Subtyping Without Reduction

Brandon Hewer and Graham Hutton

School of Computer Science,
University of Nottingham, UK

Abstract. Subtypes are useful and ubiquitous, allowing important properties of data to be captured directly in types. However, the standard encoding of subtypes gives no control over when the reduction of subtyping proofs takes place, which can significantly impact the performance of type-checking. In this article, we show how operations on a subtype can be represented in a more efficient manner that exhibits no reduction behaviour. We present the general form of the technique in Cubical Agda by exploiting its support by higher-inductive types, and demonstrate the practical use of the technique with a number of examples.

1 Introduction

Think of a subtype and some operations on it. We can give you two equivalent representations for the subtype that avoid the cost of subtyping proofs for the operations. The first representation provides an alternative internalisation for the subtyping condition, while the second does the same for the entire subtype. The purpose of this article is to introduce these two representations, explain how they compare, and formalise them in homotopy type theory.

Subtypes appear frequently in type theory, where a subtype of $A : \text{Type}$ can be characterised by a dependent sum $\sum_{(a:A)} P\,(a)$ 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 later on.

Our subtyping technique 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 types. In addition, we retain important computational properties that are absent in solutions such as Agda's abstract definitions. More specifically, the article 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 of our examples are written in Cubical Agda [13], and this is one of the first articles to exploit the power of this system; a library that formalises all of our results is available online [7]. The article is aimed at readers who are familiar with functional programming and (dependent) type theory, but we do not assume experience with homotopy type theory or Cubical Agda, and provide explanations of the necessary concepts where appropriate.

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 $\top$ or the empty type $\bot$.

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 $\mathsf{P} : A \to \mathsf{Type}$, by defining an inductive family $\mathsf{Id}_P : A \to \mathsf{Type}$ with a single constructor $\eta_P : (a : A) \to \mathsf{P}\ a \to \mathsf{Id}_P\ a$. 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.

3 Path types

In this section we review *path types*, a key concept in homotopy type theory (HoTT) that will be used throughout the article. For any type A, the type of *paths* between two terms $x, y : A$ is written $x \equiv_A y$, or simply $x \equiv y$ when A is clear. The underlying definition of a path type varies in different models of HoTT. For example, in the Cohen, Coquand, Huber, and Mörtberg (CCHM) model [3], on which Cubical Agda is based, a path type on elements of A corresponds to a continuous function from the real interval $[0, 1]$ to A.

Readers unfamiliar with HoTT can think of the path type $x \equiv_A y$ as having the same behaviour as the inductively defined identity type. In particular, the eliminator for $\equiv_A$ is given by the J-rule, which states that for any $x : A$, and

family $M : (y : A) \to x \equiv_A y \to \text{Type}$, if we have a proof $t : M\,(x, \mathbf{refl})$ that M is inhabited for reflection on x, then for all $y : A$ and $p : x \equiv_A y$, we can construct a term $J_{M,t}\,(x, y, p) : M\,(x, y, p)$. 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.

4 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 [12]. 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! $n \to$ Type we can lift a function $f : (x : \text{isEven? } n) \to \text{B}\,(\eta\ x)$ on isEven? n to a function $g : (x : \text{isEven! } n) \to \text{B}\ x$ 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 $n \to$ isEven? n maps each constructor of isEven n to its namesake, and $f :$ isEven? $n \to$ isEven n behaves similarly while mapping $\eta\,(\text{even-+}\ p\ q)$ to isEven+ $(f\ p)\ (f\ q)$. Given

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

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

hold definitionally, i.e. they are the defining equations for isEven+, the equalities

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

only hold up to a path given in terms of the squash constructor. Consequently, our definition of 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 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 isEven! and isEven are isomorphic families, one can always be replaced by the other by appropriately transporting along the isomorphism. However, when transporting from 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)) → 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! [8]. The details of each of these constructions for div2 can be found in our Cubical Agda library.

5 Higher-inductive recursive even numbers

In this section we introduce our second approach to the problem identified in Section 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 [6], 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 to$\mathbb{N}$: Even → $\mathbb{N}$. 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 $\sum[\ n \in \mathbb{N}\ ]$ 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 to$\mathbb{N}$, we can recognise that it should behave in a similar manner to the first projection on the type $\sum[\ n \in \mathbb{N}\ ]$ isEven! n, i.e. by mapping every even number to its underlying value as a natural number. With this in mind, to$\mathbb{N}$ can be defined as follows:

```
toℕ zero = 0
toℕ (2+ x) = suc (suc (toℕ x))
```

$$\mathsf{to}\mathbb{N}\ (x +\mathsf{E}\ y) = \mathsf{to}\mathbb{N}\ x + \mathsf{to}\mathbb{N}\ y$$
$$\mathsf{to}\mathbb{N}\ (\mathsf{eq}\ x\ y\ p\ i) = p\ i$$

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 to$\mathbb{N}$. 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 [3], 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 $i : \mathsf{I}$, 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 Even and $\sum[\ n \in \mathbb{N}\ ]\ \mathsf{isEven}\ n$, 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 $\sum[\ n \in \mathbb{N}\ ]\ \mathsf{isEven}\ n$ by constructing a bijection between them. In particular, this is equivalent to constructing an injection $f : \mathsf{Even} \to \mathbb{N}$ together with a family of functions $h : \mathsf{isEven}\ n \to \mathsf{Even}$, 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' [11].

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 un : $(x\ y : \mathsf{U}) \to \mathsf{El}\ x \simeq \mathsf{El}\ y \to x \equiv y$. It can be shown that un $x\ y$ is always an equivalence, and hence El is an embedding. Our approach generalises this idea by considering recursive functions U → 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 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.

6 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 [9] 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 8. 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.

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 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) =
  let p = isEven-2^ n in isEven+ p (isEven+ p even-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 4, 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 [5]. 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.

7 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 [2].

We can alternatively think of $\mathsf{Fin}\ n$ as a subtype of $\mathbb{N}$, 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 $\mathsf{to}\mathbb{N} : \sum[\ k \in \mathbb{N}\ ]\ k < n \to \mathbb{N}$ 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 $\mathsf{suc}\ n$ is by applying the constructor $\mathsf{s<s}$ a total of m times to the constructor $\mathsf{z<s}$.

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 $\neg\mathsf{m<0} : m < 0 \to \bot$ and $\mathsf{<\text{-}suc\text{-}monic} : \mathsf{suc}\ m < \mathsf{suc}\ n \to m < n$. 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 $\mathsf{trans<} : (n : \mathbb{N}) \to m < n \to n < o \to m < o$. We then extend the function $\neg\mathsf{m{<}0}$ by defining $\neg\mathsf{m{<}0}\ (\mathsf{trans<}\ n\ p\ q) = \neg\mathsf{m{<}0}\ q$, and in turn extend the function $\mathsf{<\text{-}suc\text{-}monic}$:

```
<-suc-monic (trans< 0 p q)       = absurd (¬m<0 p)
<-suc-monic (trans< (suc n) p q) = trans< (<-suc-monic p) (<-suc-monic 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 $\mathsf{n{<}sn} : (n : \mathbb{N}) \to n < \mathsf{suc}\ n$. Notably, our definition of $\neg\mathsf{m{<}0}$ does not need to be extended since 0 never unifies with $\mathsf{suc}\ n$, and we can extend $\mathsf{<\text{-}suc\text{-}monic}$ by simply mapping a term $\mathsf{n{<}sn}\ (\mathsf{suc}\ n)$ to $\mathsf{n{<}sn}\ n$.

While we have shown that the elimination principle for $\_<\_$ can be preserved after extension with constructors $\mathsf{trans<}$ and $\mathsf{n{<}sn}$, 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 $\neg\mathsf{m{<}0}$ and $\mathsf{<\text{-}suc\text{-}monic}$ 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 $\mathsf{<\text{-}suc\text{-}monic}$ was extended over the inductive constructors $\mathsf{trans<}$ and $\mathsf{n{<}sn}$, we can see that it does not 'expand' proofs built from these constructors into proofs built only from $\mathsf{z{<}s}$ and $\mathsf{s{<}s}$. 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.

8 IF formalisation

As discussed in Section 6, 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* [10,1], 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*.

8.1 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 = ∑[ c ∈ Com C o ] (∀ r → X (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 = ∑[ (c , f) ∈ ⟦ C ⟧ A o ] (∀ r → 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:

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

8.2 Propositional monads

We now introduce propositional monads, which will be used to formalise our technique. Given any $O : \mathsf{Type}$, 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 $A\ B : O \to \mathsf{Type}$, we can construct both the functorial action of M on morphisms, and its monad multiplication map

$$\mathsf{join} : (B : O \to \mathsf{Type}) \to \forall\ o \to M\ \ (M\ B)\ \ o \to M\ B\ o$$

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

Given propositional monads $F\ G : (O \to \mathsf{Type}) \to O \to \mathsf{Type}$, 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 $B : O \to \mathsf{Type}$. 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

$$\mathsf{isPropM}\ G\ B : (o : O)\ (x\ y : G\ B\ o) \to x \equiv y$$

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

8.3 Free propositional monad

For every indexed container $C : \mathsf{Container}\ O\ O$, 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
  fix : (o : O) → ⟦ C ⟧ (FreePM C P) o → FreePM C P o
  squash : (o : O) (x y : FreePM C P o) → x ≡ y
```

To show that 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 bind $P\ Q\ o\ x\ g : \mathsf{FreePM}\ C\ Q\ o$ follows by induction on x:

```
bind P Q o (η o p) g = g o p
bind P Q o (fix o (c , f)) g = fix o (c , λ r → 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 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 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 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 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 FreePM $C\ P\ o$ by simply mapping $\eta\ o\ p$ to $\eta\ o\ p$ and fix $o\ (c\ ,\ f)$ to:

```
fix o (⟨ α ⟩ (FreePM C B) (c , lift α ∘ 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 FreePM $D\ P\ o$ is a proposition.

To show that 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 FreePM C, which can be defined as follows:

```
unit : (B : O → Type) → ∀ o → ⟦ C ⟧ B o → FreePM C B o
unit B o (c , f) = fix o (s , λ r → η (next C c r) (f r))
```

Furthermore, for all containers $C\ D : \mathsf{Container}\ O\ O$ for which there is a term $\delta : \mathsf{isPropMonad}\ [\![\ D\ ]\!]$ witnessing that $[\![\ D\ ]\!]$ is a propositional monad, and for every morphism $\alpha : C \Rightarrow D$ and family $B : O \to \mathsf{Type}$, we can construct a unique family of functions fold $\alpha\ B : \forall\ \{o\} \to \mathsf{FreePM}\ C\ B\ o \to [\![\ D\ ]\!]\ B\ o$. 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 $\mathsf{fold}\ \alpha\ B\ o\ (\mathsf{unit}\ B\ o\ x) \equiv \langle\ \alpha\ \rangle\ B\ o\ x$ witnessing the left adjunct (fold) interacts with the unit map in the expected way; $\mathsf{fold}\ \alpha\ B$ 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. $\mathsf{FreePM}\ C\ P$, and P itself. Notably, as both P and $\mathsf{FreePM}\ C\ P$ are families of propositions, it suffices to construct a family of functions in both directions. The family from P to $\mathsf{FreePM}\ C\ P$ 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 $\mathsf{FreePM}\ C\ P$ 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, $f : \forall\ \{o\} \to \mathsf{FreePM}\ C\ P\ o \to P\ o$, by mapping $\eta\ o\ p$ to p and $\mathsf{fix}\ o\ (c\ ,\ g)$ to $\alpha\ o\ (c\ ,\ \lambda\ r \to f\ \ (g\ r))$.

8.4 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 6. 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 $\_\sim\_ : A \to A \to \mathsf{Type}$. 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 $\_\sim\_$ is a mere relation, isRelated is a family of h-props. Notably, if $\_\sim\_$ 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 $\_\sim\_$ 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 $\_\sim\_$ 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:

```
next C (filterC P? as) t = as
next C (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 FreePM C isRelated : List A → Type, which we denote by isRelatedF. Importantly, we can prove that for any list xs : List A, the h-props isRelatedF xs and 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 α : (xs : List A) → ⟦ C ⟧ isRelated xs → isRelated xs, by constructing proofs that filtering and removal respect isRelated. Of course, this is only true when the relation _∼_ is transitive.

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

```
filterF : ∀ P? → isRelatedF xs → isRelatedF (filter P? xs),
removeF : ∀ i → isRelatedF xs → isRelatedF (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:

```
filterF P? xs = fix (filter P? xs) (filterC P? xs , λ t → 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.

9 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 [4], but these two approaches are equivalent [6] 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.

9.1 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. $\sum[\ U \in \mathsf{Type}\ ]\ (U \to A)$. 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.

9.2 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 decode C P is an injective function. 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 $f : (x : \mathsf{FreeSTExt}\ C\ P) \to B\ x$ this means that for all $x, y : \mathsf{FreeSTExt}\ C\ P$ and $p : \mathsf{decode}\ C\ P\ x \equiv \mathsf{decode}\ C\ P\ y$ we provide a construction for the path

$$\mathsf{cong}\ f\ (\mathsf{eq}\ x\ y\ p)\ :\ \mathsf{PathP}\ (\lambda\ \mathrm{i} \to B\ (p\ i))\ (f\ x)\ (f\ y).$$

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.

9.3 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

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

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:

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

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 1. In this figure, the variables $i, j, k : \mathsf{I}$ are terms

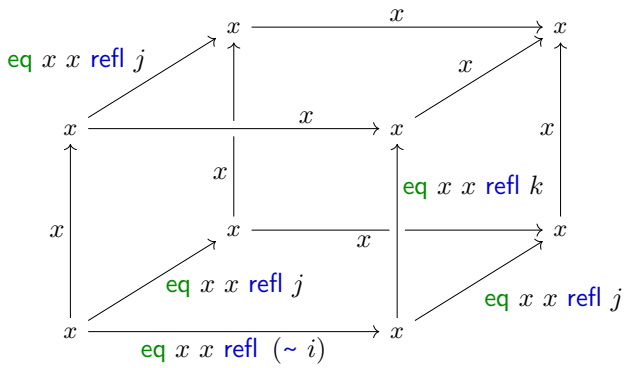


Fig. 1. A cube whose lid is eqRefl

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 ψ 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 $\vee : \mathsf{I} \to \mathsf{I} \to \mathsf{I}$ and $\sim : \mathsf{I} \to \mathsf{I}$, which together with $\wedge : \mathsf{I} \to \mathsf{I} \to \mathsf{I}$ form a De Morgan algebra on the interval type whose laws are given definitionally. To express the side faces of our cube in Cubical Agda, we will construct the following term:

```
sides : (i j k : I) → Partial (i ∨ ~ i ∨ j ∨ ~ j) (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 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:

```
sides i j k (i = i0) = eq x x refl j
sides i j k (i = i1) = eq x x refl (j ∨ k)
sides i j k (j = i0) = eq x x refl (~ i ∨ k)
sides i j k (j = i1) = x
```

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:

```
base : PathP φ (eq x x refl) (eq x x refl)
base i j = eq (eq x x refl (~ i)) x refl j
```

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 Fiber (decode C P) is a family of 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-η : ∀ x → PathP (Ψ x) (fib (unwrap x)) x
unwrap-η (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) (Ψ x j)
sides a x i j (i = i0) = fib (eq (unwrap x) a (unwrap-β x) j)
sides a x i j (i = i1) = unwrap-η 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 : ∀ 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.

9.4 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 $f : \forall\ y \to$ Fiber IR→IF y. We begin by observing that the function $\pi_1 \circ$ IR→IF, i.e. the composition with the first projection, is definitionally equal to decode C P, which we have already shown to have propositional fibers. It is provable in HoTT that for any A B : Type and $C : B \to$ 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 FreePM C P is a family of propositions, it then follows that IR→∑IF has propositional fibers.

To complete the proof that the fibers of IR→IF are contractible, it suffices to construct a term Fiber IR→IF (o, x) for each $(o, x) : \sum[\ o \in O\ ]$ 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. Fiber IR→IF (o, x). For the case where $x = \eta\ o\ p$, we construct the term fib $(\eta\ o\ p)$. If $x =$ fix o $(c\ ,\ g)$, we first construct the following path:

```
p : (o , fix o (c , IR→IF ∘ IF→IR ∘ g)) ≡ (o , fix o (c , g))
p i = o , squash o (fix o (c , IR→IF ∘ IF→IR ∘ g)) (fix o (c , g))
```

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

9.5 Example

We conclude this section by applying the formalisation of the IR encoding of subtypes to the same example used in Section 8.4 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 $\mathsf{isRelated}\ \_\sim\_$, i.e. the type $\mathsf{RelList} = \mathsf{FreeSTExt}\ C\ (\mathsf{isRelated}\ \_\sim\_)$. 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)
```

10 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 $\mathsf{vect}\ (\eta\ as) = as$, this is enough to establish that the constructor η and the function vect witness a family of isomorphisms between $\mathsf{Vec}\ A$ and $\mathsf{AVec}\ A$. Consequently, it is sufficient to establish an equivalence between the types $\sum[\ n \in \mathbb{N}\ ]\ \mathsf{Vec}\ A\ n$ and $\sum[\ n \in \mathbb{N}\ ]\ \mathsf{AVec}\ A\ n$.

By encoding vectors using the type $\mathsf{AVec}\ A$, we can precisely control when concatenation is normalised. In particular, it is possible to define a function $\mathsf{normalise} : \mathsf{AVec}\ A\ n \to \mathsf{AVec}\ A\ n$ 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 $\mathsf{map} : (A \to B) \to \mathsf{AVec}\ A\ n \to \mathsf{AVec}\ B\ n$.

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 $\mathsf{FreeRep}\ C\ A\ \alpha : O \to \mathsf{Type}$ and a recursive function $\mathsf{rec}\ \alpha : \forall\ \{o\} \to \mathsf{FreeRep}\ C\ A\ \alpha\ o \to A\ o$. First, we define $\mathsf{FreeRep}\ C\ A\ \alpha$ by:

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

Then the function $\mathsf{rec}\ \alpha$ is constructed as follows:

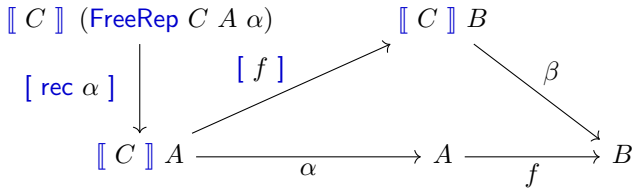
$$
\begin{aligned}
&\mathsf{rec}\ \alpha\ (\eta\ o\ a) = a\\
&\mathsf{rec}\ \alpha\ (\mathsf{fix}\ o\ (c\ ,\ g)) = \alpha\ o\ (c\ ,\ \lambda\ r \to \mathsf{rec}\ \alpha\ (g\ r))\\
&\mathsf{rec}\ \alpha\ (\mathsf{eq}\ o\ c\ g\ i) = \alpha\ o\ (c\ ,\ \lambda\ r \to \mathsf{rec}\ \alpha\ (g\ r))
\end{aligned}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

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

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

The functions $\mathsf{rec}\ \alpha$ and $\eta\ o$ establish an isomorphism between $\mathsf{FreeRep}\ C\ A\ \alpha\ o$ 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:



That is, under the image of $[\ \mathsf{rec}\ \alpha\ ]$, the family of functions f must be an algebra homomorphism between α and β. The construction of the eliminator itself is given in our Cubical Agda library. An interesting application for this idea 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 FreeRep $C\ B\ \beta$ is an algebra homomorphism between the algebras given by rec α and rec β. 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 $\alpha : (o : O) \to [\![\ C\ ]\!]\ A\ o \to A\ o$ we can mutually define a higher inductive type FreeRepIR $C\ A\ \alpha$: Type,

```
data FreeRepIR C A α where
  η : ∀ o → A o → FreeRepIR C A α
  fix : ∀ o → [[ C ]] (Fiber (π₁ ∘ rec α)) o → FreeRepIR C A α
  eq : ∀ o c g → η o (α o (c , recπ₂ α ∘ g)) ≡ fix o (c , g)
```

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

```
recπ₂ α (fib x) = π₂ (rec α x)
```

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

```
rec α (η o a) = o , a
rec α (fix o (c , g)) = o , α o (c , recπ₂ α ∘ g)
rec α (eq o c g i) = o , α o (c , recπ₂ α ∘ g)
```

In a similar manner to the generalisation of the IF approach, the eq path constructor can be extended to a family of paths eq-$\eta\ x$: uncurry η (rec $\alpha\ x$) $\equiv x$. In particular, this means that for all $o : O$, the functions rec α and uncurry η witness an isomorphism between FreeRepIR $C\ A\ \alpha$ and $\sum[\ o \in O\ ]\ A\ o$.

11 Related approaches

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 4.

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 [5]. 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.

12 Conclusion

In this article we have presented a new technique that allows 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 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 [2].

Acknowledgements We would like to thank Nicolai Kraus for many interesting discussions, and the anonymous reviewers for their useful comments and

suggestions. This work was funded by the EPSRC grant EP/P00587X/1, *Mind the Gap: Unified Reasoning About Program Correctness and Efficiency.*

References

1. Altenkirch, T., Ghani, N., Hancock, P., McBride, C., Morris, P.: Indexed Containers. Journal of Functional Programming **25** (2015)
2. Cockx, J., Devriese, D., Piessens, F.: Pattern Matching Without K. In: Proceedings of the 19th ACM SIGPLAN international conference on Functional programming. pp. 257–268 (2014)
3. Cohen, C., Coquand, T., Huber, S., Mörtberg, A.: Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. arXiv preprint arXiv:1611.02108 (2016)
4. Dybjer, P., Setzer, A.: Induction-Recursion and Initial Algebras. Annals of Pure and Applied Logic **124**(1-3) (2003)
5. Gilbert, Gaëtan and Cockx, Jesper and Sozeau, Matthieu and Tabareau, Nicolas: Definitional Proof-Irrelevance without K. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). `https://doi.org/10.1145/3290316`, `https://doi.org/10.1145/3290316`
6. Hancock, P., McBride, C., Ghani, N., Malatesta, L., Altenkirch, T.: Small Induction Recursion. In: International Conference on Typed Lambda Calculi and Applications. Springer (2013)
7. Hewer, B.: Subtyping Cubical Agda Library (2022), available online from `https://tinyurl.com/f8pezwxd`
8. Kraus, N.: Ph.D. thesis, University of Nottingham (2015)
9. Malatesta, L., Altenkirch, T., Ghani, N., Hancock, P., McBride, C.: Small Induction Recursion, Indexed Containers and Dependent Polynomials are Equivalent (2012)
10. Morris, P., Altenkirch, T.: Indexed Containers. In: IEEE Symposium in Logic in Computer Science (2009)
11. Shulman, M.: Higher Inductive-Recursive Univalence and Type-Directed Definitions (Jun 2014), `https://homotopytypetheory.org/2014/06/08/hiru-tdd/`
12. The Univalent Foundations Program: Homotopy Type Theory: Univalent Foundations of Mathematics. Tech. rep., Institute for Advanced Study (2013)
13. Vezzosi, A., Mörtberg, A., Abel, A.: Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types. Journal of Functional Programming **31** (2021)