. These examples are demonstrated in Quotient Haskell, which is an implementation of the presented ideas that extends Liquid Haskell;

- We present a core language λ_Q for quotient types, by extending the core language λ_L for liquid types with typing (Section 3.5) and subtyping (Section 3.6) rules for quotients;

- We show how the notion of equality in the underlying liquid type system can be extended in λ_Q to make use of the equalities introduced by quotients (Section 3.7);

- We outline how Quotient Haskell is implemented, with a particular focus on how the new quotient typing features are realised (Section 3.9).

Finally, in Chapter 4, we make the following contributions:

- Internalise the notion of both planar and non-planar operads in homotopy type theory, and provide practical examples of each form;

- Demonstrate how the free operad can be constructed as a higher inductive family, and provide examples of how this construction can be used;

- Prove that every operad gives rise to a canonical monad, that consequently presents an *operadic* style of constructing programs from a small collection of composable terms.

1.3 Publications

As previously detailed in Section 1.1, the contents of this thesis is comprised of three bodies of work each of which corresponds to a publication:

Chapter 2: Brandon Hewer and Graham Hutton. 2022. Subtyping Without Reduction. In Mathematics of Program Construction, Ekaterina Komendantskaya (Ed.). Springer International Publishing, Cham, 34–61.

Chapter 3: Brandon Hewer and Graham Hutton. 2024. Quotient Haskell: Lightweight Quotient Types for All. Proc. ACM Program. Lang. 8, POPL, Article 27 (Jan 2024), 31 pages. https://doi.org/10.1145/3632869

Chapter 4: Brandon Hewer and Graham Hutton. 2024. HoTT Operads. Available online: `http://www.cs.nott.ac.uk/~pszgmh/bib.html#operads`

I am the principle author of each publication.

1.4 Background

The key field of study to which this thesis contributes is *type theory*. More specifically, this thesis builds upon and makes extensive use of ideas from both *homotopy type theory* [Univalent goundations Program, 2013] and *refinement types* [Rushby et al., 1998]. Chapter 3 assumes basic knowledge of Haskell and its syntax. Moreover, category-theoretic constructions [Eilenberg and MacLane, 1945] are used throughout Chapters 2 and 4. We assume basic knowledge of intuitionistic dependent type theory throughout this thesis, however, constructions specific to both category theory and homotopy type theory are introduced when they are first required.

1.5 Path types

Of key importance in homotopy type theory is the notion of an internal equality type, also known as a *path type*. The particular semantics of path types varies in different models of HoTT. For example, in the CCHM model of cubical type theory [Cohen et al., 2016], a path between terms of a type A corresponds to a continuous function from the real interval $[0, 1]$ to A. In Chapters 2 and 4, key ideas and formalisations are presented in the metatheory of Cubical Agda which is an implementation of HoTT based upon the CCHM model. In this section we present a review of path types and describe how they are represented in Cubical Agda.

In Cubical Agda, for any type A, we can construct the type $x \equiv y$ of *paths* between two terms $x, y : A$. A path of type $x \equiv y$ can be constructed as a function out of the non-fibrant interval type I that must map the endpoints $i0, i1 : I$ to x, y respectively. For example, the identity on x can simply be constructed as $\lambda\ i \to x$. Readers unfamiliar with HoTT can think of the path type $x \equiv y$ as behaving identically to the inductively defined identity type without the K-rule. In particular, the usual eliminator for path types is given by the J-rule, which states that for any term $x : A$ and family of types $M : (y : A) \to x \equiv y \to \text{Type}$, if we have a proof $t : M\ x\ \textbf{refl}$ then for all $y : A$ and $p : x \equiv y$ we can construct a term $J_{M,t}\ x\ y\ p : M\ y\ p$. Intuitively, the J-rule states that if the end-point y of the path p can vary, then we can substitute p for reflection on x.

Cubical Agda also provides a primitive construction for heterogeneous path types which are presented in the form PathP $(\lambda\ i \to T)\ a\ b$ for terms $a : A$, $b : B$ and T : Type. The first argument to PathP is a continuous function out of the built-in interval type I for which the endpoints i0 i1 : I must be mapped to a and b respectively and such that $a : T[i \mapsto \text{i0}]$, $b : T[i \mapsto \text{i1}]$ are verifiable typing judgements.

Chapter 2

Subtyping Without Reduction

In a typed programming language, *subtyping* is a relationship between types that describes when terms of one type can be considered as of terms of another without changing the meaning of a program. In this way, subtyping is often introduced as a language feature that improves code reusability in a sound manner. Various modern programming languages support a form of subtyping such as the nominal subtyping of object-oriented languages or the structural subtyping of refinement type systems.

One approach to subtyping is to support a mechanism by which types can be equipped with additional properties or *predicates* that their terms must conform to. For example, the type of even numbers can be defined by equipping the type of integers with the property that each number must be divisible by two. This flavor of subtyping is present in both homotopy type theory and refinement type systems such as Liquid Haskell. When adopting this form of subtyping, a type checker must verify that terms satisfy any predicate that is equipped to their asserted type. The performance cost of this additional verification is highly variable and can require normalisation or *reduction* of large programs.

In this chapter, we describe a technique in homotopy type theory for encoding subtypes in a manner that avoids unnecessarily verifying the subtyping condition for a chosen collection of operations. In particular, this technique can significantly improve the performance of type-checking by providing fine-grained control over the reduction behaviour of terms.

2.1 Subtypes in type theory

Before we introduce our encoding, we begin by reviewing the type-theorteic notion of a subtype. In type theory, a subtype of a type A is characterised by a dependent sum over a family of propositions $P : A \to \text{Prop}$. For example, the even natural numbers can easily be seen as a subtype of the naturals by defining a family $\text{isEven} : \mathbb{N} \to \text{Prop}$ which maps every even number to true, and every odd number to false. Another ubiquitous example is that of the (totally-ordered) finite sets $\text{Fin} : \mathbb{N} \to \text{Type}$, which can be defined as subtypes of the natural numbers by means of the family $<: \mathbb{N} \to \mathbb{N} \to \text{Prop}$.

Operations defined over subtypes must respect the subtyping condition. For example, addition restricts to an operation over even numbers because the addition of two even numbers is even. As a more complex example, the dependent sum over a family of finite types indexed by a finite type will always be finite for some sensible notion of 'finite'. In type theory, proving that an operation respects a subtyping condition involves computing a term of the respective proposition. For addition of even numbers, this means that given terms of the propositions $\text{isEven}\,(m)$ and $\text{isEven}\,(n)$, we would compute a term of $\text{isEven}\,(m + n)$.

In practice, computing subtyping proofs can be costly, and in a dependently typed setting this has performance consequences not just at runtime, but also during type checking. The impact of this problem can even be seen in simple examples such as addition on finite sets. In particular, the addition operation $+ : \text{Fin}\ m \to \text{Fin}\ n \to \text{Fin}\ \ (m + n)$ requires a proof that if $x < m$ and $y < n$ then $x + y < m + n$, which typically proceeds by induction, and therefore the use of $+$ may result in reduction taking place during type checking.

It is natural to see this issue as part of a much larger problem in type theory: proofs whose content does not matter can drastically slow down type-checking as a result of reduction behaviour. This problem is so prevalent that many dependently typed languages have special features for addressing it, such as Agda's abstract definition mechanism. However, these language-specific features usually come with their own problems, which we discuss in Section 2.5.

In this chapter, we introduce a technique that differs from existing solutions as it does not require special-purpose language extensions, and can be used in any implementation of type-theory that supports quotient inductive types. In addition, we retain important computational properties that are absent in solutions such as Agda's abstract definitions. More specifically, this chapter makes the following contributions:

- We introduce and formalise a general technique for translating a subtype with given operations into two representations that are isomorphic to the original subtype, but avoid the need to compute the proof of the subtyping condition for the operations;

- We discuss the differences between the two representations, compare their advantages and disadvantages, and provide practical examples of each;

- We describe a generalisation of our method for $\sum$-types in which the second component is an arbitrary type, rather than just a proposition.

All examples in this chapter have been formalised in a Cubical Agda [Vezzosi et al., 2019] library that is available online [Hewer, 2022].

2.2 Example: even numbers

In this section we introduce the basic type-theoretic encoding of even natural numbers, which is used as the motivating example for the application of our technique. In (Cubical) Agda we can define a recursive family of types that witness whether a given natural number is even as follows:

```
isEven : ℕ → Type
isEven 0 = ⊤
isEven 1 = ⊥
isEven (suc (suc n)) = isEven n
```

Note that the definition uses Type rather than Prop, because in HoTT the definition of a subtype merely requires the subtyping condition to be a family of weak propositions, i.e. *h-propositions*, a family of types for which any two inhabitants are propositionally equal. We can observe that isEven is such a family because the resulting type is always either the singleton type ⊤ or the empty type ⊥.

We can also define isEven in a different but equivalent way as an inductive family, because a proof of evenness can be uniquely constructed from a proof that 0 is even and a proof that if n is even then so is $n + 2$. We can then introduce these proofs as constructors even-z and even-ss of an inductive family:

```
data isEven : ℕ → Type where
  even-z : isEven 0
  even-ss : isEven n → isEven (suc (suc n))
```

This translation from a family of propositions to an inductive family is necessary for applying our technique. It is always possible for an arbitrary family P : A → Type, by defining an inductive family Id_P : A → Type with a single constructor η_P : $(a : A)$ → P a → Id_P a. However, as illustrated above, we can often inline the definition of a propositional family as the constructors of an inductive family.

As with any type, we are also interested in the *operations* that can be used to construct terms of a subtype. Concretely, such an operation comprises a function that constructs a term of the underlying type, together with a proof that this term is an element of the subtype. We will often refer to the proof that an operation preserves a particular subtyping condition as a *closure property* of that condition. For example, one such operation on even numbers is addition, whose closure property we can construct in Agda as follows:

```
isEven+ : isEven m → isEven n → isEven (m + n)
isEven+ even-z q = q
isEven+ (even-ss p) q = even-ss (isEven+ p q)
```

We can think of isEven+ as a *non-canonical* constructor of isEven. Such constructors exhibit reduction behaviour by unfolding definitional equalities. In practice, the use of these constructors may have a significant impact on the performance of type-checking, due to an arbitrary number of reduction steps taking place.

2.3 Higher-inductive evenness

In this section we introduce our first approach to solving the above problem, which is based on the use of higher-inductive families. As an initial step, we might consider defining a new inductive family isEven?, by simply adding the proof that addition preserves evenness as a constructor:

```
data isEven? : ℕ → Type where
  even-z : isEven? 0
  even-ss : isEven? n → isEven? (suc (suc n))
  even-+ : isEven? m → isEven? n → isEven? (m + n)
```

However, isEven? and isEven are not isomorphic families, and hence cannot be used interchangeably. This is evident by observing that isEven? is not a family of propositions. For example, the type isEven? 0 is inhabited by the (provably) distinct terms even-z and even-+ 0 0.

Fortunately, there is a simple way to modify isEven? to obtain the desired isomorphism, by exploiting *higher-inductive* families [Univalent goundations Program, 2013]. These generalise inductive families by introducing the notion of *path constructors*, which internalise the idea of a (higher) quotient in type theory. While a data constructor for an inductive type A introduces a term of type A, a path constructor introduces a path of one of the iterated path types on A, e.g. $x \equiv y$ for terms $x, y : A$, or $p \equiv q$ for paths $p, q : x \equiv y$. In Cubical Agda, we can quotient our current definition of isEven? to obtain a family of propositions as follows:

```
data isEven! : ℕ → Type where
  η : isEven? n → isEven! n
  squash : (x y : isEven! n) → x ≡ y
```

That is, isEven! is given by *propositionally truncating* the family isEven?, where the path constructor squash asserts that all elements of isEven! n must be treated identically. As such, the eliminator for isEven! requires that the type being eliminated into is a proposition, thus ensuring all terms of isEven! n are mapped to provably equal terms. Formally, this means that for any family of h-propositions B : isEven! n → Type we can lift a function f : (x : isEven? n) → B (η x) on isEven? n to a function g : (x : isEven! n) → B x on isEven! n.

For example, we can use this eliminator to construct a function from isEven! n to isEven n. In this case, we begin by defining B to be the constant family choosing the proposition isEven n. The function g : isEven n → isEven? n maps each constructor of isEven n to its namesake, and f :

$\mathsf{isEven?}\ n \to \mathsf{isEven}\ n$ behaves similarly while mapping $\eta\,(\mathsf{even\text{-}+}\ p\ q)$ to $\mathsf{isEven+}\ \ (f\ p)\ \ (f\ q)$. Given that $\mathsf{isEven}\ n$ and $\mathsf{isEven!}\ n$ are provably propositions, our construction of a function in both directions is enough to establish an isomorphism.

Given two isomorphic types, it is natural to consider how they compare. Recall that $\mathsf{isEven!}$ encodes the proof that addition preserves evenness as a canonical constructor, which maps proofs $p : \mathsf{isEven!}\ m$ and $q : \mathsf{isEven!}\ n$ that m and n are even to a proof $\eta\,(\mathsf{even\text{-}+}\ p\ q)\ :\ \mathsf{isEven!}\,(m+n)$ that their addition is even. Intuitively, we can understand this encoding as hiding the definitional equalities introduced by the function $\mathsf{isEven+}$, and instead presenting them only as propositional equalities. Namely, while the equations

$$\begin{aligned}
\mathsf{isEven+}\ \mathsf{even\text{-}z}\ q \quad &= \quad q \\
\mathsf{isEven+}\,(\mathsf{even\text{-}ss}\ p)\ \ q \quad &= \quad \mathsf{even\text{-}ss}\,(\mathsf{isEven+}\ p\ q)
\end{aligned}$$

hold definitionally, i.e. they are the defining equations for $\mathsf{isEven+}$, the equalities

$$\begin{aligned}
\eta\,(\mathsf{even\text{-}+}\ \mathsf{even\text{-}z}\ q) \quad &\equiv \quad \eta\ q \\
\eta\,(\mathsf{even\text{-}+}\,(\mathsf{even\text{-}ss}\ p)\ \ q) \quad &\equiv \quad \eta\,(\mathsf{even\text{-}ss}\,(\mathsf{even\text{-}+}\ p\ q))
\end{aligned}$$

only hold up to a path given in terms of the squash constructor. Consequently, our definition of $\mathsf{isEven!}$ realises our original goal of encoding the proof that addition preserves evenness in a way that exhibits no reduction behaviour.

At this point one might be concerned that the more involved definition of $\mathsf{isEven!}$ compared to isEven makes it more difficult to define functions on even numbers. In practice, common patterns arise that simplify the use of subtypes encoded by our technique, which we describe below using an example.

Given that $\mathsf{isEven!}$ and isEven are isomorphic families, one can always be replaced by the other by appropriately transporting along the isomorphism. However, when transporting from $\mathsf{isEven!}$ to isEven, an irreducible term of the form $\eta\,(\mathsf{even\text{-}+}\ p\ q)$ may be mapped to a term that exhibits reduction behaviour. Therefore, it can often be preferable to more directly use the encoded family of propositions, rather than transport along the derived isomorphism.

For example, consider the function div2 : isEven! $n \to \mathbb{N}$ that divides an even natural number by two. Because $\mathbb{N}$ is not an h-proposition, div2 cannot be defined simply by applying the eliminator for isEven to a function div2? : isEven? $n \to \mathbb{N}$. Instead, there are several possible constructions for div2?, each of which makes intermediate use of the eliminator on isEven!. We highlight a few of these approaches below, which reveal common patterns for defining functions on the higher-inductive families arising from our technique.

One possible approach is to first eliminate isEven! n into an intermediate type B_n : Type that *is* an h-proposition. Then we can compose with a function $f_n : B_n \to \mathbb{N}$ for which the composition has the expected behaviour. For example, for div2 n we can take B_n as the h-proposition $\sum$[$k \in \mathbb{N}$] 2 $*$ $k \equiv n$, which when composed with the first projection is sufficient to construct div2.

Another approach arises by considering how to directly re-obtain the typical induction principle for evenness on our definition of isEven!. In particular, this requires constructing functions ¬isEven1 : isEven! 1 $\to \bot$ and even-pred : isEven! (suc (suc m)) $\to$ isEven! m. Because both of these functions eliminate into propositions, they can be constructed using the eliminator for isEven!.

A third option is to observe that div2? is constant over terms of isEven? and eliminates into an h-set. This ‘coherently constant’ function can hence be lifted to construct div2 by applying an alternative eliminator on isEven! [Kraus, 2015]. The details of each of these constructions for div2 can be found in our Cubical Agda library.

2.4 Higher-inductive recursive even numbers

In this section we introduce our second approach to the problem identified in Section 2.2, based on the use of higher-induction recursion. Thus far, we have shown how to encode that addition preserves evenness in a manner that exhibits no reduction behaviour. This allows us to define addition on even numbers encoded by the sum type $\sum$[$n \in \mathbb{N}$] isEven! n. In particular, the second component of this pair can be constructed by applying the elimination rule for propositional truncation, and then using the constructor even-+ p q.

While the proof that addition preserves evenness no longer exhibits reduction behaviour, addition itself may still be unfolded. Therefore, we might consider whether there is an encoding of even numbers that encodes addition as a single canonical constructor. Indeed, by adapting the isomorphism between inductive families and inductive recursive types [Hancock et al., 2013], we can extend our technique to achieve this aim. For even natural numbers, this translates to defining a higher-inductive type data Even : Type mutually with a recursive function toℕ : Even → ℕ. In Cubical Agda, the higher-inductive type Even can be defined by:

```
data Even where
  zero : Even
  2+_ : Even → Even
  _+E_ : Even → Even → Even
  eq : (x y : Even) → toℕ x ≡ toℕ y → x ≡ y
```

The three data constructors zero, 2+_ and _+E_ correspond to zero, the next even number, and the addition of two even numbers. As with isEven?, while every even number can be defined using these constructors, they are not defined in a *unique* way. For example, 0 can be encoded by both zero and zero +E zero. To restore uniqueness and establish an isomorphism with ∑[n ∈ ℕ] isEven! n, we again introduce an appropriate path constructor, eq, to quotient our type.

Intuitively, the eq path constructor asserts that two even numbers with the same numeric value must be treated identically. In order to define toℕ, we can recognise that it should behave in a similar manner to the first projection on the type ∑[n ∈ ℕ] isEven! n, i.e. by mapping every even number to its underlying value as a natural number. With this in mind, toℕ can be defined as follows:

```
toℕ zero = 0
toℕ (2+ x) = suc (suc (toℕ x))
toℕ (x +E y) = toℕ x + toℕ y
toℕ (eq x y p i) = p i
```

Readers unfamiliar with Cubical Agda may be surprised by the final equation: while eq appears to take three parameters, we are matching on four in toℕ. However, this is really an overloading of the pattern matching syntax, and combines the eliminator for the constructor eq with the eliminator for the path eq x y p. In Cubical Agda, which is based on the CCHM model of cubical type theory [Cohen et al., 2016], we can think of this path as a continuous function from the interval $[0, 1]$, internally represented by I, to the type Even of even numbers. Therefore, for an element of the interval i : I, the term eq x y p i has type Even, as expected.

So far, we have claimed that the introduction of the path constructor eq is enough to establish an isomorphism between the types Even and $\sum$[$n \in \mathbb{N}$] isEven n, but have not proved this. Indeed, the construction is significantly more involved than between isEven! and isEven. Crucially, however, we can construct an isomorphism between Even and $\sum$[$n \in \mathbb{N}$] isEven n by constructing a bijection between them. In particular, this is equivalent to constructing an injection f : Even → $\mathbb{N}$ together with a family of functions h : isEven n → Even, such that f only constructs even numbers and the composition $f \circ h$ is the constant function returning n.

We begin this construction by defining f to be the recursive function to$\mathbb{N}$, for which the proof of injectivity is simply the path constructor eq. The proof that to$\mathbb{N}$ only constructs even numbers follows by induction on a term of Even, and the case for the eq constructor is trivial because evenness is an h-proposition. We can then construct the family h by induction on terms of isEven n, mapping even-z to zero and even-ss p to 2+ $(h\ p)$. Finally, the proof that to$\mathbb{N}$ $\circ$ h is the constant function returning n follows definitionally.

Readers familiar with HoTT might at this point wonder whether this construction only works for subtypes of h-sets, such as the natural numbers $\mathbb{N}$. In particular, we have only used the fact that eq is an injection, but to generalise this construction for any subtype we would require that it be an embedding. Surprisingly, a path constructor of this form, for an inductive-recursive definition, is enough to establish that the recursive function is an embedding, even for higher-groupoid structures. Indeed, a similar construction has been discussed for so-called 'univalent inductive-recursive universes' [Shulman, 2014].

Formally, an *inductive-recursive universe* is an inductive type U : Type of 'codes', defined mutually with a recursive function El : U → Type which interprets each code as a type. Such a universe is *univalent* if it has a path constructor $\mathsf{un} : (x\ y : \mathsf{U}) \to \mathsf{El}\ x \simeq \mathsf{El}\ y \to x \equiv y$. It can be shown that $\mathsf{un}\ x\ y$ is always an equivalence, and hence El is an embedding. Our approach generalises this idea by considering recursive functions $\mathsf{U} \to \mathrm{A}$ for any type A, rather than just Type. Therefore, to generalise un we must replace the equivalence by a path; that is, we require a path constructor $\mathsf{eq} : (x\ y : \mathsf{U}) \to \mathsf{El}\ x \equiv \mathsf{El}\ y \to x \equiv y$. In a univalent setting such as Cubical Agda, if we take A to be Type, then the eq and un versions of the definition are equivalent.

2.5 Reflection

Now that we have introduced the basic ideas of our technique, it is useful to make some remarks about its monadic underpinnings, impact on performance, and how the two representations differ in terms of their construction.

Free monads. We begin by outlining how free monads naturally underpin our technique, and play a central role in its formalisation and generalisation.

Our technique can be seen as first defining a domain-specific language (DSL) on a chosen subtype. More concretely, this means constructing a free monad on a dependent polynomial functor [Malatesta et al., 2012] which characterises the operations of the language. For example, our encoding of even numbers as an inductive-recursive type introduced a DSL with addition as a constructor. We could have further extended this type with any (strictly-positive) operation which constructs an even number, such as multiplication, and the isomorphism with the standard encoding of even numbers would remain constructible.

The above idea gives rise to the formalisation of our technique that is described in Section 2.7. Notably, this involves taking particular quotients of free monads which then allows us to construct a calculus on subtypes. In practice this simplifies the process of working with our approach, particularly in the case when constructing maps between two encoded subtypes.

Higher induction. An essential requirement for a type theory in which our technique can be used is the presence of either higher inductive families or higher-induction recursion. In particular, in Section 2.3 we demonstrated how the even natural numbers can be encoded by means of a higher inductive family, while in Section 2.4 we instead made use of a higher inductive-recursive definition. While Cubical Agda implements both higher inductive families and higher induction-recursion, they do not have a known semantics in the CCHM model [Cohen et al., 2016] on which it is based. However, a semantics for both higher inductive types and higher inductive families are known in other models of HoTT.

The semantics for a range of higher inductive types have been presented in various models of homotopy type theory [Lumsdaine and Shulman, 2020] including in the cubical setting [Coquand et al., 2018]. Moreover, Cavallo and Harper [2018] consider a schema for higher inductive families in a model of cubical type theory that they term *computational higher type theory*. This schema notably includes that of propositionally truncated families, which is precisely what is required for the encoding of subtypes presented in Sections 2.3 and 2.7. At present, a semantics for higher induction-recursion has not been given in any model of homotopy type theory.

Performance. The most significant impact of hiding reduction behaviour for operations on a subtype is the improvement in performance during type-checking. In particular, operations encoded as constructors of an inductive type will not be unfolded. Indeed, the performance cost of unnecessarily unfolding terms during type-checking is so prevalent that Agda provides two language features to address this problem. The key insight behind these features, and indeed our technique, is that even within the context of a total language the choice of when to reduce terms is an important practical consideration that can mean the difference between type-checking a proof in reasonable time and running out of memory before type-checking concludes. We discuss the differences between our technique and existing language features of Agda in Section 2.11.

Perhaps surprisingly, our encoding can also improve the performance of normalising specific terms. This is possible as a consequence of retaining additional information about how a term of a subtype is constructed. For example, consider the following function that proves that any positive power of two is even:

```
isEven-2^ : (n : ℕ) → isEven (2 ^ suc n)
isEven-2^ zero = even-ss even-zero
isEven-2^ (suc n) =
```

$$\mathsf{let}\ p = \mathsf{isEven\text{-}2\hat{}}\ n\ \mathsf{in}\ \mathsf{isEven+}\ p\ (\mathsf{isEven+}\ p\ \mathsf{even\text{-}zero})$$

By replacing occurrences of isEven+ with the constructor even-+ and composing with η : isEven? $n \to$ isEven! n, we can similarly construct a function isEven!-2^ : $(n : \mathbb{N}) \to$ isEven! (2 ^ suc n). Using these definitions we can construct two different proofs that the natural number 65536 is even, and crucially, these proofs are equal up to a heterogeneous path.

To compare performance, we normalised these proof terms in Cubical Agda's Emacs mode, as a working compiler for the language is not currently available. Using a Macbook Pro with a 2.7GHz quad-core processor and 16GB of memory, we were unable to normalise the first term for any tested amount of time (up to 30 minutes), while the second was normalised in under 30 seconds.

It is natural to wonder whether any 'useful' computation on even numbers would first require that we translate back to the standard representation of evenness, given by the recursive family isEven. However, as described in Section 2.3, the example of dividing any even number by two reveals that this is not the case. Interestingly, this can lead to significant performance benefits when normalising the result of dividing a large even number by two. For example, we tested a number of 'reasonable' definitions for div2' : isEven $n \to \mathbb{N}$ and div2 : isEven! $n \to \mathbb{N}$ and found that we were unable to reduce the time taken to normalise div2' (isEven-2^ 15) to below three minutes, whereas our implementation of div2 (isEven!-2^ 15) took less than fifteen seconds. We refer readers interested in our implementation of div2 to our Cubical Agda library.

A similar performance impact to utilising our technique can be found by instead making use of a strict Prop universe [Gilbert, Gaëtan and Cockx, Jesper and Sozeau, Matthieu and Tabareau, Nicolas, 2019]. In particular, by defining the family of types isEven? introduced in Section 4 as a family of Props, it is then possible to construct a proof of the evenness of 65536 by instead defining isEven-2^ to construct a term of isEven? in a near identical fashion. At first glance, this may appear to be a simpler approach. However, one of the key characteristics of our approach is the preservation of computational content. In particular, this means that it interacts well with other constructions in HoTT. For example, if we were to define evenness as a family of strict Props, it would no longer be possible to show that the forgetful map from evens to naturals is an embedding, which is precisely the property we should expect from any subtype in HoTT.

Summary. We conclude by summarising our two approaches to encoding operations on a subtype. Both approaches give isomorphic representations, but differ in manner in which they are constructed.

Inductive families (IF) approach:

1. Given a subtype, select some closure properties on its subtyping condition;
2. Define the subtyping condition as an inductive family;
3. Extend the family with data constructors that encode the closure properties;
4. Extend the family with a path constructor which asserts that all subtyping proofs are equal.

Induction-recursion (IR) approach:

1. Given a subtype, select some operations on it;
2. Define the subtype as an inductive-recursive type;
3. Extend the inductive type with the operations, and extend the recursive function with their interpretations;
4. Extend the inductive type with a path constructor that asserts the function is injective, and trivially extend the function over the path constructor.

2.6 Example: ordered finite sets

In this section we show how ordered finite sets can be represented using the inductive families version of our technique. In type theory, ordered finite sets are typically defined as an inductive family $\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$ with two constructors, $\mathsf{zero} : \mathsf{Fin}\ (\mathsf{suc}\ n)$ and $\mathsf{suc} : \mathsf{Fin}\ n \to \mathsf{Fin}\ (\mathsf{suc}\ n)$. This example illustrates that in order to obtain an extensible encoding, it is important to carefully choose the operations to be encoded by our technique. Moreover, while there remains a degree of creativity required to select operations on a subtype, we can identify desirable properties for our encoding. In particular, for the example of ordered finite sets, our focus will be on recovering the typical elimination principle by dependent pattern matching in Cubical Agda [Cockx et al., 2014].

We can alternatively think of Fin n as a subtype of ℕ, with the subtyping condition on a natural number k given by $k < n$. This definition as a subtype is precisely the form our technique can be used with. Furthermore, the representation of Fin as a family of subtypes has several desirable properties over its definition as an inductive family. For example, the function toℕ : ∑[k ∈ ℕ] $k < n$ → ℕ is trivially defined as the first projection, and the proof that this function is an embedding follows simply from the fact that $k < n$ is a proposition. Concretely, we can define the total order _<_ in Agda as the following inductive family:

```
data _<_ : ℕ → ℕ → Type where
  z<s : 0 < suc n
  s<s : m < n → suc m < suc n
```

Intuitively, we can observe that _<_ defines a family of propositions since the only way to prove that a number m is less than a number suc n is by applying the constructor s<s a total of m times to the constructor z<s.

We can now proceed by applying the next step of our technique. In particular, we can identify closure properties on the inductive family _<_ and extend its definition with constructors encoding them. As a general rule, we recommend prioritising closure properties that are both ubiquitous and for which the typical elimination principle can be 'easily' extended over. In order to highlight this idea, we shall give examples of such closure properties on _<_.

We begin by observing that the typical elimination principle for the inductive family _<_ can be constructed from proofs ¬m<0 : $m < 0$ → ⊥ and <-smonic : suc m < suc n → $m < n$. Indeed, these proofs are precisely what is required to establish that the relation _<_ is well-founded. As such, it will be sufficient to define the action of these functions on any additional closure properties we include in our definition of _<_. For example, consider transitivity of the relation _<_, which can typically be proven as follows:

```
trans< : m < n → n < o → m < o
trans< z<s (s<s q) = z<s
trans< (s<s p) (s<s q) = s<s (trans< p q)
```

We begin by encoding the proof of transitivity as a constructor trans< : $(n : \mathbb{N}) \to m < n \to n < o \to m < o$. We then extend the function ¬m<0 by defining ¬m<0 (trans< $n\ p\ q$) = ¬m<0 q, and in turn extend the function <-monic:

```
<-smonic (trans< 0 p q)          =  absurd (¬m<0 p)
<-smonic (trans< (suc n) p q)    =  trans< (<-smonic p) (<-smonic q)
```

Crucially, this construction allows us to recover the typical elimination principle for _<_. As a second example, we consider the property that any natural number is less than its successor, which can typically be proven as follows:

```
n<sn : (n : ℕ) → n < suc n
n<sn zero = z<s
n<sn (suc n) = s<s (n<sn n)
```

We can encode this proof by a constructor n<sn : $(n : \mathbb{N}) \to n <$ suc n. Notably, our definition of ¬m<0 does not need to be extended since 0 never unifies with suc n, and we can extend <-smonic by simply mapping a term n<sn (suc n) to n<sn n.

While we have shown that the elimination principle for _<_ can be preserved after extension with constructors trans< and n<sn, the new inductive family is of course not isomorphic to the original. However, by applying the next step of our technique, we quotient our new inductive family to obtain this isomorphism:

```
data _<_ : ℕ → ℕ → Type where
  z<s : 0 < suc n
  s<s : m < n → suc m < suc n
  n<sn : n < suc n
  trans< : m < n → n < o → m < o
  trunc : (p q : m < n) → p ≡ q
```

We also now require that ¬m<0 and <-smonic be extended over the path constructor, which is trivial as both functions eliminate into h-propositions.

It is important to highlight how the approach in this section differs from recovering the same elimination principle by simply transporting along the isomorphism between the alternative and original representations. In particular, by observing how the definition of <-smonic was extended over the inductive constructors trans< and n<sn, we can see that it does not 'expand' proofs built from these constructors into proofs built only from z<s and s<s. That is, in contrast to transporting along the induced isomorphism from the alternative definition of _<_ to the original definition, we preserve the efficient encodings of transitivity and the proof that every natural number is less than its successor.

The finite sets example can also be represented using our induction-recursion technique, with the details being available in our Cubical Agda library.

2.7 IF formalisation

As discussed in Section 2.5, our encoding of a subtyping condition arises as a quotient of the free monad on a dependent polynomial functor. In this section, we give a formalisation of this idea using *indexed containers* [Altenkirch and Morris, 2009; Altenkirch et al., 2015], which can be seen as an internalisation of dependent polynomial functors in type theory. In particular, we show how our technique arises as the free construction, over an indexed container, of an algebraic structure we call a *propositional monad*.

Indexed containers

Indexed containers provide a generic means to capture and reason about strictly positive type families, and as such we can use them to capture collections of operations on a subtyping condition. Indexed containers can be represented in Agda as terms of the following record type:

```
record Container (I O : Type) : Type₁ where
  field
    Com : (o : O) → Type
    Res : ∀ {o} → Com o → Type
    next : ∀ {o} → (c : Com o) → Res o c → I
```

It is useful to think of such a container as a collection of trees, each with a single vertex, whose input edges are labelled by terms of I, and whose single output edge is labelled by a term of O. In this way, Com maps each $o : O$ to the type of trees whose output edge is labelled by o, Res maps a tree c : Com o to the type of its input edges, and next maps an input edge r : Res o c to its label from I.

Every C : Container I O gives rise to a dependent polynomial functor $[\![C]\!] : (I \to \mathsf{Type}) \to \mathsf{O} \to \mathsf{Type}$, by means of the following definition:

$$[\![C]\!]\ X\ o = \textstyle\sum[\ c \in \mathsf{Com}\ C\ o\ ]\ (\forall\ r \to X\ (\mathsf{next}\ C\ c\ r))$$

This family is termed the *extension* of C, and we can similarly formulate its *dependent extension* $[\![C]\!]_2\ A\ B : O \to \mathsf{Type}$ on all families $A : I \to \mathsf{Type}$, $B : \forall\ i \to A\ i \to \mathsf{Type}$, defined as follows:

$$[\![C]\!]_2\ A\ B\ o = \textstyle\sum[\ (c\ ,\ f) \in [\![C]\!]\ A\ o\ ]\ (\forall\ r \to B\ \_\ (f\ r))$$

Together with the notion of a container and its extension, our formalisation also uses the notion of a container morphism, captured by a record type:

```
record _⇒_ (C D : Container I O) : Type where
  field
    Com₁ : ∀ o → Com C o → Com D o
    Res₁ : ∀ {o} (c : Com C o) → Res D (Com₁ o c) → Res C c
    Coh : ∀ o c r → next C c (Res₁ c r) ≡ next D (Com₁ o c) r
```

Every morphism $f : C \Rightarrow D$ can be extended to a morphism between the extensions of C and D. In particular, for every $B : I \to \mathsf{Type}$, we define $\langle\ f\ \rangle\ B : \forall\ o \to [\![C]\!]\ B\ o \to [\![D]\!]\ B\ o$ by mapping $o : O$ and $(c\ ,\ g) : [\![C]\!]\ B\ o$ to:

$$\mathsf{Com}_1\ f\ o\ c\ ,\ \lambda\ r \to \mathsf{subst}\ B\ (\mathsf{Coh}\ f\ o\ c\ r)\ (g\ (\mathsf{Res}_1\ f\ c\ r)).$$

Propositional monads

We now introduce propositional monads, which will be used to formalise our technique. Given any O : Type, a type family $M : (O \to \mathsf{Type}) \to O \to \mathsf{Type}$ is a propositional monad if there is a term of the following record type:

```
record isPropMonad M : Type₁ where
  field
```

```
isPropM : ∀ A o → (x y : M A o) → x ≡ y
return : ∀ A o → A o → M A o
bind : ∀ A B o → M A o → (∀ o → A o → M B o) → M B o
```

This definition of isPropMonad presents the structure of a monad on families of propositions. That is, return is the unit map of the monad, bind combines its multiplication map with its functorial action on morphisms, and isPropM asserts that M is a family of propositions. The monadic laws are not required in our definition, as they will always provably hold as a consequence of M being a family of h-propositions. For any type families $A\ B : O \to$ Type, we can construct both the functorial action of M on morphisms, and its monad multiplication map

```
join : (B : O → Type) → ∀ o → M (M B) o → M B o
```

from bind and return in the usual way. Importantly, it is sufficient to construct a term of type isPropMonad M to prove the necessary monadic (and functorial) laws. Indeed, because M is a family of propositions, these laws trivially hold.

Given propositional monads $F\ G : (O \to$ Type$) \to O \to$ Type, a morphism from F to G is simply a morphism between the underlying type families, i.e. a family of functions $\alpha_B : (o : O) \to F\ B\ o \to G\ B\ o$ for every $B : O \to$ Type. If the family α_B exists, it will always be unique and respect the monadic structure. Both of these properties follow from appropriate application of the term

```
isPropM G B : (o : O) (x y : G B o) → x ≡ y
```

which states that for all $o : O$, the term $G\ B\ o$ is a proposition.

Free propositional monad

For every indexed container C : Container $O\ O$, we can define the free propositional monad over C by truncating the free monad on C. In particular, this can be defined in Cubical Agda by the following higher inductive family:

```
data FreePM C (P : O → Type) : O → Type where
  η : (o : O) → P o → FreePM C P o
```

$$\mathsf{fix} : (o : O) \to [\![\ C\ ]\!]\ (\mathsf{FreePM}\ C\ P)\ o \to \mathsf{FreePM}\ C\ P\ o$$
$$\mathsf{squash} : (o : O)\ (x\ y : \mathsf{FreePM}\ C\ P\ o) \to x \equiv y$$

To show that $\mathsf{FreePM}\ C$ is a propositional monad, we begin by observing that isPropM is trivially given by the path constructor squash, and similarly return is given by the data constructor η. Given type families $P\ Q : O \to \mathsf{Type}$, a term $x : \mathsf{FreePM}\ C\ P\ o$, and a family of functions $g : \forall\ o \to P\ o \to \mathsf{FreePM}\ Q\ o$, the construction of $\mathsf{bind}\ P\ Q\ o\ x\ g : \mathsf{FreePM}\ C\ Q\ o$ follows by induction on x:

$$\mathsf{bind}\ \mathrm{P}\ \mathrm{Q}\ \mathrm{o}\ (\eta\ o\ p)\ g = g\ o\ p$$
$$\mathsf{bind}\ \mathrm{P}\ \mathrm{Q}\ o\ (\mathsf{fix}\ o\ (c\ ,\ f))\ g = \mathsf{fix}\ o\ (c\ ,\ \lambda\ r \to \mathsf{bind}\ P\ Q\ o\ (f\ r)\ g)$$

Importantly, we can also extend bind over the path constructor squash, because we are eliminating into a proposition.

We recall that this construction also gives us the functorial action of $\mathsf{FreePM}\ C$ on O-indexed type families. That is, for all type families $P\ Q : O \to \mathsf{Type}$, we can lift a family of functions $f : \forall\ o \to P\ o \to Q\ o$ to a family of functions $\mathsf{map}\ f : \forall\ o \to \mathsf{FreePM}\ C\ P\ o \to \mathsf{FreePM}\ C\ Q\ o$ between their corresponding free propositional monads. Similarly, we can define the monad multiplication map $\mathsf{joinFPM} : \forall\ o \to \mathsf{FreePM}\ C\ (\mathsf{FreePM}\ C\ P)\ o \to \mathsf{FreePM}\ C\ P\ o$.

The free propositional monad construction extends to a functor by lifting any morphism $\alpha : C \Rightarrow D$ to a family of functions $\mathsf{lift}\ \alpha\ P : \forall\ \{o\} \to \mathsf{FreePM}\ C\ P\ o \to \mathsf{FreePM}\ D\ P\ o$. In particular, we define lift inductively on the constructors of $\mathsf{FreePM}\ C\ P\ o$ by simply mapping $\eta\ o\ p$ to $\eta\ o\ p$ and $\mathsf{fix}\ o\ (c\ ,\ f)$ to:

$$\mathsf{fix}\ o\ (\langle\ \alpha\ \rangle\ (\mathsf{FreePM}\ C\,\mathrm{B})\ (c\ ,\ \mathsf{lift}\ \alpha \circ \mathrm{f}))$$

It is easy to extend lift over the path constructor squash because the proof that we are eliminating into a proposition is simply given by squash. The functorial laws follow from the proof that $\mathsf{FreePM}\ D\ P\ o$ is a proposition.

To show that $\mathsf{FreePM}\ C$ is the *free* propositional monad on C, we first require a unit of the free construction. In particular, this is a propositional monad morphism between $[\![\ C\ ]\!]$ and $\mathsf{FreePM}\ C$, which can be defined as follows:

$$\mathsf{unit} : (B : O \to \mathsf{Type}) \to \forall\ o \to [\![\ C\ ]\!]\ B\ o \to \mathsf{FreePM}\ C\ B\ o$$
$$\mathsf{unit}\ B\ o\ (c\ ,\ f) = \mathsf{fix}\ o\ (s\ ,\ \lambda\ r \to \eta\ (\mathsf{next}\ C\ c\ r)\ (f\ r))$$

Furthermore, for all containers $C\ D$: Container $O\ O$ for which there is a term δ : isPropMonad $[\![\ D\ ]\!]$ witnessing that $[\![\ D\ ]\!]$ is a propositional monad, and for every morphism $\alpha : C \Rightarrow D$ and family $B : O \to$ Type, we can construct a unique family of functions fold $\alpha\ B : \forall\ \{o\} \to$ FreePM $C\ B\ o \to [\![\ D\ ]\!]\ B\ o$. To define fold $\alpha\ B$, we observe that the codomain is a proposition, and hence we can give its inductive definition on only the data constructors of FreePM $C\ B\ o$:

```
fold α B (η o p) = return δ B o p
fold α B (fix o (c , f )) =
   join δ B o (⟨ α ⟩ ([[ D ]] B) (c , fold α B ∘ f))
```

Importantly, for every $x : [\![\ C\ ]\!]\ B\ o$, it is possible to construct a suitable path fold $\alpha\ B\ o$ (unit $B\ o\ x) \equiv \langle\ \alpha\ \rangle\ B\ o\ x$ witnessing the left adjunct (fold) interacts with the unit map in the expected way; fold $\alpha\ B$ is the unique such propositional monad morphism satisfying this condition, as its type is an h-prop.

We recall that the key result of our technique is the construction of an isomorphism between our alternative representation of a family of propositions P, i.e. FreePM $C\ P$, and P itself. Notably, as both P and FreePM $C\ P$ are families of propositions, it suffices to construct a family of functions in both directions. The family from P to FreePM $C\ P$ is trivially given by the data constructor η. In the other direction, it is not in general, possible to construct a family of functions from FreePM $C\ P$ to P. Indeed, this is only the case when P is closed under the operations characterised by C, which means we can construct a C-algebra $\alpha : \forall\ o \to [\![\ C\ ]\!]\ P\ o \to P\ o$ with P as the carrier. Given such an algebra, we can inductively construct the desired family, $f : \forall\ \{o\} \to$ FreePM $C\ P\ o \to P\ o$, by mapping $\eta\ o\ p$ to p and fix $o\ (c\ ,\ g)$ to $\alpha\ o\ (c\ ,\ \lambda\ r \to f\ (g\ r))$.

Example

To demonstrate how free propositional monads formalise the inductive families version of our technique, we provide an example whose construction follows the four step process in Section 2.5. In particular, we will consider a subtype of lists where adjacent elements are related by a *mere* relation, i.e. a family of h-propositions, which we denote $\_{\sim}\_ : A \to A \to$ Type. To do this, we begin by defining the following family of propositions on lists:

```
data isRelated : List A → Type where
  nil : isRelated []
  sing : (x : A) → isRelated (x :: [])
  ind : x ∼ y → isRelated (y :: xs) → isRelated (x :: y :: xs)
```

The constructors nil, sing and ind allow us to construct proofs of the subtyping condition for the empty list, singleton lists, and lists whose first two elements are related and whose tail respects the subtyping condition. Because _∼_ is a mere relation, isRelated is a family of h-props. Notably, if _∼_ is a total order, then a term of type isRelated xs corresponds to a proof that xs is sorted. However, for our purposes we will only require that _∼_ be transitive. While this additional constraint will not be necessary to construct our alternative encoding of isRelated, it will be required to construct an isomorphism with the original definition.

The next step of our technique involves choosing closure properties on the subtyping condition isRelated. We will consider two such properties, namely that if _∼_ is transitive then isRelated is closed under filtering of a list and (safe) removal of elements. That is, given functions

```
filter : (A → Bool) → List A → List A,
remove : (xs : List A) → Fin (length xs) → List A,
```

defined in the obvious way, we can construct the following two proof terms:

```
filterR : ∀ P xs → isRelated xs → isRelated (filter P xs)
removeR : ∀ xs i → isRelated xs → isRelated (remove xs i)
```

We now proceed by capturing these closure properties with an indexed container C : Container (List A) (List A). To do this, we begin by defining the commands, Com C : List A → Type, as the following inductive family:

```
data RelCom : List A → Type where
  filterC : ∀ P? xs → RelCom (filter P? xs)
  removeC : ∀ i xs → RelCom (remove i xs)
```

The next step is to define the inductive positions or ‘responses’, Res C : ∀ {xs} → RelCom xs → Type. Here, this is simply given by Res C xs c = ⊤ for all xs : List A and c : RelCom xs. Finally, we define next C : ∀ {xs} → (c : RelCom xs) → Res C xs c → List A by induction on the constructors of RelCom:

$\mathsf{next}\ C\ (\mathsf{filterC}\ P?\ as)\ t = as$
$\mathsf{next}\ C\ (\mathsf{removeC}\ i\ as)\ t = as$

From our definition of the container C, the alternative encoding of the inductive family isRelated is simply the free propositional monad given by $\mathsf{FreePM}\ C\ \mathsf{isRelated} : \mathsf{List}\ A \to \mathsf{Type}$, which we denote by isRelatedF. Importantly, we can prove that for any list $xs : \mathsf{List}\ A$, the h-props $\mathsf{isRelatedF}\ xs$ and $\mathsf{isRelated}\ xs$ are isomorphic. The construction of this family of isomorphisms follows from the proof that isRelated is indeed closed under the two operations we have encoded. That is, we can construct a C-algebra $\alpha : (xs : \mathsf{List}\ A) \to [\![\ C\ ]\!]\ \mathsf{isRelated}\ xs \to \mathsf{isRelated}\ xs$, by constructing proofs that filtering and removal respect isRelated. Of course, this is only true when the relation $\_\sim\_$ is transitive.

As is the intended purpose of our alternative encoding of isRelated, the family of propositions isRelatedF comes equipped with two canonical constructors:

$\mathsf{filterF} : \forall\ P? \to \mathsf{isRelatedF}\ xs \to \mathsf{isRelatedF}\ (\mathsf{filter}\ P?\ xs),$
$\mathsf{removeF} : \forall\ i \to \mathsf{isRelatedF}\ xs \to \mathsf{isRelatedF}\ (\mathsf{remove}\ xs\ i),$

In particular, these constructors correspond to the proofs that isRelatedF is closed under filtering and removal. For example, filterF can be constructed as:

$\mathsf{filterF}\ P?\ xs = \mathsf{fix}\ (\mathsf{filter}\ P?\ xs)\ (\mathsf{filterC}\ P?\ xs\ ,\ \lambda\ t \to xs)$

Importantly, as long as the arguments $P?$ and xs are in canonical form, then the proof that filterF preserves the subtyping condition will correspondingly be in canonical form. We can construct a similar term corresponding to removeF, and in this way the inductive family isRelatedF efficiently encodes closure of lists with related adjacent elements under filtering and element removal.

2.8 IR formalisation

In this section, we will give a formalisation of our higher inductive-recursive encoding of subtypes. We call this encoding the *free subtype extension* on a container. This formalisation will again make use of indexed containers to capture collections of operations. We could also have encoded operations in terms of IR-codes [Dybjer and Setzer, 2003], but these two approaches are equivalent [Hancock et al., 2013] and the use of indexed containers simplifies many of our constructions. Once we have given a formalisation of the higher-inductive recursive encoding of subtypes, we then proceed by defining its eliminator and proving its equivalence to the higher-inductive family encoding of subtypes. We conclude this section by applying our formalisation to the practical example of ordered finite sets.

Fibers

In order to formalise our higher-inductive recursive encoding of subtypes in terms of indexed containers, we recall the definition of the *fiber* over a function. We can define the fiber over a function $f : A \to B$ as an inductive family:

```
data Fiber f : B → Type where
  fib : (a : A) → Fiber f (f a)
```

The inductive family Fiber f comes with eliminators unwrap $f\ b$: Fiber $f\ b \to A$ and unwrap-β $f\ b : (x :$ Fiber $f\ b) \to f$ (unwrap $f\ b\ x) \equiv b$, defined as follows:

```
unwrap f .(f a) (fib a) = a
unwrap-β f .(f a) (fib a) = refl
```

There is a well-known equivalence between A-indexed type families and fibrations over A, i.e. $\sum[\ U \in \text{Type}\ ]\ (U \to A)$. In particular, this equivalence maps an IR-definition, i.e. an inductive type U : Type defined mutually with a recursive function E : U $\to A$, to the fibers over E. The A-indexed type family corresponding to the fibers over E is precisely what is required when using indexed containers to formalise our higher-inductive recursive encoding of subtypes.

Free subtype extension

As with the previous approach, to formalise our inductive recursive encoding of subtypes we begin with an indexed container, C : Container $O\ O$, which corresponds to the operations that will be encoded as canonical constructors. We then proceed by mutually defining a higher-inductive datatype data FreeSTExt C P : Type together with a recursive function decode C P : FreeSTExt C $P \to O$, for every family $P : O \to$ Type. In particular, we define the type by

```
data FreeSTExt C P where
  η : ∀ o → P o → FreeSTExt C P
  fix : ∀ o → ⟦ C ⟧ (Fiber (decode C P)) o → FreeSTExt C P
  eq :  ∀ x y → decode C P x ≡ decode C P y → x ≡ y
```

and the recursive function as follows:

```
decode C P (η o p) = o
decode C P (fix o x) = o
decode C P (eq x y p i) = p i
```

Intuitively, the eq path constructor of FreeSTExt C P asserts that the function decode C P is injective. In this manner, we can understand FreeSTExt C P as being a subtype of O, and decode C P as corresponding to the first projection or 'underlying' map on the typical representation of a subtype as a dependent pair. The fix constructor of FreeSTExt C P extends the definition of the subtype given by η and eq with the operations characterised by the container C.

To construct functions out of the type FreeSTExt C P, we must prove that our constructions respect the path constructor eq. Concretely, for every eliminator f : (x : FreeSTExt C P) $\to B\ x$ this means that for all x, y : FreeSTExt C P and p : decode C P $x \equiv$ decode C P y we provide a construction for the path

```
cong f (eq x y p) : PathP (λ i → B (p i)) (f x) (f y).
```

It may at first seem that the eq path constructor, which appears to simply assert that decode C P is injective, is insufficient to capture higher subtypes. That is, injectivity of the underlying map is only enough when a subtype is an h-set. However, as we will show, the introduction of the eq path constructor is in fact sufficient to prove that decode C P is an embedding.

Decode is an embedding

A function $f : X \to Y$ is called an embedding if for all $x,\ y : X$, the action of f on the path space $x \equiv y$, i.e. cong $f : x \equiv y \to f\ x \equiv f\ y$, is an equivalence. This is known to be equivalent to f having propositional fibers, i.e. Fiber $f\ y$ is an h-prop for all $y : Y$. Importantly, this means that decode $C\ P$ is an embedding precisely when Fiber (decode $C\ P$) o is an h-prop for all $o : O$. In particular, this will simplify defining functions out of the free subtype extension into Fiber (decode $C\ P$) o, as it easy to show that the path constructor eq is respected when eliminating into an h-prop.

In order to show that decode $C\ P$ is an embedding, it is sufficient to prove that the following two functions

cong (decode $C\ P$) : $x \equiv y \to$ decode $C\ P\ x \equiv$ decode $C\ P\ y$,
eq $x\ y$: decode $C\ P\ x \equiv$ decode $C\ P\ y \to x \equiv y$,

establish an isomorphism between $x \equiv y$ and decode $C\ P\ x \equiv$ decode $C\ P\ y$. That is, we require constructions for both the left and right inverse:

left : $\forall\ p \to$ cong (decode $C\ P$) (eq $x\ y\ p$) $\equiv p$,
right : $\forall\ p \to$ eq $x\ y$ (cong (decode $C\ P$) p) $\equiv p$.

The family of paths left is simply given by the action of decode $C\ P$ on the path constructor eq. To construct the path right p for every path $p : x \equiv y$, it is sufficient to construct a path eqRefl : eq $x\ x$ refl $\equiv$ refl by application of the J-eliminator on p. In order to construct the path eqRefl, we shall make use of one of the foundational concepts of cubical type theory, by constructing it as the lid of the cube shown in Figure 2.1. In this figure, the variables i, j, k : I are terms of the interval that correspond to the dimensions of the cube, with i being the dimension from left to right, j from front to back, and k from bottom to top.

In Cubical Agda, the side faces of the cube can be constructed as a *partial element* of FreeSTExt $C\ P$. For any A : Type and interval formula ψ : I, the type Partial $\psi\ A$ corresponds to the type of cubes in A for which for which the formula ψ takes on the value i1 : I, where i1 is the maximum endpoint in the interval. Given the dimensions i, j, k of the cube, the formula which has value i1 only for the side faces is $i \vee \sim i \vee j \vee \sim j$. In this formula we use the connectives $\vee$: I $\to$ I $\to$ I and $\sim$: I $\to$ I, which together with $\wedge$: I $\to$ I $\to$ I form a De Morgan algebra on the interval type

$$\begin{tikzcd}
& x \arrow[rr, "x"] & & x \\
x \arrow[ur, "\mathsf{eq}\ x\ x\ \mathsf{refl}\ j"] \arrow[rr, "x"] & & x \arrow[ur, "x"] & \\
& x \arrow[uu, "x"] \arrow[rr, "x"] & & x \arrow[uu, "x"'] \\
x \arrow[uu, "x"] \arrow[ur, "\mathsf{eq}\ x\ x\ \mathsf{refl}\ j"'] \arrow[rr, "\mathsf{eq}\ x\ x\ \mathsf{refl}\ (\sim i)"'] & & x \arrow[uu, "\mathsf{eq}\ x\ x\ \mathsf{refl}\ k"'] \arrow[ur, "\mathsf{eq}\ x\ x\ \mathsf{refl}\ j"'] &
\end{tikzcd}$$

Figure 2.1: A cube whose lid is eqRefl

whose laws are given definitionally. To express the side faces of our cube in Cubical Agda, we will construct the following term:

$$\mathsf{sides} : (i\ j\ k : \mathsf{I}) \to \mathsf{Partial}\ (i \vee \sim i \vee j \vee \sim j)\ (\mathsf{FreeSTExt}\ C\ P)$$

To do this, we will use a special form of pattern matching that allows us to individually consider when each of the disjuncts in our formula are $\mathsf{i1}$. In particular, this will correspond to giving a construction for each side face. For example, the right-most face will consider the case $i = \mathsf{i1}$ and will be given by a heterogeneous path over $\lambda\ k \to \mathsf{eq}\ x\ x\ \mathsf{refl}\ k \equiv x$, between $\mathsf{eq}\ x\ x\ \mathsf{refl} : x \equiv x$ and $\mathsf{refl} : x \equiv x$. Concretely, we can can construct the side faces of our cube as follows:

$$\begin{aligned}
&\mathsf{sides}\ i\ j\ k\ (i = \mathsf{i0}) = \mathsf{eq}\ x\ x\ \mathsf{refl}\ j \\
&\mathsf{sides}\ i\ j\ k\ (i = \mathsf{i1}) = \mathsf{eq}\ x\ x\ \mathsf{refl}\ (j \vee k) \\
&\mathsf{sides}\ i\ j\ k\ (j = \mathsf{i0}) = \mathsf{eq}\ x\ x\ \mathsf{refl}\ (\sim i \vee k) \\
&\mathsf{sides}\ i\ j\ k\ (j = \mathsf{i1}) = x
\end{aligned}$$

Here $(i = \mathsf{i0})$, $(i = \mathsf{i1})$, $(j = \mathsf{i0})$ and $(j = \mathsf{i1})$ correspond to the four face maps for the sides of the cube. To construct the lid, we also require a path corresponding to the base. Concretely, this is a heterogeneous path over $\phi = \lambda\ i \to \mathsf{eq}\ x\ x\ \mathsf{refl}\ (\sim i) \equiv x$ from $\mathsf{eq}\ x\ x\ \mathsf{refl}$ to itself, constructed as follows:

$$\begin{aligned}
&\mathsf{base} : \mathsf{PathP}\ \phi\ (\mathsf{eq}\ x\ x\ \mathsf{refl})\ (\mathsf{eq}\ x\ x\ \mathsf{refl}) \\
&\mathsf{base}\ i\ j = \mathsf{eq}\ (\mathsf{eq}\ x\ x\ \mathsf{refl}\ (\sim i))\ x\ \mathsf{refl}\ j
\end{aligned}$$

Finally, we can construct the path from eq x x refl to refl, using Cubical Agda's homogeneous composition function:

```
eqRefl $i$ $j$ = hcomp (sides $i$ $j$) (base $i$ $j$)
```

Crucially, as outlined earlier, this is enough to prove that decode C P is an embedding. Furthermore, it follows that the fibers of decode C P are propositions, and indeed the O-indexed family of types Fiber (decode C P) can be seen as the subtyping condition for FreeSTExt C P.

As an alternative proof, we can instead directly show that the family of types Fiber (decode C P) is a family of h-propositions, and make use of the equivalence between a function having propositional fibers and being an embedding. To do this, we first state the eta-law of the inductive family Fiber,

```
unwrap-$\eta$ : $\forall$ $x$ → PathP ($\Psi$ $x$) (fib (unwrap $x$)) $x$
unwrap-$\eta$ (fib $a$) = refl
```

where Ψ x i = Fiber f (unwrap-β x i). The next step involves defining two sides of a square, for all a : FreeSTExt C P and x : Fiber (decode C P) o:

```
sides $a$ $x$ : ($i$ $j$ : I) → Partial ($i$ ∨ ~ $i$) ($\Psi$ $x$ $j$)
sides $a$ $x$ $i$ $j$ ($i$ = i0) = fib (eq (unwrap $x$) $a$ (unwrap-$\beta$ $x$) $j$)
sides $a$ $x$ $i$ $j$ ($i$ = i1) = unwrap-$\eta$ $x$ $j$
```

Finally, given any x, y : Fiber (decode C P) o, we can then construct a path $x \equiv y$ by induction on x, where we consider the single case o = decode C P a and x = fib a. We can then apply Cubical Agda's heterogeneous composition comp over the path Ψ y, to compute the lid of the square with sides given by sides a y and base given by fib (unwrap y).

The proof that the decode function has propositional fibers simplifies defining functions between free subtype extensions. Typically, to define a function f : FreeSTExt C P → FreeSTExt D P by induction on FreeSTExt C P, we need to show that f respects the path constructor eq. However, it is often simpler to define a family of functions g : $\forall$ o → Fiber (decode C P) o → Fiber (decode D P) o, and then construct f as unwrap ∘ g ∘ fib. To construct g, we can first induct on the single case of Fiber (decode C P) o, i.e. fib a for a : FreeSTExt C P, and then proceed by induction on a. Importantly, any construction we give by induction on

a will respect the path constructor eq, as a consequence of eliminating into a h-prop. An important example is defining the functorial action of the free subtype extension on a container morphism $h : C \Rightarrow D$, which can be found online in our Cubical Agda library.

Equivalence between IF and IR approaches

The proof that decode C P has propositional fibers is central to our construction of an equivalence between our IF and IR encodings. In particular, it will facilitate the construction of a family of isomorphisms between the O-indexed families FreePM C P and Fiber (decode C P), which correspond to the subtyping condition. Finally, we will construct an equivalence between the subtype encodings themselves, i.e. FreeSTExt C P and $\sum[\, o \in O\,]$ FreePM $C\,P\,o$. Concretely, we begin by constructing a family of functions IF→IR : $\forall\,\{o\} \to$ FreePM C P $o \to$ Fiber (decode C P) o, defined on the data constructors of FreePM C P o by:

```
IF→IR (η o p) = fib (η o p)
IF→IR (fix o (c , g)) = fib (fix o (c , λ r → IF→IR (g r)))
```

The action of IF→IR on the path constructor squash is given by the proof that decode C P has propositional fibers. In the other direction, we construct a family of functions IR→IF : $\forall\,\{o\} \to$ Fiber (decode C P) $o \to$ FreePM C P o by:

```
IR→IF (fib (η o p)) = η o p
IR→IF (fib (fix o (c , g))) = fix o (c , λ r → IR→IF (g r))
```

The proof that IF→IR and IR→IF witnesses a family of isomorphisms between FreePM C P and Fiber (decode C P) follows simply from these families being h-props. By univalence and function extensionality this is also enough to construct a path between propositions FreePM C P and Fiber (decode C P). Moreover, we can also prove the following function is an equivalence:

```
IR→∑IF : FreeSTExt C P → ∑[ o ∈ O ] FreePM C P o
IR→∑IF x = decode C P x , IR→IF x
```

In particular, it is sufficient to prove this function has contractible fibers. To this end, we will first show IR→IF has propositional fibers, and then give a construction for $f : \forall\ y \to \mathsf{Fiber}\ \mathsf{IR{\to}IF}\ y$. We begin by observing that the function $\pi_1 \circ \mathsf{IR{\to}IF}$, i.e. the composition with the first projection, is definitionally equal to $\mathsf{decode}\ C\ P$, which we have already shown to have propositional fibers. It is provable in HoTT that for any $A\ B : \mathsf{Type}$ and $C : B \to \mathsf{Type}$, if C is a family of propositions then given a function $f : A \to \sum[\, b \in B\,]\ C\ b$, if $\pi_1 \circ f$ has propositional fibers then so does f; the details can be found in our Cubical Agda library. Given that $\mathsf{FreePM}\ C\ P$ is a family of propositions, it then follows that $\mathsf{IR{\to}{\textstyle\sum}IF}$ has propositional fibers.

To complete the proof that the fibers of $\mathsf{IR{\to}IF}$ are contractible, it suffices to construct a term $\mathsf{Fiber}\ \mathsf{IR{\to}IF}\ (o, x)$ for each $o : O$ and $x : \mathsf{FreePM}\ C\ P\ o$. We proceed by induction on x, and note that our construction will respect the path constructor squash as a consequence of eliminating into a h-prop, i.e. $\mathsf{Fiber}\ \mathsf{IR{\to}IF}\ (o, x)$. For the case where $x = \eta\ o\ p$, we construct the term $\mathsf{fib}\ (\eta\ o\ p)$. If $x = \mathsf{fix}\ o\ (c\ ,\ g)$, we first construct the following path:

$$
\begin{aligned}
&p : (o\ ,\ \mathsf{fix}\ o\ (c\ ,\ \mathsf{IR{\to}IF} \circ \mathsf{IF{\to}IR} \circ g)) \equiv (o\ ,\ \mathsf{fix}\ o\ (c\ ,\ g))\\
&p\ i = o\ ,\ \mathsf{squash}\ o\ (\mathsf{fix}\ o\ (c\ ,\ \mathsf{IR{\to}IF} \circ \mathsf{IF{\to}IR} \circ g))\ (\mathsf{fix}\ o\ (c\ ,\ g))
\end{aligned}
$$

Finally, we transport the term $\mathsf{fib}\ (\mathsf{fix}\ o\ (c\ ,\ \mathsf{IF{\to}IR} \circ g))$ along p, in the family $\mathsf{Fiber}\ \mathsf{IR{\to}IF}$, to construct a term of type $\mathsf{Fiber}\ \mathsf{IR{\to}IF}\ (o\ ,\ \mathsf{fix}\ o\ (c\ ,\ g))$ as required.

Example

We conclude this section by applying the formalisation of the IR encoding of subtypes to the same example used in Section 2.7 to illustrate the IF encoding of subtyping conditions. In particular, we will consider an encoding for the subtype of lists for which any two adjacent elements are related by a mere transitive relation $\_{\sim}\_ : A \to A \to \mathsf{Type}$. Furthermore, we will make use of the same indexed container $C : \mathsf{Container}\ (\mathsf{List}\ A)\ (\mathsf{List}\ A)$ as defined previously.

We can encode the given subtype on lists as the free subtype extension on C applied to $\mathsf{isRelated}\ \_{\sim}\_$ as follows:

$$\mathsf{RelList} = \mathsf{FreeSTExt}\ C\ (\mathsf{isRelated}\ \_{\sim}\_)\,.$$

In this way, the subtyping condition is then encoded by the inductive family $\mathsf{Fiber}\ (\mathsf{decode}\ C\ (\mathsf{isRelated}\ \_{\sim}\_))$. In a similar manner to the example for the IF formalisation, and given

$$\mathsf{lengthR} = \mathsf{length} \circ \mathsf{decode}\ C\ \ (\mathsf{isRelated}\ \_{\sim}\_)\ ,$$

this encoding gives rise to the following canonical constructors:

```
filterR : (P? : A → Bool) → RelList → RelList
removeR : (xs : RelList) (i : Fin (lengthR xs)) → RelList
```

For example, we can construct filterR as follows:

```
filterR P? xs =
  let ys = decode C (isRelated _∼_) xs
   in fix ys (filterC P? ys , λ t → fib xs)
```

2.9 Generalising our technique

Our technique can be generalised in a natural manner, by considering an encoding for dependent sums whose second component is an arbitrary type, rather than just a proposition. The advantage of such an encoding for generalised dependent sums is the flexibility to control precisely when selected operations on a type are normalised. We begin this section with the motivating example of finite lists, and then provide a formalisation of the generalised approach for encoding arbitrary type families $A : O \to \mathsf{Type}$ together with a collection of operations represented by a container $C : \mathsf{Container}\ O\ O$.

As an alternative to the typical encoding of finite lists, we can instead first consider *vectors*, i.e. families of finite lists indexed by their length:

```
data Vec (A : Type) : ℕ → Type where
  [] : Vec A 0
  _::_ : A → Vec A n → Vec A (suc n)
```

Using vectors, we can encode finite lists of A as the sum over the family $\mathsf{Vec}\ A : \mathbb{N} \to \mathsf{Type}$. That is, we define $\mathsf{List}\ A = \sum[\ n \in \mathbb{N}\ ]\ \mathsf{Vec}\ A\ n$, which can be shown to be equivalent to the typical presentation of lists. In particular, this representation requires that any operation that constructs a list must also compute its length, and therefore its length can be retrieved in constant time by projecting the first component. For example, concatenation of two lists is defined by addition of the first components, and application of the operation $\_{++}\_ : \mathsf{Vec}\ A\ m \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (m + n)$ on the second components.

As an application of our generalised technique, we will consider encoding finite lists in such a manner that it is possible to control when the operation $\_{++}\_$ is normalised. We can first observe that the definition of $\mathsf{List}\ A$ using vectors does not fit the required scheme to apply the techniques we have described thus far, as for any $n > 0$ if A is not a proposition then neither is $\mathsf{Vec}\ A\ n$. However, the first step of the generalisation to our IF approach follows in an identical fashion, i.e. by defining a new inductive family $\mathsf{AVec}\ A : \mathbb{N} \to \mathsf{Type}$, with two constructors $\eta : \mathsf{Vec}\ A\ n \to \mathsf{AVec}\ A\ n$ and $\_{++}\_ : \mathsf{AVec}\ A\ m \to \mathsf{AVec}\ A\ n \to \mathsf{AVec}\ A\ (m + n)$. In particular, we intend for η to be an embedding from $\mathsf{Vec}\ A\ n$ into $\mathsf{AVec}\ A\ n$, and for the constructor $\_{++}\_$ to encode the operation $\_{++}\_$.

In order to establish an equivalence between $\mathsf{AVec}\ A\ n$ and $\mathsf{Vec}\ A\ n$, we can quotient $\mathsf{AVec}\ A\ n$ such that $\_{++}\_$ corresponds to concatenation of the *represented vectors*. To do this, it is necessary to be able to refer to the vectors being represented by terms of $\mathsf{AVec}\ A\ n$ in its own definition. This corresponds to mutually defining the family $\mathsf{AVec}\ A$ together with a family of functions $\mathsf{vect} : \mathsf{AVec}\ A\ n \to \mathsf{Vec}\ A\ n$ that map a term of $\mathsf{AVec}\ A\ n$ to the vector it represents. Concretely, we define the quotiented family $\mathsf{AVec}\ A$ as follows:

```
data AVec A where
  η : Vec A n → AVec A n
  _++_ : AVec A m → AVec A n → AVec A (m + n)
  eq : (xs : Vec A m) (ys : Vec A n) →
       η (vect xs ++ vect ys) ≡ xs ++ ys
```

We then define the function vect as:

```
vect (η as) = as
vect (xs ++ ys) = vect xs ++ vect ys
vect (eq xs ys i) = vect xs ++ vect ys
```

Importantly, the path constructor eq is sufficient to construct a path of the type $\eta\ (\mathsf{vect}\ as) \equiv as$ for all $n : \mathbb{N}$ and $as : \mathsf{AVec}\ A\ n$. Following from the definitional equality vect $(\eta\ as) = as$, this is enough to establish that the constructor η and the function vect witness a family of isomorphisms between Vec A and AVec A. Consequently, it is sufficient to establish an equivalence between the types $\sum[\, n \in \mathbb{N}\,]$ Vec $A\ n$ and $\sum[\, n \in \mathbb{N}\,]$ AVec $A\ n$.

By encoding vectors using the type AVec A, we can precisely control when concatenation is normalised. In particular, it is possible to define a function normalise : AVec $A\ n \to$ AVec $A\ n$ for this purpose, by composing vect with η. Indeed, this is similar to the approach of normalisation-by-evaluation, in which terms are evaluated and then reflected back into the syntax. In the case of our generalised encoding this reflection map is η and is an equivalence. More generally, we can use this approach to delay normalisation until after applying functions that do not require terms to be in a normalised form, e.g. the family of functions map : $(A \to B) \to$ AVec $A\ n \to$ AVec $B\ n$.

As can be observed with our finite lists example, in contrast to our definition of free propositional monads, the generalisation of our IF approach requires that we have a C-algebra $\alpha : (o : O) \to [\![\, C\,]\!]\ A\ o \to A\ o$ before defining our new representation of A. The generalised IF approach proceeds by mutually defining a higher inductive family FreeRep $C\ A\ \alpha : O \to$ Type and a recursive function rec $\alpha : \forall\ \{o\} \to$ FreeRep $C\ A\ \alpha\ o \to A\ o$. First, we define FreeRep $C\ A\ \alpha$ by:

```
data FreeRep C A α where
  η : ∀ o → A o → FreeRep C A α o
  fix : ∀ o → ⟦ C ⟧ (FreeRep C A α) o → FreeRep C A α o
  eq : ∀ o c g → η o (α o (c , rec α ∘ g)) ≡ fix o (c , g)
```

Then the function rec α is constructed as follows:

```
rec α (η o a) = a
rec α (fix o (c , g)) = α o (c , λ r → rec α (g r))
rec α (eq o c g i) = α o (c , λ r → rec α (g r))
```

We say that FreeRep $C\ A\ \alpha$ is a *freely represented family* for A over the container C. Given any $o : O$ and x : FreeRep $C\ A\ \alpha\ o$, we can extend the path constructor eq to a family of paths eq-$\eta\ x : \eta\ o$ (rec $\alpha\ x) \equiv x$ using the definition:

```
eq-η (η o a) = refl
```

$$\begin{aligned}
&\mathsf{eq\text{-}}\eta\ (\mathsf{fix}\ o\ (c\ ,\ g)) = \mathsf{eq}\ o\ c\ g \\
&\mathsf{eq\text{-}}\eta\ (\mathsf{eq}\ o\ c\ g\ i)\ j = \mathsf{eq}\ o\ c\ g\ (i \wedge j)
\end{aligned}$$

The functions $\mathsf{rec}\ \alpha$ and $\eta\ o$ establish an isomorphism between the types $\mathsf{FreeRep}\ C\ A\ \alpha\ o$ and $A\ o$. Hence the inductive family $\mathsf{FreeRep}\ C\ A\ \alpha$ gives an alternative representation of A, for which the operations encoded by C can be expressed in a manner that their computation is delayed until transporting along the isomorphism.

It is also possible to define an eliminator for $\mathsf{FreeRep}\ C\ A\ \alpha$ into a family of types $B : O \to \mathsf{Type}$. This eliminator requires a C-algebra $\beta : (o : O) \to [\![\ C\ ]\!]\ B\ o \to B\ o$ which has B as its carrier, and a family of functions $f : (o : O) \to A\ o \to B\ o$ such that the following diagram commutes in the slice category over O:

$$\begin{tikzcd}
{[\![\ C\ ]\!]\ (\mathsf{FreeRep}\ C\ A\ \alpha)} \arrow[d, "{[\ \mathsf{rec}\ \alpha\ ]}"'] & & {[\![\ C\ ]\!]\ B} \arrow[dr, "\beta"] & \\
{[\![\ C\ ]\!]\ A} \arrow[urr, "{[\ f\ ]}"] \arrow[rr, "\alpha"'] & & A \arrow[r, "f"'] & B
\end{tikzcd}$$

That is, under the image of $[\ \mathsf{rec}\ \alpha\ ]$, the family of functions f must be an algebra homomorphism from α to β. Concretely, given a family of paths $p : (o : O) \to f\ o \circ \alpha\ o \circ [\ \mathsf{rec}\ \alpha\ ] \equiv \beta\ o \circ [\ f\ o\ ] \circ [\ \mathsf{rec}\ \alpha\ ]$ along with a term $x : \mathsf{FreeRep}\ C\ A\ \alpha\ o$, we can constructor an eliminator $\mathsf{elimRep}\ \alpha\ \beta\ f\ p\ x : B\ o$ as follows:

$$\begin{aligned}
&\mathsf{elimRep}\ \alpha\ \beta\ f\ p\ (\eta\ o\ a) = f\ o\ a \\
&\mathsf{elimRep}\ \alpha\ \beta\ f\ p\ (\mathsf{fix}\ o\ (c\ ,\ g)) = \\
&\quad \beta\ o\ (c\ ,\ \lambda\ r \to \mathsf{elimRep}\ \alpha\ \beta\ f\ p\ (g\ r)) \\
&\mathsf{elimRep}\ \alpha\ \beta\ f\ p\ (\mathsf{eq}\ o\ c\ g\ i) = \\
&\quad p\ o\ i\ (c\ ,\ \lambda\ r \to \mathsf{elimRep}\ \alpha\ \beta\ f\ p\ (g\ r))
\end{aligned}$$

An interesting application of this eliminator is to define the functorial action of $\mathsf{FreeRep}\ C$ on morphisms between C-algebras. We remark that the correct notion of morphism between $\mathsf{FreeRep}\ C\ A\ \alpha$ and $\mathsf{FreeRep}\ C\ B\ \beta$ is an algebra homomorphism between the algebras given by $\mathsf{rec}\ \alpha$ and $\mathsf{rec}\ \beta$. Again, our library gives the details.

The generalised IF approach can also be translated into a generalisation of the IR approach. In particular, for every type A and C-algebra α :

$(o : O) \to [\![C]\!]\ A\ o \to A\ o$ we can mutually define a higher inductive type $\mathsf{FreeRepIR}\ C\ A\ \alpha : \mathsf{Type}$,

$$\begin{aligned}
&\mathsf{data}\ \mathsf{FreeRepIR}\ C\ A\ \alpha\ \mathsf{where}\\
&\quad \eta : \forall\ o \to A\ o \to \mathsf{FreeRepIR}\ C\ A\ \alpha\\
&\quad \mathsf{fix} : \forall\ o \to [\![C]\!]\ (\mathsf{Fiber}\ (\pi_1 \circ \mathsf{rec}\ \alpha))\ o \to \mathsf{FreeRepIR}\ C\ A\ \alpha\\
&\quad \mathsf{eq} : \forall\ o\ c\ g \to \eta\ o\ (\alpha\ o\ (c\ ,\ \mathsf{rec}\pi_2\ \alpha \circ g)) \equiv \mathsf{fix}\ o\ (c\ ,\ g)
\end{aligned}$$

a function $\mathsf{rec}\pi_2\ \alpha : \forall\ \{o\} \to \mathsf{Fiber}\ \ (\pi_1 \circ \mathsf{rec}\ \alpha)\ \ o \to A\ o$,

$$\mathsf{rec}\pi_2\ \alpha\ (\mathsf{fib}\ x) = \pi_2\ (\mathsf{rec}\ \alpha\ x)$$

and a function $\mathsf{rec}\ \alpha : \mathsf{FreeRepIR}\ C\ A\ \alpha \to \sum[\ o \in O\ ]\ A\ o$:

$$\begin{aligned}
&\mathsf{rec}\ \alpha\ (\eta\ o\ a) = o\ ,\ a\\
&\mathsf{rec}\ \alpha\ (\mathsf{fix}\ o\ (c\ ,\ g)) = o\ ,\ \alpha\ o\ (c\ ,\ \mathsf{rec}\pi_2\ \alpha \circ g)\\
&\mathsf{rec}\ \alpha\ (\mathsf{eq}\ o\ c\ g\ i) = o\ ,\ \alpha\ o\ (c\ ,\ \mathsf{rec}\pi_2\ \alpha \circ g)
\end{aligned}$$

In a similar manner to the generalisation of the IF approach, the eq path constructor can be extended to a family of paths

$$\mathsf{eq}\text{-}\eta\ x : \mathsf{uncurry}\ \eta\ \ (\mathsf{rec}\ \alpha\ x) \equiv x$$

In particular, this means that for all $o : O$, $\mathsf{rec}\ \alpha$ and $\mathsf{uncurry}\ \eta$ witness an isomorphism between $\mathsf{FreeRepIR}\ C\ A\ \alpha$ and $\sum[\ o \in O\ ]\ A\ o$.

2.10 Strictification

In addition to providing fine-grained control of unfolding behaviour, the encodings presented in this chapter can be used for *coherence constructions* that present a *strictification* for a chosen collection of operations. That is, our technique allows us to construct a model for which the interpretations of a chosen algebra hold strictly from one in which they only hold weakly. As an example, we will consider the strictification of the dependent sum construction for a universe that is internally represented à la Tarski. In particular, a Σ-closed Tarski universe is given by a type of codes $U : \mathsf{Type}$ with interpretations $[\![\_]\!] : U \to \mathsf{Type}$, and a dependent sum type former

$$\hat{\sum} : (A : U) \to ([\![A]\!] \to U) \to U,$$

together with a proof that its interpretation is equivalent to the Σ-type in the ambient type theory:

$$[\![\hat{\sum}]\!] : (A : U)\ (B : [\![\ A\ ]\!] \to U) \to [\![\ \hat{\sum}\ A\ B\ ]\!] \simeq \textstyle\sum[\ a \in [\![\ A\ ]\!]\ ]\ [\![\ B\ a\ ]\!].$$

Notably, $[\![\hat{\sum}]\!]$ need not be a family of strict equivalences and this is indeed the case for many common examples such as the universe of inductive finite types whereby $U = \mathbb{N}$ and $[\![\_]\!] = \mathsf{Fin}$. However, by application of the higher inductive-recursive encoding presented in Section 2.8 we can define the following higher inductive datatype:

```
data FreeU : Type where
  Code^f : U → FreeU
  ∑^f : (A : FreeU) → (fst (ElContr A) → FreeU) → FreeU
  eq :  ∀ x y → decode x ≡ decode y → x ≡ y
```

We define FreeU mutually with two recursive functions $\mathsf{decode} : \mathsf{FreeU} \to U$ and $\mathsf{ElContr} : (A : \mathsf{FreeU}) \to \sum[\ X \in \mathsf{Type}\ ]\ (X \simeq [\![\ \mathsf{decode}\ A\ ]\!])$. We define the decode function as follows:

```
decode (Code^f A) = A
decode (∑^f A B)
  = ∑̂ (decode A) λ a → decode (B (invEq (snd (ElContr A)) a))
decode (eq x y p i) = p i
```

In order to define the function ElContr we begin by observing that its codomain is contractible, i.e. $\sum[\ X \in \mathsf{Type}\ ]\ (X \simeq [\![\ \mathsf{decode}\ A\ ]\!])$. As such, any definition of ElContr that matches on the constructors of FreeU will respect the eq path constructor. The first projection of ElContr X is given by $[\![\ A\ ]\!]$ when $X = \mathsf{Code}^f\ A$ and $\sum[\ a \in \mathsf{fst}\ (\mathsf{ElContr}\ A)\ ]\ \mathsf{fst}\ (\mathsf{ElContr}\ (B\ a))$ when $X = \sum^f\ A\ B$. While we do not outline the second projections of ElContr X here, they are given by routine constructions of equivalences.

As shown in Section 2.8, we can construct an equivalence between the original type of codes U and the inductive-recursive type FreeU. Moreover, we can construct a new Σ-closed Tarski universe with codes given by FreeU, interpretations given by $\mathsf{fst} \circ \mathsf{ElContr} : \mathsf{FreeU} \to \mathsf{Type}$ and dependent sums given by $\sum^f$. The key result is that the interpretation of dependent sums in our newly constructed universe is now definitionally equal to the dependent sum in the ambient type theory. While we do not give the full proof here, it is possible to prove that not only is U equivalent to FreeU but also that our new 'stricter' universe is equal up to path to the original universe.

2.11 Related work

In this section we compare with two existing approaches in Agda to proofs whose content does not matter, *irrelevancy annotations* and *abstract definitions*, and with the use of the *Prop universe* in Coq and Agda.

In Agda, a term can be annotated as irrelevant in cases where its content will not be used in any proof-relevant constructions, and doing so prevents unfolding of the annotated term. The key distinction between our technique and irrelevancy annotations is the preservation of *computational content*. In particular, annotating proofs of the subtyping condition as irrelevant is problematic in a similar manner to that of using a strict universe of propositions, namely that we cannot then use the content of a proof in any proof-relevant context. For example, this means that we would be unable to use a proof that a natural number is even in order to construct the function div2 : isEven! $n \to \mathbb{N}$ that divides an even number by two, as discussed in Section 2.3.

Agda's abstract definition mechanism can be used to hide the implementation details of a subtype, and expose operations as irreducible terms. Abstract definitions are similar to our technique when defining a subtype X : Type within an abstract block, together with the first projection El : $X \to O$ and a term witnessing that the function El is an embedding. In particular, this is enough to construct paths corresponding to equational properties of operations on X outside of the block. However, abstract definitions still require explicit proofs of closure under the subtyping condition for operations within the abstract block. In contrast, our encoding only requires a proof that the encoded operations are coherent with respect to the subtyping condition in cases where we need to construct and transport along an isomorphism between our alternative representation of a subtype and its original definition. In this manner, encoding operations using our technique is similar to postulating them, with the important distinction that computational properties are preserved.

A third approach, provided in both Agda and Coq, is to use a universe of computationally irrelevant propositions, typically named Prop. Agda has a predicative hierarchy of Prop universes wherein pattern matching is restricted to only the absurd pattern on elimination into proof-relevant types, thereby preventing computational content from leaking. Meanwhile, Coq has a single impredicative Prop universe that also allows for matching on inductively defined propositions with a single constructor when eliminating into proof-relevant types. This is known as the singleton-elimination principle, and by application to the identity type in Coq it is possible to show the uniqueness of identity proofs, and is therefore inconsistent with univalence. As a closer analogue to Agda's Prop universe, Coq has an impredicative universe of definitionally proof-irrelevant propositions termed SProp that is consistent with univalence [Gilbert, Gaëtan and Cockx, Jesper and Sozeau, Matthieu and Tabareau, Nicolas, 2019]. The primary drawback to using a definitionally proof-irrelevant Prop universe to prevent unnecessary reduction behaviour of terms is the inability to use proof content in any proof-relevant context. In contrast, our approach provides the advantage of preserving proof content, while preventing unnecessary reduction behaviour.

Opaque definitions [Gratzer et al., 2022] are a recent development in type theory that provide fine-grained control over unfolding behaviour and have been implemented in Agda as of version 2.6.5. In contrast to the techniques outlined in this chapter, opaque definitions are a language-level feature and require built-in support to use. A key benefit to this approach is that significant overhead can be automatically managed by tooling. Consequently, opaque definitions are both modular and particularly easy to use. However, opaque definitions do not share precisely the same use cases as that of the technique described in this chapter. For example, our encoding of subtypes can be used to construct algorithms that are more performant at runtime as discussed in Section 2.5.

2.12 Conclusion

In this chapter we presented a new technique that allows for operations on a subtype to be represented in a manner that exhibits no reduction behaviour. Our approach does not require special-purpose language extensions, can be used in any implementation of type theory that supports quotient inductive types, and retains important computational properties.

Interesting topics for further work include generalising our approach by introducing equational properties between operations as path constructors of the encoded subtype, generalising the equivalence between inductive-recursive types and inductive families to their higher counterparts by applying the transformation used in our technique, and developing a wider range of combinators for working with subtypes.

Furthermore, from the point of view of extensibility, it would be beneficial to identify a sufficient condition on a collection of operations such that the typical elimination principle for an encoded subtype can be recovered simply by dependent pattern matching in the sense of Cockx et al. [2014].

Chapter 3

Quotient Haskell

In the previous chapter, we explored a specialised encoding for subtypes in homotopy type theory. This particular flavor of subtype involved equipping an underlying type with a proposition determining whether each term was an element of the subtype. A similar notion of subtype is introduced in the literature of *refinement types*, whereby a type is equipped with a decidable predicate. Indeed, refinement types can be formalised in homotopy type theory as subtypes whose subtyping condition is built from a decidable expression language.

Refinement types [Freeman and Pfenning, 1991] are a class of subtypes for which the subtyping predicate, or *refinement*, is SMT-decidable. Restricting ourselves to this class of subtypes has an important practical benefit: a type-checker that utilises an SMT-solver can automate many of the proof obligations that arise from refinements. Indeed, this is a central feature of Liquid Haskell [Vazou, 2016], an extension of Haskell with refinement types. A simple example of a refinement type is the even integers, expressed in Liquid Haskell by `{n:Int | n % 2 == 0}`. For any concrete integer, and for each of the common arithmetic operations, Liquid Haskell can check the evenness condition without requiring that we manually construct a proof. Importantly, the equality operator used in the definition of even numbers does not correspond to that of Haskell's `Eq` typeclass. In particular, when `==` appears in a subtyping predicate it corresponds to the notion of equality in the refinement logic of Liquid Haskell.

Subtypes also have a well-known dual construction, *quotient types*, which are given by a type together with a collection of equations. When considered in the framework of homotopy type theory, quotient types can be understood as set-truncated higher-inductive types, as introduced in Chapter 2. That is, quotient types are higher-inductive types whose proof content may only be used in the construction of propositions. While few languages currently support their use, there are many practical examples of quotient types, including algebraic structures such as monoids, and data structures such as bags. Intuitively, a subtype requires that we prove its predicate is respected on construction, while a quotient type requires that we prove that its equations are respected on elimination. In this manner, both subtypes and quotient types introduce proof-obligations, which in turn may require tedious manual proof construction in the absence of sufficient automated proof search.

In contrast to subtypes, automated proof for quotient types is much less explored. Indeed, the only languages with quotient types at present seem to be proof assistants. However, quotient types share an important practical utility with subtypes: they allow us to assert static properties that we hope can be validated by a type-checker. For example, the type of bags (multisets) can be expressed by quotienting the type of lists with a quotient `swap :: x:a -> y:a -> xs:[a] -> x:y:xs == y:x:xs`. In particular, this quotient requires that for every function `f :: Bag a -> b` that takes a bag as an argument, we must have `f (x:y:xs) == f (y:x:xs)`, i.e. the behaviour of `f` must be invariant under permutation of the elements. In the absence of automated proof tools, this means supplying a manually constructed proof of this equality. In practice, this can quickly become burdensome and is a significant barrier to the use of quotient types in general programming.

In this chapter, we introduce *Quotient Haskell*, an extension of Liquid Haskell that extends the notion of refinement types to support a class of quotient types whose equational laws are SMT-decidable. In particular, our system supports quotient inductive types [Altenkirch and Kaposi, 2016], a class of quotient types that simultaneously define an inductive data type alongside equational laws on its elements. The soundness of our class of quotient inductive types requires that we consider only the total fragment of Haskell. Therefore, we will assume throughout this chapter that Liquid Haskell's totality flag is enabled. This flag turns on totality, strict positivity and termination checking. As of version 0.9.10.1, Liquid Haskell's termination checker does not permit corecursive definitions and it is therefore not possible to soundly reason about coinductive types. Consequently, both the core language and implementation that we present in this chapter will only consider quotients of *inductive* types.

Related work is discussed in Section 3.10, and we reflect on the design, practical use and limitations of the system in Section 2.5. We assume a basic knowledge of Haskell, but to make this chapter more accessible we do not require expertise with type theory, quotient types or Liquid Haskell. While Haskell serves as the implementation vehicle in this chapter, the ideas introduced by the core language are applicable to any programming language with a liquid type system. The Quotient Haskell system itself is freely available online as supplementary material [Hewer, 2023].

3.1 Mobiles

To describe the class of quotient types that we introduce in this chapter and are implemented by Quotient Haskell, we present three increasingly involved examples. The provided examples outline the necessary concepts required to use Quotient Haskell. In this section, we explore the first of these examples, *mobiles*, which are commutative trees in which subtrees with the same parent can freely be swapped. For the purposes of this example, we will consider binary trees, defined in Haskell as follows, but the same idea also applies to rose trees and multiway trees:

```
data Tree a = Leaf | Bin a (Tree a) (Tree a)
```

In usual parlance, `Leaf` and `Bin` are the *data constructors* of the `Tree` type. A quotient type extends the typical notion of an algebraic datatype with a new kind of constructor, that we shall refer to as an *equality constructor*. In the type-theory literature, equality constructors can also be referred to as *path constructors*. However, while a data constructor introduces new terms of a data type, an equality constructor introduces new *equalities* between terms of a type. For example, in order to define mobiles we will require an equality constructor that quotients `Tree` to assert the commutativity condition. That is, we require an equality constructor of the following form:

```
swap :: x:a -> l:Tree a -> r:Tree a -> Bin x l r == Bin x r l
```

In the above definition of `swap` we have made use of the dependent syntax of Liquid Haskell. In particular, a type of the form `x:a -> P` whereby `P` is a proposition can be understood as universal quantification over the elements of `a` and can be read as 'for all x of type a, P holds'. While the syntax used for equality in the above definition is shared by Haskell's `Eq` typeclass, it instead refers to a type-level proposition and this syntax is inherited from Liquid Haskell.

Note that the target of the `swap` constructor introduces an equality between trees. In particular, `swap` asserts that two trees with swapped children must be treated identically. Concretely, we can define the datatype of binary mobiles in Quotient Haskell as follows:

```
data Mobile a
  =  Tree a
  |/ swap :: x:a -> l:Mobile a -> r:Mobile a
          -> Bin x l r == Bin x r l
```

As with any refinement in Liquid Haskell, this definition must be given within a *{−@ @−}* block. However, for brevity we omit these additional annotations throughout this chapter. As shown in the above example, a quotient type in Quotient Haskell is defined with a similar syntax to that of ADTs, with some notable differences. Firstly, an underlying type must directly follow after the equality symbol. For example, in the above definition of mobiles, the underlying type is `Tree a`. The underlying type can be a Liquid Haskell refinement type or even another quotiented type. After providing the underlying type, the equality constructors must follow, each preceded by the `/` character. Any equality constructor defined in a Liquid Haskell block, such as `swap`, is introduced as a term with the same name within the namespace in which it is defined. Consequently, equality constructors can be used in Liquid Haskell proofs. For example, `swap` can be understood as a Liquid Haskell proof function of the following type:

```
x:a -> l:Mobile a -> r:Mobile a -> { Bin x l r == Bin x r l }
```

Importantly, the left-hand term of the equality target of an equality constructor must be in *canonical form*, i.e. the left-hand term must be a valid match term. This is important in circumstances of induction-recursion, whereby a function on an inductive type appears in any of its equality constructors. A formal account of this rule and an explanation for its existence is given in Section 3.9. The right-hand term of the equality may vary freely, as long as it has the correct type.

Note that quotient types in Quotient Haskell introduce a new type constructor into the Liquid Haskell namespace. That is, a quotient type definition can be read as defining a new type rather than simply giving a refinement for an existing type. This is in contrast to the usual approach of datatype refinement in Liquid Haskell, whereby type constructors refer directly to their underlying GHC counterparts. We adopt this alternative approach for the practical purpose of avoiding having to create a newtype for every quotient type defined on an underlying type. To explain the issue, let us consider how the usual Liquid Haskell approach to data type refinement could be adopted for our `Mobile` example. In particular, this approach would involve redefining `Tree` in a Liquid Haskell block, and then appending the `swap` constructor. In keeping with the terminology used in Liquid Haskell, we say that the type `Tree` has been *refined* to the type of mobiles. Consequently, any function defined on the type `Tree` must be a function on mobiles, i.e. must respect the `swap` constructor. As such, to define multiple quotient types on the same underlying `Tree` type, such as sets or bags, we would need require a distinct type definition for each.

Crucially, every binary tree is a binary mobile. As described in more detail in Section 3.5, this imposes an ordering relation on types subject to quotienting. A key practical concept of quotient types is that, unlike subtypes, there are no proof obligations imposed by quotienting on term construction. Rather, an equality constructor introduces proof obligations on term elimination. Intuitively, subtyping imposes conditions when *building* a term and quotienting imposes conditions when *using* a term. For example, for the `swap` equality constructor this means that for any function of the form `f :: Mobile a -> b`, we require that `f (Bin x l r) = f (Bin x r l)`. A function that does not respect this law is not a valid function on mobiles, and we should expect this to be verified by our type checker. Indeed, this is precisely the verification that Quotient Haskell can be used for. For example, the following function on trees cannot be refined to a function on mobiles:

```
isLeft :: Tree Bool -> Bool
isLeft Leaf = False
isLeft (Bin p Leaf z) = p
isLeft (Bin p (Bin q x y) z) = q
```

Evidently, the function `isLeft` distinguishes between the left and right subtrees when they do not contain the same logical value. In contrast, an example of a function on trees that can be refined to a function on mobiles is the following summation function:

```
sum :: Tree Int -> Int
sum Leaf = 0
sum (Bin n l r) = n + sum l + sum r
```

Because addition is commutative, we should expect that the `sum` function can be refined over binary mobiles. That is, we should expect

```
sum :: Mobile Int -> Int
```

to be a valid typing judgement. The necessary proof obligation imposed by the `swap` equality constructor is then given by

```
sum (Bin n l r) == sum (Bin n r l)
```

for all `n :: Int`, `l :: Mobile Int` and `r :: Mobile Int`. After unfolding the definition of `sum` on both sides of the equality, we can observe that the necessary proof follows from commutativity of addition. Indeed, a constraint of this form can be solved by Liquid Haskell with no further intervention.

A second example of a function on mobiles trees is that of the `map` function. We recall that the typical definition for the `map` function on binary trees is given as follows:

```
map :: (a -> b) -> Tree a -> Tree b
map f Leaf = Leaf
map f (Bin x l r) = Bin (f x) (map f l) (map f r)
```

We expect the type of `map` to refine to `(a -> b) -> Mobile a -> Mobile b`. In this example, after unfolding definitions and eliding quantification, the necessary condition that must hold is:

```
Bin (f x) (map f l) (map f r) == Bin (f x) (map f r) (map f l)
```

This equality is witnessed by the term `swap (f x) (map f l) (map f r)`, and our implementation of Quotient Haskell is capable of automatically applying this single proof step.

By making use of Liquid Haskell's refinement types, we can define the general `fold` on mobiles. Recall that the general `fold` on binary trees is given as follows:

```
fold :: (a -> b -> b -> b) -> b -> Tree a -> b
fold f z Leaf = z
fold f z (Bin x l r) = f x (fold f z l) (fold f z r)
```

We cannot naively refine `fold` by replacing `Tree` with `Mobile`, as in general we do not have:

```
fold f z (Bin x l r) == fold f z (Bin x r l)
```

After unfolding definitions on both sides of the equality we can observe that this condition only holds when the function `f` is symmetric in its second and third arguments. More specifically, this means that we have a family of equalities of the following form:

```
x:a -> l:b -> r:b -> f x l r = f x r l
```

In Liquid Haskell, we can define the type of such functions as follows:

```
type Fun a b
  = ( f : a -> b -> b -> b
    , x:a -> l:b -> r:b -> { f x l r == f x r l }
    )
```

Unfortunately, this approach to defining a subtype of the function space is not particularly ergonomic, and requires explicitly carrying proof witnesses. However, Liquid Haskell necessarily does not permit higher-order propositions to inhabit the type of propositions `Prop` and as such, this is a work-around. With this definition of `Fun`, we can define a general `fold` on mobiles by

```
fold :: Fun a b -> b -> Mobile a -> b
fold (f, p) z Leaf = Leaf
fold (f, p) z (Bin x l r)
  = f x (fold (f, p) z l) (fold (f, p) z r)
```

The witness that the `swap` constructor is respected by our new `fold` function follows from the second component of the pair. However, the version of Quotient Haskell introduced by this chapter cannot implicitly make use of the explicit proof term. As such, this definition will not type-check without additional intervention. In particular, we must explicitly construct the necessary witness that the above `fold` function respects the `swap` equality constructor. To achieve this in Quotient Haskell, we first define the following unrefined proof:

```
foldSwap :: Fun a b -> b -> a -> Tree a -> Tree a -> Proof
foldSwap (f, p) z x l r
  = p x (fold (f, p) z l) (fold (f, p) z r)
```

This is precisely the proof that `fold` respects the `swap` equality constructor. Notably, the arguments of `swap` are expanded as arguments to `foldSwap`. Any explicit proof that an equality constructor is respected must follow this form in Quotient Haskell. The next step then is to introduce the following refinement inside of a Liquid Haskell block:

```
respects<fold, swap> foldSwap
  :: f:Fun a b -> z:b -> x:a -> l:Mobile a -> r:Mobile a
  -> { fold f z (Bin x l r) == fold f z (Bin x r l) }
```

The function `foldSwap` will be checked in the same manner as any other Liquid Haskell proof. In addition, by adding the prefix `respects<fold, swap>`, Quotient Haskell will check whether `foldSwap` is a valid witness that `fold` respects the `swap` constructor. A valid witness of a quotient being respected is simply a function whose type precisely matches the relevant respectability theorem. As our definition of `foldSwap` is both of the correct type and a valid proof, it can be used as an explicit witness that `fold` respects the `swap` equality constructor.

Notably, there are circumstances under which the constraints generated for an equality constructor cannot be automatically verified by Quotient Haskell. This particular limitation is inherited from Liquid Haskell which only implements a limited form of inductive reasoning by means of its proof by logical evaluation algorithm. Indeed, this use case is one of the primary reasons that Liquid Haskell offers a mechanism for writing manual proofs. As such, Quotient Haskell expands upon this framework to support writing manual proofs for respectfulness theorems that cannot at present be automatically verified. For many practical cases, such as the examples provided through this chapter, this limitation does not arise.

As an example of using our general `fold` function for mobiles, we can consider using it to redefine our previous `sum` function. To do this, we first define a function `add3 :: Int -> Int -> Int -> Int` that adds three integers, which can then be reflected to define the following proof witness:

```
add3Comm :: x:Int -> y:Int -> z:Int
         -> { add3 x y z == add3 x z y }
add3Comm x y z = trivial *** QED
```

In Liquid Haskell, the triviality of the above proof follows from the triviality of the commutativity of addition. Finally, we can simply define `sum = fold (add3, add3Comm) 0`.

3.2 Boom hierarchy

In this section, we explore the family of datatypes comprising trees, lists, bags and sets, which are collectively known as the *Boom hierarchy* [Meertens, 1986]. These examples will demonstrate how the ordering relation on quotient types can be used to practical effect.

We begin this section with a notion of a tree data structure that varies from that used in our mobiles example, in which data is only found in the leaves:

```
data Tree a = Empty | Leaf a | Join (Tree a) (Tree a)
```

This notion of tree forms the basis for defining all the other types in the Boom hierarchy. We begin this exploration with a less familiar definition of the list datatype by means of quotienting:

```
data List a
  =  Tree a
  |/ idl :: x:List a -> Join Empty x == x
  |/ idr :: x:List a -> Join x Empty == x
  |/ assoc :: x:List a -> y:List a -> z:List a
          -> Join (Join x y) z == Join x (Join y z)
```

This definition captures the idea that lists can be obtained from trees by requiring that the `Join` constructor is associative, and has `Empty` as the identity element. Intuitively, this definition of a list can be seen as a direct translation of the algebraic definition of a monoid structure on a given type. Indeed, this is precisely why lists are the *free* monoid on the parameter type.

The above formulation of lists might seem rather complex when compared to the standard version: `data List a = Nil | Cons a (List a)`. The benefit of the above quotiented formulation is that concatenation is given simply by the constructor `Join`, and thus has asymptotic runtime complexity of $O(1)$. For example, when we are primarily building lists by their monoid interface, such as when using the `Writer` monad, this can be a more performant representation. Furthermore, using the monoid laws on `[a]` we can give a well-typed unfolding `toList :: List a -> [a]` of tree-based lists into standard lists, and this has runtime complexity of $O(n)$. As such, it can sometimes be more performant to use this tree representation of lists when constructing them through repeated concatenation, and subsequently apply `toList` when required.

Examples of functions on trees that can be refined to the quotiented type `List` include:

```
sum :: Tree Int -> Int
map :: (a -> b) -> Tree a -> Tree b
filter :: (a -> Bool) -> Tree a -> Tree b
```

In contrast, an example of a function on the `Tree` datatype that cannot be refined to a function on lists is the following inductive subtraction function:

```
subtr :: Tree Int -> Int
subtr Empty = 0
subtr (Leaf n) = n
subtr (Join x y) = subtr x - subtr y
```

In particular, `subtr` does not respect the associativity condition introduced by the `assoc` equality constructor, because integer subtraction is not associative.

The next datatype in the Boom hierarchy is multisets, also known as *bags*, which can be used to count the number of occurrences of elements from a collection. Bags can intuitively be thought of as lists for which the order of the elements cannot be used, which means that the `Join` constructor must be commutative. Consequently, while `List` characterises the free monoid construction on the parameter type, our definition for `Bag` will characterise the free commutative monoid construction. In Quotient Haskell, we can define the `Bag` datatype as a quotient of `List` as follows:

```
data Bag a
  = List a
  |/ comm :: xs:Bag a -> ys:Bag a -> Join xs ys == Join ys xs
```

The definition of `Bag` is an example of further quotienting an already defined quotient type. Indeed, this is precisely how the datatypes of the Boom hierarchy can be understood to form a hierarchy. In Quotient Haskell, this hierarchy is made explicit by a subtyping relation derived from the evident ordering of quotient types, i.e. `Tree a <: List a <: Bag a`. Intuitively, `a <: b` can be read as 'every element of `a` is an element of `b`'. Furthermore, by contravariance we also have

```
(Bag a -> b) <: (List a -> b) <: (Tree a -> b)
```

Importantly, this means that in Quotient Haskell it is only necessary to refine a function to the greatest quotient type possible in a given hierarchy. For example, if we were to refine the `sum` function on `Tree` to a function on `Bag a`, then it would also be possible to apply `sum` to an element of type `List a`.

Alongside `sum`, `map` and `filter`, another useful function on trees that can be refined to a function on bags counts the number of elements that satisfy a given property:

```
countIf :: (a -> Bool) -> Tree a -> Int
countIf p Empty = 0
countIf p (Leaf a) = if p a then 1 else 0
countIf p (Join x y) = countIf p x + countIf p y
```

In order to verify that `countIf` can be refined to a function on bags, it is necessary to check that it respects both the `comm` equality constructor and all of the equality constructors given in the definition of `List`. In particular, these laws follow from the fact that integers form a commutative monoid under addition, and can be automatically verified by Quotient Haskell.

An example of a function on trees that can be refined to a function on lists but not on bags, is the following simple function that converts a tree into a list:

```
toList :: Tree a -> [a]
toList Empty = []
toList (Leaf a) = [a]
toList (Join x y) = toList x ++ toList y
```

In this example, we eliminate trees into the non-commutative monoid of the Haskell list type equipped with concatenation. Indeed, because the `toList` function constructs lists using their monoidal interface, the laws introduced by the equality constructors of `List` are evidently satisfied. However, concatenation is not commutative and hence `toList` cannot be refined to a function on bags. Of course, we should not expect to be able to convert an arbitrary bag into a list, as this would require a unique identification of an ordering on its elements.

The final type in the Boom hierarchy is *sets*, unordered collections that can only contain each element once. Sets can also be understood as bags for which any repeated occurrences of elements are forgotten. In Quotient Haskell, we can define the type of sets as follows:

```
data Set a
  = Bag a
  |/ idem :: xs:Set a -> Join xs xs == xs
```

In this definition, the `idem` equality constructor asserts that we cannot distinguish between a set and that same set unioned with itself. Alongside the `comm` equality constructor introduced in the definition of bags, this guarantees that repeated occurrences of an element cannot be used to alter the behaviour of a function on sets. Note that the term `Join xs xs` is not a valid match term in Haskell because of the duplicated variable `xs`, and therefore does not satisfy the necessary requirement for appearing on the left-hand side of an equality constructor. However, this is indeed valid syntax in Quotient Haskell, and is equivalent to the following formulation:

```
idem :: xs:Set a -> ys:Set a -> {xs == ys} -> Join xs ys == xs
```

However, the original and more concise formulation of `idem` is generally preferable in practice, as it does not introduce the extra precondition `{xs == ys}` to the type-checker.

A simple example of a function on trees that can be refined to a function on sets determines if a given value is contained in a tree, and can be defined as follows:

```
contains :: Eq a => a -> Tree a -> Bool
contains x Empty = False
contains x (Leaf y) = x == y
contains x (Join t u) = contains x t || contains x u
```

All of the necessary properties that are required to verify that `contains` refines to a function on sets follows from the fact that `Bool` forms an idempotent commutative monoid under logical disjunction. For example, in order to show that the `comm` equality constructor is satisfied we would require that `contains x (Join xs xs) == contains x xs`, which unfolds to:

```
contains x xs || contains x xs == contains x xs
```

This property is then trivially true because disjunction is idempotent. We have already seen an example of a function on trees that can be refined to a function on bags but not on sets, namely `countIf`, because the number of occurrences of an element matters for this function.

3.3 Rational numbers

In this section we show how the *rational numbers*, a classic example of a quotient type, can be captured in Quotient Haskell. The approach is quite different from their typical representation in a language such as Haskell, where a well-behaved interface is presented on an abstract type with a hidden data representation. In contrast, by defining the rationals as a quotient type, their implementation details can be exposed without compromising correctness. As discussed in this chapter, this increased flexibility comes at the cost of extra proof obligations arising from quotient respectfulness theorems. However, in Quotient Haskell such proof obligations are automatically resolved by the type checker when operating within the SMT-decidable logic of Liquid Haskell.

A well-known constructive definition of the rational numbers involves quotienting pairs of integers representing the numerator and denominator, with the proviso that the second element is non-zero to preclude division by zero. We can define the type of non-zero integers in Liquid Haskell by `type NonZero = { n : Int | n /= 0 }`. Using this type definition, we can then proceed to define the rational numbers with the following quotient type:

```
data Rational
  =  (Int, NonZero)
  |/ eqR :: w:Int -> x:Int -> y:NonZero -> z:NonZero
         -> {w * z == x * y} -> (w , y) == (x , z)
```

Note that the underlying type of `Rational` uses a refinement predicate, `n /= 0`, to ensure that the second element of a pair of integers is non-zero. More generally, Quotient Haskell allows any rank 1 liquid type to be used as the underlying type in the definition of a quotient type. The above definition uses the 'cross multiplication' approach to decide if two rationals are equal, but this is not the only possible approach. For example, we could use a greatest common divisor function `gcd :: Int -> Int -> Int` to define the following equality constructor:

```
eqGCD :: x:Int -> y:NonZero ->
          (x, y) == (x `div` gcd x y, y `div` gcd x y)
```

While the equality constructor `eqR` quantifies over four variables, `eqGCD` only requires quantification over two variables. However, this alternative presentation of rational numbers requires that the definition of `gcd` be unfolded when the type checker validates many proof obligations that arise for common applications of rational numbers. Moreover, `eqGCD` requires additional work to validate its well-formedness. In particular, it must provably be the case that if `y /= 0` then ``y `div` gcd x y /= 0``. In practice, this approach is often less performant than simply quotienting by `eqR`, and can often result in constraints being generated that require an unreasonable amount of time to be validated by an SMT solver. Indeed, as highlighted by the example of rational numbers, performance can be an important consideration in choosing a particular quotient representation. As such, we proceed with our original presentation of the rationals.

As a first example of defining functions on rational numbers, we consider a simple function that decides if a pair of integers represents a negative rational:

```
isNegative :: (Int, Int) -> Bool
isNegative (m, n) = (m < 0 && n >= 1) || (m > 0 && n <= -1)
```

We can observe that `isNegative` is a valid function on rational numbers, and should therefore expect that we can refine its type to `Rational -> Bool`. A simple but important observation in Liquid Haskell is that subtyping constraints can always be added to function inputs. The dual observation in Quotient Haskell is that outputs of functions can always be quotiented. Both observations follow from the subtyping rules of the type system. In this particular case, `isNegative` can trivially be refined to a function of type `(Int, NonZero) -> Bool`, and the type checker of Quotient Haskell need only consider whether it respects the `eqR` equality constructor. Concretely, the type checker will automatically verify that given variables `w:Int`, `x:Int`, `y:NonZero` and `z:NonZero` along with a precondition `w * z == x * y` the following unfolded condition holds:

```
((w < 0 && y >= 1) || (w > 0 && y <= -1))
  == ((x < 0 && z >= 1) || (x > 0 && z <= -1))
```

Included in the large collection of functions that can be defined on the rational numbers are the standard arithmetic operations. Notably, for operations that rationals are closed under, such as addition and multiplication, it is not necessary to simplify the pair of integers to satisfy the `eqR` equality constructor. As demonstrated in previous examples, Quotient Haskell can make use of equality constructors such as `eqR` when eliminating into a quotient type. However, this is not the case if we were to instead attempt to extract the numerator and denominator from a rational number. For example, the following projection cannot be refined to a function on rationals:

```
numerator :: (Int, Int) -> Int
numerator (m, _) = m
```

In particular, it is not the case that if `w * z == x * y` then `w == x`, and attempting to refine `numerator` to the type `Rational -> Int` will yield a type error in Quotient Haskell. Instead, we consider refining the type of the following function, which reduces a rational number to its simplest form:

```
reduce :: (Int, Int) -> (Int, Int)
reduce (m, n)
  | m == 0    = (0, 1)
  | m < 0     = (-m `div` d, -n `div` d)
  | otherwise = (m `div` d, n `div` d)
  where d = gcd m n
```

At first glance, it is natural to ask why the `reduce` function negates both elements of the pair when the first element is negative. In particular, it is not immediately evident why `-1 / 2` should be considered anymore reduced than `1 / -2`. Indeed, if we were to change the condition `m < 0` to `n < 0` the definition of `reduce` would remain equally valid. However, we cannot omit this line altogether, and it is required in order to refine the type of `reduce` to `Rational -> (Int, Int)`.

The above behaviour is not a bug in Quotient Haskell's type checker, but is essential to the correctness of the `reduce` function. In particular, the respectfulness theorem resulting from the `eqR` equality constructor ensures that we cannot distinguish between rational numbers such as `-1 / 2` and `1 / -2`, and hence we must choose a uniform way to reduce such terms. That is, we must choose a uniform way to handle the sign of the rational number. The approach used in the above definition of `reduce` is to always move the sign to the denominator, however it is equally valid to instead always move the sign to the numerator. There are other possible ways to define a `reduce` function that uniformly handles the sign of a rational number, such as by constructing a triple `(Bool, Nat, Nat)` in which a Boolean is used to represent the sign.

In addition to the uniform handling of sign, it is crucial that the `gcd` function used in the definition of `reduce` has the property that if the property `w * z == x * y` holds then the following two propositions hold:

```
abs w `div` gcd w y == abs x `div` gcd x z
abs y `div` gcd w y == abs z `div` gcd x z
```

This is evidently the case for any correct implementation of greatest common divisor. However, Liquid Haskell does not currently reflect the standard library function `gcd` in its type system. In practice, this means that `gcd` needs to be redefined. Many of the standard approaches to such a definition, such as Euclid's algorithm or recursively applying the modulus function, verifiably exhibit the necessary property when proof by logical evaluation (PLE) is enabled in Liquid Haskell. In Quotient Haskell, PLE is enabled by default. Importantly, the described property of the `gcd` function, along with the η-rule for the product type, are precisely the equations used by Quotient Haskell to verify that `reduce` can indeed be refined to `reduce :: Rational -> (Int, Int)`. Moreover, we can choose to further refine the type to `reduce :: Rational -> (Int, NonZero)`, whereby the second component result can verifiably be shown to always be non-zero. Using this definition of `reduce`, we can compose with the projections `fst` and `snd` and to obtain the functions `numerator :: Rational -> Int` and `denominator :: Rational -> NonZero`, respectively.

3.4 Quotient inductive types

As described in earlier sections of this chapter, Quotient Haskell implements a class of quotient types known as quotient inductive types. Moreover, Quotient Haskell adopts the unique approach of allowing a user to define a quotient inductive type by simply extending a previously defined base type with equality constructors. A key benefit to this approach is that in conjunction with the implicit subtyping rules for quotient types, this permits functions that are defined on a quotient type to be reused on any type further down in the quotient hierarchy. For example, any function on sets as defined through the Boom hierarchy can also be applied to bags, lists and trees. In this section, we provide a general overview of how Quotient Haskell supports quotient inductive types.

In order to describe the key idea behind Quotient Haskell's support for quotient inductive types, we will consider the example of mobiles that were first presented in Section 3.1. We recall that mobiles are defined by extending the following type of binary trees

```
data Tree a = Leaf | Bin a (Tree a) (Tree a)
```

in the following manner

```
data Mobile a
  =  Tree a
  |/ swap :: x:a -> l:Mobile a -> r:Mobile a
        -> Bin x l r == Bin x r l
```

However, this definition of `Mobile` raises a key issue, namely that the `swap` equality is seemingly ill-typed as a consequence of applying `Bin` to mobiles rather than trees. Indeed, without any change to the typing rules for data constructors, this would not be a valid definition.

Unfortunately, in order to correctly define the type of mobiles it is not sufficient to simply quantify the `swap` equality constructor over trees rather than mobiles. In particular, if we were to adopt this approach there would be no way to inductively apply `swap` within the structure of a mobile. Moreover, without adapting the typing rules for data constructors, we would be unable to apply the `Bin` data constructor to build a mobile from two submobiles. These issues arise as a consequence of `Mobile` necessarily being a quotient inductive type, whereby its equality constructor should be mutually defined alongside its data constructors.

Quotient Haskell adds support for quotient inductive types by introducing new typing rules for data constructors that allow them to be applied to quotient types. In particular, when the underlying type of a quotient type A is of the form $T\ \tau_1\ ...\ \tau_n$ where T is a type constructor, new typing rules for the data constructors of T are introduced such that they may be applied to elements of A. More specifically, we:

1. Find each strictly positive occurrence of T in each of its respective data constructors;

2. Check whether each occurrence of T is applied to type arguments $\gamma_1\ ...\ \gamma_n$ such that for all i, τ_i unifies with γ_i with substitution σ_i;

3. If there is a unifying substitution, we replace the type application of T with the quotient type A and apply the unifying substitutions σ_1, ... σ_n to each type argument of A.

To understand this approach, we will consider how it applies to the example of mobiles. We begin by confirming that the underlying type of `Mobile a` is given by the application of a type constructor `Tree` to an argument `a`, and as such the preliminary condition is satisfied. Consequently, we proceed with the first step whereby we find each strictly positive occurrence of `Tree` in the data constructors `Leaf` and `Bin`. The `Leaf` constructor has only one such occurrence, namely its output type. Meanwhile, the `Bin` constructor has three such occurrences that include the types for both its left and right subtrees as well as its output type. In the second step, we filter out any occurrences of `Tree` that are not applied to a type argument that unifies with `a`, of which there are none. Finally, we apply the trivial unifying substitution to `a` and replace each suitable application of `Tree` with `Mobile a` to produce the following two additional typing rules:

$\vdash$ `Leaf` : `forall a. Mobile a`,

$\vdash$ `Bin` : `forall a. a -> Mobile a -> Mobile a -> Mobile a`.

In order to support multiple typing rules for a single data constructor, it is necessary that we are able to manage and resolve ambiguity in the type system. For example, we should be able to apply `Bin 0` to either a mobile or a tree, but given a term `t : Mobile a` we should only be able to apply `Bin 0 t` to a second mobile. Quotient Haskell achieves this by internally maintaining all possible types for a data constructor and simply adapting the application rule for data constructors to resolve their type when possible. Moreover, in cases of ambiguity such as typing an expression `\n -> Bin (n + 1)`, Quotient Haskell will use the default type of the data constructor, i.e. `\n -> Bin (n + 1)` will be inferred to have type `forall a. Num a => Tree a -> Tree a -> Tree a`.

3.5 Core language

In this section we present a core language λ_Q, which is a variant of the lambda calculus with patterns [Klop et al., 2008], and a Hindley-Milner type system extended with liquid and quotient types. We introduce λ_Q to give a precise account of how Quotient Haskell extends the type system of Liquid Haskell. In particular, λ_Q is formulated as a conservative extension to a generic underlying liquid type system. In this way, Quotient Haskell can be understood as an implementation of λ_Q for which the underlying refinement type system is Liquid Haskell, while λ_Q can be understood as an extension of λ_L, which was introduced to capture liquid types [Rondon et al., 2008].

The crucial features that λ_Q introduces to an existing liquid type system are quotients and their typing rules. Moreover, constructors and pattern matching by means of λ-case functions are included as the mechanism by which quotients can be used. Extending a Hindley-Milner type system with constructors and pattern matching is straightforward, so we do not discuss the details here. Furthermore, we only introduce the syntax and typing rules of λ_Q that are either novel, or important for understanding key ideas. The full collection of typing rules for λ_Q are presented in Section 3.8 alongside the subtyping and equality rules.

A type environment Γ for λ_Q is a sequence of type bindings $x : \tau$, guard predicates ϕ, and typed quotients $Q :: \tau$. The notation $Q :: \tau$ should be read as 'Q is a well-formed quotient on the underlying type τ'. Quotients that appear in a type environment alter the notion of equality between terms of the underlying type, and this context-sensitive notion of equality is described in Section 3.7. Notably, as with the core language λ_L, a complete account of the typing rules for λ_Q requires that we also allow *guard predicates* to inhabit an environment. Intuitively, guard predicates are propositions in the refinement logic that correspond to assumptions that hold true within the branches of conditionals. That is, guard predicates appear in a type environment to represent the known truth values of conditionals while checking the branches of an if-expression.

In addition to guard predicates, λ_Q inherits the notion of a qualifier set from λ_L. A qualifier set consists of all well-formed Boolean expressions that can be constructed from the environment variables. In an environment with a qualifier set $\mathbb{Q}$, a well-formed refinement predicate is a conjunction over any subset of $\mathbb{Q}$. We write $\Gamma \vdash_{\mathbb{Q}} x : \tau$ to mean that x has a liquid type τ in the environment Γ with qualifier set $\mathbb{Q}$, and $\Gamma \models \tau$ to mean that the type τ is well-formed in Γ. Moreover, for refinement predicates we write

$\Gamma \vdash_{\mathbb{Q}} \phi$ to mean that ϕ is well-formed in Γ with qualifier set $\mathbb{Q}$ and $\Gamma \models \phi$ to mean that the refinement predicate ϕ is well-formed and provable in Γ. Finally, we write $\Gamma \vdash_{\mathbb{Q}} Q :: \sigma$ to mean that the quotient Q is defined on an underlying type σ in context Γ with qualifier set $\mathbb{Q}$. The details of liquid typing can be found in the original work on λ_L [Rondon et al., 2008].

A quotient in λ_Q is represented by a sequence of quantified variables followed by a triple of a refinement predicate ϕ, a pattern p, and a term e. This quantified triple is written as

$$\textbf{forall } v_1 :: \tau_1 \textbf{ in } \cdots \textbf{forall } v_k :: \tau_k \textbf{ in } \phi \Rightarrow p == e$$

and we can think of this as an equality constructor in which v_i are variables, ϕ is the quotient precondition, p is the left-hand side of the target equality, and q is the right-hand side. This is characterised by the two following introduction rules:

$$\frac{\Gamma \models \sigma \qquad \Gamma, v : \tau \vdash_{\mathbb{Q}} Q :: \sigma}{\Gamma \vdash_{\mathbb{Q}} \textbf{forall } v : \tau \textbf{ in } Q :: \sigma}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} \phi \qquad \Gamma \vdash_{\mathbb{Q}} p : \sigma \qquad \Gamma \vdash_{\mathbb{Q}} e : \sigma}{\Gamma \vdash_{\mathbb{Q}} \phi \Rightarrow p == e :: \sigma}$$

Note that the first of these typing rules implicitly makes use of the weakening rule in the type system of λ_Q. In particular, the judgement $\Gamma, v : \tau \vdash_{\mathbb{Q}} Q :: \sigma$ assumes that $\Gamma, v : \tau \models \sigma$, which requires weakening of the premise $\Gamma \models \sigma$. The necessity of the weakening rule is a consequence of the underlying type not being allowed to depend on the terms over which the quotient varies. That is, λ_Q does not permit quotient inductive families. Furthermore, λ_Q adds the following three weakening rules:

$$\frac{\Gamma \vdash_{\mathbb{Q}} t : \tau \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \sigma}{\Gamma, t : \tau \vdash_{\mathbb{Q}} Q :: \sigma} \qquad \frac{\Gamma \vdash_{\mathbb{Q}} \phi \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \sigma}{\Gamma, \phi \vdash_{\mathbb{Q}} Q :: \sigma}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} Q_1 :: \sigma_1 \qquad \Gamma \vdash_{\mathbb{Q}} Q_2 :: \sigma_2}{\Gamma, Q_1 :: \sigma_1 \vdash_{\mathbb{Q}} Q_2 :: \sigma_2}$$

Intuitively, these rules simply assert that every well-typed quotient in a context Γ remains well-typed in any extension of Γ.

Refinements in a refinement type system may depend on terms and hence come equipped with a notion of substitution in types. In λ_Q, the substitution operation is extended over quotient types by first defining substitution in quotients. In particular, for every substitution σ, i.e. a finite map from variables to terms $v_1 \mapsto e_1; \cdots ; v_k \mapsto e_k$, we define substitution for quotients as follows:

$$\begin{aligned}
(\textbf{forall } v : \tau \textbf{ in } Q)[\sigma] &:= (\textbf{forall } w : \tau \textbf{ in } Q[v \mapsto w][\sigma]) \quad w \text{ not free in } \sigma \\
(\phi \Rightarrow p == e)[\sigma] &:= \phi[\sigma] \Rightarrow p[\sigma] == e[\sigma]
\end{aligned}$$

That is, we construct $Q[\sigma]$ in a standard way by avoiding the capture of bound variables and by applying σ to both the precondition and equality terms that constitute Q. This definition of substitution for quotients can in turn be used to define substitution in quotient types by simply applying a given substitution to both the underlying type and the quotient.

For every type τ and every quotient $Q :: \tau$, we can form a new type that represents the quotient of τ by Q and is denoted $\tau \ / \ Q$. This type formation or well-formedness rule is given as follows:

$$\frac{\Gamma \models \tau \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \models \tau \ / \ Q}$$

An important property of quotient types is that every term that inhabits an underlying type τ must also inhabit $\tau \ / \ Q$. This simple property is captured by the following introduction rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \quad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ Q}$$

Note that the above rule introduces an implicit type conversion from τ to $\tau \ / \ Q$, and hence this can lead to ambiguity in the meaning of expressions in the refinement logic. For example, the proposition that two terms $x, y : \tau$ are equal can be expressed as $x == y$, however this may not be logically equivalent to the same expression where x and y are instead considered to be elements of $\tau \ / \ Q$. In particular, it is possible that the quotient Q may equate previously distinct elements. As such, we will write $x \equiv_\sigma y$ for a given type σ to denote an equality proposition in the refinement logic between the two terms x, y when considered as elements of σ, or simply

$x \equiv y$ if the quotient is clear from the context. This extended notion of equality is discussed in more detail in Section 3.7.

We note that together with the introduction rules for quotients, the introduction rule for quotient types allows us to apply a *sequence* of quotients. That is, any valid quotient Q on an underlying type τ is also a valid quotient on the type $\tau \ / \ P$ for any quotient P. It is essential that any sequence of quotients of a type must be both idempotent and invariant under reordering, i.e. multiple quotients of an underlying type must be considered together as a *set*. We can give a formal characterisation of these two rules, which we term the 'idempotent' and 'permutation' rule, as follows:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P \ / \ P}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P_{\sigma(1)} \ / \ \cdots / \ P_{\sigma(n)} \qquad \sigma \text{ is a permutation } n \simeq n}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P_1 \ / \ \cdots / \ P_n}$$

Alternatively, we could have used sets directly in the syntactic construction rule for quotient types in λ_Q. This alternative approach would allow us to replace the two rules above with a single typing rule corresponding to the union of sets. However, this would in turn complicate the λ-case formation rule for quotient types that we introduce later in this section. Consequently, we continue with the above approach in our presentation of the typing rules of λ_Q. We note that while a naive implementation approach of this rule such as exhaustively checking every permutation of n quotients is $n!$, in practice the number of quotients used is typically very small.

Another essential property of quotient types is a generalisation of the λ-formation rule. In particular, we expect that any function defined on a quotient type that does not match on its input is always well-typed. Concretely, this generalised λ-formation rule can be expressed as follows:

$$\frac{\Gamma, x : \tau \ / \ Q \vdash_{\mathbb{Q}} e : \sigma \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} \lambda x.e : (v : \tau \ / \ Q) \to \sigma}$$

In order to give the typing rule for functions that match on their inputs, we will first require a formal notion as to what it means for a particular 'case' of a matching function to respect a quotient. To do this, we will define a context-sensitive binary relation $\Gamma \models \bullet \rightsquigarrow \bullet$ whose first element is a finite sequence of pairs of patterns and terms and whose second element is a quotient. We write elements of this relation in the form

$$\lambda\ \{p_1 \to e_1;\ \cdots\ ;\ p_k \to e_k\} \rightsquigarrow Q,$$

or simply $\lambda\ \{p \to e\} \rightsquigarrow Q$. This relation will characterise precisely when a particular λ-case term respects a given quotient. In order to define this relation, we first introduce a number of auxiliary definitions.

First of all, we make use of unification or *matching* of patterns. This is a well-known concept for lambda calculus with patterns, and we do not reintroduce this idea here. We write $x \sim_\sigma y$ to denote that the terms x and y can be unified, with the *most general unifier* given by the substitution σ. The property that σ is the most general unifier is crucial to the correctness of the core language and is necessary for the proof of Proposition 3, which requires uniqueness of the most general unifier. With this in mind, we continue with an inductive definition of the context-sensitive relation $\Gamma \models \lambda\ \{\ p \to e\ \} \rightsquigarrow Q$. We begin with the more involved base case, given by the following rule:

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} \rho : \tau \qquad \Gamma \vdash_{\mathbb{Q}} p_1,\ \ldots,\ p_k : \tau \qquad \Gamma \vdash_{\mathbb{Q}} e_1,\ \ldots,\ e_k : \upsilon \\ \Gamma \vdash_{\mathbb{Q}} \rho \sim_\sigma p_k \qquad \forall\ i\ j\ \sigma'.\ p_i \sim_{\sigma'}\ p_j \Rightarrow i = j \\ \Gamma,\ \phi[\sigma] \models e_k[\sigma] \equiv_\upsilon \lambda\ \{p_1 \to e_1;\ \ldots;\ p_k \to e_k\}\ t[\sigma]\end{array}}{\Gamma \models \lambda\ \{\ p \to e\ \} \rightsquigarrow (\phi \Rightarrow \rho == t)}$$

Importantly, this rule makes use of the extended notion of equality in λ_Q described earlier in this section. The condition $\forall i\ j\ \sigma'.\ p_i \sim_{\sigma'}\ p_j \Rightarrow i = j$ asserts that each pattern must be distinct up to unification. By imposing this condition we ensure there is no overlap between different branches of a λ-case, and as such there can be at most one p_i that unifies with the left-hand side of the quotient's equality target. Intuitively, the above rule can be understood as first identifying a pattern p_i from a sequence p that unifies with the left-hand side of the equality target of a quotient. After applying the unifying substitution, this rule checks in the refinement logic whether the precondition of the quotient implies the corresponding expression e_i from a sequence e is equal to the right-hand side of the equality target applied to the lambda-case function. If the outlined condition is met, then the relation holds between the given sequences p, e and the considered quotient.

We next consider the inductive case, which simply extends the environment with a quantified variable and is characterised by the following rule:

$$\frac{\Gamma, v : \tau \models \lambda \; \{ \; p \rightarrow e \; \} \rightsquigarrow Q}{\Gamma \models \lambda \; \{ \; p \rightarrow e \; \} \rightsquigarrow (\textbf{forall } v : \tau \textbf{ in } Q)}$$

The above definitions allow us to proceed with a formal characterisation of the λ-case formation rule for quotient types. In particular, this rule is given as follows:

$$\frac{\begin{array}{c}\Gamma \models (x : \tau \; / \; Q) \rightarrow \sigma \qquad \Gamma \models \lambda \; \{ \; p \rightarrow e \; \} \rightsquigarrow Q \\ p_1, \; \ldots, \; p_k \text{ is a complete case analysis of } \tau\end{array}}{\Gamma \vdash_{\mathbb{Q}} \lambda \; \{ \; p_1 \rightarrow e_1; \ldots; p_k \rightarrow e_k \; \} : (x : \tau \; / \; Q) \rightarrow \sigma}$$

Note that we impose that any sequence of patterns used to construct a λ-case term must form a complete case analysis of the underlying type. The formal description of this idea is well understood for a type theory with algebraic data types, and we do not reintroduce it here. In the absence of this condition, the partiality of λ-case functions can be used to contradict the necessary correctness result for quotients. The λ-case formation rule is the crucial component of the typing rules for λ_Q and ensures that the equality constructors of quotient types are respected by functions that match on their input. Importantly, when only the total functions of λ_Q are considered it must always be the case that there exists a pattern p_i that unifies with the left-hand target of a quotient Q.

To conclude our introduction of the typing system of λ_Q, we formulate and prove a correctness result for quotients. Notably, soundness of quotient type checking follows directly from the soundness of the underlying refinement type system and the assumption that a liquid type judgement $\Gamma \vdash_{\mathbb{Q}} e : \tau$ implies $\Gamma \vdash e : \tau$. In particular, the proof of soundness for quotient type checking follows in the same manner as described for λ_L by Rondon et al. [2008]. The correctness result for quotient type checking states that from the typing rules for λ_Q, we can conclude that function congruence correctly extends over quotients. Indeed, this is the essential and defining property of quotient types. In order to state this correctness result, we begin by defining precisely what it means for an arbitrary function on a quotient type to respect the relevant quotient.

Definition. Given an environment Γ and a function $\Gamma \vdash_{\mathbb{Q}} f : (x : \tau) \rightarrow \sigma$, we write $\Gamma \models f * Q$ to denote that f respects the quotient Q in the environment Γ, which is inductively defined by:

$$\frac{\Gamma, v : \tau \models f * Q}{\Gamma \models f * (\textbf{forall } v : \tau \textbf{ in } Q)}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} p : \tau \qquad \Gamma \vdash_{\mathbb{Q}} e : \tau \qquad \Gamma, \phi \models f\ p \equiv_{\sigma} f\ e}{\Gamma \models f * (\phi \Rightarrow p == e)}$$

The above definition directly corresponds to the functorial action of a function on an equality that is constructed by means of a quotient. An important property of the proposition $\Gamma \models f * Q$ is invariance with respect to context extension by a binding, which is captured as follows.

Proposition 1. Given $\Gamma \models f * Q$ and $\Gamma \vdash_{\mathbb{Q}} e : \tau$ then we can conclude $\Gamma, e : \tau \models f * Q$. This can be understood as a weakening rule for the property that every function must respect the quotients of its inputs. The proof follows by induction on the quotient Q:

- For $Q = \textbf{forall } v : \gamma \textbf{ in } P$, by the inductive hypothesis we can weaken $\Gamma, v : \gamma \models f * P$ to obtain $\Gamma, v : \gamma, e : \tau \models f * P$ as required;

- For $Q = \phi \Rightarrow p == h$, we apply the standard weakening rule for typing judgements to both $\Gamma \vdash_{\mathbb{Q}} p : \tau$ and $\Gamma \vdash_{\mathbb{Q}} h : \tau$ to extend their context with $e : \tau$, and similarly apply the weakening rule for equality judgements in λ_Q to $\Gamma, \phi \models f\ p \equiv_{\sigma} f\ h$.

An important property of the typing system of λ_Q and a key component in the proof of our correctness theorem is preservation of equality under quotient rewriting. That is, when a quotient $\phi \Rightarrow p == e$ appears in a context Γ then given any expression in Γ the two expressions that can be constructed by substituting a free variable for either p or e should be considered equal by the type system of λ_Q. We formally express this property in the following proposition.

Proposition 2. Given a typing judgement $\Gamma, v : \tau \ / \ (\phi \Rightarrow p == h) \vdash_{\mathbb{Q}} e : \sigma$, we can conclude $\Gamma; (\phi \Rightarrow p == h) :: \tau \models e[v \mapsto p] \equiv_\sigma e[v \mapsto h]$. The proof follows by induction on the expression e, for which the interesting cases proceeds as follows:

- For $e = x$, if $x = v$ then x must have type $\tau \ / \ (\phi \Rightarrow p == h)$ and we can conclude $p \equiv h$ from the equality checking rules of λ_Q;

- For $e = \lambda \ \{p_1 \to e_1; \ \ldots; \ p_k \to e_k\}$, for each p_i where v appears freely, applying either $[v \mapsto p]$ or $[v \mapsto h]$ to e does not change e_i and the equality trivially holds, otherwise we continue our induction on e_i in the appropriately extended context to obtain $e_i[v \mapsto p] \equiv_\sigma e_i[v \mapsto h]$ and finally we aggregate the resulting equalities in the obvious manner to obtain an equality between lambda-case functions.

In advance of stating our correctness result for quotients in λ_Q, we first provide a proof of a contextual version. In particular, in Proposition 3 we assume that we are working within a context Γ whereby all functions in Γ that are defined on quotient types respect the relevant quotient. From this assumption, we can prove that any other well-typed function on quotient types that can be constructed from this context will also respect its quotients.

Proposition 3. Given an environment Γ such that all functions in Γ respect the relevant quotient, i.e. every $g : (x : \alpha \ / \ P) \to \beta \in \Gamma$ satisfies $\Gamma \models g * P$, it follows that every constructible function $\Gamma \vdash_{\mathbb{Q}} f : (x : \tau \ / \ Q) \to \tau'$ has the property $\Gamma \models f * Q$. The proof follows by induction on Q and f:

- For $Q = \textbf{forall} \ v : \gamma \ \textbf{in} \ R$, we first apply weakening (Proposition 1) to our initial assumption to conclude that every $g : (x : \alpha \ / \ P) \to \beta \in \Gamma$ has the property $\Gamma, v : \gamma \models g * P$, then we continue our induction with this new assumption alongside the extended environment $\Gamma, v : \gamma$ and the weakened term $\Gamma, v : \gamma \vdash_{\mathbb{Q}} f : (x : \tau \ / \ R) \to \tau'$;

- For $Q = \phi \Rightarrow p == h$, we consider each case for f:

- For $f = x$ and $f = c$, the result follows from the initial assumption;

- For $f = \lambda x.e$, after unfolding definitions it suffices to show that $\Gamma, \phi \models e[x \mapsto p] \equiv_{\tau'} e[x \mapsto h]$ which follows directly from Proposition 2;

- For $f = \lambda\ \{p_1 \to e_1;\ \ldots;\ p_k \to e_k\}$, by the condition that a well-typed λ-case must perform a complete case analysis, there must exist precisely one p_i such that $p_i \sim_\sigma p$ and it consequently suffices to show that $\Gamma, \phi \models e_i[\sigma] == \lambda\ \{p_1 \to e_1;\ \ldots;\ p_k \to e_k\}\ h[\sigma]$ and since σ is the most general unifier, this follows from the typing rule for λ-case functions.

Finally, we can now conclude our introduction of the core language λ_Q with the key correctness result for quotient types, which is captured by the following theorem.

Theorem 1 (*Correctness of quotienting*). Every closed function in λ_Q of the form $f : (x : \tau\ /\ Q) \to \sigma$ respects the relevant quotient Q, i.e. $\emptyset \models f{*}Q$. That is, function congruence correctly extends over quotients. The proof is given simply by specialising Γ to the empty context in Proposition 3.

3.6 Subtyping

Quotient types in the core language λ_Q come equipped with an ordering relation that we use to extend our type system with subtyping rules. In practice, these subtyping rules provide us with a framework for deciding when it is sound to substitute one quotiented type for another. As such, the subtyping rules for quotient types improve the reusability of code by allowing functions on quotient types to be applied to more 'weakly' quotiented types subject to the ordering relation. For example, a valid function on the propositional truncation of a type, i.e. the quotient which relates all terms, must also respect any other quotient on the same underlying type. In this section, we introduce the subtyping rules for quotient types in λ_Q.

The first of the λ_Q subtyping rules is for quotient generalisation, and asserts that any quotient type is a supertype of its underlying type. For example, the type of lists can be understood as a subtype of bags, or bags as a subtype of sets. This rule is formally expressed as follows:

$$\frac{\Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} \tau <: \tau \ / \ Q}$$

We write $\tau <: \sigma$ above to denote that τ is a subtype of σ. When considered alongside the λ_Q typing rule for subtypes, this rule is equivalent to the introduction form for quotient types.

To introduce the next subtyping rule for quotient types, we first present an ordering relation on quotients such that $P <: Q$ means that P is a *subquotient* of Q, defined using four rules. The first rule specifies when the precondition and equality of a quotient are subsumed by another. For this rule, we make use of a join-semilattice on patterns, whereby $p \subseteq_\sigma q$ iff the pattern p is subsumed by q with substitution σ. Consequently, when $p \subseteq_\sigma q$ we have a definitional equality $q[\sigma] = p$. For example, given a constructor $S : \tau \to \tau$ and variables $v : \tau$, the term $S\ (S\ v)$ is subsumed by $S\ v$ with substitution $[v \mapsto S\ v]$. We write $\Gamma \models p \subseteq_\sigma q$ to denote that p is subsumed by q in context Γ, and introduce the following subquotient rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} p, q, e, f : \tau \qquad \Gamma \models p \subseteq_\sigma q \qquad \Gamma, \phi \models \psi[\sigma] \qquad \Gamma, \phi \models e \equiv_\tau f[\sigma]}{\Gamma \models (\phi \Rightarrow p == e) <: (\psi \Rightarrow q == f)}$$

In this definition, $\equiv_\tau$ is an extended notion of equality between terms of type τ that can make use of substitutions presented by quotients, which we will define in Section 3.7. Intuitively, the above rule states that if one quotient is implied by another in the refinement logic, then the former is a subquotient of the latter. For example, given a constructor $S : \tau \to \tau$ and variables $u, v : \tau$, we can conclude $(\textbf{true} \Rightarrow S\ (S\ (S\ v)) == S\ v) <: (\textbf{true} \Rightarrow S\ (S\ u) == u)$ using the substitution $[u \mapsto S\ v]$.

When two quotients quantify over a variable of the same type, the ordering relation is defined by the following rule, which simply extends the context by the quantified variables and checks whether the remaining body of one quotient is a subquotient of the other:

$$\frac{\Gamma, v : \tau, u : \tau \vdash_{\mathbb{Q}} P <: Q}{\Gamma \vdash_{\mathbb{Q}} (\textbf{forall}\ v : \tau\ \textbf{in}\ P) <: (\textbf{forall}\ u : \tau\ \textbf{in}\ Q)}$$

In contrast, when two quotients are quantified over types that are not definitionally equal, it is possible that one is a subquotient of the other, by means of the following permutation rule:

$$\frac{\Gamma,\ v_1 : \tau_1,\ \ldots,\ v_n : \tau_n \vdash_{\mathbb{Q}} P <: Q \qquad \Gamma \models \forall\ i.\ \tau_i \neq \tau_{\sigma(i)} \qquad \sigma \text{ is a permutation on } n}{\Gamma \vdash_{\mathbb{Q}} (\textbf{forall}\ v_1 : \tau_1\ \textbf{in}\ \ldots\ \textbf{forall}\ v_n : \tau_n\ \textbf{in}\ P) <: (\textbf{forall}\ v_{\sigma(1)} : \tau_{\sigma(1)}\ \textbf{in}\ \ldots\ \textbf{forall}\ v_{\sigma(n)} : \tau_{\sigma(n)}\ \textbf{in}\ Q)}$$

In this rule, $\Gamma \models \forall\ i.\ \tau_i \neq \tau_{\sigma(i)}$ means that when τ is reordered under the permutation σ, the type in position i is not definitionally equal to the old type in the same position. This condition ensures that the rule does not overlap with the prior rule for quotients that quantify over the same type. That is, by imposing this constraint, we can translate these rules to well-founded induction on two quotients. Understood in this manner, an implementation of the above rule involves constructing two sequences of types $\tau_1, \ldots, \tau_n$ and $\rho_1, \ldots, \rho_n$, whereby we continue until there are no more quantifiers or we find $\tau_{n+1} = \rho_{n+1}$, and finally we check if ρ is a permutation of τ.

The final case we consider is when a subquotient quantifies over a variable and the superceding quotient does not. In particular, we introduce the following ordering rule for quotients:

$$\frac{\Gamma \models \tau \qquad \Gamma, v : \tau \vdash_{\mathbb{Q}} P <: (\phi \Rightarrow p == e) \qquad v \text{ not free in } \phi,\ p \text{ or } e}{\Gamma \vdash_{\mathbb{Q}} (\textbf{forall}\ v : \tau\ \textbf{in}\ P) <: (\phi \Rightarrow p == e)}$$

In particular, this rule holds because every quotient in λ_Q specifies a *proposition*. That is, we should consider **forall** $v : \tau$ **in** P to be analogous to $\forall\ (v : \tau).\ P$, rather than the dependent function type $\Pi\ (v : \tau)\ P$. If we were instead to consider a calculus for higher quotients, this subquotient rule would not be correct. It is important that the quantified variable is not free in each term of the superceding quotient, otherwise context extension may change its meaning.

From the definition of our ordering relation on quotients, we can extend our subtyping rules for quotient types in the language λ_Q with the following rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} P <: Q}{\Gamma \vdash_{\mathbb{Q}} \tau \ / \ P <: \tau \ / \ Q}$$

That is, a subquotienting relationship can be lifted directly to a subtyping relationship. Combining this rule with the subtyping rule for functions yields the following important typing derivation:

$$\frac{\Gamma \vdash_{\mathbb{Q}} f : (\tau \ / \ Q) \to \tau' \qquad \Gamma \vdash_{\mathbb{Q}} P <: Q}{\Gamma \vdash_{\mathbb{Q}} f : (\tau \ / \ P) \to \tau'}$$

That is, if we know that P is a subquotient of Q and that a function f respects Q, then it must also respect P. We can similarly derive a typing rule that allows us to apply any function on a quotient type to a term of the underlying type. For practical purposes, such as in Quotient Haskell, this means that only the most restrictive quotient is needed in the type definition of a function, which can subsequently be applied to terms inhabiting any subquotients.

Finally, the subquotient relationship can be shown to satisfy a correctness theorem that states that if a function respects a quotient Q it must also respect any subquotient of Q. In order to prove this correctness rule for the subquotienting relationship, we make use of Proposition 4, which is presented in Section 3.7. In particular, an equality $x \equiv y$ is invariant with respect to substitution. We can then proceed with our correctness theorem for the subtyping rules of λ_Q.

Theorem 2 (*Correctness of subquotienting*). Given a function $\Gamma \vdash_{\mathbb{Q}} f : (x : \tau) \to \tau'$ and quotients $\Gamma \models P, Q :: \tau$ such that $\Gamma \vdash_{\mathbb{Q}} P <: Q$, then if $\Gamma \models f * Q$ it follows that $\Gamma \models f * P$. The proof is by induction on the subquotienting relation, for which the interesting case proceeds as follows:

- For $\phi \Rightarrow p == e <: \psi \Rightarrow q == r$, unfolding the definitions for the judgement $\Gamma \models f * Q$ and the subquotient $P <: Q$ yields the following four assumptions:

$$(1)\ \ \Gamma, \psi \models f\ q \equiv_{\tau'} f\ r \qquad\qquad (2)\ \ p \subseteq_{\sigma} q$$

$$(3)\ \ \Gamma, \phi \models e \equiv_{\tau} r \qquad\qquad (4)\ \ \Gamma, \phi \models \psi[\sigma]$$

As stated in Proposition 4, a key property of equality is invariance with respect to term substitution, and consequently by function congruence and assumption (1) we can conclude $\Gamma, \psi[\sigma] \models f\ q[\sigma] \equiv_{\tau'} f\ r[\sigma]$. From assumption (2) it follows that $p := q[\sigma]$, and by function congruence and transitivity of equality we can conclude $\Gamma, \psi[\sigma] \models f\ p \equiv_{\tau'} f\ r[\sigma]$. We proceed by applying function congruence and symmetry of equality to (3) to show that $\Gamma, \psi[\sigma] \models f\ r[\sigma] \equiv_{\tau} f\ e$. Finally, by transitivity of equality and by generalising the guard predicate through (4), we can conclude $\Gamma, \phi \models f\ p \equiv_{\tau'} f\ e$, as required.

3.7 Equality

Quotients that appear in a typing context introduce a substitution rule that can be used for deciding the equality of terms during type-checking. This rule changes the notion of equality between terms, and we write $x ==_{\tau} y$ to represent an equality between terms of type τ in the underlying refinement type system, and $x \equiv_{\tau} y$ to represent the extended notion of equality in λ_Q that can make use of substitutions presented by quotients. The precise definition of equality in the underlying refinement type system can vary, but it remains crucial that it is an equivalence relation. Furthermore, we assume several additional rules hold in the underlying refinement type system:

- *(Function congruence)* For every closed function $\emptyset \models f : (x : \tau) \to \tau'$, if we have an equality $\Gamma \models x ==_{\tau} y$ then we must also have an equality $\Gamma \models f\ x ==_{\tau'} f\ y$;

- *(Equality substitution)* For every term $\Gamma \vdash_{\mathbb{Q}} e : \tau$, every equality $\Gamma \models x ==_{\tau'} y$ and every variable v, it must hold that $\Gamma \models e[v \mapsto x] ==_{\tau} e[v \mapsto y]$;

- *(Substitution invariance)* For every equality $\Gamma \models x ==_{\tau} y$ and substitution σ, the equality must also hold under the substitution, i.e. $\Gamma \models x[\sigma] ==_{\tau} y[\sigma]$.

Crucially, while $x ==_{\tau} y$ is a proposition in the refinement logic, $x \equiv_{\tau} y$ is a judgement in the type system. Consequently, in λ_Q the definition of

$\Gamma \models \phi$ differs from the underlying liquid type system. In particular, any equality $x ==_\tau y$ that appears in the logical proposition ϕ is lifted to the extended notion of equality $x \equiv_\tau$ y. For example, we check the judgement $\Gamma \models x == y \Rightarrow \psi$ by checking whether either $\Gamma \not\models x \equiv y$ or $\Gamma \models \psi$. Intuitively, our extended notion of equality can make use of quotients that appear in a context in order to transform the original equality by means of substitutions. This decidable process results in building a new equality in the refinement logic which can then be checked by an SMT-solver.

Importantly, our extended notion of equality is strictly weaker than that of the underlying liquid type system, and this property is expressed by the following rule:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \qquad \Gamma \models x ==_\tau y \qquad \tau \text{ is not a quotient type}}{\Gamma \models x \equiv_\tau y}$$

That is, two terms are considered equal in the language λ_Q if they are equal in the underlying refinement type system. Consequently, for any term that inhabits a type that has not been quotiented, the notion of equality in λ_Q and the underlying refinement logic is identical. However, for terms that inhabit a quotient type, it is necessary to extend our notion of equality to make use of the equalities introduced by quotients. To do this, we begin by giving a formal account of when a context is extended by a quotient in the process of equality checking.

In contrast to types and guard predicates, we make use of a more involved form of context extension for quotients that ensures a given context contains only the most general form of a quotient. As discussed later in this section, this is necessary to ensure termination of equality checking in our type system. As such, given a context Γ and quotient $Q :: \tau$, we write Γ, Q for regular context extension and $\Gamma; Q$ for the more involved notion. In order to define $\Gamma; Q$, we will consider three separate cases concerning whether Γ already contains a quotient $P :: \tau$ that is related to Q by a subquotient relationship. A complete account of the ordering rules for quotients is given in Section 3.6, and we write $Q <: P$ to mean that Q is a subquotient of P. We begin by considering the case when there does not exist a quotient $P :: \tau$ in Γ such that Q is related to P by means of the subquotient relationship. In this case, we simply apply the usual notion of context extension and as such we have $\Gamma; Q = \Gamma, Q$. The two remaining cases consider when there exists a quotient $P :: \tau$ in Γ such that either $Q <: P$ or $P <: Q$. In the case when $Q <: P$, then Γ is left unchanged and we define $\Gamma; Q = \Gamma$. In turn, for $P <: Q$ we replace P in Γ with Q. Using this notion of extending a context by a quotient, we can state the following equality rule for quotient types:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \ / \ (\phi \Rightarrow p == e) \qquad \Gamma; (\phi \Rightarrow p == e) :: \tau \models x \equiv_\tau y}{\Gamma, \phi \models x \equiv_{\tau/(\phi \Rightarrow p == e)} y}$$

This rule states that two terms inhabiting a quotiented type are to be considered equal when they are equal as inhabitants of their underlying type in a context extended by their quotient. Note that the rule formulated above only considers quotients that do not contain bindings, and indeed, this is the only form of a quotient that may be used to extend a context. For the case of quotients that contain bindings, the following equality rule applies:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \ / \ (\textbf{forall } v : \sigma \textbf{ in } Q) \qquad \Gamma, v : \sigma \vdash_{\mathbb{Q}} x \equiv_{\tau/Q} y}{\Gamma \vdash_{\mathbb{Q}} x \equiv_{\tau/(\textbf{forall } v:\sigma \textbf{ in } Q)} y}$$

That is, for quotients with bindings we simply consider equality in the context extended by the bindings. The final equality rule we introduce to the typing system of λ_Q corresponds to checking the equality of terms in a context containing a compatible quotient. This rule allows us to rewrite an equality using a quotient in a given context, and is formulated as follows:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y, p, e : \tau \qquad \Gamma \models \phi[\sigma] \qquad \Gamma \models y \equiv_\tau e[\sigma] \qquad x \subseteq_\sigma p}{\Gamma; (\phi \Rightarrow p == e) :: \tau \models x \equiv_\tau y}$$

We write $x \subseteq_\sigma p$ here to denote that a term x is subsumed by the pattern p with substitution σ. With this in mind, the above rule states that any two terms $x, y : \tau$ are considered equal in a context Γ if there is a compatible quotient $\phi \Rightarrow p == e :: \tau$ in Γ such that p subsumes x with substitution σ, and such that y can be show to be equal in the underlying refinement type system to the term obtained by applying σ to e. Furthermore, we have an additional symmetric rule whereby we check if p instead subsumes y and $x \equiv_\tau e[\sigma]$, which is otherwise identical to the above rule.

In order to establish that our extended notion of equality is an equivalence on terms, we require an additional axiomatic rule for transitivity of equality as defined on quotient types:

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y, z : \tau \ / \ Q \qquad \Gamma \models x \equiv_{\tau/Q} y \qquad \Gamma \models y \equiv_{\tau/Q} z}{\Gamma \models x \equiv_{\tau/Q} z}$$

Crucially, if the equality of the underlying refinement type system is an equivalence relation then so is our extended notion of equality in λ_Q. Reflexivity of equality follows evidently from the rule that every equality in the underlying refinement type system infers equality under our extended notion of equality. Transitivity is given by the axiomatic rule postulated above, while symmetry follows from the fact that the third equality rule can be symmetrised. Later in this chapter, we make use of the fact that our extended notion of equality is indeed an equivalence relation in correctness proofs for both quotients and a subquotienting relation.

A practical implementation for type-checking that applies the equality rules for λ_Q requires that we handle cases for which there is more than one compatible quotient. Importantly, the manner in which we extend a context by a quotient ensures that a context will only ever contain a single instance of a quotient. In addition, when a quotient is applied during equality checking it is removed from the context. When considered together, these properties ensure that every equality checking path will terminate for terms that inhabit a quotient type. As such, given that a context Γ can only contain a finite number of quotients of the form $Q :: \tau$, then if Γ has n such quotients there can be at most $n!$ different paths to consider for equality checking. This pathological case occurs when every permutation of quotients $Q_1 :: \tau, \ldots, Q_n :: \tau \in \Gamma$ corresponds to a sequence of rewrites. For example, if the left-hand side of the equality produced by each Q_k were simply a variable, then each quotient would evidently unify with any term of type τ, and consequently every permutation would indeed specify a valid sequence of rewrites. In practice, however, the number of quotients that appear in a context for a type is typically small, and as such exhaustively checking each valid permutation does not present an issue even in the pathological case.

As an example of how the extended equality rule of λ_Q is applied in practice, recall that in Section 3.2 we defined a type `List` by quotienting an underlying type `Tree`. To check that the `map` function on trees refines to a function on lists, we must show it respects the quotients of `List`. One such quotient is the left-identity law `idl :: x:List a -> Join Empty x == x`, and to show that `map` respects `idl` we must show

```
map f (Join Empty x) == map f x
```

After normalising the term on the left, the stated equality simplifies to

```
Join Empty (map f x) == map f x
```

Importantly, when considering the refinement of `map` to a function on lists, this is an equality between terms of the quotient type `List`. Consequently, when checking whether this equality holds we first extend our context by the quotients of `List`, and then check if the pattern appearing on the left of the equality target of any of these quotients subsumes either `Join Empty (map f x)` or `map f x`. We can observe that the left identity law has precisely this form, as `Join Empty x` subsumes `map f x` with substitution `x` $\mapsto$ `map f x`. Finally, we check whether `map f x == x`[`x` $\mapsto$ `map f x`], or simply `map f x == map f x`, which holds definitionally. This same sequence of steps can similarly be used to show that `map` respects each of the quotients of `List` and as such `map` is refinable to a function on lists.

We conclude our discussion on the extended notion of equality used in the type system of λ_Q by discussing and giving a formal account of three essential properties of equality substitution, substitution invariance and function congruence. The first of these properties we consider is substitution invariance which is critical for the correctness of the subquotienting relation defined in Section 3.6. Concretely, we state this property as follows:

Proposition 4. Given a context Γ and a substitution σ, then for every equality $\Gamma \models x \equiv_\tau y$ we have $\Gamma \models x[\sigma] \equiv_\tau y[\sigma]$. In the case when τ is not a quotient type, the proof follows from the substitution invariance property of the underlying equality. Otherwise, the proof follows by inversion whereby we consider each of the extended equality rules.

In addition to being invariant under substitution, equality in λ_Q must also be respected by the substitution operator. Intuitively, this means that if two terms are equal then substituting them for a variable in any expression should produce two equal terms, and is stated as follows:

Proposition 5. For every equality $\Gamma \models x \equiv_\tau y$, term $\Gamma \vdash_{\mathbb{Q}} e : \sigma$ and any choice of variable u, we can construct an equality $\Gamma \models e[u \mapsto x] \equiv e[u \mapsto y]$. The proof follows by induction on e and is evident for each case after suitably unfolding the definition of substitution.

The final property of equality we consider is function congruence, which we describe in two parts. Firstly, in Proposition 6 we consider open terms whereby we assume that we are working in a context which respects the congruence law for functions. We then specialise to the empty context in Proposition 7 to state the true congruence property for closed functions.

Proposition 6. Given a context Γ with function congruence, i.e. for every function $f : (x : \tau) \to \tau' \in \Gamma$ and every equality $\Gamma \models x \equiv_\tau y$ we can construct an equality $\Gamma \models f\ x \equiv_{\tau'} f\ y$, then every constructible function in Γ must also obey the function congruence rule. In order to prove this we must show that for every function $\Gamma \vdash_{\mathbb{Q}} f : (x : \tau) \to \tau'$ and every equality $\Gamma \models x \equiv_\tau y$, we can conclude $\Gamma \models f\ x \equiv_{\tau'} f\ y$. The proof follows by induction on f and we consider only the interesting cases whereby f is either a λ function or a λ-case function. In both cases, after unfolding definitions we can observe that the proof follows evidently from Proposition 5.

Proposition 7. For every closed function $\emptyset \vdash_{\mathbb{Q}} f : (x : \tau) \to \tau'$ and every equality $\emptyset \vdash_{\mathbb{Q}} x \equiv_\tau y$, we can show that there is an equality $\emptyset \vdash_{\mathbb{Q}} f\ x \equiv_{\tau'} f\ y$. The proof follows immediately by simply instantiating the context in Proposition 6 to the empty context.

3.8 Typing rules

In this section, we present the full collection of novel typing, subtyping and equality rules for our core language λ_Q. In particular, we present the typing rules in Figure 3.1, the subtyping rules in Figure 3.2, and the equality rules in Figure 3.3.

$\boxed{\Gamma \vdash_{\mathbb{Q}} Q :: \sigma}$ *(Quotient typing and weakening)*

$$\frac{\Gamma \models \sigma \qquad \Gamma, v : \tau \vdash_{\mathbb{Q}} Q :: \sigma}{\Gamma \vdash_{\mathbb{Q}} \textbf{forall } v : \tau \textbf{ in } Q :: \sigma} \qquad \frac{\Gamma \vdash_{\mathbb{Q}} \phi \qquad \Gamma \vdash_{\mathbb{Q}} p : \sigma \qquad \Gamma \vdash_{\mathbb{Q}} e : \sigma}{\Gamma \vdash_{\mathbb{Q}} \phi \Rightarrow p == e :: \sigma}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} t : \tau \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \sigma}{\Gamma, t : \tau \vdash_{\mathbb{Q}} Q :: \sigma} \qquad \frac{\Gamma \vdash_{\mathbb{Q}} \phi \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \sigma}{\Gamma, \phi \vdash_{\mathbb{Q}} Q :: \sigma}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} Q_1 :: \sigma_1 \qquad \Gamma \vdash_{\mathbb{Q}} Q_2 :: \sigma_2}{\Gamma, Q_1 :: \sigma_1 \vdash_{\mathbb{Q}} Q_2 :: \sigma_2}$$

$\boxed{\Gamma \models \tau \ / \ Q}$ *(Well-formedness)*

$$\frac{\Gamma \models \tau \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \models \tau \ / \ Q}$$

$\boxed{\Gamma \vdash_{\mathbb{Q}} e : \tau \ / \ P}$ *(Quotient set rules)*

$$\frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P \ / \ P}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P_{\sigma(1)} \ / \ \cdots / \ P_{\sigma(n)} \qquad \sigma \text{ is a permutation } n \simeq n}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ P_1 \ / \ \cdots / \ P_n}$$

$\boxed{\Gamma \models \lambda \ \{ \ p \to e \ \} \rightsquigarrow Q}$ *(Respectfulness relation)*

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} \rho : \tau \qquad \Gamma \vdash_{\mathbb{Q}} p_1, \ \ldots, \ p_k : \tau \\ \Gamma \vdash_{\mathbb{Q}} e_1, \ \ldots, \ e_k : \upsilon \qquad \Gamma \vdash_{\mathbb{Q}} \rho \sim_\sigma p_k \qquad \forall \ i \ j \ \sigma'. \ p_i \sim_{\sigma'} \ p_j \Rightarrow i = j \\ \Gamma, \ \phi[\sigma] \models e_k[\sigma] \equiv_\upsilon \lambda \ \{p_1 \to e_1; \ \ldots; \ p_k \to e_k\} \ t[\sigma]\end{array}}{\Gamma \models \lambda \ \{ \ p \to e \ \} \rightsquigarrow (\phi \Rightarrow \rho == t)}$$

$$\frac{\Gamma, v : \tau \models \lambda \ \{ \ p \to e \ \} \rightsquigarrow Q}{\Gamma \models \lambda \ \{ \ p \to e \ \} \rightsquigarrow (\textbf{forall } v : \tau \textbf{ in } Q)}$$

$\boxed{\Gamma \vdash_{\mathbb{Q}} e : \tau}$ *(Typing rules)*

$$\frac{\Gamma \vdash_{\mathbb{Q}} x : \tau \quad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} x : \tau \ / \ Q} \qquad \frac{\Gamma, x : \tau \ / \ Q \vdash_{\mathbb{Q}} e : \sigma \qquad \Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} \lambda x.e : (\upsilon : \tau \ / \ Q) \to \sigma}$$

$$\frac{\begin{array}{c}\Gamma \models (x : \tau \ / \ Q) \to \sigma \\ \Gamma \models \lambda \ \{ \ p \to e \ \} \rightsquigarrow Q \qquad p_1, \ \ldots, \ p_k \text{ is a complete case analysis of } \tau\end{array}}{\Gamma \vdash_{\mathbb{Q}} \lambda \ \{ \ p_1 \to e_1; \ \ldots; p_k \to e_k \ \} : (x : \tau \ / \ Q) \to \sigma}$$

Figure 3.1: Typing rules for λ_Q

$\boxed{\Gamma \vdash_{\mathbb{Q}} P <: Q}$ *(Subquotienting rules)*

$$\frac{\Gamma \vdash_{\mathbb{Q}} Q :: \tau}{\Gamma \vdash_{\mathbb{Q}} \tau <: \tau \ / \ Q}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} p, q, e, f : \tau \qquad \Gamma \models p \subseteq_{\sigma} q \qquad \Gamma, \phi \models \psi[\sigma] \qquad \Gamma, \phi \models e \equiv_{\tau} f[\sigma]}{\Gamma \models (\phi \Rightarrow p == e) <: (\psi \Rightarrow q == f)}$$

$$\frac{\Gamma, v : \tau, u : \tau \vdash_{\mathbb{Q}} P <: Q}{\Gamma \vdash_{\mathbb{Q}} (\textbf{forall } v : \tau \textbf{ in } P) <: (\textbf{forall } u : \tau \textbf{ in } Q)}$$

$$\frac{\begin{array}{c}\Gamma, \ v_1 : \tau_1, \ \ldots, \ v_n : \tau_n \vdash_{\mathbb{Q}} P <: Q \\ \Gamma \models \forall \ i. \ \tau_i \neq \tau_{\sigma(i)} \qquad \sigma \text{ is a permutation on } n\end{array}}{\begin{array}{c}\Gamma \vdash_{\mathbb{Q}} \ (\textbf{forall } v_1 : \tau_1 \textbf{ in } \ldots \textbf{ forall } v_n : \tau_n \textbf{ in } P) <: \\ (\textbf{forall } v_{\sigma(1)} : \tau_{\sigma(1)} \textbf{ in } \ldots \textbf{ forall } v_{\sigma(n)} : \tau_{\sigma(n)} \textbf{ in } Q)\end{array}}$$

$$\frac{\Gamma \models \tau \qquad \Gamma, v : \tau \vdash_{\mathbb{Q}} P <: (\phi \Rightarrow p == e) \qquad v \text{ not free in } \phi, \ p \text{ or } e}{\Gamma \vdash_{\mathbb{Q}} (\textbf{forall } v : \tau \textbf{ in } P) <: (\phi \Rightarrow p == e)}$$

$\boxed{\Gamma \vdash_{\mathbb{Q}} \tau <: \sigma}$ *(Subtyping rules)*

$$\frac{\Gamma \vdash_{\mathbb{Q}} P <: Q}{\Gamma \vdash_{\mathbb{Q}} \tau \ / \ P <: \tau \ / \ Q}$$

Figure 3.2: Subquotienting and subtyping rules for λ_Q

$\boxed{\Gamma \models x \equiv_\tau y}$ *(Equality introduction)*

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \qquad \Gamma \models x ==_\tau y \qquad \tau \text{ is not a quotient type}}{\Gamma \models x \equiv_\tau y}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \;/\; (\phi \Rightarrow p == e) \qquad \Gamma; (\phi \Rightarrow p == e) :: \tau \models x \equiv_\tau y}{\Gamma, \phi \models x \equiv_{\tau/(\phi \Rightarrow p == e)} y}$$

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y : \tau \;/\; (\textbf{forall } v : \sigma \textbf{ in } Q) \qquad \Gamma, v : \sigma \vdash_{\mathbb{Q}} x \equiv_{\tau/Q} y}{\Gamma \vdash_{\mathbb{Q}} x \equiv_{\tau/(\textbf{forall } v:\sigma \textbf{ in } Q)} y}$$

$\boxed{\Gamma; Q \models x \equiv_\tau y}$ *(Quotient equalities)*

$$\frac{\Gamma \vdash_{\mathbb{Q}} x, y, p, e : \tau \qquad \Gamma \models \phi[\sigma] \qquad \Gamma \models y \equiv_\tau e[\sigma] \qquad x \subseteq_\sigma p}{\Gamma; (\phi \Rightarrow p == e) :: \tau \models x \equiv_\tau y}$$

Figure 3.3: Equality rules in λ_Q

3.9 Implementation

In order to demonstrate the utility of the quotient types of the core language λ_Q in a practical setting, we have developed Quotient Haskell. In particular, Quotient Haskell is an extension to Liquid Haskell that adds quotient types through the mechanism of datatype refinement, and checks whether functions defined on quotient types respect the necessary equations by generating constraints for an SMT solver. More specifically, our implementation of Quotient Haskell makes notable changes to Liquid Haskell parsing, syntax trees, equality constraint generation for quotient types, and type-checking and constraint generation for case expressions.

The first step of the Liquid Haskell pipeline that is modified by our implementation is the parser. In particular, the parser is extended to support the syntax illustrated by the examples in Sections 3.1, 3.2 and 3.3. As expressed in these examples and the core language presented in Section 3.5, in Quotient Haskell we require that the left-hand side of an equality appearing in the codomain of an equality constructor is a *pattern*. In practice, this requirement ensures that when type-checking a case expression for which an element of a quotiented type is the scrutinee, proof requirements corresponding to equality constructors can be either unfolded or erased. The syntax for equality patterns in Quotient Haskell is formally specified by the following grammar:

$$\begin{array}{lcll}
\mathit{lpat} & ::= & \mathit{var} & \text{variables} \\
& \mid & \mathit{literal} & \text{literals} \\
& \mid & \mathit{qcon}\ \mathit{lpat}_1\ ...\ \mathit{lpat}_k & \text{constructors } (k \geq 0) \\
& \mid & (\ \mathit{lpat}\ ) & \text{parenthesised pattern} \\
& \mid & (\ \mathit{lpat}_1,\ ...,\ \mathit{lpat}_k\ ) & \text{tuples } (k \geq 1)
\end{array}$$

In the above formulation of the *lpat* syntax, we write *var*, *literal*, *qvar*, *qcon* to denote the Haskell non-terminals for variables, literals, qualified variables and qualified constructors, respectively. The *lpat* rule can be understood as a more restricted version of the Haskell grammar rule for patterns that can appear on the left-hand side of a function. In particular, some patterns do not make sense for equality constructors, such as wildcards, irrefutable patterns and as-patterns.

We now turn to the syntax for equality constructors. For this definition, we make use of the Haskell non-terminals *expr* and *cxt*, which correspond to Haskell expressions and typeclass contexts. In addition, we use the Liquid Haskell non-terminal *rbind*, which corresponds to dependent bindings of a variable, which can appear to the left of an arrow. We also use *bpred*, which corresponds to bare predicates in Liquid Haskell, which are propositions surrounded by braces such as `{x == y}`. The syntax for an equality constructor is then given by the following rule, where $j, k \geq 0$:

$$\begin{aligned}\textit{eqcons} ::= \textit{var}\ \texttt{::}\ [\texttt{(cxt) =>}]\ \textit{rbind}_1\ \texttt{->}\ \cdots\ \texttt{->}\ \textit{rbind}_j \\ \texttt{->}\ \textit{pred}_1\ \cdots\ \texttt{->}\ \textit{pred}_k\ \texttt{->}\ \textit{lpat}\ \texttt{==}\ \textit{expr}\end{aligned}$$

Notably, the *eqcons* rule allows for quantified variables to be constrained by typeclasses. A typeclass context that appears in an equality constructor can be used to further constrain when respectability theorems are generated. In particular, after matching a case alternative with the left-hand side of an equality, Quotient Haskell will only generate the resulting respectability theorem if there exists the necessary instances for each of the matched variables.

To define the syntax for quotient types, we use the Liquid Haskell non-terminal *stype*, which corresponds to refinement types excluding bindings and bare predicates, and the Haskell non-terminal *simpletype*, which corresponds to the name of a type followed by type variables. The syntax for quotient type definitions is then given by the following rule, where $k \geq 1$:

$$\textit{quotty} \quad ::= \quad \textbf{data}\ \textit{simpletype} = \textit{stype}\ |/\ \textit{eqcons}_1\ |/\ \cdots\ |/\ \textit{eqcons}_k$$

In addition to the above syntax for quotient constructors, Quotient Haskell has syntax for providing explicit proofs that equality constructors are respected, where $k \geq 0$:

$$\begin{aligned}\textit{quotproof} ::= \texttt{respects}\ \texttt{<}\textit{qvar}\texttt{,}\textit{qvar}\texttt{>}\ \textit{var}\ \texttt{::}\ \textit{rtype}_1\ \texttt{->}\ \cdots\ \texttt{->}\ \textit{rtype}_k \\ \texttt{->}\ \texttt{\{}\ \textit{expr}\ \texttt{==}\ \textit{expr}\texttt{\}}\end{aligned}$$

Quotient type definitions and explicit proofs of respectability, as specified by *quotty* and *quotproof* respectively, are only valid within a Liquid Haskell block.

To support the above extensions, the syntax trees within Liquid Haskell are modified to include both quotient constructors and explicit quotient respectability proofs. In Liquid Haskell, a datatype refinement must typically have the same name as the datatype being refined. However, for quotient types the name must instead be unique with respect to any other type constructors in scope. As such, we extend the type-checking environment with the names of any quotient type constructors. Moreover, the names of quotient constructors must be unique with respect to any term identifiers in scope. After parsing and construction of the type-checking environment, each of the required checks associated with quotients are performed during the existing refinement checking phase. In the remainder of this section we describe the core steps involved in quotient type checking.

Quotient Wellformedness. The wellformedness check for each quotient ensures that the domain of each equality constructor is well-formed, and that both sides of the target equality are terms of the underlying type. We also check that both the name of a quotient type and its quotient constructors are unique in the scope. The name of a quotient type must be unique with respect to type constructors, while quotient constructors must be unique with respect to term identifiers.

Equality Lifting. As described in Section 3.7, to make use of the equalities introduced by quotients within arbitrary refinements, it is necessary to extend the notion of equality between terms that inhabit a quotient type. To do this, Quotient Haskell includes a transformation step that traverses every refinement type in the type-checking environment. We proceed with a description of this transformation as it applies to each refinement type. Firstly, we consider the underlying refinement predicates which contain an equality between terms of a quotient type. For each such equality, we generate a set of expressions by rewriting the equality with suitable quotients, as described in Section 3.7, and adding the preconditions of each used quotient by means of a conjunction. Finally, the new equality is constructed as a disjunction over this set of expressions. For example, consider the following quotient type that represents a list of integers for which any integer ≤ 0 acts as a unit:

```
data Positives
  =  [Int]
  |/ unit :: xs:[Int] -> x:{ x:Int | x <= 0 } -> x :: xs == xs
```

Intuitively, we can understand this type as ensuring that we can only consider the strictly positive integers within a list. In addition, we consider a function `f` of the following form:

```
f :: xs:Positives -> x:{ x:Int | (x + 1) :: xs == xs } -> ...
```

The precise codomain and definition of `f` are not important here. Instead, we focus on the argument `x`, which has a refinement type whose predicate is given by an equality between terms of a quotient type. In particular, `x` must be an integer such that when `x+1` is prepended to a term of type `Positives` it does not change the list. This equation evidently does not hold for the underlying type of lists of integers. However, as we are considering an equality between terms of a quotient type, we must also consider the equality constructors, in this case `unit`. The equality lifting transformation is precisely the approach by which Quotient Haskell includes equality constructors in its logic.

Equality lifting involves considering a restricted set of the possible sequences of rewrites by means of suitable equality constructors. In particular, an equality constructor can be used to rewrite an equation of the form `x == y` precisely when the left-hand side of the equality constructor's target equality unifies with either `x` or `y`. This rewrite occurs by first applying the unifying substitution to both equalities, then composing them, and finally building a conjunction over the preconditions of the equality constructor and the composite equality. At present, Quotient Haskell makes use of a naive rewriting algorithm that only permits each quotient to be used once during an equality checking pass through a term. Each pass makes use of a different permutation of the available quotients to allow for a different order in which rewrites may be applied.

As an example of how equality rewriting works in Quotient Haskell, we will consider rewriting the equation `(x + 1) :: xs == xs` by means of the `unit` equality constructor. This rewrite is possible because `(x + 1) :: xs` unifies with the left-hand side `x :: xs` of the target equality of `unit`, with substitution `x` := `x + 1`. We then proceed to apply this substitution to the equality target of `unit` to obtain the equation `(x + 1) :: xs == xs`. Finally, we compose the two equations and build an implication from the precondition of `unit` to the composite equality and obtain the proposition `x + 1 <= 0 && xs == xs`. Notably, this logical proposition is simply equivalent to `x + 1 <= 0` and will be simplified by Liquid Haskell. This proposition is combined by disjunction with all other valid sequences of rewrites. For example, after equality lifting is applied, the type of the function `f` becomes

```
f :: xs:Positives
  -> x:{ x:Int | (x + 1) :: xs == xs || x + 1 <= 0 }
  -> ...
```

Note that rewriting using a quotient may change the number of free variables in a refinement predicate. This will occur when there are variables bound in a quotient that do not appear on the left hand side of the target equality. Importantly, these free variables are only used in the construction of logical constraints to be handled either by an external SMT solver or proof by logical evaluation.

Quotient Respectability. Our implementation of Quotient Haskell covers two possible cases when considering quotient respectability: refined functions that take a term of a quotient type as input and do not match on that input, and case expressions that match on a single variable. We are interested only in cases where we have been provided with a type declaration that asserts that the input being considered must inhabit a quotient type. For functions that do not match on their quotiented input, we require no additional checks beyond those imposed by Liquid Haskell. As such, we only need consider matching functions and case expressions.

In the GHC API that is used by Liquid Haskell, both matching functions and case expressions are represented by single argument case expressions that match at most one level deep. In particular, we intuitively think of this representation as taking the form

```
let x = e in case e of { p_1 -> e_1; ...; p_k -> e_k }
```

where each p_i is either a variable or a constructor applied to a finite number of variables. In the proceeding transformation phase, Liquid Haskell transforms such case expressions into a sequence of conditional expressions, which make use of both distinguished testing predicates and selection functions for constructors to remove the let-binding and the free variables introduced by each case pattern. However, in the checking phase, the prior representation of case expressions are maintained and this representation is more suitable for checking quotient respectability.

A particularly useful consequence of working with case expressions that only permit matching on a single layer is the simplification of the unification check between the patterns of the case expression and the left-hand side of a quotient's equality target. In particular, we need only check whether either pattern is a mere variable, or the leading constructors match. The construction of the unifying substitution is similarly straightforward in this setting. With this in mind, we proceed to check quotient respectability of single argument case expressions as follows:

- We first check if the type of the scrutinee of the case expression is in the typing environment, and if this is a quotient type we proceed to check respectability, otherwise we are done;

- For each match $p \to e$ of the case expression we filter the relevant quotients by whether the left-hand side of their target equality unifies with p;

- For each quotient that passes the filter we apply the unifying substitution to both the right-hand side of the quotient's target equality and the expression e;

- We extract the refinements from the quotient's domain, from which we construct a logical implication that states that if the conjunction of these refinements hold then e is equal to the original case expression applied to the right-hand side of the quotient's target equality;

- If e is known to inhabit a quotient type, then for each quotient such that e unifies with the left-hand side of its equality target we apply the equality lifting transformation;

- We include the constructed constraint in the `.smt2` output generated by Liquid Haskell.

Note that the constraint generated by the above procedure will be a proposition quantified over the free variables that appear in the disjunction of the generated logical expressions. It is important that we normalise the terms that appear in the generated constraints, and as such the proof-by-logical-evaluation feature of Liquid Haskell is always enabled in Quotient Haskell.

To demonstrate the above procedure, we consider the example of the `map` function as refined on the quotient type `List` given in Section 3.2. In particular, only the definition

```
map f (Join x y) = Join (map f x) (map f y)
```

is relevant here, as the remaining definitions for `map` do not match with any of the quotients of `List`. When checking whether `map` is well-typed when refined to `List`, the first step of the above procedure is to check respectability, because `List` is a quotient type. In the next step, we filter the quotients of `List` to obtain only those that match with `Join x y`, which in this case happens to be every quotient for `List`. After the third step of the procedure, we obtain equations corresponding to the quotients `idl`, `idr` and `assoc`. For example, after proof-by-logical-evaluation the equation generated for `idl` is `Join (map f x) Empty == map f x`. As the quotients of `List` do not contain any refinements in the domain, we continue with the quotient substitution step. In the proceeding steps, we first confirm that the result type of `map` is a quotient type, which in this case is again `List`. We then apply all possible rewrites, as constructed from the quotients of `List`, to each of the generated equations. For example, we can rewrite `Join (map f x) Empty == map f x` using the `idl` quotient to obtain a new equation `map f x == map f x`. Finally, we generate constraints by taking the disjunction of equations built for each quotient. Continuing with our example of `idl`, we would generate a respectability constraint of the following form:

```
f:(a -> b) -> x:Mobile a ->
   Join (map f x) Empty == map f x || map f x == map f x || ...
```

This constraint will pass the SMT checking phase because the expression `map f x == map f x` holds definitionally. Consequently, the `map` function is shown to respect the `idl` function, and similar constraints are generated and shown to hold for both `idr` and `assoc`.

Quotient Subtyping. In practice, when types are checked rather than being inferred, subtyping rules need only be considered when type-checking function application. Because quotient types cannot yet be inferred in Quotient

Haskell, this translates to a simple extension of the function application case of Liquid Haskell's constraint generation algorithm. In particular, our extension is relevant in the case when the function being applied takes a quotient type as input, and the argument it is applied to is of a different quotient type. We proceed by checking whether for every quotient of the argument's type, there is a quotient of the function's input type that supercedes it with respect to the ordering relation on quotients given in Section 3.6. If this is the case, then we retype the argument to the input type of the function and continue with constraint generation. Otherwise, constraint generation fails and we report a type-checking error.

To reduce the performance cost of automatically deciding quotient subtyping, it is possible to cache the computed results of the subtyping relation for quotient types. In particular, quotient types in Quotient Haskell can be uniquely identified by their name, and caching can hence be achieved using a map from pairs of names to the results of the subtyping relation. Consequently, the performance cost of automatically inferring subtyping between quotient types is primarily determined by a one-time check of the subquotienting rules described in Section 3.6. In practice, this means that automatically inferring subtyping for quotient types does not usually have a significant impact on the runtime of the type checker, while improving ease of use.

Performance of Type Checking. The implementation of Quotient Haskell includes simple optimisations such as caching of the subtyping relation for quotient types, and erasure of trivial respectability theorems. In practice, we have found that the additional typing features of Quotient Haskell do not usually have a significant impact on time performance. For most practical examples, including those presented in this chapter, time performance is dominated by the external SMT solver.

3.10 Related work

Refinement and quotient types, along with their implementations, have been extensively studied in the literature. This prior work underpins the development of Quotient Haskell, which can be understood as extending a refinement type system with quotient types.

Refinement Types are types equipped with a subtyping predicate from an SMT-decidable logic [Bengtson et al., 2011; Rushby et al., 1998]. As such, refinement types utilise a restricted form of dependency in which bound variables can appear in the body of a predicate. Implementations of refinement type systems have been developed for many popular languages, including ML [Freeman and Pfenning, 1991], OCaml [Kawaguchi et al., 2010], and Haskell [Vazou et al., 2013]. Our core language λ_Q is introduced as a conservative extension to a generic underlying refinement type system, and supports the typing extensions of our practical implementation Quotient Haskell. The key idea is to translate the equational laws required by functions defined on quotient types into predicates in the underlying refinement logic. With this translation, we can utilise any suitable solver for the refinement logic to assist in the proof of quotient laws. Moreover, in order to make use of the equations described by quotients, we presented an extension of equality from a mere proposition in the refinement logic to a statement in the type system.

Liquid Haskell is an implementation of a bounded liquid type system [Rondon et al., 2008; Vazou et al., 2015] for Haskell, which adds termination checking to ensure correctness of refinement typing in a lazy setting [Vazou et al., 2014]. As introduced in this chapter, Quotient Haskell is an extension of the Liquid Haskell type system with quotient typing rules from the core language λ_Q. By developing Quotient Haskell as an extension to Liquid Haskell, quotients and subtypes can be utilised together and the elimination laws for quotients can make use of existing automation and rewriting developed for Liquid Haskell [Grannan et al., 2022; Vazou et al., 2017].

Quotient Types are types that are equipped with a distinguished notion of equality that may differ from the trivial, definitional equality for that type [Li, 2015; Hofmann, 1995]. Developing a well-behaved theory of quotient types for dependently typed languages has been an ongoing subject in the literature [Abbott et al., 2004; Nogin, 2002]. A key difficulty that arises when introducing quotient types to intensional type theories is the preservation of canonicity. In particular, when quotients are added to a type system by means of axiomatic rules, it may be possible to construct closed terms that do not compute to a canonical form by means of the elimination rule for the equality type. This axiomatic approach to quotients is precisely the approach adopted by the type system of λ_Q. However, this issue does not arise in λ_Q, because equality does not constitute a type with its own elimination form but rather a judgement in the type system.

Quotient types can be further generalised to quotient inductive families, quotient inductive-recursive types, or quotient inductive-inductive types [Altenkirch et al., 2018; Kaposi et al., 2019; Altenkirch and Kaposi, 2016]. Of particular note is quotient inductive families, whereby an inductive family can be thought of as a generalised algebraic datatype (GADT) that can additionally be indexed by a type, for example lists indexed by their length. A possible extension to λ_Q is the addition of typing rules for generalised algebraic quotient types, which in turn would describe how to extend Quotient Haskell with quotients for GADTs.

Implementations of Quotient Types include the higher-inductive types of Cubical Agda [Vezzosi et al., 2019], the HoTT library of Lean [van Doorn et al., 2017] and the axiomatic quotients in the Lean standard library [De Moura et al., 2015], a quotient types library for Coq [Cohen, 2013], various implementations in Isabelle [Slotosch, 1997; Paulson, 2006; Kaliszyk and Urban, 2011], quotient types by means of equivalence relations in NuPRL [Constable et al., 1986], and laws in Miranda [Thompson, 1986]. Existing implementations have largely focused on the use of quotient types in proof assistants. However, several practical use cases for quotient types have been highlighted in the literature that may translate to general-purpose functional programming, such as the Boom hierarchy [Meertens, 1986] and domain-specific languages with equational laws [Altenkirch and Kaposi, 2016]. A crucial drawback of existing implementations for quotient types are the manual proof obligations required by every function that is defined on a quotient type. In practice, these proof obligations can become unnecessarily burdensome, especially in cases where the proofs can be derived in a systematic manner. Quotient Haskell, and the core language λ_Q, were developed to address this drawback by making use of the well-known theory of refinement type systems to introduce quotients whose elimination laws can be handled by a suitable solver for the underlying refinement logic.

Automation for Quotient Types has been explored in the Isabelle quotient package, with a focus on transferring terms and properties from an underlying type to a corresponding quotient type [Kaliszyk and Urban, 2011; Huffman and Kunčar, 2013]. In particular, implicit coercion from an underlying type to a quotient type is termed ‘lifting’, while automatically translating properties from one to the other is referred to as ‘transfer’. As a consequence of quotient types being implemented as an internalised package, the automation of lifting and transfer is achieved in Isabelle by non-trivial proof automation tactics. In contrast, Quotient Haskell adds support for quotient inductive types by extending the Liquid Haskell typechecker. Consequently, in Quotient Haskell the lifting property simply arises from the introduction rule for quotient types, and the transferal of proofs follows from the definition of the conservative extension of equality in Section 3.7. As such, while the automation for quotient types in Isabelle addresses the problems of lifting and transfer, Quotient Haskell instead addresses the problem of automating the proofs of respectability theorems for quotient types.

NuPRL is a proof development system that provides support for both quotient types and proof automation [Constable et al., 1986; Nogin, 2002]. In NuPRL, quotient types are constructed by defining a new equality for a type using an equivalence relation, and this construction is provided as an operation on types. However, this approach is known to have a key drawback, namely that quotients of inductive types with a non-finite number of inductive positions often require the axiom of choice to construct their elimination map. In contrast, the quotient inductive types of Quotient Haskell do not have this issue. For example, such types can be used to prove properties of the Cauchy real numbers [Univalent goundations Program, 2013] and the weak delay monad [Chapman et al., 2019] without the need for the axiom of choice. Moreover, in contrast to Quotient Haskell, the quotient operation in NuPRL requires that a user constructs an explicit equivalence relation with proof witnesses for the necessary laws.

In summary, Quotient Haskell introduces a novel approach for supporting quotient types by extending a liquid type system with support for quotient inductive types whose generated respectfulness theorems are SMT-decidable. This approach is distinctly different from the setoid quotient types in languages such as Isabelle and NuPRL, and from the higher inductive types of Cubical Agda. Notably, this extension enables interoperability between quotient types and subtypes by generating constraints within a shared refinement logic. Moreover, in contrast to a higher inductive type in Cubical Agda, a quotient type in Quotient Haskell is defined by extending an existing type with inductive equalities, as detailed in Section 3.4. A practical consequence of this approach is the ability to define hierarchies of quotient types with implicit coercions that correspond to their canonical surjections. This feature follows from the subtyping rules of Quotient Haskell, as detailed in Section 3. For instance, every tree can be trivially coerced into a mobile without the need for explicit construction in Quotient Haskell.

3.11 Reflection

Existing systems that support quotient types have primarily focused on the goal of formalising mathematical theories. Quotient Haskell adds a new point to the design space, focusing on the use of quotient inductive types in a general purpose programming language. This is achieved by integrating quotient types into Haskell in a manner that reduces proof obligations arising from their use, thereby allowing users to focus on programming rather than proof. In this section, we reflect on the design, practical use and limitations of the Quotient Haskell system.

Usability of quotient types in a general purpose language was the guiding principle in the design choices of Quotient Haskell. There are three key design choices that were made. First of all, and most importantly, the system is built on top of a refinement type system, in this case Liquid Haskell. This approach allows us to take advantage of existing work and infrastructure on proof automation for refinement types, such as the generation of subtyping constraints and their translation into an SMT-decidable form. Moreover, extending Liquid Haskell with support for quotient types allows for interoperability with subtypes, which provides a more expressive type system while retaining the benefits of proof automation. In practice, many programming use cases involve both quotient types and subtypes, so being able to use them in combination is important.

Secondly, the system supports a particular class of quotient types known as quotient inductive types. This approach allows users to define equational laws alongside an underlying data type without requiring the explicit construction of an equivalence relation. In particular, quotient inductive types can be understood as extending the notion of equality in a manner that implicitly preserves its underlying properties, such as the equivalence relation and function congruence laws. As we have seen in the practical example sections, using quotient inductive types provides a simple and natural approach to defining types with equational laws.

And finally, the system introduces syntax and typing rules for quotient types that allow the reuse of existing data definitions without the need to redefine the constructors. For example, the type of mobiles is defined in Section 3.1 by quotienting an existing type for trees, with the same constructor names `Leaf` and `Bin` used for both the original and quotiented type. Crucially, this means that any function on mobiles can also be used on trees, without the need to explicitly convert between the two types. In contrast, existing systems that implement quotient inductive types, such as Cubical Agda, require unique data constructors for each type. Consequently, functions on a quotient type cannot be reused on a type that only holds the underlying data without explicit conversion. For example, in the case of mobiles, this would mean introducing new constructors for the mobile type, and using conversion functions when applying a function on mobiles to trees.

Applications were the central motivation for the development of Quotient Haskell. In addition to the applications presented in Sections 3.1, 3.2 and 3.3 we have explored a range of further examples, including modular arithmetic, coordinate systems, efficient data structures, and domain specific languages with equational laws. We briefly describe two of these examples below, and plan to focus on additional applications of quotient types as part of our further work.

Polar coordinates are typically represented by a pair of numbers corresponding to a magnitude and an angle. For example, in Haskell we might choose to define `type Polar = (Double, Int)`, where the angle is given to one degree of accuracy. However, this definition allows multiple representations of the same point in the space. In particular, the point represented by `(r, a)` can also be represented by `(-r, -a)`, and by any pair constructed by adding or subtracting a multiple of 360 degrees. This problem can be avoided by bounding the two polar components using subtyping, by defining `type Magnitude = {r : Double | r >= 0}` and `type Angle = {a : Int | a >= 0 && a < 360}`. However, this representation brings its own problem, namely that operations on angles may need to normalise their results. To avoid this issue, we can represent polar coordinates as a quotient type, which can be achieved in Quotient Haskell as follows:

```
data Polar
  =  (Magnitude, Int)
  |/ turn :: r:Magnitude -> a:Int -> (r, a) == (r, a `mod` 360)
```

In this definition, the `turn` equality constructor captures the property that the full rotation of a point around the origin does not change that point. To illustrate the utility of representing polar coordinates as a quotient type, consider the following rotation operation:

```
rotate :: Int -> Polar -> Polar
rotate x (r, a) = (r, a + x)
```

This definition is well-typed in Quotient Haskell, because the `turn` equality constructor is respected as a consequence of eliminating into the `Polar` type. In contrast, the definition would be ill-typed if polar coordinates were instead represented by the type `(Magnitude, Angle)` of bounded magnitudes and angles. Making the definition type correct would require that the `rotate` function normalises the resulting angle, which impacts on both simplicity and efficiency.

Another application of quotient types, and in particular quotient inductive types, is the representation of domain-specific languages with equational laws. To demonstrate this, we consider how Quotient Haskell can be used to add η-expansion to the (untyped) lambda calculus. In particular, we use a de Bruijn representation where variables are natural numbers given by `type Nat = {n : Int | n >= 0}`, and define terms of the lambda calculus as follows:

```
data Expr = Var Nat | Lam Nat Expr | App Expr Expr
```

To state the η-expansion law, we require a predicate `isNotFree :: Nat -> Expr -> Bool`, which asserts that a given variable is not free in a given term. This predicate can easily be defined by induction on the terms of `Expr`. Using these two definitions, we can then define the following quotient type in which η-expansion is explicitly given by an equality constructor:

```
data LambdaExpr
  =  Expr
  |/ eta :: f:LambdaExpr -> v:Int -> {isNotFree v f}
        -> Lam v (App f v) == f
```

A key operation for the lambda calculus is β-reduction, which for the underlying type `Expr` can take the form of a function `reduce :: Expr -> Expr`. A correct definition of `reduce` should have the property

```
  reduce (Lam v e) == Lam v (reduce e)
```

Indeed, this property typically forms the defining equation for `reduce` in the `Lam` case. As such, if we assume a correct definition, the type of `reduce` can be refined to `LambdaExpr -> LambdaExpr` in Quotient Haskell. Notably, this refinement is only possible by eliminating into `LambdaExpr` and not `Expr`. To construct a well-typed function from `LambdaExpr` to `Expr` would require η-expansion to be explicitly applied.

Limitations are an important practical consideration for users of Quotient Haskell. Below we consider a number of limitations and their practical consequences.

First of all, without function extensionality, i.e. the principle that two functions are equal if they are equal for all arguments, it is not possible to prove some equalities that might otherwise be expected to hold. In practice, this issue can arise when the type checker attempts to resolve an equality that involve higher-order functions. For example, in Liquid Haskell the equation `map (1+) xs == map (+1) xs` cannot be shown without function extensionality. Naively extending Liquid Haskell by adding function extensionality as an axiom is known to be inconsistent, however, solutions to this problem have been explored Vazou and Greenberg [2022]. At present, Quotient Haskell does not support reasoning with function extensionality when checking respectability theorems for quotient types. As we have seen, the lack of this feature does not preclude interesting and useful examples of quotient types, but we plan to consider in future work how function extensionality can be added to Quotient Haskell without compromising consistency.

Secondly, quotients of Generalised Algebraic Data Types (GADTs) are not currently supported in Quotient Haskell. GADTs allow inductive types to be indexed by other types, which enhances the expressivity of the type system by allowing data constructors to make use of additional type information. For example, a GADT of kind `Expr :: Type -> Type` that represents a simple form of well-typed expressions can be defined as follows:

```
data Expr a where
```

```
Val :: a -> Expr a
Add :: Expr Int -> Expr Int -> Expr Int
Eq :: Expr Int -> Expr Int -> Expr Bool
If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

In particular, the type of each constructor ensures that it can only be applied to arguments of suitable types, ensuring that expressions are always well-formed. Support for quotients of GADTs would allow equations to be introduced between expressions indexed by the same type. In the case of `Expr`, this would allow equations such as the following to be added:

```
ifTrue :: t:Expr a -> f:Expr a -> If (Val True) t f == t
commute :: m:Expr Int -> n:Expr Int -> Add m n == Add n m
```

Such equations would then need to be respected by functions defined on expressions, such as a well-typed evaluation function `eval :: Expr a -> a`. Adding support for GADTs to Quotient Haskell would enhance the space of examples that can be considered.

Thirdly, while the underlying logic of both Liquid Haskell and Quotient Haskell is SMT-decidable, the type checker may still be unable to automatically prove some statements. This can occur for two key reasons: generated constraints may not be solvable in reasonable time, and theorems that require inductive reasoning cannot always be solved. The first of these problems can require users to make design choices to minimise the complexity of generated constraints. In Quotient Haskell, this can mean designing equality constructors with fewer quantified variables and preconditions. Further work on automatically optimising and minimising generated SMT constraints can assist in reducing the design burden for users. Meanwhile, the second problem can be observed in Liquid Haskell by considering associativity of list concatenation, which cannot be automatically proved. This issue is similarly present in Quotient Haskell, and while proof by logical evaluation (PLE) can assist in some cases, there is not yet a precise classification of which inductive proofs can be automated. In practice, this means that generated respectability theorems that require inductive reasoning to prove will often require manual proofs. Quotient Haskell provides such a manual proof mechanism for respectability theorems, as demonstrated in Section 3.1.

And finally, at present Quotient Haskell has a limited error reporting system. In the case that a respectability theorem cannot be proven by the type checker and is not manually provided, a user is given a simple error message describing which equality constructor was not respected, followed by a generic Liquid Haskell error detailing the constraints. Improvements to the error reporting system of Quotient Haskell are a subject for future development work.

3.12 Conclusion and further work

In this chapter, we presented a core language that supports practical programming with quotient types. This is achieved by extending an established core language that supports liquid types with a class of quotients whose proof obligations can be automatically discharged by the type checker. In particular, this class has the property that the equational laws that functions on quotients must satisfy can be translated into a collection of constraints that can be decided by an SMT solver. Furthermore, we showed how the equations constructed by quotients can be exploited by the type system, which is an essential aspect of having proper support for quotient types. More specifically, we extended the notion of equality in the refinement logic of the underlying liquid type system by adding substitution rules corresponding to each quotient.

The above ideas are realised in practice by Quotient Haskell, a proof-of-concept extension of Liquid Haskell with quotient types. We presented a range of examples demonstrating the use of quotients for practical programming, including mobiles (commutative binary trees), the Boom hierarchy (lists, trees, bags and sets), and the rational numbers.

There are many interesting topics for further work. First of all, at present Quotient Haskell requires explicit typing declarations when using quotient types, and cannot infer them from untyped expressions. As such, a possible improvement is to extend the constraint generation phase to include the possibility of typing judgements that include quotients, and consequently to allow quotient types to be automatically inferred. Secondly, we could generalise the range of types that can be quotiented by including additional features of (Liquid) Haskell, such as GADTs and refinement polymorphism. Moreover, we could also generalise the subquotient relationship given in Section 3.6 to relate a wider range of quotients, and hence improve the reusability of functions defined on quotient types.

Thirdly, we could build upon previously explored ideas for reasoning about coinduction in Liquid Haskell [Mastorou et al., 2022] in order to introduce reason about quotients of coinductive structures such as streams. And finally, it is important to consider possible improvements to the practical aspects of the system, such as error messages and IDE support.

Chapter 4

HoTT Operads

Formal theories of datatypes such as polynomial functors and strictly positive types have been extensively studied by type theorists. For example, containers were developed as an internal theory of datatypes with the form a generalised polynomial [Abbott, 2003]. As first introduced in the previous chapter, the theory of containers captures both the strictly positive types [Abbott et al., 2005] and ordinary polynomial functors. Similarly, combinatorial species have been used to capture finitely-labelled structures [Yorgey, 2014]. While containers represent datatypes by their 'shapes' and 'positions', combinatorial species describe the construction of a structure from a finite set of labels. One of the most prominent applications of both containers and species is generic programming in a dependent type theory. In this way, both theories give rise to an internalised calculus of datatypes.

With the existence of several internal theories of datatypes it appears natural to consider a similar calculus of *operations over datatypes*. However, perhaps surprisingly, this particular approach is much less explored in the type theory literature. Meanwhile, category theory provides a well-known tool for describing such algebraic structures. In particular, the notion of an *operad* [May, 2006] is a generalisation of the standard concept of a category to a 'multicategory' (with one object), in which the source of a morphism is given by a finite sequence of objects. The theory of operads can be viewed as an extension to the theory of combinatorial species, in which an operad is simply a species together with a well-behaved notion of composition.

In this chapter we present an internalised calculus of composable operations by realising the categorical notion of an operad in a suitable intensional type theory. More concretely, in this chapter we:

- Internalise the notion of both planar (section 4.2) and symmetric (section 4.3) operads in homotopy type theory, and provide practical examples of each form;
- Introduce a generalised notion of operad whose operations are indexed over a univalent category [Ahrens et al., 2015];
- Prove that our notion of a generalised operad induces a canonical monad (section 4.7), which provides an *operadic* style of constructing programs from a small collection of composable terms;
- Demonstrate how the free operad can be constructed as both a higher inductive family and a higher inductive-recursive type (section 4.8).

While our formalisation work is presented in the meta-theory of Cubical Agda, the constructions and proofs outlined in this chapter are applicable to any implementation of homotopy type theory with higher inductive families. Our results are formalised in a Cubical Agda library that is freely available online [Hewer, 2020]. We refer to our library for proofs consisting of substantial technical detail that are not required for understanding the key ideas of this chapter.

4.1 Basic idea

To present a type-theoretic calculus of operations, we first require a notion of both an operation and a collection of operations. In this section, we consider a simple motivating example to introduce these notions, and describe what it means for a collection of operations to be operadic.

Informally, we can think of an *operation* as a map from elements of a structured collection to an element of the same collection. Two examples of such operations are addition of natural numbers and disjoint union of finite sets. In particular, these are both examples of a *binary* operation, where the domain is given by the binary product on the underlying collection. More generally, we will consider any domain that can be expressed as a strictly positive endofunctor on the underlying collection, e.g. finite and countable products. That is, we can understand an operation on a type A to simply be given by a strictly positive endofunctor F : Type → Type, together with an F-algebra $f : F\ A \to A$. For example, consider the following simple expression language in Agda:

```
data Expr : Type where
  val : ℕ → Expr
  add : Expr → Expr → Expr
```

The constructors val and add can be understood as operations on Expr — val is an algebra on the constant functor choosing ℕ, and add is a (curried) algebra on the binary diagonal functor.

While the above definition of an operation is sufficient to discuss properties of operations in isolation, such as associativity or preservation of limits, it is not sufficient to describe how operations interact, and more specifically compose. For example, consider the collection of operations which are constructed as finite compositions of the two constructors val and add, together with the identity function id : Expr → Expr. Importantly, the inclusion of the identity function will allow us to introduce variables in constructed operations. An example of a term that can be constructed in this manner is $\lambda\ x\ y\ z \to$ add (add x (val 1)) (add y z) of type Expr → Expr → Expr → Expr. Operations constructed in this way will evidently always be finitary, i.e. their domain will be a finite product of expressions, and they will always use each of their arguments precisely once.

In order to represent the desired collection of operations on the type Expr, we first observe that the type ∑[n ∈ ℕ] ((Fin n → Expr) → Expr), where Fin : ℕ → Type is the family of totally-ordered finite sets, corresponds to all planar finitary operations on Expr. However, our goal is to capture only those operations which are constructible as compositions of val, add and id. In order to do this, we might search for a subtype of all finitary operations which precisely corresponds to such a collection. Alternatively, it is perhaps easier and more natural to give an abstract representation of this collection, together with an interpretation as concrete operations. Indeed, we can do this by first defining the following inductive family:

```
data IExpr : ℕ → Type where
  id↑ : IExpr 1
  val↑ : ℕ → IExpr 0
```

$$\mathsf{add}\!\uparrow\ :\ \mathsf{IExpr}\ m \to \mathsf{IExpr}\ n \to \mathsf{IExpr}\ (m + n)$$

For every $n : \mathbb{N}$, we can think of the type $\mathsf{IExpr}\ n$ as representing the n-ary operations that are constructible as finite compositions of id, val and add. In particular, the constructors $\mathsf{id}\!\uparrow$ and $\mathsf{val}\!\uparrow$ correspond directly to the operations id and val, while $\mathsf{add}\!\uparrow$ describes how the outputs of an m-ary and n-ary operation can be plugged into the inputs of add to construct an $(m+n)$-ary operation. In order to interpret terms of $\mathsf{IExpr}\ n$ as concrete n-ary functions, we will make use of the projections $\pi^1 : (\mathsf{Fin}\ (m + n) \to \mathsf{Expr}) \to \mathsf{Fin}\ m \to \mathsf{Expr}$ and $\pi^2 : (\mathsf{Fin}\ (m + n) \to \mathsf{Expr}) \to \mathsf{Fin}\ n \to \mathsf{Expr}$, which project the first m and last n elements from an $(m + n)$-fold product respectively. We can then interpret terms of our abstract representation $\mathsf{IExpr}\ n$ as concrete operations as follows:

$$\begin{aligned}
&[\![\_]\!]\ :\ \mathsf{IExpr}\ n \to (\mathsf{Fin}\ n \to \mathsf{Expr}) \to \mathsf{Expr}\\
&[\![\ \mathsf{id}\!\uparrow\ ]\!]\ es = es\ \mathsf{zero}\\
&[\![\ \mathsf{val}\!\uparrow\ n\ ]\!]\ es = \mathsf{val}\ n\\
&[\![\ \mathsf{add}\!\uparrow\ e_1\ e_2\ ]\!]\ es = \mathsf{add}\ ([\![\ e_1\ ]\!]\ (\pi^1\ es))\ ([\![\ e_2\ ]\!]\ (\pi^2\ es))
\end{aligned}$$

Notably, the interpretation function $[\![\_]\!]$ is provably injective, i.e. we can construct a family of paths $(x\ y : \mathsf{IExpr}\ n) \to [\![\ x\ ]\!] \equiv [\![\ y\ ]\!] \to x \equiv y$. The proof of injectivity follows by induction on x and y, and in each case where the outer constructor of x and y differ the proof follows from absurdity. For example, when $x = \mathsf{id}\!\uparrow$ and $y = \mathsf{val}\!\uparrow\ n$ our injectivity assumption is $[\![\ \mathsf{id}\!\uparrow\ ]\!] \equiv [\![\ \mathsf{val}\!\uparrow\ n\ ]\!]$ or more simply $(\lambda\ es \to es\ \mathsf{zero}) \equiv (\lambda\ es\ \to \mathsf{val}\!\uparrow\ n)$, which is evidently false. As such, it is only necessary to consider the three inductive cases whereby the constructors considered are the same. The proof is trivial in the case where both x and y are $\mathsf{id}\!\uparrow$ and the remaining cases follow from injectivity of the constructors $\mathsf{val}\!\uparrow$ and $\mathsf{add}\!\uparrow$.

In addition to $[\![\_]\!]$ being an injective function, its codomain is an h-set and it therefore constitutes an embedding. In particular, this means that there is an equivalence between the path spaces $[\![\ x\ ]\!] \equiv [\![\ y\ ]\!]$ and $x \equiv y$ for any $x, y : \mathsf{IExpr}\ n$. That is, two abstract operations of type $\mathsf{IExpr}\ n$ are equal if and only if they have equal interpretations. In this way, the family of abstract operations $\mathsf{IExpr}\ n$ can be understood as a subtype of the concrete finitary operations on Expr. More specifically, the inductive family IExpr represents precisely the finite compositions of id, val and add.

It is natural to identify a reasonable condition for which a countable family such as $\mathsf{IExpr} : \mathbb{N} \to \mathsf{Type}$ can be considered an abstract collection of finitary operations without appealing to a specific interpretation function such as $[\![\_]\!]$. Importantly, this condition should capture the compositional structure of the operations described by a family such as IExpr. The minimal such condition we will require is for the family in question to be equipped with an identity operation and closed under an n-ary composition map that respects identity and associativity laws. In order to give a definition of an n-ary composition, we first require a family of functions $\mathsf{sum}\ n : (\mathsf{Fin}\ n \to \mathbb{N}) \to \mathbb{N}$ for calculating the sum of n natural numbers, which can be given inductively as follows:

$$
\begin{aligned}
&\mathsf{sum}\ 0\ ns = 0 \\
&\mathsf{sum}\ 1\ ns = ns\ \mathsf{zero} \\
&\mathsf{sum}\ (\mathsf{suc}\ (\mathsf{suc}\ n))\ ns = ns\ \mathsf{zero} + \mathsf{sum}\ (\mathsf{suc}\ n)\ (\lambda\ i \to ns\ (\mathsf{suc}\ i))
\end{aligned}
$$

For every $n : \mathbb{N}$ and n-ary product of natural numbers $ns : \mathsf{Fin}\ n \to \mathbb{N}$, we can now construct a family of functions $\mathsf{comp}\ n\ ns : \mathsf{IExpr}\ n \to ((i : \mathsf{Fin}\ n) \to \mathsf{IExpr}\ (ns\ i)) \to \mathsf{IExpr}\ (\mathsf{sum}\ n\ ns)$:

$$
\begin{aligned}
&\mathsf{comp}\ 1\ ns\ \mathsf{id}{\uparrow}\ es = es\ \mathsf{zero} \\
&\mathsf{comp}\ 0\ ns\ (\mathsf{val}{\uparrow}\ k)\ es = \mathsf{val}{\uparrow}\ k \\
&\mathsf{comp}\ .(m+n)\ ns\ (\mathsf{add}{\uparrow}\ e_1\ e_2)\ es = \\
&\quad \mathsf{let}\ es_1 = \mathsf{comp}\ m\ (\pi^1\ ns)\ e_1\ (\Pi^1\ es) \\
&\qquad\quad es_2 = \mathsf{comp}\ m\ (\pi^2\ ns)\ e_2\ (\Pi^2\ es) \\
&\quad\ \ \mathsf{in}\ \mathsf{subst}\ \mathsf{IExpr}\ (\pi^1{+}\pi^2\ ns)\ (\mathsf{add}{\uparrow}\ es_1\ es_2)
\end{aligned}
$$

In the definition of comp, we make use of auxiliary dependent projection functions,

$$
\begin{aligned}
\Pi^1 &: ((i : \mathsf{Fin}\ (m+n)) \to \mathsf{IExpr}\ (ns\ i)) \to (i : \mathsf{Fin}\ m) \to \mathsf{IExpr}\ (\pi^1\ ns\ i), \\
\Pi^2 &: ((i : \mathsf{Fin}\ (m+n)) \to \mathsf{IExpr}\ (ns\ i)) \to (i : \mathsf{Fin}\ n) \to \mathsf{IExpr}\ (\pi^2\ ns\ i),
\end{aligned}
$$

together with the path

$$
\pi^1{+}\pi^2\ ns : \mathsf{sum}\ m\ (\pi^1\ ns) + \mathsf{sum}\ n\ (\pi^2\ ns) \equiv \mathsf{sum}\ (m+n)\ ns.
$$

The constructions for Π^1, Π^2 and $\pi^1{+}\pi^2$ can be found in our Cubical Agda formalisation.

Intuitively, our composition map plugs the outputs of n operations into the inputs of an n-ary operation. Importantly, we can show that our definition of comp preserves associativity and identity laws, where the identity operaton is given by id↑. The left and right unit laws assert that given any n-ary operation, either plugging its output into the input of the identity operation represented, or plugging the identity's output into each of its n inputs, leaves the operation unchanged. Associativity asserts that composing a finitely branching tree of operations 'bottom-up' is the same as composing 'top-down'. We provide a formal characterisation of these laws in Section 4.

We can also observe that a similar compositional structure to that described by comp exists for the family of all concrete finitary operations (Fin n → Expr) → Expr, where the composition map is simply given by n-ary function composition and the unit operation is given by the term λ es → es zero : (Fin 1 → Expr) → Expr. The interpretation function ⟦_⟧ can then be understood as a homomorphism between such structures, respecting both the composition map and the identity operation. This common compositional structure on collections of operations, is precisely the structure described by planar operads.

4.2 Planar operads

A collection of finitary operations with a distinguished unit, closed under an n-ary composition map that respects unitality and associativity, is also known as a *planar operad*. Indeed, the compositional structure detailed for the family IExpr : ℕ → Type in the previous section precisely describes a planar operad. In particular, planar in this context simply means that our operadic composition map comes equipped with a choice of a total order on the input operations. Consequently, the composition map of a planar operad does not necessarily respect reordering of the input operations. In this section we provide a formalisation of planar operads in Cubical Agda, together with an example of such an operad and several important properties of our definition.

We begin by recalling that we can understand a countable family of types K : ℕ → Type as a family of finitary operations indexed by their number of inputs. In order to avoid higher coherences, we will consider only families of h-sets, i.e. $K : \mathbb{N} \to \mathsf{hSet}$ where hSet is the universe of h-sets. In Cubical Agda, hSet is not a built-in universe but is instead expressed as a dependent sum $\sum[\ A \in \mathsf{Type}\ ]\ \mathsf{isSet}\ A$, where $\mathsf{isSet}\ A$ is the proposition that A is indeed an h-set. However, given a family of sets $K : \mathbb{N} \to \mathsf{hSet}$, we will simply write $K\ n : \mathsf{Type}$ for the first projection, and will make it clear when we use the proof that this type is an h-set.

Given any family of h-sets $K : \mathbb{N} \to \mathsf{hSet}$, we can define the planar operads on K as a record type $\mathsf{record}\ \mathsf{PlanarOperad}\ K : \mathsf{Type}$, i.e. a finitely iterated sigma type with named projections. In particular, a planar operad $O : \mathsf{PlanarOperad}\ K$ comes equipped with a distinguished identity operation $\mathsf{id}\ O : K\ 1$ and a family of composition maps

$$\mathsf{comp}\ O\ n\ ns : K\ n \to ((i : \mathsf{Fin}\ n) \to K\ \ (ns\ i)) \to K\ \ (\mathsf{sum}\ n\ ns)\,,$$

for every $n : \mathbb{N}$ and $ns : \mathsf{Fin}\ n \to \mathbb{N}$. The composition map of an operad tells us how to plug the outputs of n operations, each with its own number of inputs ns, into the inputs of an operation with n inputs. Furthermore, we require paths witnessing that this composition map respects the identity and associativity laws. In particular, the left identity law is witnessed by a path

$$\mathsf{idl}\ O\ n\ k : \mathsf{comp}\ O\ 1\ \ (\lambda\ i \to n)\ \ (\mathsf{id}\ O)\ \ (\lambda\ i \to k)\ \equiv k,$$

for every $n : \mathbb{N}$ and $k : K\ n$. The path $\mathsf{idl}\ O\ n\ k$ witnesses that the operation plugging the output of k into the identity is equal to k. Right identity is then witnessed by a heterogeneous path

$$\begin{aligned}\mathsf{idr}\ O\ n\ k : \mathsf{PathP}\ &(\lambda\ i \to K\ \ (\mathsf{sum\text{-}idr}\ n\ i))\\ &(\mathsf{comp}\ O\ n\ (\lambda\ i \to 1)\ k\ (\lambda\ i \to \mathsf{id}\ O))\\ &k,\end{aligned}$$

where $\mathsf{sum\text{-}idr}\ n : \mathsf{sum}\ n\ \ (\lambda\ i \to 1)\ \equiv n$ is the path witnessing that the sum of n ones is n. We can understand $\mathsf{idr}\ O\ n\ k$ as witnessing that the operation plugging the output of the identity into each of the n inputs of k is equal to k. Finally, for every collection of natural numbers

$$n : \mathbb{N},\ ns : \mathsf{Fin}\ n \to \mathbb{N},\ nss : (i : \mathsf{Fin}\ n) \to \mathsf{Fin}\ \ (ns\ i) \to \mathbb{N},$$

and every collection of operations

$$\begin{aligned}&k : K\ n,\\ &ks : (i : \mathsf{Fin}\ n) \to K\ \ (ns\ i)\,,\\ &kss : (i : \mathsf{Fin}\ n)\,(j : \mathsf{Fin}\ \ (ns\ i)) \to K\ \ (nss\ i\ j)\,,\end{aligned}$$

the associativity law of comp O is witnessed by a heterogeneous path

```
assoc O n ns nss k ks kss :
  PathP (λ i → sum-assoc n ns nss i)
        (comp O n (λ i → sum (ns i) (nss i))
              k (λ i → comp O (ns i) (nss i) (ks i) (kss i)))
        (comp O (sum n ns) (uncurry nss ∘ sumΣ)
              (comp O n ns k ks) (uncurry kss ∘ sumΣ))
```

where the function

$$\mathsf{sum}\Sigma\ ns : \mathsf{Fin}\ \ (\mathsf{sum}\ n\ ns) \to \Sigma[\ i \in \mathsf{Fin}\ n\ ]\ \mathsf{Fin}\ \ (ns\ i)$$

is an equivalence witnessing that the totally ordered finite sets are closed under dependent sums, uncurry is the usual dependent uncurrying function, and the path

```
sum-assoc n ns nss :
          sum n (λ i → sum (ns i) (nss i))
        ≡ sum (sum n ns) (uncurry nss ∘ sumΣ)
```

witnesses the associativity of finite sums of natural numbers. The associativity condition asserts that the two approaches to building a composition tree from a collection of operations k, ks and kss, i.e. top down or bottom up, are equal under the composition map comp O.

We have already seen one example of a planar operad, namely the family of h-sets IExpr : ℕ → hSet equipped with the n-ary composition map comp described in the previous section. As a second example, we will consider lists with 'holes' which we call *partial lists* and define as follows:

```
data PartialList (A : Type) : ℕ → Type where
```

$$\begin{aligned}
&[] \; : \; \mathsf{PartialList}\; A\; 0\\
&\_{::}\_ \; : \; A \to \mathsf{PartialList}\; A\; n \to \mathsf{PartialList}\; A\; n\\
&\mathsf{poke} \; : \; \mathsf{PartialList}\; A\; n \to \mathsf{PartialList}\; A\; (\mathsf{suc}\; n)
\end{aligned}$$

Intuitively, the family $\mathsf{PartialList}\; A$ corresponds to partial lists indexed by their number of holes. The idea of 'poking holes' in a data structure is similar to that of previous work on zippers [Huet, 1997] and derivatives of containers [Abbott et al., 2003; McBride, 2001, 2008]. The constructors $[]$ and $\_{::}\_$ can be understood in the usual way, while poke allows us to introduce a hole at the start of a given list. Partial lists come equipped with a concatenation operation $\_{++}\_ : \mathsf{PartialList}\; A\; m \to \mathsf{PartialList}\; A\; n \to \mathsf{PartialList}\; A\; (m + n)$ defined as follows:

$$\begin{aligned}
&[] \mathbin{++} ys = ys\\
&(x :: xs) \mathbin{++} ys = x :: (xs \mathbin{++} ys)\\
&\mathsf{poke}\; xs \mathbin{++} \mathrm{ys} = \mathsf{poke}\; (xs \mathbin{++} ys)
\end{aligned}$$

Given any h-set $A : \mathsf{Type}$, the countable family of h-sets $\mathsf{PartialList}\; A$ comes equipped with a planar operadic structure $O : \mathsf{PlanarOperad}\; (\mathsf{PartialList}\; A)$, with n-ary composition map:

$$\begin{aligned}
&\mathsf{comp}\; O\; 0\; ns\; xs\; xss = xs\\
&\mathsf{comp}\; O\; (\mathsf{suc}\; n)\; ns\; (x :: xs)\; xss = x :: \mathsf{comp}\; O\; (\mathsf{suc}\; n)\; ns\; xs\; xss\\
&\mathsf{comp}\; O\; 1\; ns\; (\mathsf{poke}\; xs)\; xss\\
&\quad = \mathsf{subst}\; (\mathsf{PartialList}\; A)\; (\mathsf{+\text{-}zero}\; (ns\; \mathsf{zero}))\; (xss\; \mathsf{zero} \mathbin{++} xs)\\
&\mathsf{comp}\; O\; (\mathsf{suc}\; (\mathsf{suc}\; n))\; ns\; (\mathsf{poke}\; xs)\; xss\\
&\quad = xss\; \mathsf{zero} \mathbin{++} \mathsf{comp}\; O\; (\mathsf{suc}\; n)\; (ns \circ \mathsf{suc})\; xs\; (xss \circ \mathsf{suc})
\end{aligned}$$

The identity operation for the compositional structure on partial lists is then simply defined as $\mathsf{id}\; O = \mathsf{poke}\; []$. We refer readers interested in the corresponding proofs of identity and associativity laws to our Cubical Agda formalisation but we do not provide them here.

We might consider whether the operations characterised by the operadic structure on $\mathsf{PartialList}\; A$ corresponds to a collection of concrete n-ary operations. Indeed, there is an interpretation map $\mathsf{fill} : \mathsf{PartialList}\; A\; n \to (\mathsf{Fin}\; n \to \mathsf{List}\; A) \to \mathsf{List}\; A$ which fills the n holes of a partial list with the provided n lists. In a similar fashion to the interpretation function $[\![\_]\!] : \mathsf{IExpr}\; n \to (\mathsf{Fin}\; n \to \mathsf{Expr}) \to \mathsf{Expr}$ described in the previous section, the function fill can be shown to be a homomorphism between planar operads and respects both the compositional structure and identity operation. We discuss how the notion of an operad homomorphism can be internalised in HoTT in Section 8.

4.3 Symmetric operads

In the previous section, we introduced planar operads in HoTT, as a generalisation of the usual notion of operads whose operations also come equipped with actions of symmetric groups that are compatible with the operad composition map. In this section, we will discuss how our existing notion of a planar operad can be specialised to the symmetric case. Naively, this means extending our definition of a planar operad $O : \mathsf{PlanarOperad}\ K$ with a field

$$\mathsf{permute}\ O\ n : \mathsf{Fin}\ n \simeq \mathsf{Fin}\ n\ \to K\ n \simeq K\ n,$$

for every $n : \mathbb{N}$, where $A \simeq B$ is the type of isomorphisms between the h-sets A and B. An isomorphism $\sigma : A \simeq B$ has projections $\mathsf{fun}\ \sigma : A \to B$, $\mathsf{inv}\ \sigma : B \to A$, $\mathsf{linv} : (a : A) \to \mathsf{inv}\ \sigma\ (\mathsf{fun}\ \sigma\ a) \equiv a$ and $\mathsf{rinv} : (b : B) \to \mathsf{fun}\ \sigma\ (\mathsf{inv}\ \sigma\ b) \equiv b$. Moreover, $\mathsf{permute}\ O\ n$ must be a group homomorphism between the automorphism groups $\mathsf{Fin}\ n \simeq \mathsf{Fin}\ n$ and $K\ n \simeq K\ n$, but we omit the evident fields to witness preservation of identity, composition and symmetry. Furthermore, for all $ns : \mathsf{Fin}\ n \to \mathbb{N}$, $k : K\ n$, $ks : (i : \mathsf{Fin}\ n) \to K\ (ns\ i)$ and $\sigma : \mathsf{Fin}\ n \simeq \mathsf{Fin}\ n$, we require an additional field

$$\begin{aligned}\mathsf{symm}\ O\ n\ ns\ k\ ks\ \sigma : \mathsf{PathP}\ &(\lambda\ i \to K\ (\mathsf{sum\text{-}permute}\ n\ ns\ \sigma\ i))\\ &(\mathsf{comp}\ O\ n\ ns\ (\mathsf{fun}\ (\mathsf{permute}\ O\ n\ \sigma)\ k)\ ks)\\ &(\mathsf{comp}\ O\ n\ ns\ k\ (ks \circ \mathsf{fun}\ \sigma))\end{aligned}$$

witnessing that $\mathsf{permute}\ O\ n$ is compatible with the operad composition map.

This naive presentation of symmetric operads can be internalised in standard Martin-Löf type theory, but requires an explicit construction of the permutation homomorphism and the proof of compatibility with the operad composition map. However, this ad-hoc definition of planar operads can be unwieldy in practice. This problem arises because a countable family of h-sets $K : \mathbb{N} \to \mathsf{Type}$ does not, in general, come equipped with actions of symmetric groups. Intuitively, this means that the composition map of a planar operad can freely make use of the order in which operations are provided. This is an expected consequence of indexing operations by the natural numbers which are in equivalence with the type of finite sets equipped with a total order, i.e. the type $\sum[\ A \in \mathsf{Type}\ ]\ \sum[\ n \in \mathbb{N}\ ]\ A \simeq \mathsf{Fin}\ n$. This can readily be seen by recognising that for any $n : \mathbb{N}$, the type $\sum[\ A \in \mathsf{Type}\ ]\ A \simeq \mathsf{Fin}\ n$ is a singleton and consequently contractible.

As an alternative approach, we can instead exploit additional principles present in HoTT in order to consider operadic structures on collections of operations indexed over the groupoid of finite sets and bijections, i.e. finite sets that do not, in general, come equipped with a total-ordering on their elements. A common presentation of this groupoid in HoTT, known as Bishop-finite sets, internalises finiteness as a predicate on types, $\mathsf{isFinite} : \mathsf{Type} \to \mathsf{Type}$, as follows:

$$\mathsf{isFinite}\ A = \sum[\ n \in \mathbb{N}\ ]\ \|\ A \simeq \mathsf{Fin}\ n\ \|$$

where $\|\ X\ \|$ is the propositional truncation of a type X. For any type A, there is a well-known equivalence between the type $\mathsf{isFinite}\ A$ and the h-proposition $\|\ \sum[\ n \in \mathbb{N}\ ]\ A \simeq \mathsf{Fin}\ n\ \|$, which follows from the pigeonhole principle. Intuitively, this equivalence witnesses that every finite set has a unique cardinality, and therefore this notion of finiteness is an h-proposition.

The universe of Bishop-finite sets can be internalised à la Tarski as a large type of codes $\mathsf{FinSet} = \sum[\ A \in \mathsf{Type}\ ]\ \mathsf{isFinite}\ A$ together with an interpretation map $\mathsf{El} : \mathsf{FinSet} \to \mathsf{Type}$ which is simply given as the first projection map. Importantly, this interpretation map is an embedding which follows from the proof that the finiteness of a type is an h-proposition. That is, for any finite sets $A\ B : \mathsf{FinSet}$ the functorial action of El on paths is an equivalence between the path space of codes $A \equiv B$ and the path space of the underlying types $\mathsf{El}\ A \equiv \mathsf{El}\ B$. A Tarski universe with this property is also known as a univalent universe. We recall that the property that a given type A is an h-set, i.e. $(x\ y : A)\ (p\ q : x \equiv y) \to p \equiv q$, is itself an h-proposition. Therefore, given a finite set $(A\ ,\ n\ ,\ p) : \mathsf{FinSet}$, it follows by application of the elimination principle for propositional truncation to p, and from the underlying isomorphism between A and $\mathsf{Fin}\ n$, that since

$\mathsf{Fin}\ n$ is an h-set so is A. That is, for every finite set $A : \mathsf{FinSet}$ the underlying type $\mathsf{El}\ A$ is an h-set. Therefore, given any two finite sets $A\ B : \mathsf{FinSet}$, by application of univalence the type of paths between them $A \equiv B$ is equivalent to the type of isomorphisms $\mathsf{El}\ A \simeq \mathsf{El}\ B$ between their underlying types. We will express the forward direction of this equivalence as follows:

$$\mathsf{un} : (A\ B : \mathsf{FinSet}) \to \mathsf{El}\ A \simeq \mathsf{El}\ B \to A \equiv B$$

The universe FinSet is closed under various common type formers including both dependent sums and products. In particular, for every finite set $A : \mathsf{FinSet}$ and family of finite sets $B : \mathsf{El}\ A \to \mathsf{FinSet}$, we have both $\hat{\sum}\ A\ B : \mathsf{FinSet}$ and $\hat{\prod}\ A\ B : \mathsf{FinSet}$ together with definitional equalities $\mathsf{El}\ (\hat{\sum}\ A\ B) = \sum[\ a \in \mathsf{El}\ A\ ]\ \mathsf{El}\ (B\ a)$ and $\mathsf{El}\ (\hat{\prod}\ A\ B) = (a : \mathsf{El}\ A) \to \mathsf{El}\ (B\ a)$. The finiteness proofs for both $\mathsf{isFinite}\ (\sum[\ a \in \mathsf{El}\ A\ ]\ \mathsf{El}\ (B\ a))$ and $\mathsf{isFinite}\ ((a : \mathsf{El}\ A) \to \mathsf{El}\ (B\ a))$ can be found in our Cubical Agda formalisation and in Voevodsky's unimath Coq library. The universe of Bishop-finite sets is also closed under isomorphism, i.e. for any two finite sets $A\ B : \mathsf{FinSet}$ we can construct $A \mathrel{\hat{\equiv}} B : \mathsf{FinSet}$ with a definitional equality $\mathsf{El}\ (A \mathrel{\hat{\equiv}} B) = \mathsf{El}\ A \simeq \mathsf{El}\ B$ and a proof of $\mathsf{isFinite}\ (\mathsf{El}\ A \simeq \mathsf{El}\ B)$. We again refer interested readers to our Cubical Agda formalisation.

From this notion of Bishop-finite sets internalised in HoTT, we can define a collection of finitary operations as a FinSet-indexed family of h-set. Indeed, such families are also known as *combinatorial species* or equivalently *finitary containers*. Every such family $K : \mathsf{FinSet} \to \mathsf{Type}$ comes equipped with a canonical right action of symmetric groups of the form $\mathsf{El}\ A \simeq \mathsf{El}\ A$, or equivalently $A \equiv A$, for $A : \mathsf{FinSet}$. Importantly, not only are $\mathsf{El}\ A \simeq \mathsf{El}\ A$ and $A \equiv A$ equivalent as sets, but their canonical group structures are preserved by this equivalence. That is, they are also equivalent as groups. In particular, for every symmetric group element $\sigma : A \equiv A$ this action is given by the automorphism witnessed by $\mathsf{subst}\ K\ \sigma : K\ A \to K\ A$ and $\mathsf{subst}\ K\ (\mathsf{sym}\ \sigma) : K\ A \to K\ A$. We note that as K is a family of h-sets, the group of paths $K\ A \equiv K\ A$ is equivalent to the group of automorphisms $K\ A \simeq K\ A$. It follows from the laws of substitution over paths, that this construction from $A \equiv A$ to $K\ A \equiv K\ A$ is provably a group homomorphism. Concretely, for any $X : \mathsf{Type}$ and $Y : X \to \mathsf{Type}$, together with terms $x\ y\ z : X$, $b : Y\ x$, and paths $p : x \equiv y$ and $q : y \equiv z$, we make use of the following standard coherence laws for substitution over paths:

$$\begin{aligned} &\mathsf{subst}\ Y\ \mathsf{refl}\ b \equiv b && \text{(identity)} \\ \mathsf{subst}\ Y\ q\ \ (&\mathsf{subst}\ Y\ p\ b) \equiv \mathsf{subst}\ Y\ \ (p \cdot q)\ \ b && \text{(composition)} \end{aligned}$$

The proof of the symmetry law follows from both the identity and composition laws along with the symmetry law on paths, i.e. $p \cdot \mathsf{sym}\ p \equiv \mathsf{refl}$.

Thus far, we have seen how FinSet-indexed families come equipped with a canonical notion of right action of symmetric groups, in contrast to that of $\mathbb{N}$-indexed families. However, we have yet to describe what it means for a family of types indexed over FinSet to have an operadic structure. To do this, we begin by considering the definition of planar operads and replacing the finite summation function sum with the dependent sum type former of finite sets $\hat{\sum}$. The intuitive connection is that sum is the dependent sum type former for the Tarski universe with codes given by $\mathbb{N}$ and interpretation map $\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$, i.e. the universe of totally-ordered finite sets. For example, given a collection of finitary operations $K : \mathsf{FinSet} \to \mathsf{Type}$, a finite set $A : \mathsf{FinSet}$, and family of finite sets $B : \mathsf{El}\ A \to \mathsf{FinSet}$, the composition map of an operad $O : \mathsf{Operad}\ K$ is given by a field

$$\mathsf{comp}\ O\ A\ B : K\ A \to ((a : \mathsf{El}\ A) \to K\ \ (B\ a)) \to K\ (\hat{\textstyle\sum}\ A\ B)$$

Moreover, given the singleton finite set $\top : \mathsf{FinSet}$ and a family of finite sets $C : (a : \mathsf{El}\ A) \to \mathsf{El}\ \ (B\ a) \to \mathsf{FinSet}$, the right identity and associativity laws are now given as heterogeneous paths over the following canonical paths:

$$\hat{\textstyle\sum}\text{-}\mathsf{idr}\ A : \hat{\textstyle\sum}\ A\ (\lambda\ a \to \top) \equiv A$$

$$\begin{gathered} \hat{\textstyle\sum}\text{-}\mathsf{assoc}\ A\ B\ C : \\ \hat{\textstyle\sum}\ A\ (\lambda\ a \to \hat{\textstyle\sum}\ (B\ a)\ (Ca)) \equiv \hat{\textstyle\sum}\ (\hat{\textstyle\sum}\ A\ B)\ (\lambda\ (a\ ,\ b) \to C\ a\ b) \end{gathered}$$

We can construct these paths by appropriately applying un to the corresponding isomorphisms between types. That is, for all $A : \mathsf{Type}$, $B : A \to \mathsf{Type}$ and $C : (a : A) \to B\ a \to \mathsf{Type}$, we construct $\hat{\sum}$-idr and $\hat{\sum}$-assoc by respectively applying un to the following canonical isomorphisms:

$$\textstyle\sum\text{-}\mathsf{idr} : A \times \top \simeq \mathsf{El}\ A$$

$$\textstyle\sum\text{-}\mathsf{assoc} :$$
$$\textstyle\sum[\ a \in A\ ]\ \sum[\ b \in B\ a\ ]\ C\ a\ b \simeq \sum[\ (a\ ,\ b) \in \sum[\ a \in A\ ]\ B\ a\ ]\ C\ a\ b$$

Furthermore, given any operation $k : K\ A$ it is also now necessary to witness the left identity as a heterogeneous path as follows:

$$\begin{aligned}\mathsf{idl}\ O\ A\ k : \mathsf{PathP}\ (\lambda\ i \to K\ (\hat{\textstyle\sum}\text{-}\mathsf{idl}\ A\ i))\\ (\mathsf{comp}\ O\ A\ (\lambda\ a \to \top)\ k\ (\lambda\ a \to \mathsf{id}\ O))\ k,\end{aligned}$$

where $\hat{\sum}\text{-}\mathsf{idl}\ A : \hat{\sum}\ \top\ (\lambda\ t \to A) \equiv A$ is constructed by application of $\mathsf{un}\ (\hat{\sum}\ \top\ (\lambda\ t \to A))\ A$ to the canonical isomorphism between $\top \times \mathsf{El}\ A$ and $\mathsf{El}\ A$.

To complete our definition of a symmetric operad, we might consider the addition of a field witnessing that our previously described right actions of symmetric groups on the collection of operations K is compatible with the composition map. That is, we might introduce an analogue to the symm field for FinSet-indexed families. In particular, for every finite set $A : \mathsf{FinSet}$, family of finite sets $B : \mathsf{El}\ A \to \mathsf{FinSet}$, operation $k : K\ A$, family of operations $ks : (a : \mathsf{El}\ A) \to K\ (\mathsf{El}\ (B\ a))$ and symmetric group element $\sigma : A \equiv A$, we could introduce the following path as a field:

$$\begin{aligned}&\mathsf{symm}\ O\ A\ B\ k\ ks\ \sigma :\\ &\quad \mathsf{PathP}\ (\lambda\ i \to K\ (\hat{\textstyle\sum}\ (\sigma\ i)\ (\lambda\ a \to B\ (\mathsf{transp}\ (\lambda\ j \to \mathsf{El}\ (\sigma\ (i \vee j)))\ i\ a))))\\ &\qquad\qquad (\mathsf{comp}\ O\ A\ (B \circ \mathsf{subst}\ \mathsf{El}\ \sigma)\ k\ (ks \circ \mathsf{subst}\ \mathsf{El}\ \sigma))\\ &\qquad\qquad (\mathsf{comp}\ O\ A\ B\ (\mathsf{subst}\ K\ \sigma\ k)\ ks)\end{aligned}$$

However, it turns out that this formulation of symm, where the right action of a symmetric group is defined by path substitution, can already be constructed as follows:

```
symm O A B k ks σ i
      = comp O (σ i) (λ a → B (transp (λ j → El (σ (i ∨ j))) i a))
                         (transp (λ j → K (σ (i ∧ j))) (∼ i) k)
                         (λ a → ks (transp (λ j → El (σ (i ∨ j))) i a))
```

That is, by switching from ℕ to FinSet indexed families, and changing finite summation sum to the dependent sum type former $\hat{\sum}$, our notion of an operad restricts to the classical definition which includes compatability with the right actions of symmetric groups. Importantly, we can similarly translate our notion of a planar operad morphism to use FinSet and $\hat{\sum}$, and we do not require a field witnessing preservation of the actions of symmetric groups. In particular, for any two families K L : FinSet → Type, family of maps $(A$: FinSet$)$ → K A → L A, operation $k : K\ A$, and symmetric group element $\sigma : A \equiv A$, a proof witnessing that $f\ A$ (subst $K\ \sigma\ k$) $\equiv$ subst $L\ \sigma\ (f\ A\ k)$ follows directly from the property that substitution commutes with morphisms in slice categories.

We now recall our example of the planar operadic structure on the type family IExpr : ℕ → Type, and introduce the following, near identical, example of a FinSet indexed family:

```
data SymExpr : FinSet → Type₁ where
  id↑ : SymExpr ⊤
  val↑ : ℕ → SymExpr ⊥
  add↑ : SymExpr A → SymExpr B → SymExpr (A ⊎ B)
```

In this example, the type former _⊎_ : FinSet → FinSet → FinSet corresponds to coproducts in the universe of finite sets, i.e. El $(A \uplus B)$ = El $A \uplus$ El B, and $\bot$ is the empty type (the finite set with cardinality 0). The proof that finite sets are indeed closed under finite coproducts can again be found both in our Cubical Agda formalisation and in Voevodsky's unimath library. Notably, SymExpr is a family of large types which is a consequence of indexing on the large type FinSet and the constructor add↑ being parametrised over a large type. Toward the end of this section, we will discuss an alternative encoding of finite sets when defining families of h-sets, that will allow us to instead define SymExpr as a family of small types.

In order to define the operad composition map for the family $\mathsf{SymExpr}$, for all types $A\ B : \mathsf{Type}$ and families $C : A \uplus B \to \mathsf{Type}$, we will make use of the following canonical isomorphism:

$$\begin{aligned}
&\uplus\text{-}\mathsf{distr}\ A\ B\ C :\\
&\qquad (\textstyle\sum[\ a \in A\ ]\ C\ (\mathsf{inl}\ a)) \uplus \textstyle\sum[\ b \in B\ ]\ C\ (\mathsf{inr}\ b) \simeq\\
&\qquad \textstyle\sum[\ x \in A \uplus B\ ]\ C\ x
\end{aligned}$$

where $\uplus\text{-}\mathsf{distr}\ A\ B\ C$ is constructed in the evident way. Moreover, for every families $A : \top \to \mathsf{Type}$ and $B : \bot \to \mathsf{Type}$, we will make use of the following additional two canonical isomorphisms:

$$\begin{aligned}
\textstyle\sum\text{-}\mathsf{idl}\ A &: A\ \mathsf{tt} \simeq \textstyle\sum[\ x \in \top\ ]\ A\ x\\
\textstyle\sum\text{-}\bot\ B &: \bot \simeq \textstyle\sum[\ x \in \bot\ ]\ B\ x
\end{aligned}$$

We can then construct the composition map for a symmetric operad O : $\mathsf{Operad}\ \mathsf{SymExpr}$ as follows:

$$\begin{aligned}
&\mathsf{comp}\ O\ .\top\ B\ \mathsf{id}{\uparrow}\ es\\
&\quad = \mathsf{subst}\ \mathsf{SymExpr}\\
&\qquad\qquad (\mathsf{un}\ (\mathsf{El}\ \ (B\ \mathsf{tt}))\ (\textstyle\sum[\ x \in \top\ ]\ B\ x)\ (\textstyle\sum\text{-}\mathsf{idl}\ B))\\
&\qquad\qquad (es\ \mathsf{tt})\\
&\mathsf{comp}\ O\ .\bot\ B\ (\mathsf{val}{\uparrow}\ n)\ es =\\
&\quad = \mathsf{subst}\ \mathsf{SymExpr}\ (\mathsf{un}\ \bot\ (\textstyle\sum[\ x \in \bot\ ]\ B\ x)\ (\textstyle\sum\text{-}\bot\ B))\ (\mathsf{val}{\uparrow}\ n)\\
&\mathsf{comp}\ O\ .(A_1 \uplus A_2)\ B\ (\mathsf{add}{\uparrow}\ e_1\ e_2)\ es =\\
&\quad \mathsf{let}\ es_1 = \mathsf{comp}\ O\ A_1\ (B \circ \mathsf{inl})\ e_1\ (es \circ \mathsf{inl})\\
&\qquad\quad es_2 = \mathsf{comp}\ O\ A_2\ (B \circ \mathsf{inr})\ e_2\ (es \circ \mathsf{inr})\\
&\quad\ \mathsf{in}\ \mathsf{subst}\ \mathsf{SymExpr}\ (\mathsf{un}\ \_\ \_\ (\uplus\text{-}\mathsf{distr}\ A_1\ A_2\ B))\ (\mathsf{add}{\uparrow}\ es_1\ es_2)
\end{aligned}$$

While we do not provide the proofs here that this construction respects the necessary identity and associativity laws, the details can be found in our Cubical Agda formalisation.

4.4 Small FinSet

As previously highlighted, in contrast to the typical inductive presentation of $\mathbb{N}$, we have presented the universe of finite sets, i.e. FinSet, as a *large* type. Consequently, many of the FinSet-indexed families we will construct, such as SymExpr, will themselves necessarily be families of large types. Moreover, it is also necessary to address this size issue in the definition of a symmetric operad, that will instead need to be defined over large families of types of the form FinSet → Type_1. However, as we will demonstrate in this section, this size issue arises as a consequence of the choice of representation of FinSet. In particular, we will introduce two small equivalent encodings of FinSet, the first as a higher-inductive type and the second using higher-induction recursion.

Intuitively, we can observe that the underlying type of points or *connected components* of FinSet is equivalent to the natural numbers. That is, we can construct an equivalence between the set-truncation of FinSet and the type ℕ, whose explicit construction can be found in our Cubical Agda formalisation. Another way to understand this connection between FinSet and ℕ, is to construct the universe of finite sets by starting with ℕ and 'adding' in the necessary paths corresponding to the permutations of finite sets. Indeed, this is the key insight behind the two small encodings of finite sets presented in this chapter. The first of these encodings is a standard construction that defines the finite sets as a *groupoid quotient* [Sojakova, 2016] of the natural numbers by the transitive relation mapping $m\ n$: ℕ to the h-set of permutations Fin $m \simeq$ Fin n. Concretely, we can define this as a higher-inductive type in Cubical Agda as follows:

```
data FinSet : Type where
  size : ℕ → FinSet
  eq : Fin n ≃ Fin n → size n ≡ size n
  eqr : (p q : Fin n ≃ Fin n) →
          PathP (λ i → size n ≡ eq q i) (eq p) (eq (p · q))
  trunc : isGroupoid FinSet
```

In this definition of FinSet, the data constructor size characterises the points of the groupoid universe of finite sets. We then include a path constructor eq that introduces a path between size n and itself for every permutation of type Fin $n \simeq$ Fin n. However, this structure alone is not sufficient as it does not, for example, satisfy the necessary groupoid law eq (id n) ≡ refl or more generally eq $(p \cdot q)$ ≡ eq p · eq q, where id n : Fin $n \simeq$ Fin n is the identity permutation. In order to obtain the desired groupoid structure on FinSet, with respect to permutations, we could have considered directly adding a constructor of the form eq $(p \cdot q)$ ≡ eq p · eq q. However, in contrast to heterogeneous paths, path composition is not typically a primitive notion in implementations of cubical type theory such as Cubical Agda. Consequently, it can be more practical to work with the provided path constructor eqr from which eq $(p \cdot q) \equiv$ eq $p \cdot$ eq q can be derived from the contractibility of path composition. Finally, we include a groupoid truncation path constructor to remove any higher structure introduced by the eqr constructor.

To complete our alternative encoding of the universe of finite sets, we also require a FinSet-indexed family of types corresponding to the underlying finite types. We can define this family by eliminating into the groupoid of h-sets and mapping each point size n to Fin n. The full details of this construction can be found in our Cubical Agda formalisation together with closure properties such as finite products and coproducts along with dependent sums. Unfortunately, in practice it can be challenging to define eliminators on this definition of finite sets, as a consequence of the added path constructors. In a type theory with higher-induction recursion, we can encode finite sets by applying the technique first introduced in Section 2.8. Concretely, we mutually define a higher-inductive type FinSet : Type together with an interpretation function El : FinSet → Type, where FinSet is given as follows:

```
data FinSet : Type where
  size : ℕ → FinSet
  un : (A B : FinSet) → El A ≃ El B → A ≡ B
```

The function El is then mutually defined as follows:

```
El (size n) = Fin n
El (un A B p i) = ua (El A) (El B) p i
```

where for any types X and Y, the function ua $X\ Y : X \simeq Y \to X \equiv Y$ is the inverse of the canonical map from paths to equivalences pathToEquiv $X\ Y$: $X \equiv Y \to X \simeq Y$, as follows from univalence.

It can be shown that this small higher-inductive recursive representation of FinSet is indeed equivalent to our original encoding as a large type. The construction of this equivalence relies on the observation that for any finite sets $A\ B$: FinSet, the function un $A\ B$: El $A \simeq$ El $B \to A \equiv B$ is provably inverse to the composition of the functorial action of El on paths with the canonical map from paths to equivalences, i.e. the function pathToEquiv (El A) (El B) $\circ$ cong El. Consequently, it can be shown that the function El is in an embedding, i.e. the functorial action of El on paths establishes an equivalence between the path spaces El $A \equiv$ El B and $A \equiv B$. Equivalently, the function El has propositional fibers, which we recall means that for any X : Type the type $\sum$[$A \in$ FinSet] El $A \equiv X$ is an h-proposition. The full details of this proof can be found in our Cubical Agda formalisation. The proof that El has propositional fibers can subsequently be used to construct a family of isomorphisms between the fibers of El and the propositional family isFinite : Type → Type, and we again refer to our formalisation for the details. By contractibility of singletons, the dependent sum taken over the fibers of El is equivalent to FinSet, and therefore we can establish that the higher-inductive recursive encoding FinSet is indeed equivalent to our original large encoding of finite set.

In this chapter, we proceed by using the higher-inductive recursive definition of FinSet of finite sets, which is considerably easier to work with in practice. However, when working in a type theory with higher-inductive types but without higher-induction recursion, the issue of size can instead be address by the first alternative encoding discussed in this section.

4.5 Generalised operad universes

Thus far, we have seen how symmetric and planar operads can be defined in HoTT by varying the universe (à la Tarski) with which we index collections of operations. For the planar case we considered operations indexed by the universe of totally-ordered finite sets ($\mathbb{N}$, Fin) and for the symmetric case, we considered the more general grouopid of finite sets and bijections (FinSet, El). To describe how an operadic composition map acts on indices, we required both of these universes be both closed under dependent sums and have a corresponding unit element. That is, for the planar case we required the finite summation map sum together with the unit 1, while for the symmetric case we instead used closure of finite sets under dependent sums $\hat{\sum}$ together with the singleton finite set $\top$. Indeed, as both the symmetric and planar operad definitions are otherwise uniform, this suggests a generalisation for collections of operations indexed by any universe with sufficient closure properties. In this section, we will introduce an internal notion of such universes in HoTT which will allow us to work with a generalised notion of operads. This generalised formulation of operads will enable us to give an account of many of the important constructions in the theory of operads without requiring separate details for the symmetric and planar variations.

We begin by recalling that a Tarski type universe comprises a type of codes U : Type and an interpretation family $E : U \to \mathsf{Type}$. For example, the groupoid of finite sets and bijections can be presented as a universe with $U = \mathsf{FinSet}$ and $E = \mathsf{El}$. In Agda, we can present the type of such universes as a record type $\mathcal{U}$: $\mathsf{Universe}$ with a field (projection) for the type of codes $\mathsf{Code}\ \mathcal{U} : \mathsf{Type}$ and for every code $A : \mathsf{Code}\ \mathcal{U}$, a field $[\![\ \mathcal{U} \ni A\ ]\!] : \mathsf{Type}$ corresponding to the underlying type. As a first step to capturing only those universes closed under dependent sums, we can introduce the requirement that a universe $\mathcal{U}$ must come equipped with the following type former:

$$\hat{\sum} : (A : \mathsf{Code}\ \mathcal{U}) \to ([\![\ \mathcal{U} \ni A\ ]\!] \to \mathsf{Code}\ \mathcal{U}) \to \mathsf{Code}\ \mathcal{U}$$

such that for every $A : \mathsf{Code}\ \mathcal{U}$ and family $B : [\![\ \mathcal{U} \ni A\ ]\!] \to \mathsf{Code}\ \mathcal{U}$ there is an equivalence

$$[\![\hat{\sum}]\!]\ A\ B : [\![\ \mathcal{U} \ni (\hat{\sum}\ A\ B)\ ]\!] \simeq \sum[\ a \in [\![\ \mathcal{U} \ni A\ ]\!]\ ]\ [\![\ \mathcal{U} \ni B\ a\ ]\!].$$

In addition to closure under dependent sums, we require a field $\hat{\top} : \mathsf{Code}\ \mathcal{U}$ corresponding to the code for the unit type, together with an equivalence $[\![\ \mathcal{U} \ni \hat{\top}\ ]\!] \simeq \top$. However, these properties are not sufficient to show that $\mathcal{U}$ is closed under dependent sums and a unit (terminal) object. For example, for every $A : \mathsf{Code}\ \mathcal{U}$, $B : [\![\ \mathcal{U} \ni A\ ]\!] \to \mathsf{Code}\ \mathcal{U}$ and $C : (a : [\![\ \mathcal{U} \ni A\ ]\!]) \to [\![\ \mathcal{U} \ni B\ a\ ]\!] \to \mathsf{Code}\ \mathcal{U}$, these fields are insufficient to construct a proof of associativity:

$$\begin{gathered}\hat{\textstyle\sum}\text{-}\mathsf{assoc}\ A\ B\ C : \\ \hat{\sum}\ A\ (\lambda\ a \to \hat{\sum}\ (B\ a)\ (C\ a)) \equiv \hat{\sum}\ (\hat{\sum}\ A\ B)\ (\mathsf{uncurry}\ C \circ [\![\hat{\sum}]\!]\ A\ B).\end{gathered}$$

In order to address this, we will require that for any two codes $A\ B : \mathsf{Code}\ \mathcal{U}$ we can lift equivalences between their underlying types to paths between them by means of the following additional field

$$\mathsf{Inject}\ A\ B : [\![\ \mathcal{U} \ni A\ ]\!] \simeq [\![\ \mathcal{U} \ni B\ ]\!] \to A \equiv B$$

Indeed, a map of this form is typically introduced in HoTT to define the notion of being Rezk-complete and consequently univalent categories [Ahrens et al., 2015]. However, being Rezk-complete would require that Inject can be used to prove that the canonical map from the path space $A \equiv B$ to the the type of equivalences $[\![\ \mathcal{U} \ni A\ ]\!] \simeq [\![\ \mathcal{U} \ni B\ ]\!]$ is itself an equivalence, which is too strong for our purposes. For example, this would not allow us to capture the universe of totally ordered finite sets $(\mathbb{N}, \mathsf{Fin})$ that is in turn used to index the operations of planar operads.

While being Rezk-complete is too strong of a property for our purposes, it remains necessary that we can reason about the paths constructed by means of Inject. As such, we will instead consider a weaker but similar property that is suitable to the application of an operadic theory in HoTT. In particular, we recall that the notion of a generalised operad universe will be used to index a family of h-sets corresponding to the operations of an operad. That is, given a universe $\mathcal{U}$: Universe, a $\mathcal{U}$-operad has operations given by a family of h-sets K : Code $\mathcal{U} \to$ Type. Importantly, the type of h-sets is itself an h-groupoid and consequently the higher path structure of codes beyond that of the 1-groupoid structure, i.e. paths between paths, is trivially respected by K. Consequently, we need only care about the 1-groupoid structure on the equivalences between the underlying types of codes in the universe $\mathcal{U}$.

Hence, it is sufficient to include a coherence condition witnessing that Inject preserves this 1-groupoid structure. For all codes $A\ B\ C$: Code $\mathcal{U}$ and equivalences $e_1 : [\![\ \mathcal{U} \ni A\ ]\!] \simeq [\![\ \mathcal{U} \ni B\ ]\!]$ and $e_2 : [\![\ \mathcal{U} \ni B\ ]\!] \simeq [\![\ \mathcal{U} \ni C\ ]\!]$, we can achieve this by adding the field:

$$\begin{gathered}\mathsf{InjectComp}\ A\ B\ C\ e_1\ e_2 : \\ \mathsf{Inject}\ A\ C\ (e_1 \cdot e_2) \equiv \mathsf{Inject}\ A\ B\ e_1 \cdot \mathsf{Inject}\ B\ C\ e_2\end{gathered}$$

Intuitively, the family of paths InjectComp asserts that Inject must preserve composition of equivalences, i.e. by mapping composition of equivalences to the composition of paths. Moreover, this condition is sufficient to witness that up to a path Inject maps identity equivalences to reflection paths.

Concretely, this means that for every code A : Code we can construct the following path:

$$\mathsf{InjectRefl}\ A : \mathsf{Inject}\ A\ A\ (\mathsf{id}{\simeq}\ [\![\ \mathcal{U} \ni A\ ]\!]) \equiv \mathsf{refl},$$

where $\mathsf{id}{\simeq}\ [\![\ \mathcal{U} \ni A\ ]\!] : [\![\ \mathcal{U} \ni A\ ]\!] \simeq [\![\ \mathcal{U} \ni A\ ]\!]$ is the identity equivalence on $[\![\ \mathcal{U} \ni A\ ]\!]$.

To construct the path $\mathsf{InjectRefl}$, we first apply $\mathsf{InjectComp}\ A\ A\ A$ to the identity equivalence (twice) on $[\![\ \mathcal{U} \ni A\ ]\!]$ and precompose with the functorial action of $\mathsf{Inject}\ A\ A$ on paths applied to the path witnessing that the identity equivalence composed with itself is the identity equivalence. With this composition, we have constructed a path between $\mathsf{Inject}\ A\ A\ (\mathsf{id}{\simeq}\ [\![\ \mathcal{U} \ni A\ ]\!])$ and the same path composed with itself. It follows from the groupoid structure of paths that for any type A, term $x : A$, and path $p : x \equiv x$, if there is a path of type $p \cdot p \equiv p$ then we can construct a path of type $p \equiv \mathsf{refl}$. Consequently, by applying this rule we can construct a path between $\mathsf{Inject}\ A\ A\ (\mathsf{id}{\simeq}\ [\![\ \mathcal{U} \ni A\ ]\!])$ and refl. Furthermore, for any types $X\ Y$: Type if we let $\mathsf{pathToEquiv}\ X\ Y : X \equiv Y \to X \simeq Y$ be the canonical map from paths to equivalences then by applying the J-elimination rule to the composition of the path witnessing that $\mathsf{pathToEquiv}$ maps reflection to the identity equivalence and the path $\mathsf{InjectRefl}\ A$, we can construct the following family of paths:

$$\begin{gathered}\mathsf{InjectSec}\ A\ B : \\ (p : A \equiv B) \to \mathsf{Inject}\ A\ B\ (\mathsf{pathToEquiv}\ (\mathsf{cong}\ [\![\ \mathcal{U} \ni \_]\!]\ p)) \equiv p\end{gathered}$$

In particular, $\mathsf{InjectSec}$ witnesses that Inject has all sections given by the composition of the functorial action of the interpretation map $[\![\ \mathcal{U} \ni \_]\!]$ on paths and the canonical map from paths to equivalences. Intuitively, this condition asserts that the path spaces of the representing codes of a universe reflect the path spaces of the underlying types and no more.

Notably, the family of paths $\mathsf{InjectComp}$ only witness sufficient coherencies to ensure that 1-groupoid structure is preserved by Inject. That is, we do not include additional coherencies to witness preservation of higher paths beyond the 1-groupoid structure. This would suggest we should also require that the type of codes $\mathsf{Code}\ \mathcal{U}$ be an h-groupoid and that for every code A : $\mathsf{Code}\ \mathcal{U}$ its interpretation $[\![\ \mathcal{U} \ni A\ ]\!]$ be an h-set. Indeed, if our intention was to reason covariantly about such universes then these would be necessary conditions. However, for our particular application to operads the codes of a universe are only used to index the groupoid of h-sets and their interpretations are only used to index the h-sets themselves. It is a standard result in HoTT that for any type X : Type, concrete h-level n and n-type Y that there is an equivalence between the types $X \to Y$ and $\| X \|_n \to$. Consequently, our notion of an operad that omits the requirement that $\mathsf{Code}\ \mathcal{U}$ is an h-groupoid and that $[\![\mathcal{U} \ni A]\!]$ is an h-set is equivalent to the definition where these witnesses are present.

To conclude our description of generalised operad universes, we first recall that the definition of an operadic structure on a collection of operations includes heterogeneous paths that witness that the composition map respects left and right identity laws along with an associativity law. In particular, for generalised operad universes these paths will necessarily be heterogeneous over Inject applied to the following canonical equivalences:

$$\hat{\sum}\mathsf{Idl}{\simeq}\ \mathcal{U}\ A : [\![\mathcal{U} \ni \hat{\sum}\ \hat{\top}\ (\lambda\ t \to A)]\!] \simeq [\![\mathcal{U} \ni A]\!] \qquad \text{(left identity)}$$

$$\hat{\sum}\mathsf{Idr}{\simeq}\ \mathcal{U}\ A : [\![\mathcal{U} \ni \hat{\sum}\ A\ (\lambda\ a \to \hat{\top})]\!] \simeq [\![\mathcal{U} \ni A]\!] \qquad \text{(right identity)}$$

$$\begin{aligned} &\hat{\sum}\mathsf{Assoc}{\simeq}\ \mathcal{U}\ A\ B\ C : && \text{(associativity)}\\ &\qquad [\![\mathcal{U} \ni \hat{\sum}\ A\ (\lambda\ a \to \hat{\sum}\ (B\ a)\ (C\ a))]\!] \simeq \\ &\qquad [\![\mathcal{U} \ni \hat{\sum}\ (\hat{\sum}\ A\ B)\ (\mathsf{uncurry}\ C \circ \mathsf{fun}\ ([\![\hat{\sum}]\!]\ A\ B)) \end{aligned}$$

for all A : $\mathsf{Code}\ \mathcal{U}$, B : $[\![\mathcal{U} \ni A]\!] \to \mathsf{Code}\ \mathcal{U}$ and C : $(a : [\![\mathcal{U} \ni A]\!]) \to [\![\mathcal{U} \ni B\ a]\!] \to \mathsf{Code}\ \mathcal{U}$. In particular, these are precisely the equivalences that are constructed from the underlying canonical equivalences on the dependent sum construction in the meta-theory appropriately composed with equivalences constructed from application of $[\![\hat{\sum}]\!]$. In order to construct $\mathcal{U}$-operadic structures, we require that each of the above equivalences are represented by the paths they are sent to by Inject. Concretely, this means that we require paths between the above equivalences and the application of the map that sends each $X\ Y$: $\mathsf{Code}\ \mathcal{U}$ and e : $[\![\mathcal{U} \ni A]\!] \simeq [\![\mathcal{U} \ni A]\!]$ to $\mathsf{pathToEquiv}$ (cong $[\![\mathcal{U} \ni\_]\!]$ (Inject X Y e)) to the same equivalences. Equivalently, given the inverse map to $\mathsf{pathToEquiv}$ following from univalence, i.e. ua : $X \simeq Y \to X \equiv Y$, we can capture the required conditions with the following fields:

$$\llbracket \hat{\sum}\mathsf{Idl} \rrbracket\ \mathcal{U}\ A :$$
$$\mathsf{ua}\ (\hat{\sum}\mathsf{Idl}{\simeq}\ \mathcal{U}\ A) \equiv \mathsf{cong}\ \llbracket\ \mathcal{U} \ni\_\rrbracket\ (\mathsf{Inject}\ \_\ \_\ (\hat{\sum}\mathsf{Idl}{\simeq}\ \mathcal{U}\ A))$$

$$\llbracket \hat{\sum}\mathsf{Idr} \rrbracket\ \mathcal{U}\ A :$$
$$\mathsf{ua}\ (\hat{\sum}\mathsf{Idr}{\simeq}\ \mathcal{U}\ A) \equiv \mathsf{cong}\ \llbracket\ \mathcal{U} \ni\_\rrbracket\ (\mathsf{Inject}\ \_\ \_\ (\hat{\sum}\mathsf{Idr}{\simeq}\ \mathcal{U}\ A))$$

$$\llbracket \hat{\sum}\mathsf{Assoc} \rrbracket\ \mathcal{U}\ A\ B\ C :$$
$$\mathsf{ua}\ (\hat{\sum}\mathsf{Assoc}{\simeq}\ \mathcal{U}\ A\ B\ C) \equiv$$
$$\mathsf{cong}\ \llbracket\ \mathcal{U} \ni\_\rrbracket\ (\mathsf{Inject}\ \_\ \_\ (\hat{\sum}\mathsf{Assoc}{\simeq}\ \mathcal{U}\ A\ B\ C))$$

While we do not fill in all of the details here, both the universe of totally-ordered finite sets and the groupoid of finite sets and bijections satisfy all of the outlined criteria. Another evident example of a universe for which all of the above terms can be constructed, given careful consideration of size issues, is the universe Type itself, with the trivial implementation. Similarly, for any homotopy-level $n \geq -1$ (i.e. propositions or higher), the universe of n-types also satisfies these criteria.

From this characterisation of universes closed under both dependent sums and a unit type, our internalisation of operads can be generalised by parametrising over these universes. That is, for every such universe $\mathcal{U}$: Universe and for every collection of operations K : Code $\mathcal{U} \to$ Type, where K is a family of h-sets, we can internalise the notion of a generalised operadic structure on K as a term of a parametrised record type Operad $\mathcal{U}$ K : Type. Given such an operadic structure O : Operad $\mathcal{U}$ K, its fields are almost identical to those detailed in Sections 4 and 5, with finite summation or dependent sum over finite sets replaced by $\hat{\sum}$. For example, for every A : Code $\mathcal{U}$ and B : $\llbracket\ \mathcal{U} \ni A\ \rrbracket \to$ Code $\mathcal{U}$ the composition map for the operad O is given by the following field:

$$\mathsf{comp}\ O\ A\ B : K\ A \to ((a : \llbracket\ \mathcal{U} \ni A\ \rrbracket) \to K\ (B\ a)) \to K\ (\hat{\sum}\ A\ B)$$

Similarly, the unit operation is given by a field id O : $K\ \hat{\top}$. The families of heterogeneous paths witnessing the left and right unit laws along with the associativity law can be defined by appropriately applying Inject to the respective laws on dependent sums. For example, for every A : Code $\mathcal{U}$ and $k : K\ A$ the corresponding left unit law for O is witnessed by the following field:

```
idl O A k :
    PathP (λ i → K (Inject (∑̂ ⊤̂ (λ u → A)) A (∑Idl ⟦ 𝓤 ∋ A ⟧) i))
          (comp O ⊤̂ (λ u → A) (id O) (λ u → k)) k
```

To conclude our discussion on generalised operads, we describe the groupoid structure on $\mathcal{U}$-operads, for any appropriate universe $\mathcal{U}$: Universe, which arises from the path space on the type of $\mathcal{U}$-operads. In HoTT, given any type A and family of h-propositions $P : A \to$ Type, the sum type $\sum[\ a \in A\ ]\ P\ a$ can be considered a *subtype* of A as witnessed by the first projection $\pi_1 : \sum[\ a \in A\ ]\ P\ a \to A$ being an embedding. Concretely, this means that the functorial action of the first projection map on paths, i.e. cong $\pi_1 : x \equiv y \to \pi_1\ x \equiv \pi_1\ y$, is an equivalence. That is, the path spaces of a subtype are entirely characterised by the path spaces of its underlying type. Given a universe $\mathcal{U}$: Universe and a collection of operations $K : \mathcal{U} \to$ Type, where K is a family of h-sets and is equipped with an operad structure O : Operad $\mathcal{U}\ K$, it follows that the types of each of the paths idl $O\ n\ k$, idr $O\ n\ k$ and assoc $O\ n\ ns\ nss\ k\ ks\ kss$ are h-propositions. Consequently, an operad can be understood as a subtype of a record type with only an identity id and composition map comp. That is, the operad laws are not important in characterising the path spaces between operads, and we make use of this fact to simplify the following definition of these path spaces.

Given any two collections of operations $K\ L : \mathcal{U} \to$ Type, where both K and L are families of h-sets and come equipped with operad structures O^K : Operad $\mathcal{U}\ K$ and O^L : Operad $\mathcal{U}\ L$, and given any family of paths $K{\equiv}L$: (A : Code $\mathcal{U}$) $\to K\ A \equiv L\ A$, the heterogeneous path space between O^K and O^L is equivalent to the following record type:

```
record Path^O K≡L O^K O^L : Type where
  field
    id≡ : PathP (λ i → K≡L ⊤̂ i) (id O^K) (id O^L)
    comp≡ :  (A : Code 𝓤)  (B : ⟦ 𝓤 ∋ A ⟧ → Code 𝓤) →
```

$$\begin{aligned}&\mathsf{PathP}\ (\lambda\ i \to (k :\ K{\equiv}L\ A\ i)\\ &\qquad\qquad (ks : (a : [\![\ \mathcal{U} \ni A\ ]\!]) \to K{\equiv}L\ (B\ a)\ i)\\ &\qquad\qquad K{\equiv}L\ (\hat{\textstyle\sum}\ A\ B)\ i)\\ &\qquad (\mathsf{comp}\ O^K\ A\ B)\ (\mathsf{comp}\ O^L\ A\ B)\end{aligned}$$

Importantly, it follows from K and L being families of h-sets that the type $\mathsf{Path}^O\ K{\equiv}L\ O^K\ O^L$ is an h-proposition. That is, the path space between two operads is an h-proposition. Notably, we do not require that this definition explicitly encode paths between the proofs of unitality and associativity of composition in the operads O^K and O^L. In particular, it is a standard result that the type of these paths is contractible as a consequence of the unitality and associativity proofs being h-propositions.

The heterogeneous path space between two operads presents a groupoid structure on the (large) type $\sum[\ K \in\ (\mathsf{Code}\ \mathcal{U} \to \mathsf{hSet})\ ]\ \mathsf{Operad}\ \mathcal{U}\ K$ of all $\mathcal{U}$-operads. In particular, the hom-sets of this groupoid are characterised by the path spaces of this type, i.e. for any two terms

$$(K\ ,\ O^K)\ \ (L\ ,\ O^L) : \textstyle\sum[\ K \in\ (\mathsf{Code}\ \mathcal{U} \to \mathsf{hSet})\ ]\ \mathsf{Operad}\ K,$$

their hom-set is given by the path space $(K\ ,\ O^K) \equiv (L\ ,\ O^L)$, or equivalently by the h-set $\sum[\ p \in\ ((A : \mathsf{Code}\ \mathcal{U}) \to K\ \ A \equiv L\ \ A)\ ]\ \mathsf{Path}^O\ p\ O^K\ O^K$. The full groupoid structure of this construction follows from the group structure on path spaces.

We conclude this section with a brief discussion on the utility of the generalised operad construction that we have introduced. In particular, by developing a generalised formulation of operads, we can introduce further key ideas in the theory of operads by parametrising over the notion of a generalised operad universe where possible. This will allow us to consider and define concepts such as operad morphisms as well as algebras and monads over an operad, each of which are uniform over the universe, without specialising for both the planar and symmetric cases. Indeed, as we will discuss in a later section of this chapter, even the free operad construction can be parametrised over the universe of codes. This highlights one of the key benefits to studying operads in HoTT, namely that the functoriality of type families with respect to paths allows us to present a generalised notion of an operad which in turn allows us to give uniform definitions for many typical constructions in the theory of operads. For example, the *associa-*

tive, *commutative* and endomorphism operads can all be defined uniformly over generalised operad universes, such that specialising to the universe of totally ordered finite sets or Bishop-finite sets gives the expected variations as planar and symmetric operads respectively. Indeed, these examples are found in our accompanying Cubical Agda library.

In the following sections of this chapter, we develop our theory of operads in HoTT uniformly over the indexing universe. Notably, this demonstrates that many of the standard operadic constructions such as the endomorphism operad (Section 4.6), the monad over an operad (Section 4.7) and the free operad (Section 4.8) do not require that we constrain ourselves to collections of finitary operations. Indeed, finiteness is only necessary for operads whose *operation data* is constructed by induction over the natural numbers. For example, the operations of a symmetric operad K : FinSet → hSet can equivalently be given by a countable family f : ℕ → hSet together with permutations σ_f : Fin $n \simeq$ Fin $n \to f\ n \simeq f\ n$ and a family of paths witnessing that σ_f preserves groupoid structure c_f : $(p\ q$: Fin $n \simeq$ Fin $n) \to \sigma_f\ (p \cdot q) \equiv \sigma_f\ p \cdot \sigma_f\ q$. In particular, the family of h-sets f corresponds to the data of the operations defined by a symmetric operad and can induct on its natural input to build operations.

4.6 Category of operads

In order to study and work with operads in HoTT, we require notions for both structure-preserving interactions between them and concrete interpretations of the abstract collection of composable operations that operads represent. In particular, this corresponds to operad homomorphisms and operad algebras respectively. A concrete example of both an operad homomorphism and algebra is that of the fill function described in Section 4, which can be understood as a morphism from the PartialList operad to the *endomorphism operad* on lists. In this section, we will begin by presenting the categorical structure of operads within HoTT. We will then introduce the notion of an endomorphism operad on a given type, whereby morphisms in the category of operads into the endomorphism operad constitute the notion of an operad algebra.

In addition to the core groupoid structure described in the previous section, the (large) type ∑[K ∈ (Code $\mathcal{U}$ → hSet)] Operad $\mathcal{U}$ K comes equipped with a more general category structure wherein a morphism between operads is any family of maps between their operations that preserves both compositional structure and the identity operation. In particular, given two families of h-sets $K\ L$: Code 𝓤 → Type we define the corresponding family of hom-sets between 𝓤-operads by:

```
record _⇒_ (Oᴷ : Operad 𝓤 K) (Oᴸ : Operad 𝓤 L) : Type where
  field
    ⟨_⟩ : (A : Code 𝓤) → K A → L A
    id-resp : ⟨_⟩ ⊤̂ (id Oᴷ) ≡ id Oᴸ
    comp-resp :
      (A : Code 𝓤)  (B : ⟦ 𝓤 ∋ A ⟧ → Code 𝓤)
      (k : K A) (ks : (a : ⟦ 𝓤 ∋ A ⟧) → K (B a)) →
      ⟨_⟩ (Σ̂ A B) (comp Oᴷ A B k ks) ≡
       comp Oᴸ A B (⟨_⟩ A k) (λ a → ⟨_⟩ (B a) (ks a))
```

In a similar fashion to that of the path space between two operads, this type can be seen as a subtype of the type of slice morphisms between K and L, i.e. the type $(A :$ Code 𝓤$) \to K\ A \to L\ A$. That is, the types of the fields id-resp and comp-resp are h-propositions and consequently the path space between any two morphisms of operads is equivalent to the path space of the action of these morphisms on the underlying operations. Intuitively, two operad morphisms are equal (up to a path) if they act identically on operations. Concretely, this means that given two morphisms $f\ g : O^K \Rightarrow O^L$, the function cong ⟨_⟩ : $f \equiv g \to$ ⟨ f ⟩ $\equiv$ ⟨ g ⟩ is an equivalence.

Given a third family of h-sets J : Code 𝓤 → Type with an operadic structure O^J : Operad 𝓤 J, then for any two operad morphisms $f : O^J \Rightarrow O^K$ and $g : O^K \Rightarrow O^L$ we can construct their composition $g \bullet f : O^J \Rightarrow O^K$ as follows:

```
⟨ g • f ⟩ A k = ⟨  g ⟩ A (⟨ f ⟩ A k)
id-resp (g • f ) = cong (⟨ g ⟩ ⊤̂) (id-resp f ) · id-resp g
comp-resp (g • f ) A B k ks
  = cong (⟨ g ⟩ (Σ̂ A B))
          (comp-resp f A B k ks) ·
           comp-resp g A B (⟨ f ⟩ A k) (λ a → ⟨ f ⟩ (B a) (ks a))
```

The identity operation $\mathsf{id}^{op}\ O : O \Rightarrow O$ on an operad O : Operad 𝓤 K

is then given by defining $\langle\ O\ \rangle\ n\ k = k$, with both the path id-resp and the family of paths comp-resp holding definitionally. In order to prove that the composition of operad morphisms is associative and unital, we recall that the path space of operad morphisms is equivalent to the path space of their underlying operations. Consequently, the associativity and identity laws trivially follow from the associativity and unitality of composition of morphisms of slices, i.e. indexed function composition.

An example of a planar operad morphism can be seen by considering how partial lists of natural numbers can be interpreted as finite operations on Expr. Concretely, we start by constructing a family of functions build : $(n : \mathbb{N}) \to$ PartialList $\mathbb{N}$ $n \to$ IExpr n as follows:

```
build 0 [] = val↑ 0
build n (k :: ks) = add↑ (val↑ k) (build n ks)
build 1 (poke []) = id↑
build (suc n) (poke (k :: ks)) = add↑ id↑ (add↑ (val↑ k) (build n ks))
build (suc (suc n)) (poke (poke ks))
  = add↑ id↑ (add↑ id↑ (build n ks))
```

We can understand the function build as interpreting partial lists of natural numbers as a finite sum expression, where holes correspond to variables. Indeed, the build operad morphism highlights a useful intuition for understanding the partial list operad; as an abstract characterisation of linear expressions with a finite number of variables or ‘holes’ that are context independent. By recalling that poke [] and id↑ are the identity operations for the PartialList and IExpr operads respectively, we can observe that build definitionally respects the identity operation, and the proof that it also respects the composition map is provided in our Cubical Agda formalisation.

We have already been introduced to two further examples of planar operad morphisms in Section 4, namely the functions ⟦_⟧ : IExpr $n \to$ (Fin $n \to$ Expr) $\to$ Expr and fill : PartialList A $n \to$ (Fin $n \to$ List A) $\to$ List A. Both of these operation maps can be shown to be operad morphisms into a particular instance of an *endomorphism operad* which is an operad whose operations are ‘n-ary like’ functions on a given h-set and whose composition map is given by function composition. Concretely, given any universe $\mathcal{U}$: Universe and any h-set X : Type we begin by defining a collection of operations EndoOps : Code $\mathcal{U} \to$ Type that maps each A : Code $\mathcal{U}$ to $(⟦\ \mathcal{U} \ni A\ ⟧ \to X) \to X$. Notably, when specialising to the universe of totally-ordered finite sets, i.e. planar operads, this precisely corresponds to the family of n-ary functions on X. Given that X must be a h-set, it follows that the collection of operations EndoOps is a collection of h-sets. We denote the endomorphism $\mathcal{U}$-operad on X by Endo $\mathcal{U}$ X : Operad $\mathcal{U}$ EndoOps and its unit operation id (Endo $\mathcal{U}$ X) : ($\llbracket$ $\mathcal{U}$ $\ni$ $\hat{\top}$ $\rrbracket$ → X) → X is given as follows:

```
id (Endo 𝓤 X) f = f (inv (⟦⊤̂⟧ 𝓤) tt)
```

Intuitively, the unit operation extracts the only element from a unitary collection f : $\llbracket$ $\mathcal{U}$ $\ni$ $\hat{\top}$ $\rrbracket$ → X. For every A : Code $\mathcal{U}$, B : $\llbracket$ $\mathcal{U}$ $\ni$ A $\rrbracket$ → Code $\mathcal{U}$ together with an output operation f : ($\llbracket$ $\mathcal{U}$ $\ni$ A $\rrbracket$ → X) → X and input operations fs : (a : $\llbracket$ $\mathcal{U}$ $\ni$ A $\rrbracket$) → ($\llbracket$ $\mathcal{U}$ $\ni$ B a $\rrbracket$ → X) → X, we define the composition map of the endomorphism operad on X as follows:

```
comp (Endo 𝓤 X) A B f fs : (⟦ 𝓤 ∋ Σ̂ A B ⟧ → X) → X
comp (Endo 𝓤 X) A B f fs xs
  = f (λ a → fs a (λ b → xs (inv (⟦Σ̂⟧ 𝓤 A B) (a , b))))
```

In order to construct proofs of the identity and associativity laws for the composition map of the endomorphism operad, we first give an account of how the univalence map from equivalences to paths, i.e. ua : $X \simeq Y \to X \equiv Y$, acts when appearing twice to the left of the function arrow. Concretely, for any types X_1 X_2 Y Z : Type together with an equivalence $e : X_1 \simeq X_2$ and a function $f : (X_1 \to Y) \to Z$ the action of ua $e : X_1 \equiv X_2$ on f is witnessed by a heterogeneous path:

```
ua→→ f e : PathP (λ i → (ua e i → Y) → Z) f (λ ys → f (ys ∘ fun e))
ua→→ f e i ys = f (ys ∘ ua-gluePt e i)
```

where ua-gluePt e i : A → ua e i is a path from the identity function to fun e and is a standard result that can be found in the Cubical Agda standard library. While we do not give the explicit constructions here, the proofs that the composition map of the endomorphism operad respects the identity and associativity laws proceeds by application of ua→→ to the equivalences $\hat{\Sigma}$Idl≃ A, $\hat{\Sigma}$Idr≃ A and $\hat{\Sigma}$Assoc≃ A B C followed by substitution along the paths $\llbracket\hat{\Sigma}$Idl$\rrbracket$, $\llbracket\hat{\Sigma}$Idr$\rrbracket$ and $\llbracket\hat{\Sigma}$Assoc$\rrbracket$ respectively. The full details can be found in our Cubical Agda formalisation.

4.7 Monad over an operad

To construct and compute with operadic collections of operations, it is often useful to dependently build operations from data. Indeed, this is precisely the behaviour of the canonical monad that arises over any operad. In particular, an element of the monad over an operad is an operation together with data stored at each of its inputs. Concretely, for any universe $\mathcal{U}$: Universe and collection of operations K : Code $\mathcal{U}$ → Type, where K is a family of h-sets, the monad over any $\mathcal{U}$-operad can be internalised in Cubical Agda by first introducing the following record type:

```
record OpM (O : Operad 𝓤 K) (X : Type) : Type where
  constructor _▷_▷_
  field
    Index : Code 𝓤
    Op : K Index
    Data : ⟦ 𝓤 ∋ Index ⟧ → X
```

Intuitively, Op corresponds to an abstract operation whose type of inputs is represented by the field Index. The Data field corresponds to the data elements that are stored at each input of this operation.

When the type of codes for the universe $\mathcal{U}$ is an h-groupoid, then for any operad O : Operad $\mathcal{U}$ K the family of types OpM O : Type → Type can be shown to be equipped with a (strict) 2-monadic structure on the 2-category of h-groupoids where 1-morphisms are simply functions and 2-morphisms are paths between such functions. In particular, this monadic structure is simply referred to as the monad over the operad O. The unit map for this monad corresponds to storing an element at the output of the input operation, and can be defined in Cubical Agda as follows:

```
return : X → OpM O X
return x = ⊤̂ 𝓤 ▷ unit O ▷ λ t → x
```

The composition map for the monad over an operad intuitively describes how given an output operation with the data stored at its leaves being input operations with their own further attached data, we can use the operadic composition map to compose the output operation with the input operations and retain the data attached to the inputs. In order to formalise this monadic structure, we begin by introducing an dependent variation of the usual monadic bind. In particular, we will index the kleisli morphism of the monadic bind function over each of the inputs provided to the output operation. Concretely, given a term ox : OpM O X, a function f : ⟦ $\mathcal{U}$ ∋ Index ox ⟧ → X → OpM O Y, and the following family of inputs

```
In : ⟦ 𝓤 ∋ Index ox ⟧ → OpM O Y
In i = f i (Data ox i)
```

we can construct our indexed monadic bind function as follows:

```
compM ox f : OpM O Y
Index (compM ox f) = Σ̂ 𝓤 (Index ox) (Index ∘ In)
Op (compM ox f)
  = comp O (Index ox) (Index ∘ In) (Op ox) (Op ∘ In)
Data (compM ox f) k
  = let (i , j) = fun (⟦Σ̂⟧ 𝓤 (Index ox) (Index ∘ In)) k
     in Data (In i) j
```

From this definition of compM, we can then define the usual monad multiplication map as follows:

```
join : OpM (OpM O X) → OpM O X
join o = compM o (λ i x → x)
```

Although it is similarly possible to define the functorial map _<\$>_ : $(X \to Y) \to$ OpM O X → OpM O Y by mapping each $f : X \to Y$ and o : OpM O X to compM o (λ i → return ∘ f), the resulting term is more involved than the one obtained by individually defining _<\$>_ in the obvious way. For example, by defining _<\$>_ in terms of compM the index of f <\$> o is given by $\hat{\sum}$ $\mathcal{U}$ (Index o) (λ i → $\hat{\top}$ $\mathcal{U}$) while using the more direct definition the index is instead simply Index o. By using the more direct definition of the functorial map, the usual 2-functorial laws hold definitionally, as required. Moreover, the proofs that return and join are strict 2-natural transformations similarly hold by definition.

To describe the 2-monadic structure of OpM O, we also require proofs of the usual coherence laws. We begin by giving an equivalent characterisation of the path space on OpM O X for any X : Type, which we will use to construct the necessary witnesses of the coherence laws. In particular, for every x y : OpM O X, the path space between x and y is trivially equivalent to the type of triples consisting of a path p : $\mathsf{Index}\ x \equiv \mathsf{Index}\ y$ between the indices, together with a heterogeneous path $\mathsf{PathP}\ (\lambda\ i \to K\ (p\ i))\ (\mathsf{Op}\ x)\ (\mathsf{Op}\ y)$ between operations and a heterogeneous path $\mathsf{PathP}\ (\lambda\ i \to [\![\ \mathcal{U} \ni p\ i\ ]\!] \to X)\ (\mathsf{Data}\ x)\ (\mathsf{Data}\ y)$ between the data.

By making use of the Rezk-complete structure of the universe $\mathcal{U}$, we can define an equivalent and more practical characterisation of the path space over $\mathsf{OpM}\ O\ X$. In particular, it is often the case that the path between indices, i.e. p, is necessarily constructed by applying Inject to lift an equivalence between types to a path between their corresponding codes. As such, we can alternatively represent the path space between two terms $x, y : \mathsf{OpM}\ O\ X$ by the type of triples consisting of an equivalence between indices

$$e : [\![\ \mathcal{U} \ni \mathsf{Index}\ x\ ]\!] \simeq [\![\ \mathcal{U} \ni \mathsf{Index}\ y\ ]\!]$$

a heterogeneous path between operations of type

$$\mathsf{PathP}\ (\lambda\ i \to K\ (\mathsf{Inject}\ (\mathsf{Index}\ x)\ (\mathsf{Index}\ y)\ e\ i))\ (\mathsf{Op}\ x)\ (\mathsf{Op}\ y)$$

and a family of paths between the data of type

$$(i : [\![\ \mathcal{U} \ni \mathsf{Index}\ x\ ]\!]) \to \mathsf{Data}\ x\ i \equiv \mathsf{Data}\ y\ (\mathsf{fun}\ e\ i)$$

Importantly, the proof that this is equivalent to our original characterisation of the path space over $\mathsf{OpM}\ O\ X$ relies on a known characterisation of the heterogeneous path space over a function type whose domain ranges over a path constructed by application of univalence. Concretely, given any types $X\ Y$: Type, functions $f : X \to Z$, $g : Y \to Z$ and equivalence $e : X \simeq Y$ there is an equivalence between the heterogeneous path space $\mathsf{PathP}\ (\lambda\ i \to \mathsf{ua}\ e\ i \to X)\ f\ g$ and families of paths $(x : X) \to f\ x \equiv g\ (\mathsf{fun}\ e\ x)$. The details of this equivalence can be found in the Cubical Agda standard library.

We proceed by giving a general overview for the constructions of the paths witnessing the necessary 2-monadic laws.

Identity. For all $x : \mathsf{OpM}\ O\ X$, it is necessary to prove that the following two identity laws hold:

$$\mathsf{join\text{-}return}_1\ x : \mathsf{join}\ (\mathsf{return}\ x) \equiv x,$$
$$\mathsf{join\text{-}return}_2\ x : \mathsf{join}\ (\mathsf{return}\ \mathsf{<\$>}\ x) \equiv x$$

We begin by constructing the following equivalences between indices:

$$\hat{\sum}\mathsf{Idl}{\simeq}\ \mathcal{U}\ (\mathsf{Index}\ x) : \hat{\sum}\ \mathcal{U}\ (\hat{\top}\ \mathcal{U})\ (\lambda\ i \to \mathsf{Index}\ x) \simeq \mathsf{Index}\ x$$
$$\hat{\sum}\mathsf{Idr}{\simeq}\ \mathcal{U}\ (\mathsf{Index}\ x) : \hat{\sum}\ \mathcal{U}\ (\mathsf{Index}\ \mathcal{U})\ (\lambda\ i \to \hat{\top}\ \mathcal{U}) \simeq \mathsf{Index}\ x$$

The corresponding heterogeneous paths between operations are given by $\mathsf{idl}\ O\ (\mathsf{Index}\ x)\ (\mathsf{Op}\ x)$ and $\mathsf{idr}\ O\ (\mathsf{Index}\ x)\ (\mathsf{Op}\ x)$. Finally, the family of paths required between data for each of $\mathsf{join\text{-}return}_1$ and $\mathsf{join\text{-}return}_2$ holds by definition, i.e. is a family of refl paths.

Associativity. For all $x : \mathsf{OpM}\ O\ (\mathsf{OpM}\ O\ (\mathsf{OpM}\ O\ X))$, it is necessary to prove that the following associativity condition holds:

$$\mathsf{join\text{-}assoc} : \mathsf{join}\ (\mathsf{join}\ y) \equiv \mathsf{join}\ (\mathsf{join}\ \mathsf{<\$>}\ y)$$

As before, we first construct the following equivalence between indices:

$$\hat{\sum}\mathsf{Assoc}\ \mathcal{U}\ (\mathsf{Index}\ y)\ (\mathsf{Index} \circ \mathsf{Data}\ y)\ (\lambda\ a \to \mathsf{Index} \circ \mathsf{Data}\ (\mathsf{Data}\ y\ a))$$

The corresponding path between operations is then given as follows:

$$\mathsf{assoc}\ O\ (\mathsf{Op}\ y)\ (\mathsf{Op} \circ \mathsf{Data}\ y)\ (\lambda\ a \to \mathsf{Op} \circ \mathsf{Data}\ (\mathsf{Data}\ y\ a))$$

Finally, the family of paths between data once again holds definitionally.

For any operad $O : \mathsf{Operad}\ \mathcal{U}\ K$, the algebras over the strict 2-monad $\mathsf{OpM}\ O$ or simply *the algebras over* O describe how the abstract collection of operations represented by O can be interpreted as concrete operations on a carrier type. Concretely, an operad algebra over O is given by a carrier h-set $A : \mathsf{Type}$ together with an operad morphism $\alpha : O \Rightarrow \mathsf{Endo}\ \mathcal{U}\ A$. Examples of operad algebras include both the function $[\![\_]\!] : \mathsf{IExpr}\ n \to (\mathsf{Fin}\ n \to \mathsf{Expr}) \to \mathsf{Expr}$ with carrier Expr and the function $\mathsf{fill} : \mathsf{PartialList}\ A\ n \to (\mathsf{Fin}\ n \to \mathsf{List}\ A) \to \mathsf{List}\ A$ with carrier $\mathsf{List}\ A$. Moreover, this notion of an operad algebra gives rise to the following function:

```
runAlg : (O ⇒ Endo 𝓤 A) → OpM O A → A
runAlg α o = ⟨ α ⟩ (Index x) (Op x) (Data x)
```

To conclude our discussion on the monads over an operad, we provide a concrete example of using the monad over the partial list operad in order to select . We begin by letting $\hat{\mathbb{N}}$: Universe be the generalised operad universe of totally ordered finite sets, i.e. Code $\hat{\mathbb{N}} = \mathbb{N}$ and $[\![\, \hat{\mathbb{N}} \ni n \,]\!]$ = Fin n, and we let PList A : Operad $\hat{\mathbb{N}}$ (PartialList A) be the partial list operad on an h-set A. We then observe that we can monadic liftings of the _::_ and poke constructors. In particular, for every h-groupoid X : Type we construct the monadic lifting of _::_ as follows:

```
consM : A → OpM (PartialList A) X → OpM (PartialList A) X
consM a o = Index o ▷ (a :: Op o) ▷ Data o
```

We similarly define the monadic lifting of poke as follows:

```
pokeM : X → OpM (PartialList A) X → OpM (PartialList A) X
pokeM x o
  = suc (Index o)
  ▷ poke (Op o)
  ▷ λ { zero → x ; (suc i) → Data o i }
```

We can combine these two functions to construct a function that allows us to extract a selection of elements from a list that pass a particular predicate, and moreover preserve the original list with holes in place of the selected elements. Concretely, we can define this function as follows:

```
select : (A → Bool) → List A → OpM (PartialList A) A
select p [] = 0 ▷ [] ▷ λ ()
select p (a :: as)
  = if p a then pokeM a (select p as)
           else  consM a (select p as)
```

Intuitively, select highlights elements of a list that satisfy a given predicate, and will allow us to modify them while leaving the rest of the list intact. Indeed, if we let fill⇒ : PList A ⇒ Endo $\mathcal{U}$ (List A) be the operad morphism with ⟨ fill⇒ ⟩ = fill, then for every predicate $p : A \to$ Bool and list xs : List A

we can construct a path of type runAlg fill⇒ ([_] <$> select p xs) $\equiv xs$, where [_] : $A \to$ List A is the function constructing the singleton list. That is, reinserting the highlighted elements back into the list without modifying them will reconstruct the original list.

4.8 Free operad

Thus far, we have seen how a generalised form of operads can be formalised in HoTT, and how this encompasses the usual notions of planar and symmetric operads. Using this framework, we now introduce and formalise one of the key constructions in the theory of operads, namely the *free* operad on a given collection of operations. In particular, this is precisely the left adjoint construction to the forgetful functor from operads to their underlying operations. Intuitively, the operations of free operads correspond to trees whose nodes are labelled by elements of a collection of operations depending on their arity, where the composition map is given by grafting of trees. Free operads provide an approach to constructing operation trees independently from the choice of how to compose the underlying operations. In this section, we will give an account of the free construction for generalised operads in HoTT, together with some important properties and examples.

We begin with an introduction to the notion of a free planar operad on a countable family of h-sets $K : \mathbb{N} \to$ Type and then show how the relevant constructions can be generalised. The operations of the free planar operad on K are characterised by planar, finitely-branching trees in which nodes with n-input vertices are labelled by an element of $K\ n$. Furthermore, we require a distinguished unit tree with both a single input and output vertex that acts as both a left and right unit for tree composition. For example, one possible valid tree that corresponds to an operation of the free planar operad on K can be depicted as follows:

$$\begin{tikzcd}
{} \arrow[d] & & {} \arrow[d] & {} \arrow[d] & {} \arrow[d] & {} \arrow[d] & \\
\mathsf{id} \arrow[dr] & & \mathsf{id} \arrow[dl] & \mathsf{id} \arrow[dr] & \mathsf{id} \arrow[d] & \mathsf{id} \arrow[dl] & \\
& k_0 : K\ \mathsf{2} \arrow[drrr] & & & k_1 : K\ \mathsf{3} \arrow[d] & & k_2 : K\ \mathsf{0} \arrow[dll] \\
& & & & k : K\ \mathsf{3} & &
\end{tikzcd}$$

In this diagram, we use id to denote the distinguished unit tree. The depicted operation has precisely 5 inputs and represents an operation built by composing the outputs of the operations k_0, k_1 and k_2 into the three inputs of the operation k. As previously described, we also require that the distinguished unit tree acts as both a left and right unit for tree composition. Intuitively, this means that we can collapse internal uses of the unit tree so, for example

$$\begin{tikzcd}
\cdots \arrow[d] & & & \\
x : K\ 1 \arrow[d] & \cdots \arrow[dr] & \cdots \arrow[d] & \cdots \arrow[dl] \\
\mathsf{id} \arrow[dr] & & y : K\ 3 \arrow[dl] & \\
& z : K\ 2 & &
\end{tikzcd}
\qquad \text{reduces to} \qquad
\begin{tikzcd}
\cdots \arrow[d] & \cdots \arrow[dr] & \cdots \arrow[d] & \cdots \arrow[dl] \\
x \arrow[dr] & & y \arrow[dl] & \\
& z & &
\end{tikzcd}$$

For readers familiar with the notion of *rose trees*, this structure may seem familiar. To recall, rose trees are the planar finitely-branching trees and are typically labelled with elements of a parametrised type. We can define rose trees in Cubical Agda as follows:

```
data RoseTree (A : Type) : Type where
  leaf : RoseTree A
  node : (n : ℕ) → A → (Fin n → RoseTree A) → RoseTree A
```

We can easily define a function count : RoseTree A → ℕ that counts the total number of leaves of any given rose tree. The fibers of this count function, i.e. the countable family of types mapping n : ℕ to the type ∑[t ∈ RoseTree A] count $t \equiv n$ of rose trees with n leaves, comes equipped with an operad structure given by tree composition with unit leaf. In the specific case where the parametrised type A is the unit type, the induced operad is also known as the *planar tree operad*. The full details of this construction can be found in our Cubical Agda formalisation and is near identical to the following construction of the free planar operad. The key difference between rose trees and the free planar operad, is how nodes are labelled. In particular, for the free planar operad we are given a countable family of h-sets K : ℕ → Type and a node with n-inputs is labelled by an element of $K\ n$. Concretely, we can define the operations of the free planar operad as follows:

```
data FreePLOps (K : ℕ → Type) : ℕ → Type where
  unit : FreePLOps K 1
  comp : (n : ℕ) (ns : Fin n → ℕ) → K n →
           ((∀ i → FreePLOps K (ns i)) → FreePLOps K (Σ n ns)
```

This definition might at first appear rather different from that of rose trees, however this arises as a consequence of choosing to index our inductive definition over the number of leaves rather than defining a seperate count function whose fibers are equivalent to FreePLOps K. This is similar to choosing to use the usual inductive definition of length-indexed vectors in contrast to equivalently defining them as the fibers over the length function on lists, and in practice is easier to compute with. Given that K is a family of h-sets, it follows that FreePLOps is also a family of h-sets. We do not give the explicit proof of this here, but it can be evidently observed from the inductive definition of FreePLOps and the full details can be found in our Cubical Agda formalisation.

As we should expect, our inductive construction for the operations of the free planar operad comes equipped with a planar operadic structure with the identity operation given by unit : FreePLOps K 1. While we do not give an account of the full operadic structure on FreePLOps here, it follows as a specific case of the more general free operad construction. In particular, given a universe $\mathcal{U}$: Universe, a first attempt at defining the operations of the free $\mathcal{U}$-operad is the following translation of FreePLOps to the generalised case:

```
data FreeOps (K : Code 𝓤 → Type) : Code 𝓤→ Type where
  leaf : FreeOps K (⊤̂ 𝓤)
  node : (A : Code 𝓤) (B : ⟦ 𝓤 ∋ A ⟧ → Code 𝓤) →
           K A → ((∀ a → FreeOps K (B i)) → FreeOps K (Σ̂ 𝓤 A B)
```

However, this definition of FreeOps is not a sufficient characterisation of the free operad as it is not, in general, a family of h-sets. This is not immediately evident as although the node constructor parametrises over a type of codes Code $\mathcal{U}$ that may not be an h-set, i.e. FinSet, the possible codes are then restricted as a consequence of being used to index the node constructor. Indeed, it is possible to show that the assumption that FreeOps K A is an h-set for every family of h-sets K : Code $\mathcal{U}$ → Type and every code A : Code $\mathcal{U}$ leads to a contradiction. We begin by defining a family of types

$\mathsf{Partition} : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ as follows:

$$\begin{aligned}\mathsf{Partition}\ X = {} & \textstyle\sum[\ A \in \mathsf{Code}\ \mathcal{U}\ ] \\ & \textstyle\sum[\ B \in (⟦\ \mathcal{U} \ni A\ ⟧ \to \mathsf{Code}\ \mathcal{U})\ ] \\ & (X \equiv \hat{\textstyle\sum}\ \mathcal{U}\ A\ B)\end{aligned}$$

We then proceed by constructing a family of types $\mathsf{FOps} : (\mathsf{Code}\ \mathcal{U} \to \mathsf{Type}) \to \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ that is evidently equivalent to the family $\mathsf{FreeOps}$ as follows:

$$\begin{aligned}& \mathsf{FOps}\ K\ X \\ & \quad = (X \equiv \hat{\top}\ \mathcal{U}) \uplus \\ & \qquad (\textstyle\sum[\ (A\ ,\ B\ ,\ p) \in \mathsf{Partition}\ X\ ] \\ & \qquad\qquad K\ A \times ((a : ⟦\ \mathcal{U} \ni A\ ⟧) \to \mathsf{FreeOps}\ K\ (B\ a)))\end{aligned}$$

While we do not provide the construction of the family of equivalences between $\mathsf{FreeOps}\ K\ A$ and $\mathsf{FOps}\ K\ A$ for all $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ and $A : \mathsf{Code}\ \mathcal{U}$ here, it is a routine construction and can be found in our Cubical Agda formalisation. As equivalences preserve h-levels, it is therefore sufficient to show that for any choice of universe $\mathcal{U}$, family of h-sets $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ and code $A : \mathsf{Code}\ \mathcal{U}$, the assumption that $\mathsf{FOps}\ K\ A$ is an h-set leads to a contradiction. In order to do this, we consider the case where $\mathcal{U}$ is the universe of Bishop-finite sets, i.e. $\mathsf{Code}\ \mathcal{U} = \mathsf{FinSet}$ and $⟦\ \mathcal{U} \ni A\ ⟧ = ⟦\ A\ ⟧$, K is the constant family to the unit type $\top$, and A is the finite set $\mathsf{size}\ 0 : \mathsf{FinSet}$. We then proceed by recalling that retractions preserve h-levels and therefore it is sufficient to choose any type $X : \mathsf{Type}$ such that we have a term $\neg\mathsf{isSetX} : \mathsf{isSet}\ X \to \bot$ together with functions $f : X \to \mathsf{FOps}\ (\lambda\ a \to \top)\ (\mathsf{size}\ 0)$ and $g : \mathsf{FOps}\ (\lambda\ a \to \top)\ (\mathsf{size}\ 0) \to X$ such that we can construct a family of paths $p : (x : X) \to g\ (f\ x) \equiv x$. In particular, we choose X to be FinSet which can be shown to not be an h-set. For example, the path space $\mathsf{size}\ 2 \equiv \mathsf{size}\ 2$ is equivalent to $\mathsf{Fin}\ 2 \simeq \mathsf{Fin}\ 2$ which is not an h-proposition. In order to define the function f, we first note that for every $A : \mathsf{FinSet}$ we can construct a path $\hat{\sum}0\ A : \mathsf{size}\ 0 \equiv \hat{\sum}\ A\ (\lambda\ a \to \mathsf{size}\ 0)$ by applying the un path constructor to the canonical equivalence between $\mathsf{Fin}\ 0$ and $⟦\ A\ ⟧ \times \mathsf{Fin}\ 0$. The function f is then defined as follows:

$$\begin{aligned}f\ A = {} & \mathsf{inr}\ (A\ ,\ (\lambda\ a \to \mathsf{size}\ 0)\ ,\ \hat{\textstyle\sum}0\ A \\ & \quad ,\ \mathsf{tt}\ ,\ (\lambda\ a \to \mathsf{node}\ (\mathsf{size}\ 0)\ (\lambda\ ())\ \mathsf{tt}\ (\lambda\ ())))\end{aligned}$$

In the other direction, we define g by induction on each of the left and right cases, whereby for the left case we have a proof of $\mathsf{size}\ 0 \equiv \mathsf{size}\ 1$ and by the functorial action of $[\![\_]\!]$ on paths this gives an equivalence $\mathsf{Fin}\ 0 \simeq \mathsf{Fin}\ 1$ and thus leads to contradiction, leaving only the right case for consideration. For the right case, we simply project out the first component of the partition, i.e. the first finite set. Following from this construction, the family of paths p that witnesses that g is a retraction of f follows definitionally, and we can therefore conclude that neither FOps or more importantly FreeOps are families of h-sets.

As a consequence of FreeOps not being a family of h-sets, in order to internalise the underlying operations of the free operad it is necessary to add a set-truncation path constructor

$$\mathsf{set} : (A : \mathsf{Code}\ \mathcal{U}) \to \mathsf{isSet}\ (\mathsf{FreeOps}\ K\ A)$$

to our inductive definition. Notably, this is not the same as defining a new family of types which maps each $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ and $A : \mathsf{Code}\ \mathcal{U}$ to $\|\ \mathsf{FreeOps}\ K\ A\ \|_0$, i.e. the set or 0-truncation of $\mathsf{FreeOps}\ K\ A$. Moreover, in the case where the type of codes for the considered universe is an h-set, such as for the universe of totally-ordered finite sets, then the versions of the FreeOps with and without the set path constructor are equivalent.

As a consequence of presenting the operations of the free operad on K as a higher-inductive family $\mathsf{FreeOps}\ K$, the definition of the operad composition map requires explicit substitutions along paths on codes in order to construct operations with the expected labelling given to their inputs. In practice, this can lead to "transport hell" whereby operations of the free operad that are constructed by applying the composition map are built from nested substitutions that consequently cannot be unfolded in further computations. That is, the application of an inductively-defined function to a term that is constructed by substitution along a path cannot always be unfolded along the definitional equalities of that function. In the presence of (small) induction-recursion, it is possible to address this issue by presenting an alternative encoding of the operations of the free operad that separates the shape of a labelled tree from the indexing of its leaves. This alternative presentation allows us to avoid "transport hell" when defining functions inductively on the shape of a tree corresponding to an operation of the free operad.

Our alternative encoding of the operations of the free operad makes use of a known equivalence between inductive families and small induction-recursion. However, this equivalence has not been extended to higher-inductive families and correspondingly higher small induction-recursion. In order to highlight this issue, we first consider a naive translation of only the data constructors of the higher inductive family FreeOps K, for every K : Code $\mathcal{U}$ → Type, to a small inductive-recursive definition. Concretely, we mutually define an inductive type FreeOpsIR K : Type together with a recursive function CodeOps K : FreeOpsIR K → Code $\mathcal{U}$ as follows:

```
data FreeOpsIR K : Type where
  leaf : FreeOpsIR K
  node : (A : Code 𝓤) → K A →
         (⟦ 𝓤 ∋ A ⟧ → FreeOpsIR K) → FreeOpsIR K
```

```
CodeOps K leaf = ⊤̂ 𝓤
CodeOps K (node A k ts) = Σ̂ 𝓤 A (CodeOps ∘ ts)
```

In particular, there is an equivalence between the fibers of CodeOps K and the inductive family FreeOps K without its set path constructor. However, as the type of codes Code $\mathcal{U}$ is not necessarily an h-set, the fibers of CodeOps K cannot, in general, be shown to be h-sets and are therefore not equivalent to the higher-inductive family FreeOps K.

In the absence of the axiom of choice, it is not sufficient to simply consider the set truncated fibers over CodeOps K, i.e. the family mapping each code A : Code $\mathcal{U}$ to the following 0-truncated type:

$$\| \textstyle\sum[\ t \in \text{FreeOpsIR}\ K\ ]\ \text{CodeOps}\ K\ t \equiv A\ \|_0$$

Instead, it is necessary to add an appropriate higher path constructor to the definition of FreeOpsIR K together with the action of CodeOps K on this constructor, to ensure that the fibers of CodeOps K are h-sets. To do this, we begin by noting that for any types X, Y : Type and function $f : X \to Y$ the proposition that the fibers of f are all h-sets is equivalent to the proposition that the fibers of the functorial action of f on paths are all h-propositions. Moreover, this is equivalent to the proposition that the functorial action of f is an embedding, i.e. for all terms $x_1\ x_2 : X$ and paths $p\ q : x_1 \equiv x_2$, the function cong (cong f) : $p \equiv q \to$ cong f $p \equiv$ cong f q

is an equivalence. Indeed, this embedding is precisely what we will aim to establish by adding an appropriate path constructor to FreeOpsIR K.

In order to establish that the functorial action of CodeOps K is an embedding, it is sufficient to present an isomorphism between the types $p \equiv q$ and cong (CodeOps K) $p \equiv$ cong (CodeOps K) q, for all $p\ q : t \equiv u$, and $t\ u$: FreeOpsIR K, where the function cong (cong (CodeOps K)) : $p \equiv q \to$ cong (CodeOps K) $p \equiv$ cong (CodeOps K) q is the 'forward' map of this isomorphism. Concretely, this means constructing a function from cong (CodeOps K) $p \equiv$ cong (CodeOps K) q to $p \equiv q$ that is both a section and retraction of the function cong (cong (CodeOps K)). From our current definition of FreeOpsIR it is not possible to construct such a function. However, as this function should construct a higher path within the type FreeOpsIR K, we can first attempt to simply include it as the following path constructor of FreeOpsIR K:

```
set  : (t u : FreeOpsIR K) (p q : t ≡ u) →
         cong (CodeOps K) p ≡ cong (CodeOps K) q → p ≡ q
```

In Cubical Agda, the above path constructor will not be accepted on account of not being considered strictly positive. Fortunately, the solution to this problem is as simple as flipping squares of the form

```
cong (CodeOps K) p ≡ cong (CodeOps K) q
```

along their diagonal and instead representing them with the following type of heterogeneous paths:

```
PathP (λ i → CodeOps K (p i) ≡ CodeOps K (q i)) refl refl
```

With this alternative 'flipped' representation, we can then extend the definition of CodeOps K over the path constructor set as follows:

```
CodeOps K (set t u p q r i j) = r j i
```

The next step to ensuring that the fibers of CodeOps K are all h-sets is to ensure that set $t\ u\ p\ q$ is indeed both a section and retraction of cong (cong (CodeOps K)). However, this happens to already be provable with no further adjustments to our definition. That is, the set path constructor is sufficient to establish that the fibers of CodeOps K are all h-sets.

The proof of this is a generalisation of a proof used in the construction of *univalent inductive-recursive* types, whereby a path constructor asserting injectivity of the mutually defined function is sufficient to show that it has propositional fibers. The full details of this proof, together with a proof that the fibers of CodeOps K are equivalent to the higher inductive family FreeOps, can be found in our Cubical Agda formalisation.

Notably, we can always eliminate terms of FreeOpsIR K into any h-set whereby the path constructor set is necessarily respected. Importantly, this includes eliminating into the fibers of the function CodeOps K which is necessary to define the composition map for the free operad. Indeed, the fibers over CodeOps K are precisely the operations of the free operad. As such, we define the following family of types in order to simplify definitions in this section:

```
FreeOperad K A = fiber (CodeOps K) A
```

In order to construct the composition map over FreeOperad K, we first introduce the following two required families of paths:

```
∑̂Idl₁ 𝓤
  : (B : ⟦ 𝓤 ∋ ⊤̂ ⟧ → Code 𝓤) →
    B (inv (⟦⊤̂⟧ 𝓤) tt) ≡ ∑̂ 𝓤 (⊤̂ 𝓤) B

∑̂Assoc₁ 𝓤
  : (A : Code 𝓤) (B : ⟦ 𝓤 ∋ A ⟧ → Code 𝓤)
    (C : ⟦ 𝓤 ∋ ∑̂ 𝓤 A B ⟧ → Code 𝓤) →
    ∑̂ 𝓤 A (λ a → ∑̂ 𝓤 (B a) (λ b → C (inv (∑̂ 𝓤 A B ⟧) (a , b))))
      ≡ ∑̂ 𝓤 (∑̂ 𝓤 A B) C
```

While we do not give the full constructions for these paths here, they follow from the paths ∑̂Idl and ∑̂Assoc respectively. Given an abstract operation t : FreeOpsIR K and a collection of abstract operations ts : ⟦ 𝓤 ∋ CodeOps K t ⟧ → FreeOpsIR K, we can define a non-indexed version of the operad composition map as follows:

```
graft t ts : FreeOperad K (∑̂ 𝓤 (CodeOps K t) (CodeOps K ∘ ts))
graft leaf ts
  = ts (inv (⟦⊤̂⟧ 𝓤) tt) ,
        Σ̂Idl₁ 𝒰 (CodeOps K ∘ ts) (inv (⟦⊤̂⟧ 𝒰) tt)
    graft (node A k ts) tss
      = let us : (a : ⟦ 𝒰 ∋ A ⟧) →
                   fiber (CodeOps K) (Σ̂ 𝒰 (CodeOps K (ts a)) _)
            us = graft (ts a)
                       λ b → tss (inv (⟦Σ̂⟧ 𝒰 A (CodeOps K ∘ ts)) (a , b))
        in node A k (fst ∘ us) ,
          (λ i → Σ̂ 𝒰 A λ a → snd (us a) i) ·
          Σ̂Assoc₁ 𝒰 (CodeOps K ∘ ts) (CodeOps K ∘ tss)
```

We note that the action of the graft function on the set path constructor follows from the proof that the fibers of CodeOps K are h-sets.

The indexed composition map for the free operad follows from the definition of graft by reinserting the indexing of the input operations. In particular, for every code A : Code $\mathcal{U}$, family of codes $B : ⟦ \mathcal{U} \ni A ⟧ \to$ Code $\mathcal{U}$, indexed operation t : fiber (CodeOps K) A, and family of indexed operations $ts : (a : ⟦ \mathcal{U} \ni A ⟧) \to$ fiber (CodeOps K) $(B\ a)$, we can construct the operad composition map fcomp $A\ B\ t\ ts$: fiber (CodeOps K) ($\hat{\sum}\ \mathcal{U}\ A\ B$) as follows:

```
    fcomp A B t ts =
      let
        q : Σ̂ 𝒰 (CodeOps K (fst t)) (CodeOps K ∘ fst ∘ ts) ≡ Σ̂ 𝒰 A B
        q i = Σ̂ 𝒰 (snd ts i) (λ a → snd (ts a) i)

        (u , p) = graft (fst t) (fst ∘ ts)
      in u , p · q
```

The unit operation for the free operad is simply given by a pair of the term leaf : FreeOpsIR together with the reflection path witnessing that CodeOps K leaf $\equiv \hat{\top}\ \mathcal{U}$. We omit the constructions for the paths witnessing the identity and associativity operad laws for the composition map fcomp as they involve significant technical detail that is not necessary for understanding the general construction. However, these proofs can be found in our Cubical Agda formalisation.

We have thus far described the operadic structure on the fibers over the function CodeOps K for every $\mathcal{U}$-species K : Code $\mathcal{U} \to$ Type, however, we have yet to show that this is indeed the *free* operad on K. Indeed to do this, we will first highlight the categories on which we will define the adjunction corresponding to the free construction. In particular, the objects of our underlying category are $\mathcal{U}$-species which themselves can be understood as functors from the h-groupoid structure of $\mathsf{Code}\ \mathcal{U}$ to the category of h-sets and functions. Concretely, the functorial action of a $\mathcal{U}$-species $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ is given by path substitution in K, i.e. a path $p : A \equiv B$ is lifted to $\mathsf{subst}\ K\ p : K\ A \to K\ B$. It is a standard result of path substitution in HoTT that this indeed satisfies the functorial laws. As might be expected, morphisms in the category of $\mathcal{U}$-species are given by natural transformations. That is, for any two $\mathcal{U}$-species $K\ L : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ their corresponding hom is given by $(A : \mathsf{Code}) \to K\ A \to L\ A$ where naturality follows from substitution commuting with morphisms in slice categories. Finally, the target category for the free construction of a $\mathcal{U}$-operad is precisely the category of $\mathcal{U}$-operads detailed in Section 8.

There is an evident forgetful functor from the category of $\mathcal{U}$-operads to the category of $\mathcal{U}$-species which simply forgets the operadic structure and acts similarly on operad morphisms by means of the projection $\langle \_ \rangle : (O^K \Rightarrow O^L) \to (A : \mathsf{Code}) \to K\ A \to L\ A$. The free operad on a $\mathcal{U}$-species $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ is, up to a path, the operad that is constructed by the left-adjoint to this forgetful functor. As we will show, the left-adjoint is the functor mapping each $K : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ to the collection of operations $\mathsf{fiber}\ (\mathsf{CodeOps}\ K) : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$ equipped with its previously described operadic structure. The functoriality of this construction follows from our proof that it satisfies the universal property of being a left adjoint in the usual way. The first step to showing that this is indeed left adjoint to the forgetful functor from $\mathcal{U}$-operads to $\mathcal{U}$-species, is to define the following unit natural transformation:

$$\begin{aligned}
&\eta : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to \mathsf{fiber}\ (\mathsf{CodeOps}\ K)\ A \\
&\eta\ A\ k = \mathsf{node}\ A\ k\ (\lambda\ a \to \mathsf{leaf})\ ,\ \hat{\textstyle\sum}\mathsf{Idr}\ \mathcal{U}\ A
\end{aligned}$$

Moreover, we require that for every operad $L : \mathsf{Code}\ \mathcal{U} \to \mathsf{Type}$, $O : \mathsf{Operad}\ \mathcal{U}\ L$, and every morphism of $\mathcal{U}$-species $f : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to L\ A$, that we can construct an operad morphism from the operadic structure of $\mathsf{FreeOpsIR}$ to O. In particular, we begin by defining a non-indexed variation of the interpretation function $\mathsf{interpret}_1\ O\ f : (t : \mathsf{FreeOpsIR}\ K) \to L\ (\mathsf{CodeOps}\ K\ t)$ as follows:

$$\begin{aligned}
&\mathsf{interpret}_1\ O\, f\, \mathsf{leaf} = \mathsf{id}\ O \\
&\mathsf{interpret}_1\ O\, f\, (\mathsf{node}\ A\ k\ ts) \\
&\quad = \mathsf{comp}\ O\ A\ (\mathsf{CodeOps}\ K \circ ts)\ (f\ A\ k)\ (\mathsf{interpret}\ O\, f \circ ts)
\end{aligned}$$

From this definition, we can define the standard indexed interpretation map $\mathsf{interpret} : (A : \mathsf{Code}\ \mathcal{U}) \to \mathsf{fiber}\ (\mathsf{CodeOps}\ K)\ A \to L\ A$ as follows:

$$\mathsf{interpret}\ O\, f\, A\ (t\ ,\ p) = \mathsf{subst}\ L\ p\ (\mathsf{interpret}_1\ O\, f\, t)$$

As can be observed, $\mathsf{interpret}$ maps the identity $(\mathsf{leaf}\ ,\ \mathsf{refl})$ to the term $\mathsf{subst}\ L\ \mathsf{refl}\ (\mathsf{id}\ O)$ which is provably equal to $\mathsf{id}\ O$ and thus it respects operad identity. Meanwhile, the proof that the interpretation map $\mathsf{interpret}$ respects operad composition proceeds by induction on t and involves significant technical detail. As this is not necessary to describe the general structure of the free operad, we refer interested readers to our Cubical Agda formalisation for the details.

Moreover, our formalisation necessarily includes a construction for the universal property of the free operad, namely a proof that for all $f : (A : \mathsf{Code}\ \mathcal{U}) \to K\ A \to L\ A$ the type

$$\begin{aligned}
&\textstyle\sum[\ g \in (A : \mathsf{Code}\ \mathcal{U}) \to \mathsf{FreeOperad}\ K\ A\ \to L\ A\ ] \\
&\quad ((A : \mathsf{Code}\ \mathcal{U})\ (k : K\ A) \to g\ A\ (\eta\ A\ k) \equiv f\ A\ k)
\end{aligned}$$

is contractible. In particular, the base point is given by a pair of the function $\mathsf{interpret}\ O\ f$ together with a witness of the proposition that for all $A : \mathsf{Code}\ \mathcal{U}$ and $k : K\ A$, $\mathsf{interpret}\ O\ f\ A\ (\eta\ A\ k) = f\ A\ k$. After unfolding the left-hand side, we can observe that it suffices to construct a path of the following type:

$$\mathsf{subst}\ L\ (\hat{\textstyle\sum}\mathsf{Idr}\ \mathcal{U}\ A)\ (\mathsf{comp}\ O\ A\ (\lambda\ a \to \hat{\top}\ \mathcal{U})\ (f\ A\ k)\ (\lambda\ a \to \mathsf{leaf})) \equiv f\ A\ k$$

In Cubical Agda, it is a standard result that this is provably equivalent to the following heterogeneous path type:

$$\begin{aligned}
&\mathsf{PathP}\ (\lambda\ i \to \hat{\textstyle\sum}\mathsf{Idr}\ \mathcal{U}\ A\ i) \\
&\qquad (\mathsf{comp}\ O\ A\ (\lambda\ a \to \hat{\top}\ \mathcal{U})\ (f\ A\ k)\ (\lambda\ a \to \mathsf{leaf})) \\
&\qquad (f\ A\ k)
\end{aligned}$$

Notably, this is precisely the type of the right identity law, i.e. $\mathsf{idr}\ O$, for the operad O applied to $f\ A\ k$. As such, the path idr $O\ A\ (f\ A\ k)$ is a witness of the required uniqueness property for our construction of the interpretation map.

4.9 Related work

There have been several generic representations of datatypes developed in the type theory literature. The idea of separating structure from data first arose in the work on shapely types [Jay and Cockett, 1994]. More recently, the theory of containers was developed to capture the idea of the strictly positive types [Abbott, 2003]. The more general presentation of indexed containers was later developed to capture inductive families [Altenkirch and Morris, 2009]. Containers are closed under finite products and exponentials [Altenkirch et al., 2010], and have a sensible notion of derivative [Abbott et al., 2003]. The theory of containers also captures the shapely types. In particular, shapely type constructors are precisely given by the extensions of discretely-finite containers.

Kock [2012] introduces a generalisation of containers that captures polynomial functors over groupoids. More specifically, while the standard notion of a container is defined over a universe of sets, Kock introduces a similar construction that instead makes use of a universe of groupoids. Interestingly, groupoid containers can also be used to describe the underlying collection of operations of generalised operads as introduced in Section 2.9. In particular, we represent a collection of operations as families of of h-sets, i.e. Code $\mathcal{U}$ → Type, however they can equivalently be presented as groupoid containers of the form $\sum$[A ∈ hSet] (A → Code $\mathcal{U}$). Indeed, groupoid containers as introduced by Kock were developed for precisely the same reason as our notion of generalised operad universes, i.e. to exploit the higher structure of types to capture symmetries.

A key aspect of our theory of operads in HoTT is the notion of a finite set, and the corresponding choice of a suitable definition of finiteness. Such a choice only manifests in constructive mathematics, where various classically equivalent notions of finiteness are distinct. In this chapter, we adopted a notion of finiteness used in previous work on internalising the theory of combinatorial species in HoTT [Yorgey, 2014], namely Bishop-finiteness. However, there has also been work on internalising 'enumerated' types and Kuratowski-finite types in the framework of HoTT [Spiwack and Coquand, 2010; Frumin et al., 2018]. In particular, a Kuratowski-finite type is a type A for which there merely exists a finite list that contains every element of A. Moreover, a *Kuratowski-finite set* is a type that is both an h-set and Kuratowski-finite. It can be shown that the Bishop-finite sets are precisely the Kuratowski-finite types that have decidable equality.

Adopting a different notion of finiteness impacts both which collection of operations are definable as a species, and which proofs about operads can be internalised. In this chapter, this distinction does not materialise as a consequence of developing our theory of operads uniformly over the notion of generalised operad universes as introduced in Section 4.5. However,

The theory of combinatorial species were first introduced to unify existing approaches for analysing generating functions of discrete structures [Joyal, 1981]. Intuitively, the idea of a species arose from the idea of building mathematical structures from a finite collection of labels. Yorgey [2014] formalises combinatorial species in HoTT to capture specific classes of data types, analogous to shapely types and the theory of containers. Our formalisation of symmetric operads in Section 4.3 can be seen as an extension of this formalisation work, whereby we equip a combinatorial species with a compositional structure that respects unitality and associativity conditions.

Flores et al. [2023] have recently developed a formalisation of coloured operads [Yau, 2018] in the proof assistant Coq with the goal of reasoning about the denotational semantics of programming languages. Coloured operads are a particular presentation of a multicategory [Lambek, 1969], and can be seen as a generalisation of the notion of operad introduced in this chapter. In particular, our definition of an operad only presents the structure of a single object multicategory. Indeed, this observation is also made by Flores et al., who compare their formalisation to an earlier version of the work presented in this chapter. In keeping with the standard presentation of operadic theory, Flores et al. index operations with natural numbers. However, our notion of an operad can be understood to generalise in a different direction whereby we permit operations to be indexed over any universe that satisfies the criteria outlined in Section 4.5.

The theory of operads first originated in the field of algebraic topology [May, 2006; Boardman and Vogt, 2006] for describing operations on iterated loop-spaces. Cyclic operads [Getzler and Kapranov, 1995] enrich the standard notion of an operad by equipping it with an action that allows interchanging any of the inputs of an operation with its output. That is, a cyclic operad can be viewed as a symmetric operad that does not distinguish between its inputs and outputs. This enrichment of operads was first introduced in order to study structures in cyclic homology and examples include the associative and commutative operads [Kimura et al., 1995]. This additional cyclic structure cannot be captured merely by selecting a sufficient operad universe as described in Section 4.5, as this does not capture symmetries that would permit interchanging inputs and outputs.

To capture cyclic operads within the framework presented in this chapter, we can use a standard approach from classical operad theory. Specifically, this involves equipping a symmetric operad with a 'well-behaved' involution on the derivative of its underlying species [Curien and Obradovic, 2016]. Intuitively, the derivative of a species introduces a distinguished point corresponding to the position of the output in an operation. In particular, given a combinatorial species $K : \mathsf{FinSet} \to \mathsf{hSet}$, we construct its derivative as follows $\partial\ K\ A = K\ (A \uplus \top)$ and we define the family of exchange isomorphisms $\mathsf{ex} : (A : \mathsf{FinSet}) \to \partial\ (\partial\ K\ A) \to \partial\ (\partial\ K\ A)$ by substitution over the equivalence that swaps the two distinct points. A cyclic operad can then be defined as a symmetric operad that is equipped with a family of involutions $\mathsf{cycle} : (A : \mathsf{FinSet}) \to \partial\ K\ A \to \partial\ K\ A$ together with a collection of three laws witnessing that cycle respects both exchange and operad composition. We omit the additional laws for cyclic operads since they are detailed in Curien and Obradovic [2016] and are not the focus of this chapter.

4.10 Conclusion and further work

In this chapter we have demonstrated how operads can be internalised in HoTT, and how this gives rise to a generic calculus of operations. Notably, we show how discretely finite containers, which represent inductive data types whose constructors have a finite arity, give rise to a natural operadic interpretation. Intuitively this provides an operation-centric approach to inductive types, in contrast to the traditional data-centric approach. Our results open up a new line of work on investigating the properties and applications of operads from a type-theoretic perspective.

A natural extension of our work is generalising our formalisation of an operad to capture coloured operads [Yau, 2018]. The relationship between operads and coloured operads is analagous to that of containers and indexed containers. In particular, a coloured operad can intuitively be understood as an operad whose operations are equipped with a simple type system. Indeed, just as our notion of an operad can be understood as equipping a particular class of containers with a compositional structure, a formalisation of coloured operads should do the same for indexed containers.

As is typical for algebraic structures, operads have a natural dual in 'cooperads' [Ching, 2012]. Cooperads are defined by the evident dualisation of operads, i.e. by reversing the direction of the composition and unit maps. Intuitively, a cooperad over a collection of operations describes a means by which an operation can be partitioned into 'smaller' operations. In this way, cooperads are to operads as coinductive types are to inductive types. A formalisation of the theory of cooperads would seem to follow by a trivial dualisation of the definitions given in this paper. However, an important asymmetry arises when considering an appropriate implementation of the cofree cooperad. From the definition of the free operad, we can infer that the cofree cooperad should be isomorphic to a type $\mathbb{G}_{\mathcal{U}}$ of graphs, in which each node has a label, a distinguished output edge, and a collection of input edges indexed by a code in $\mathcal{U}$. Equivalently, $\mathbb{G}_{\mathcal{U}}$ is the type of coinductive (infinite), $\mathcal{U}$-branching, labelled trees. We postulate that, in contrast to its inductive counterpart, counitality and coassociativity can be proven for the 'de-composition' operation of $\mathbb{G}_{\mathcal{U}}$-graphs without the need for quotients.

Chapter 5

Conclusion

In this thesis, we have developed and explored three key ideas that are based upon equipping types with extra structure: predicates, equations, and composition. Each of chapters 2, 3 and 4 already incorporates their own conclusion and future work section. This final chapter presents a high-level reflection upon each idea, summarises our key contributions, and considers connections that suggest interesting areas for further work.

Chapter 2 introduced and formalised two equivalent encodings for subtypes that support fine-grained control over the unfolding of a chosen collection of operations. In particular, the first of these encodings involved defining the subtyping condition as a higher inductive family while the second defined the entire subtype as a higher inductive-recursive type. This fine-grained control over the unfolding of operations on subtypes can significantly improve the performance of type checking within any proof assistant that supports either higher inductive families or higher induction-recursion.

In contrast to existing solutions for controlling unfolding behaviour such as Agda's abstract definitions, the techniques outlined in this chapter can also be used to improve runtime performance. Importantly, this is achieved without discarding computational content and we can construct an internal equivalence between a subtype its encodings. Indeed, our approach can be seen as a particular instance of using higher inductive types to efficiently encode data with a more performant eliminator. In Section 2.9 we considered how this more general pattern could be formalised by generalising our alternative encodings of subtypes to arbitrary dependent sums.

Notably, the techniques outlined in this chapter are not specific to homotopy type theory. That is, we do not require support for types with higher structure and only make use of a limited form of higher induction. In particular, we need only be able to introduce equalities between terms and not between the equalities themselves. Consequently, these techniques are applicable in any type system that supports either quotient inductive families or quotient induction-recursion. We will consider an example of how our technique can be applied to practical programming in our summary of Chapter 3.

Chapter 3 introduced Quotient Haskell: a type system that extends Liquid Haskell with support for a class of quotient inductive types whose respectfulness theorems can be automatically verified by an SMT solver. Moreover, this chapter presented an idealised core language that added support for quotient inductive types to the liquid type system λ_L introduced by Rondon et al. [2008]. We also considered several practical examples of quotient inductive types in Quotient Haskell such as rational numbers, polar coordinates and the Boom hierarchy.

In addition to supporting the automated verification of respectfulness theorems, Quotient Haskell adds subtyping rules to Liquid Haskell that capture quotient type hierarchies. For example, binary trees can be considered a subtype of binary mobiles. The addition of these decidable subtyping rules permits the reuse of functions along a quotient hierarchy. Furthermore, Quotient Haskell incorporates a rewriting system through which the equalities introduced by equality constructors are automatically reified into the refinement logic. In conjunction with the automated verification of respectfulness theorems, the rewriting system of Quotient Haskell eliminates the manual proof burden of quotient inductive types for many practical use cases. These features were designed in order to facilitate a more lightweight form of quotient types without comprimising on correctness.

Several possible extensions of Quotient Haskell were suggested in the conclusion of chapter 3 including quotient polymorphism and the inference of quotient types. Included amongst these ideas for further work was adding support for quotient induction-recursive types. Notably, support for quotient inductive-recursive types would allow for the techniques described in chapter 2 to be utilised in Quotient Haskell. For example, in order to encode subtypes of lists that permit filtering we first define the following type of templates for lists:

```
data ListT a <p :: [a] -> Bool>
  = Normal { xs : [a] | p xs }
  | Filter (a -> Bool) (ListT a <p>)
```

In the above Liquid Haskell definition we parametrise the type `ListT a` over predicates on lists `p :: [a] -> Bool`. The predicate `p` corresponds to the choice of subtyping condition. The `Normal` constructor consists of an unfolded list, while `Filter` corresponds to filtering a list. By encoding `Filter` as a constructor, we perform optimisations such as fusion before computing along the structure of the represented list.

The type of list templates `ListT a <p>` is evidently not equivalent to the type of lists for which the predicate `p` holds. In particular, it is certainly possible to write functions on `ListT a <p>` that are not lawful functions on lists. For example, we can easily construct a function that counts the number of times `Filter` is applied. If Quotient Haskell were extended to support quotient inductive-recursive types we would expect to be able to mutually define the following inductive type

```
data List a <p :: a -> Bool>
  =  ListT a
  |/ eq :: xs:List a <p> -> ys:List a <p>
        -> { toList xs == toList ys } -> xs == ys
```

alongside the following recursive function:

```
toList :: forall a <p :: a -> Bool>. List a <p> -> [a]
toList (Normal xs)   = xs
toList (Filter p xs) = filter p (toList xs)
```

The above construction is a direct translation of the technique introduced in section 2.4 to Quotient Haskell. Importantly, the `toList` function and `Normal` data constructor witness an isomorphism between `List a <p>` and the type of lists for which `p` holds.

Chapter 4 presented a constructive theory of operads internal to homotopy type theory. Moreover, we demonstrated how a theory of operads internal to HoTT presents a framework for reasoning about generic classes of operations. We recall that the classical notion of an operad typically indexes its operations over the category of ordered finite sets and that different 'classes' of operads such as symmetric operads are defined by means of adding equational laws. In contrast, in section 4.5 we presented a notion of operad in HoTT in which operations were indexed over any choice of h-groupoid. This generalised notion of an operad allows for additional equational laws to be captured by the higher structure of the indexing type rather than explicit proofs. For example, families of types indexed over `FinSet` can capture the property of symmetry under permutations.

In section 4.7 we formalised the construction of the canonical strict 2-monad over an operad. Moreover, we introduced the key notion of an algebra over an operad and provide an example of using such an algebra in practice by means of constructing selections over partial lists.

Finally, in section 4.8 we introduced two constructions in HoTT for the free operad, as a higher inductive family and a higher inductive-recursive type. We also proved the necessity of set-truncating the family of operations represented by our free operad construction. Intuitively, we can understand the free operad as providing us with a framework for writing a domain specific language over a collection of operations for which we can decide how to interpret. Indeed, this is analagous to how the free monad construction is typically used for data in functional programming.

Bibliography

Michael Abbott. 2003. *Categories of containers.* Ph. D. Dissertation. University of Leicester.

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: constructing strictly positive types. *Theoretical Computer Science* 342, 1 (2005), 3–27.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2003. Derivatives of containers. In *Typed Lambda Calculi and Applications*, Martin Hofmann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 16–30.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. 2004. Constructing polymorphic programs with quotient types. In *Mathematics of Program Construction: 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004. Proceedings 7.* Springer, Berlin, Heidelberg, 2–15.

Benedikt Ahrens, Kryzsztof Kapulkin, and Michael Shulman. 2015. Univalent categories and the Rezk completion. *Mathematical Structures in Computer Science* 25, 5 (Jan. 2015), 1010–1039. https://doi.org/10.1017/s0960129514000486

Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. 2018. Quotient inductive-inductive types. In *International Conference on Foundations of Software Science and Computation Structures.* Springer International Publishing, Cham, 293–310.

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015), e5.

Thorsten Altenkirch and Ambrus Kaposi. 2016. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices* 51, 1 (2016), 18–29.

Thorsten Altenkirch, Paul Levy, and Sam Staton. 2010. Higher-order containers. In *Programs, Proofs, Processes*, Fernando Ferreira, Benedikt Löwe, Elvira Mayordomo, and Luís Mendes Gomes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 11–20.

Thorsten Altenkirch and Peter Morris. 2009. Indexed containers. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*. IEEE, United States, 277–285.

Robert Atkey. 2018. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Sergio Maffeis. 2011. Refinement types for secure implementations. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 33, 2 (2011), 1–45.

Yves Bertot and Pierre Castéran. 2013. Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions.

John Michael Boardman and Rainer M Vogt. 2006. *Homotopy invariant algebraic structures on topological spaces*. Vol. 347. Springer, Berlin, Heidelberg.

Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593. https://doi.org/10.1017/S095679681300018X

Evan Cavallo and Robert Harper. 2018. Computational Higher Type Theory IV: Inductive Types. arXiv:1801.01568 [cs.LO] https://arxiv.org/abs/1801.01568

James Chapman, Tarmo Uustalu, and Niccolò Veltri. 2019. Quotienting the delay monad by weak bisimilarity. *Mathematical Structures in Computer Science* 29, 1 (2019), 67–92.

Michael Ching. 2012. A note on the composition product of symmetric sequences. *Journal of Homotopy and Related Structures* 7 (2012), 237–254.

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern matching without K. *SIGPLAN Not.* 49, 9 (aug 2014), 257–268. `https://doi.org/10.1145/2692915.2628139`

Cyril Cohen. 2013. Pragmatic quotient types in Coq. In *Interactive Theorem Proving: 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings 4*. Springer, Berlin, Heidelberg, 213–228.

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2016. Cubical type theory: a constructive interpretation of the univalence axiom. arXiv:1611.02108 [cs.LO]

R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. 1986. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, Inc., USA.

Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. On Higher Inductive Types in Cubical Type Theory. arXiv:1802.01170 [cs.LO] `https://arxiv.org/abs/1802.01170`

Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *COLOG-88*, Per Martin-Löf and Grigori Mints (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 50–66.

Pierre-Louis Curien and Jovana Obradovic. 2016. On the various definitions of cyclic operads. (Jan. 2016). `https://hal.science/hal-01254649` working paper or preprint.

Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. 2015. The Lean theorem prover (system description). In *Automated Deduction-CADE-25: 25th International Conference on*

Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25. Springer, Berlin, Heidelberg, 378–388.

Peter Dybjer and Anton Setzer. 2003. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic* 124, 1-3 (2003).

Samuel Eilenberg and Saunders MacLane. 1945. General theory of natural equivalences. *Trans. Amer. Math. Soc.* 58 (1945), 231–294.

Zachary Flores, Angelo Taranto, Eric Bond, and Yakir Forman. 2023. A formalization of operads in Coq. arXiv:2303.08894 [math.CT]

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. Association for Computing Machinery, New York, NY, United States, 268–277.

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*. Association for Computing Machinery, New York, NY, United States, 201–214.

E. Getzler and M.M. Kapranov. 1995. *Cyclic Operads and Cyclic Homology*. International Press, Cambridge, 167–201.

Gilbert, Gaëtan and Cockx, Jesper and Sozeau, Matthieu and Tabareau, Nicolas. 2019. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.* 3, POPL, Article 3 (jan 2019), 28 pages. `https://doi.org/10.1145/3290316`

Zachary Grannan, Niki Vazou, Eva Darulova, and Alexander J Summers. 2022. REST: integrating term rewriting with program verification (Extended Version). (2022). arXiv preprint arXiv:2202.05872.

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

Peter Hancock, Conor McBride, Neil Ghani, Lorenzo Malatesta, and Thorsten Altenkirch. 2013. Small induction recursion. In *Typed Lambda*

Calculi and Applications, Masahito Hasegawa (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 156–172.

Brandon Hewer. 2020. HoTT Operads. `https://github.com/brandonhewer/Operads-HoTT`.

Brandon Hewer. 2022. Subtyping Cubical Agda library. Available online from `https://tinyurl.com/f8pezwxd`.

Brandon Hewer. 2023. Quotient Haskell. `https://github.com/brandonhewer/QuotientHaskell/tree/develop`.

Brandon Hewer and Graham Hutton. 2022. Subtyping without reduction. In *Mathematics of Program Construction*, Ekaterina Komendantskaya (Ed.). Springer International Publishing, Cham, 34–61.

Brandon Hewer and Graham Hutton. 2024. Quotient Haskell: lightweight quotient types for all. *Proc. ACM Program. Lang.* 8, POPL, Article 27 (jan 2024), 31 pages. `https://doi.org/10.1145/3632869`

Martin Hofmann. 1995. A simple model for quotient types. In *International Conference on Typed Lambda Calculi and Applications*. Springer, Berlin, Heidelberg, 216–234.

Gérard Huet. 1997. The zipper. *Journal of functional programming* 7, 5 (1997), 549–554.

Brian Huffman and Ondřej Kunčar. 2013. Lifting and transfer: a modular design for quotients in Isabelle/HOL. In *Certified Programs and Proofs: Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings 3*. Springer, Berlin, Heidelberg, 131–146.

C. Barry Jay and J. R. B. Cockett. 1994. Shapely types and shape polymorphism. In *Programming Languages and Systems — ESOP '94*, Donald Sannella (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 302–316.

André Joyal. 1981. Une théorie combinatoire des séries formelles. *Advances in Mathematics* 42, 1 (1981), 1–82. `https://doi.org/10.1016/0001-8708(81)90052-9`

Cezary Kaliszyk and Christian Urban. 2011. Quotients revisited for Isabelle/HOL. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. Association for Computing Machinery, New York, NY, United States, 1639–1644.

Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. 2019. Constructing quotient inductive-inductive types. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–24.

Ming Kawaguchi, Patrick M Rondon, and Ranjit Jhala. 2010. Dsolve: safety verification via liquid types. In *International Conference on Computer Aided Verification*. Springer, Berlin, Heidelberg, 123–126.

Takashi Kimura, Jim Stasheff, and Alexander A. Voronov. 1995. On operad structures of moduli spaces and string theory. *Communications in Mathematical Physics* 171, 1 (1995), 1–25. `https://doi.org/10.1007/BF02103769`

Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. 2008. Lambda calculus with patterns. *Theoretical Computer Science* 398, 1-3 (2008), 16–31.

Joachim Kock. 2012. Data types with symmetries and polynomial functors over groupoids. *Electronic Notes in Theoretical Computer Science* 286 (2012), 351–365. `https://doi.org/10.1016/j.entcs.2013.01.001` Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII).

Nicolai Kraus. 2015. *Truncation levels in homotopy type theory.* Ph. D. Dissertation. University of Nottingham.

Joachim Lambek. 1969. Deductive systems and categories II. Standard constructions and closed categories. In *Category Theory, Homology Theory and their Applications I*, Peter J. Hilton (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 76–122.

Nuo Li. 2015. *Quotient types in type theory.* Ph. D. Dissertation. University of Nottingham.

Peter LeFanu Lumsdaine and Michael Shulman. 2020. Semantics of higher inductive types. In *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 169. Cambridge University Press, 159–208.

Lorenzo Malatesta, Thorsten Altenkirch, Neil Ghani, Peter Hancock, and Conor McBride. 2012. Small induction recursion, indexed containers and dependent polynomials are equivalent. (2012).

Per Martin-Löf. 1975. An intuitionistic theory of types: predicative part. In *Studies in Logic and the Foundations of Mathematics*. Vol. 80. Elsevier, Amsterdam, 73–118.

Per Martin-Lof. 1984. *Intuitionistic type theory*. Vol. 6. Bibliopolis Naples.

Lykourgos Mastorou, Nikolaos Papaspyrou, and Niki Vazou. 2022. Coinduction inductively: mechanizing coinductive proofs in Liquid Haskell. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium* (Ljubljana, Slovenia) *(Haskell 2022)*. Association for Computing Machinery, New York, NY, USA, 1–12. `https://doi.org/10.1145/3546189.3549922`

J Peter May. 2006. *The geometry of iterated loop spaces*. Vol. 271. Springer, Berlin, Heidelberg.

Conor McBride. 2001. The derivative of a regular type is its type of one-hole contexts. *Unpublished manuscript* (2001), 74–88.

Conor McBride. 2008. Clowns to the left of me, jokers to the right (pearl) dissecting data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 287–295.

Lambert Meertens. 1986. Algorithmics: towards programming as a mathematical activity. In *Proceedings of the CWI symposium on Mathematics and Computer Science*, Vol. 1. North-Holland Publishing Company, Amsterdam, 289–334.

Aleksey Nogin. 2002. Quotient types: a modular approach. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, Berlin, Heidelberg, 263–280.

Ulf Norell. 2007. Towards a practical programming language based on dependent type theory.

Lawrence C Paulson. 2006. Defining functions on equivalence classes. *ACM Transactions on Computational Logic (TOCL)* 7, 4 (2006), 658–675.

Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation.* Association for Computing Machinery, New York, NY, United States, 159–169.

John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for specifications: predicate subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (1998), 709–720.

Mike Shulman. 2014. Higher inductive-recursive univalence and type-directed definitions. `https://homotopytypetheory.org/2014/06/08/hiru-tdd/`

Oscar Slotosch. 1997. Higher order quotients and their implementation in Isabelle HOL. In *Theorem Proving in Higher Order Logics: 10th International Conference, TPHOLs' 97 Murray Hill, NJ, USA, August 19–22, 1997 Proceedings 10.* Springer, Berlin, Heidelberg, 291–306.

Kristina Sojakova. 2016. *Higher Inductive Types as Homotopy-Initial Algebras.* Ph. D. Dissertation. Carnegie Mellon University.

Arnaud Spiwack and Thierry Coquand. 2010. Constructively finite? , 217-230 pages. `https://inria.hal.science/inria-00503917` Link to the full book here: http://www.unirioja.es/servicios/sp/catalogo/monografias/vr77.shtml.

Simon Thompson. 1986. Laws in miranda. In *Proceedings of the 1986 ACM conference on LISP and functional programming.* Association for Computing Machinery, New York, NY, United States, 1–12.

UniMath. 2021. UniMath library. https://tinyurl.com/2j29fwp2.

The Univalent goundations Program. 2013. *Homotopy type theory: univalent foundations of mathematics.* `https://homotopytypetheory.org/book`, Institute for Advanced Study.

Floris van Doorn, Jakob von Raumer, and Ulrik Buchholtz. 2017. Homotopy type theory in Lean. In *Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings 8*. Springer, Berlin, Heidelberg, 479–495.

Niki Vazou. 2016. *Liquid Haskell: Haskell as a theorem prover.* Ph. D. Dissertation. University of California San Diego.

Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, New York, NY, United States, 48–61.

Niki Vazou and Michael Greenberg. 2022. How to safely use extensionality in Liquid Haskell. In *Proceedings of the 15th ACM SIGPLAN International Haskell Symposium*. Association for Computing Machinery, New York, NY, United States, 13–26.

Niki Vazou, Patrick M Rondon, and Ranjit Jhala. 2013. Abstract refinement types. In *European Symposium on Programming*. Springer, Berlin, Heidelberg, 209–228.

Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. Association for Computing Machinery, New York, NY, United States, 269–282.

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement reflection: complete verification with SMT. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–31.

Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: a dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–29.

Donald Yau. 2018. *Operads of wiring diagrams*. Springer International Publishing. `https://doi.org/10.1007/978-3-319-95001-3`

Brent Abraham Yorgey. 2014. *Combinatorial species and labelled structures.* Ph. D. Dissertation. University of Pennsylvania.