JFP, pages, 2020.
doi:10.1017/xxxxx

Cubical Agda: A Dependently Typed Programming Language with Univalence and Higher Inductive Types

ANDREA VEZZOSI
IT University of Copenhagen, Denmark. e-mail: avez@itu.dk

ANDERS MÖRTBERG
Stockholm University, Sweden. e-mail: anders.mortberg@math.su.se

ANDREAS ABEL
Chalmers and Gothenburg University, Sweden. e-mail: andreas.abel@gu.se

Abstract

Proof assistants based on dependent type theory provide expressive languages for both programming and proving within the same system. However, all of the major implementations lack powerful extensionality principles for reasoning about equality, such as function and propositional extensionality. These principles are typically added axiomatically which disrupts the constructive properties of these systems. Cubical type theory provides a solution by giving computational meaning to Homotopy Type Theory and Univalent Foundations, in particular to the univalence axiom and higher inductive types. This paper describes an extension of the dependently typed functional programming language Agda with cubical primitives, making it into a full-blown proof assistant with native support for univalence and a general schema of higher inductive types. These new primitives allow the direct definition of function and propositional extensionality as well as quotient types, all with computational content. Additionally, thanks also to copatterns, bisimilarity is equivalent to equality for coinductive types. The adoption of cubical type theory extends Agda with support for a wide range of extensionality principles, without sacrificing type checking and constructivity.

1 Introduction

A core idea in programming and mathematics is *abstraction*: the exact details of how an object is represented should not affect its abstract properties. In other words, the implementation details should not matter. This is exactly what the principle of univalence captures by extending the equality on the universe of types to incorporate equivalent types.[1] This then gives a form of abstraction, or invariance up to equivalence, in the sense that equivalent types will share the same structures and properties. The fact that equality is *proof relevant* in dependent type theory is the key to enabling this; the data of an equality proof

[1] For the sake of this introduction, "equivalent" may be read as "isomorphic". In Homotopy Type Theory (HoTT), *isomorphism* coincides with equivalence for *sets* (in the sense of HoTT). Equivalence for *types* in general is a refinement of the concept of isomorphism.

can store the equivalence, and transporting along this equality should then apply the function underlying the equivalence. In particular, this allows programs and properties to be transported between equivalent types, hereby increasing modularity and decreasing code duplication. A concrete example are the equivalent representations of natural numbers in unary and binary format. In a univalent system it is possible develop the theory of natural numbers using the unary representation, but compute using the binary representation, and as the two representations are equivalent they share the same properties.

The principle of univalence is the major new addition in Homotopy Type Theory and Univalent Foundations (HoTT/UF) (Univalent Foundations Program, 2013). However, these new type-theoretic foundations add univalence as an *axiom* which disrupts the good constructive properties of type theory. In particular, if we transport addition on binary numbers to the unary representation we will not be able to compute with it as the system would not know how to reduce the univalence axiom. Cubical type theory (Cohen *et al.*, 2018) addresses this problem by introducing a novel representation of equality proofs and thereby providing computational content to univalence. This makes it possible to constructively transport programs and properties between equivalent types. This representation of equality proofs has many other useful consequences, in particular function and propositional extensionality (pointwise equal functions and logically equivalent propositions are equal), and the equivalence between bisimilarity and equality for coinductive types (Vezzosi, 2017).

Dependently typed functional languages such as `Agda` (2018), `Coq` (2019), `Idris` (Brady, 2013), and `Lean` (de Moura *et al.*, 2015), provide rich and expressive environments supporting both programming and proving within the same language. However, the extensionality principles mentioned above are not available out of the box and need to be assumed as axioms just as in HoTT/UF. Unsurprisingly, this suffers from the same drawbacks as it compromises the computational behavior of programs that use these axioms, and even make subsequent proofs more complicated.

So far, cubical type theory has been developed with the help of a prototype `Haskell` implementation called `cubicaltt` (Cohen *et al.*, 2015), but it has not been integrated into one of the main dependently typed functional languages. Recently, an effort was made, using `Coq`, to obtain effective transport for restricted uses of the univalence axiom (Tabareau *et al.*, 2018), because, as the authors mention, *"it is not yet clear how to extend [proof assistants] to handle univalence internally"*.

This paper achieves this, and more, by making `Agda` into a cubical programming language with native support for univalence and higher inductives types (HITs). We call this extension `Cubical Agda` as it incorporates and extends cubical type theory. In addition to providing a fully constructive univalence theorem, `Cubical Agda` extends the theory by allowing proofs of equality by copatterns, HITs as in Coquand *et al.* (2018) with nested pattern matching, and interval and partial pre-types. This paper aims to provide a formal account of the extensions to the language of `Agda` and its type-checking algorithm needed to accommodate the new features. In particular, as it requires the most care, we will dedicate a large portion of this paper to the handling of pattern matching.

Contributions. The main contribution of this paper is the implementation of `Cubical Agda`; a fully fledged proof assistant with constructive support for univalence and

HITs. This makes a variety of extensionality principles provable and we show how these can be used for programming and proving in Sect. 2. We explain how `Agda` was extended to support cubical type theory (Sect. 3); in particular, we describe how some primitive notions of cubical type theory are internalized as pre-types: the interval (Sect. 3.1), partial elements, and cubical subtypes (Sect. 3.3). The technical contributions are:

- We extend cubical type theory by records and coinductive types (Sect. 3.2.2).
- We add support for a general schema of HITs, and extend the powerful dependent pattern-matching of `Agda` to also support pattern matching on HITs (Sect. 4).
- We include support for inductive families, which also requires extra care to handle pattern matching definitions (Sect. 4).
- We describe an optimization to the algorithm for transport in `Glue` types (Sect. 5), which gives a simpler proof of the univalence theorem compared to Cohen *et al.* (2018) (Sect. 5.2).

Using the optimization to transport for `Glue` types we discuss an improved canonicity theorem for cubical type theory with HITs (Sect. 6). The paper finishes with some concluding remarks and an overview of related and future work (Sect. 7).

A conference version of this article has appeared at the *International Conference on Functional Programming* (Vezzosi *et al.*, 2019). The support for inductive *families* (rather than just inductive *types*) and the related examples are the novel contributions of the present journal version of this article. At the time of writing the implementation of inductive families has not yet been merged into the main branch of Agda, but it is available through `https://github.com/agda/agda/tree/issue3733`. The required extension to the proof-relevant unifier (see Sect. 4.3.1) does not yet handle unification by injectivity of constructors.

Acknowledgements. The authors are grateful to the anonymous reviewers for their helpful comments on earlier versions of the paper. The second author is also grateful for the feedback from everyone in the `Agda` learning group at CMU and especially to Loïc Pujet for porting the HIT integers from `cubicaltt` and Zesen Qian for the formalization of set quotients. We would also like to thank Martín Escardó for encouraging us to develop the `agda/cubical` library and everyone who has contributed to it since then. Andrea Vezzosi was supported by a research grant (13156) from VILLUM FONDEN, and by USAF, Airforce office for scientific research, award FA9550-16-1-0029. Anders Mörtberg was supported by the Swedish Research Council (SRC, Vetenskapsrådet) under Grant No. 2019-04545. Andreas Abel acknowledges support by the SRC under grants 621-2014-4864 and 2019-04216.

Conflicts of Interest. None.

2 Programming and proving in Cubical Agda

In this section, we show some examples of how the new cubical features in Agda enable interesting and useful ways for both programming and proving in dependent type theory. No expert knowledge of HoTT/UF is assumed. By using univalence and other ideas from HoTT/UF we can:

1. Transfer programs and proofs between equivalent types. (Sect. 2.1.1)
2. Prove properties for proof-oriented datatypes using computation-oriented ones. (Sect. 2.1.2)
3. Reason about dependently typed programs using inductive families. (Sect. 2.2)
4. Treat bisimilar elements of coinductive types as equal. (Sect. 2.3)
5. Define and reason about quotient types. (Sect. 2.4)
6. Represent topological spaces as datatypes and reason about them synthetically. (Sect. 2.5)

The examples are taken from the open-source library agda/cubical hosted at https://github.com/agda/cubical.

2.1 Unary and Binary Numbers

An example of two equivalent types that are well-suited for different tasks are unary and binary numbers. The unary representation is useful for proving because of its direct induction principle and the binary representation is much better for computation as it is exponentially more compact. By utilizing computational univalence, we can transfer results between the two representations in a convenient way; this gives us the best of both worlds without having to duplicate results.

The unary numbers, ℕ, are built into Agda and are inductively generated by the constructors zero and suc (successor). We encode binary numbers as:

```
data Bin : Set where
  bin0   : Bin
  binPos : Pos → Bin

data Pos : Set where
  pos1 : Pos
  x0   : Pos → Pos
  x1   : Pos → Pos
```

A binary number is hence either zero (bin0) or a positive binary number represented as a list of zeroes and ones with no trailing zeroes. Least significant bits come first (little-endian format), thus, the number 6 is binary 011 (binPos (x0 (x1 pos1))). This way, every number has a unique binary representation, and it is straightforward to write maps to and from the unary representation (Bin→ℕ and ℕ→Bin) with proofs that they cancel (ℕ→Bin→ℕ and Bin→ℕ→Bin). This means that the two types ℕ and Bin are isomorphic which implies that they are *equivalent*, in the sense of the terminology of Voevodsky (2015) and Univalent Foundations Program (2013). Spelled out, a map $f : A \to B$ is an equivalence if the preimage of any point in B is a singleton type.

Given types A and B we write $A \simeq B$ for the type of equivalences between them, the univalence theorem[2] then states that

$$(A \equiv B) \simeq (A \simeq B).$$

In particular there is a function $\mathsf{ua} : A \simeq B \to A \equiv B$, sending a proof that two types are equivalent to an equality between these types. We use ua to turn the equivalence of $\mathbb{N}$ and Bin into an equality.

$$\begin{aligned}
&\mathbb{N}{\simeq}\mathsf{Bin} : \mathbb{N} \simeq \mathsf{Bin}\\
&\mathbb{N}{\simeq}\mathsf{Bin} = \mathsf{isoToEquiv}\ (\mathsf{iso}\ \mathbb{N}{\to}\mathsf{Bin}\ \mathsf{Bin}{\to}\mathbb{N}\ \mathsf{Bin}{\to}\mathbb{N}{\to}\mathsf{Bin}\ \mathbb{N}{\to}\mathsf{Bin}{\to}\mathbb{N})
\end{aligned}$$

$$\begin{aligned}
&\mathbb{N}{\equiv}\mathsf{Bin} : \mathbb{N} \equiv \mathsf{Bin}\\
&\mathbb{N}{\equiv}\mathsf{Bin} = \mathsf{ua}\ \mathbb{N}{\simeq}\mathsf{Bin}
\end{aligned}$$

In fact, the equality in $\mathbb{N}{\equiv}\mathsf{Bin}$ is not the regular type-theoretic equality à la Martin-Löf (in the sense of being inductively generated from constructor refl for reflexivity), but rather a *path* equality. The core idea of HoTT/UF is the close correspondence between proof-relevant equality, as in type theory, and paths, as in topology. The idea that equality corresponds to paths is taken very literally in cubical type theory; by adding a primitive interval type I, paths in a type A can be represented as functions $\mathsf{I} \to A$. Iterating these function types lets us represent squares, cubes, and hypercubes; making the type theory *cubical*.

The interval I has two distinguished endpoints i0 and i1. Since paths are functions, we introduce them using λ-abstraction and eliminate them using function application; by applying a path to i0 we get its left endpoint and by applying it to i1 we get the right one. We often want to specify the endpoints of a path (or, more generally, the boundary of a cube) in its type; in `Cubical Agda`, there is a special primitive for this:

$$\mathsf{PathP} : (A : \mathsf{I} \to \mathsf{Set}\ \ell) \to A\ \mathsf{i0} \to A\ \mathsf{i1} \to \mathsf{Set}\ \ell$$

we introduce these paths by lambda abstractions like so, $\lambda i \to t : \mathsf{PathP}\ A\ (t[\mathsf{i0}/i])\ (t[\mathsf{i1}/i])$, provided that $t : A\ i$ for $i : \mathsf{I}$. Consequently, we can apply $p : \mathsf{PathP}\ A\ a_0\ a_1$ to an $r : \mathsf{I}$ to obtain $p\ r : A\ r$. Also, no matter how p is given, we have that $p\ \mathsf{i0}$ reduces to a_0 and $p\ \mathsf{i1}$ reduces to a_1.

The PathP types should be thought of as heterogeneous equalities since the two endpoints are in different types; this is similar to the dependent paths in HoTT (Univalent Foundations Program, 2013, Sect. 6.2). We can define homogeneous non-dependent path equality in terms of PathP as follows:

$$\begin{aligned}
&_{\equiv}_ : \{A : \mathsf{Set}\ \ell\} \to A \to A \to \mathsf{Set}\ \ell\\
&_{\equiv}_\ \{A = A\}\ x\ y = \mathsf{PathP}\ (\lambda\ _ \to A)\ x\ y
\end{aligned}$$

In the previous definition, the syntax $\{A = A\}$ tells `Agda` to bind the hidden argument A of $\_{\equiv}\_$ (the first A in $\{A = A\}$) to a variable A (the second A) that can be used on the right

[2] As the univalence "axiom" is provable in `Cubical Agda` we refer to it as the *univalence theorem*.

hand side. Note also that some definitions are polymorphic in universe level ℓ which is implicitly universally quantified. Further, from now we will omit the explicit quantification over $\{A : \mathsf{Set}\ \ell\}$.

Viewing equalities as functions out of the interval allows us to reason elegantly about equality; for instance, the constant path represents a proof of reflexivity.

$$\begin{aligned}
&\mathsf{refl} : \{x : A\} \to x \equiv x\\
&\mathsf{refl}\ \{x = x\} = \lambda\ i \to x
\end{aligned}$$

We can also directly apply a function to a path in order to prove that dependent functions respect path equality, as shown in the definition of cong below. Simply by computation cong satisfies some new definitional equalities compared to the corresponding definition for the inductive equality type à la Martin-Löf (1975). For instance, cong is functorial by definition: we can prove congId and congComp by plain reflexivity (refl).

$$\begin{aligned}
&\mathsf{cong} : \forall\ \{B : A \to \mathsf{Set}\ \ell\}\ (f : (a : A) \to B\ a)\ \{x\ y\}\ (p : x \equiv y) \to\\
&\qquad \mathsf{PathP}\ (\lambda\ i \to B\ (p\ i))\ (f\ x)\ (f\ y)\\
&\mathsf{cong}\ f\ p\ i = f\ (p\ i)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{congId} : \forall\ \{x\ y : A\}\ (p : x \equiv y) \to \mathsf{cong}\ (\lambda\ a \to a)\ p \equiv p\\
&\mathsf{congId}\ p = \mathsf{refl}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{congComp} : \forall\ (f : A \to B)\ (g : B \to C)\ \{x\ y\}\ (p : x \equiv y) \to\\
&\qquad\qquad \mathsf{cong}\ (g \circ f)\ p \equiv \mathsf{cong}\ g\ (\mathsf{cong}\ f\ p)\\
&\mathsf{congComp}\ f\ g\ p = \mathsf{refl}
\end{aligned}$$

Path types also let us prove new things that are not provable in standard `Agda` with Martin-Löf propositional equality. For example, function extensionality, stating that pointwise equal functions are equal themselves, has an extremely simple proof:

$$\begin{aligned}
&\mathsf{funExt} : \{f\ g : A \to B\} \to ((x : A) \to f\ x \equiv g\ x) \to f \equiv g\\
&\mathsf{funExt}\ p\ i\ x = p\ x\ i
\end{aligned}$$

The proof of function extensionality for dependent and n-ary functions is equally direct. Since funExt is a *definable* notion in `Cubical Agda`, it is, in contrast to Martin-Löf Type Theory, not an axiom. This means that it has computational content: it simply swaps the arguments to p.

The facts that paths can be manipulated as functions and that we have heterogeneous path types makes many equality proofs simpler compared to the corresponding proofs in standard `Agda` or HoTT. For instance, the equality of second projections of dependent pair types is a simple instance of cong:

$$\begin{aligned}
&\Sigma\text{-}\mathsf{eq}_2 : \forall\ \{A : \mathsf{Set}\}\ \{B : A \to \mathsf{Set}\}\ \{p\ q : \Sigma[\ x \in A\ ]\ (B\ x)\} \to (e : p \equiv q) \to\\
&\qquad\qquad \mathsf{PathP}\ (\lambda\ i \to B\ (e\ i\ .\mathsf{fst}))\ (p\ .\mathsf{snd})\ (q\ .\mathsf{snd})\\
&\Sigma\text{-}\mathsf{eq}_2 = \mathsf{cong}\ \mathsf{snd}
\end{aligned}$$

The corresponding result in regular type theory has to be stated with the homogeneous notion of equality using transport, making equality in dependent pair types notoriously difficult to work with.

2.1.1 *Univalent Transport*

One of the key properties of type-theoretic equality is *transport*.

$$\mathsf{transport} : A \equiv B \to A \to B$$
$$\mathsf{transport}\ p\ a = \mathsf{transp}\ (\lambda\ i \to p\ i)\ \mathsf{i0}\ a$$

This is defined using another primitive of `Cubical Agda` called transp. It is a generalization of the regular transport principle which lets us specify where the transport is the identity function. In particular, when the second argument to transp is i1 it will reduce to a, which let us prove that $\mathsf{transp}\ A\ r\ a$ is always path equal to a (cf. addp later). A consequence is that whenever we have an equivalence $e : A \simeq B$ we have that $\mathsf{transport}\ (\lambda\ i \to F\ (\mathsf{ua}\ e\ i))$ is an equivalence as well. This ability to lift equivalences through arbitrary type operators F is an easily overlooked benefit of a language with computational univalence.

The substitution principle is obtained as an instance of transport.

$$\mathsf{subst} : (B : A \to \mathsf{Set}\ \ell)\ \{x\ y : A\} \to x \equiv y \to B\ x \to B\ y$$
$$\mathsf{subst}\ B\ p\ b = \mathsf{transport}\ (\lambda\ i \to B\ (p\ i))\ b$$

Function subst invokes transport with a proof of $B\,x \equiv B\,y$; this proof $\lambda\ i \to B\ (p\ i)$ is an inlining of cong, stating that families B respect equality p.

After this digression about path types, let us revisit the ℕ≡Bin path. Using transport, we can transfer zero along ℕ≡Bin, and since univalence is a theorem with computational content in `Cubical Agda`, this will reduce to bin0. In contrast, if we were working in a system with axiomatic univalence, we could still define ℕ≡Bin, but the transport of zero along that equality would be *stuck*; the system would not know how to automatically transport with the ua constant.

Having computational univalence lets us do a lot more than just transporting constructors. We can for example transport the *addition* function from unary to binary numbers in order to make it easier to prove properties about the more complex binary addition function:

$$\_\mathsf{+Bin}\_ : \mathsf{Bin} \to \mathsf{Bin} \to \mathsf{Bin}$$
$$\_\mathsf{+Bin}\_ = \mathsf{transport}\ (\lambda\ i \to \mathbb{N}{\equiv}\mathsf{Bin}\ i \to \mathbb{N}{\equiv}\mathsf{Bin}\ i \to \mathbb{N}{\equiv}\mathsf{Bin}\ i)\ \_{+}\_$$

In this case, the path that we transport along is between the function types $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $\mathsf{Bin} \to \mathsf{Bin} \to \mathsf{Bin}$. This way, we obtain an addition function on binary numbers and the fact that ua has computational content lets us run it:

$$\_ : (\mathsf{binPos}\ (\mathsf{x0}\ (\mathsf{x0}\ \mathsf{pos1})))\ \mathsf{+Bin}\ (\mathsf{binPos}\ \mathsf{pos1}) \equiv \mathsf{binPos}\ (\mathsf{x1}\ (\mathsf{x0}\ \mathsf{pos1}))$$
$$\_ = \mathsf{refl}$$

In order to reduce the left hand side `Cubical Agda` will convert all of the arguments to the unary representation, add them using _+_ and then convert the result back to binary. The main reason for defining _+Bin_ like this is that it lets us transport results about the unary addition function. For example, we transport the proof of associativity as follows:

$$\mathsf{addp}\ :\ \mathsf{PathP}\ (\lambda\ i \to \mathbb{N}{\equiv}\mathsf{Bin}\ i \to \mathbb{N}{\equiv}\mathsf{Bin}\ i \to \mathbb{N}{\equiv}\mathsf{Bin}\ i)\ \_{+}\_\ \_\mathsf{+Bin}\_$$
$$\mathsf{addp}\ i = \mathsf{transp}\ (\lambda\ j \to \mathbb{N}{\equiv}\mathsf{Bin}\ (i \wedge j) \to \mathbb{N}{\equiv}\mathsf{Bin}\ (i \wedge j) \to \mathbb{N}{\equiv}\mathsf{Bin}\ (i \wedge j))\ ({\sim}\ i)\ \_{+}\_$$

```
+Bin−assoc : (m n o : Bin) → m +Bin (n +Bin o) ≡ (m +Bin n) +Bin o
+Bin−assoc =
  transport (λ i → (m n o : ℕ≡Bin i) → addp i m (addp i n o)
                                        ≡ addp i (addp i m n) o)
            +−assoc
```

In addp we utilize the interval operators *minimum* (_∧_) and *reversal* (~_); further, `Cubical Agda` features the *maximum* operator (_∨_). The intuition is that elements of I correspond to points in the real unit interval $[0, 1]$. The _∧_ and _∨_ operations take the minimum and maximum of i, j : I while the reversal operation computes $1 - i$. These operations satisfy the laws of a *De Morgan algebra*. This means, for one, that the min/max operations form a bounded distributive lattice, with i0 and i1 as bottom and top elements. Further, the reversal is a De Morgan involution, so, for instance, $\sim(i \wedge j) = \sim i \vee \sim j$. Note that this is still *not* a Boolean algebra, since $i \wedge \sim i =$ i0 and $i \vee \sim i =$ i1 are *not* valid for points of the unit interval, except for the endpoints.

Let us assert the well-typedness of addp: When i is i0, the first argument to transp in addp is constantly ℕ → ℕ → ℕ, since i0 ∧ i is then just i0 and ℕ≡Bin i0 reduces to ℕ. The second argument ~ i becomes i1 so that the left endpoint of the path is _+_, exploiting that transp (...) is the identity function when applied to i1. On the other hand, when i is i1, then addp i reduces to transp (λ j → ℕ≡Bin j→ℕ≡Bin j→ℕ≡Bin j) i0 _+_ which is exactly the definition of _+Bin_. This establishes that addp indeed constitutes a path from _+_ to _+Bin_. Note that the path type of addp is heterogeneous as the two addition functions have different types.

The desired result is then obtained by transporting the proof that unary addition is associative along a path from

$$(m\ n\ o : \mathbb{N}) \to m + (n + o) \equiv (m + n) + o$$

to

$$(m\ n\ o : \mathsf{Bin}) \to m +\mathsf{Bin}\ (n +\mathsf{Bin}\ o) \equiv (m +\mathsf{Bin}\ n) +\mathsf{Bin}\ o\ .$$

The above proof might seem quite complex and the reader might rightfully question the scalability to more involved examples. However, one can largely simplify things by using subst in a suitable Σ-type:

```
T : Set → Set
T X = Σ[ _+_ ∈ (X → X → X) ] ((x y z : X) → x + (y + z) ≡ (x + y) + z)

TBin : T Bin
TBin = subst T ℕ≡Bin (_+_ , +−assoc)

_+Bin′_ : Bin → Bin → Bin
_+Bin′_ = fst TBin

+Bin′−assoc : (m n o : Bin) → m +Bin′ (n +Bin′ o) ≡ (m +Bin′ n) +Bin′ o
+Bin′−assoc = snd TBin
```

The operation _+Bin′_ is *definitionally* that same as _+Bin_. The user hence doesn't have to write the proof of +Bin−assoc by hand, but `Cubical Agda` can compute the addition operation with its associativity proof for them. It is now easy to imagine automatic transport of more complex operations and properties simply by modifying the type family T.

As discussed above, the _+Bin_ operation is of course a very inefficient way of adding binary numbers. However, we can also define an efficient addition function _+B_ as follows:

```
mutual
  _+P_ : Pos → Pos → Pos
  pos1 +P y     = sucPos y
  x0 x  +P pos1 = x1 x
  x0 x  +P x0 y  = x0 (x +P y)
  x0 x  +P x1 y  = x1 (x +P y)
  x1 x  +P pos1 = x0 (sucPos x)
  x1 x  +P x0 y  = x1 (x +P y)
  x1 x  +P x1 y  = x0 (x +PC y)

  -- Add with carry
  _+PC_ : Pos → Pos → Pos
  pos1 +PC pos1 = x1 pos1
  pos1 +PC x0 y  = x0 (sucPos y)
  pos1 +PC x1 y  = x1 (sucPos y)
  x0 x  +PC pos1 = x0 (sucPos x)
  x0 x  +PC x0 y  = x1 (x +P y)
  x0 x  +PC x1 y  = x0 (x +PC y)
  x1 x  +PC pos1 = x1 (sucPos x)
  x1 x  +PC x0 y  = x0 (x +PC y)
  x1 x  +PC x1 y  = x1 (x +PC y)

  _+B_ : Bin → Bin → Bin
  bin0      +B y        = y
  x         +B bin0     = x
  binPos x +B binPos y = binPos (x +P y)
```

This function is rather complicated as the helper function _+P_ for adding positive numbers has to be defined mutually with an addition with carry operation _+PC_ in order to be efficient. We don't expect the reader to understand the details, but it should be clear that directly proving properties like associativity for this operation would be very complicated as the deeply nested pattern-matching would quickly lead to an explosion of cases. Luckily, we can take advantage of the naive addition operation _+Bin_ which we know share all properties of the unary addition _+_. The key lemma we need to prove is:

```
ℕ→Bin−+B : (x y : ℕ) → ℕ→Bin (x + y) ≡ ℕ→Bin x +B ℕ→Bin y
```

As this lemma is expressed by quantification over unary numbers we largely avoid the explosion of cases. Combining function extensionality with the fact that ℕ→Bin constitutes one direction of an equivalence we can easily construct a path +B≡+Bin proving that _+B_ and _+Bin_ are equal functions:

```
+B≡+Bin : _+B_ ≡ _+Bin_
+B≡+Bin i x y = goal x y i
  where
  goal : (x y : Bin) → x +B y ≡ ℕ→Bin (Bin→ℕ x + Bin→ℕ y)
  goal x y = (λ i → Bin→ℕ→Bin x (~ i) +B Bin→ℕ→Bin y (~ i))
             • sym (ℕ→Bin−+B (Bin→ℕ x) (Bin→ℕ y))
```

The • operation is binary composition of paths which will be discussed in detail in Sect. 3.4. Finally, as the functions are proved equal they share the same properties; for example we can turn the proof of +Bin−assoc into a proof that _+B_ is also associative as follows:

```
+B−assoc : (m n o : Bin) → m +B (n +B o) ≡ (m +B n) +B o
+B−assoc m n o = (λ i → +B≡+Bin i m (+B≡+Bin i n o))
                   ∙ +Bin−assoc m n o
                   ∙ (λ i → +B≡+Bin (~ i) (+B≡+Bin (~ i) m n) o)
```

This example shows how univalent transport can be used to reason conveniently about efficient functions on computation-oriented types which would otherwise have been very complicated to do directly. Another useful consequence of being able to transport proofs between types is that we can prove some result by computation for binary numbers and then transport the proof to the unary representation where the computation might have been infeasible.

2.1.2 Univalent Program and Data Refinements

Sometimes concrete computations are necessary in proofs. For example one could imagine a situation where one needs to check an equality between two terms that are expensive to compute like:

$$2^{20} \cdot 2^{10} = 2^{5} \cdot 2^{15} \cdot 2^{10} \,.$$

When this is part of a proof, it is likely that one is using a unary representation of natural numbers. However, that makes it impossible to verify the above equation by computation. One way to resolve this dilemma would be to redo the formalization using binary numbers, but that could involve a complete rewrite of the formalization. Another alternative would be to use algebraic manipulations to prove the above equality manually. However, the latter is sometimes not feasible as the computation might be very complicated.

Such issues can be resolved by what we call univalent *program and data refinements* following Cohen *et al.* (2013). As binary numbers are equivalent to unary numbers we can prove the property for binary numbers by computation and then transport the proof to unary numbers. To this end, we define a “doubling structure” in which we can express the above equation, and instantiate with unary and binary numbers. We omit the concrete definitions, but as expected the doubling function is of linear complexity for unary numbers and constant for binary.

```
record Double (A : Set) : Set where
  constructor doubleStruct
  field
    double : A → A
    elt : A

Doubleℕ : Double ℕ
Doubleℕ = doubleStruct doubleℕ 1024

DoubleBin : Double Bin
DoubleBin = doubleStruct doubleBin bin1024
```

The equality between binary and unary numbers lifts to an equality of doubling structures. We omit the details of this definition as they are quite technical, however the whole definition is only 7 lines of code so it is not particularly difficult to write once one is familiar with all of the features of `Cubical Agda`.

```
DoubleBin≡Doubleℕ : PathP (λ i → Double (Bin≡ℕ i)) DoubleBin Doubleℕ
```

We can now formulate the equation that we originally wanted to verify. We wrap the equation in a record and use copattern matching when proving it for the Bin instance (DoubleBin). This technical trick prevents `Agda` from eagerly unfolding the ℕ instance when we transport the proof over to unary numbers along the equality of doubling structures. Sect. 3.2.2 discusses transport in record types.

```
doubles : {A : Set} (D : Double A) → ℕ → A → A
doubles D n x = iter n (double D) x

record propDouble {A : Set} (D : Double A) : Set where
  field
    proof : doubles D 20 (elt D) ≡ doubles D 5 (doubles D 15 (elt D))

propDoubleBin : propDouble DoubleBin
proof propDoubleBin = refl

propDoubleℕ : propDouble Doubleℕ
propDoubleℕ = transport (λ i → propDouble (DoubleBin≡Doubleℕ i))
                        propDoubleBin
```

The fact that equivalences of types lift to equivalences of structures is called the *structure identity principle* (SIP) in HoTT/UF (Univalent Foundations Program, 2013, Sect. 9.8). Combining this with univalence lets us lift equalities of types to equalities of structures on these types. This was originally formalized in `Agda` for algebraic structures and isomorphisms by Coquand & Danielsson (2013). Recently another variation of the SIP, due to Escardó (2019), was implemented in `Cubical Agda` by Angiuli *et al.* (2020). This cubical SIP extracts the pattern described here so that a user need not repeat this construction when considering new structures. Using the cubical SIP, Angiuli *et al.* (2020) have developed a variety of more substantial examples from computer science and mathematics, including for example queues and finite multisets.

2.2 Inductive families

When programming with dependent types in `Agda`, it is very common to use inductive families as they allow the programmer to encode various information using indices in the type. The classic example is vectors—length indexed lists—which allow the programmer to write for example a safe head function that extracts the first element of a non-empty list. While such types are ubiquitous in dependently typed programming, they also cause some headache when reasoning formally about the functions written using them. For instance, one cannot naively state associativity of concatenation for vectors because the two ways of associating concatenation lead to terms of different type. To even *state* the equation, we need to substitute along the proof of associativity for addition of natural numbers. This can quickly become complicated, making reasoning about dependently typed programs rather bureaucratic. `Cubical Agda` offers a solution to this as the built-in PathP equality is heterogeneous, making it more natural to express such equations. In this section, we will show how this helps with dependently typed programming by developing some basic results about vectors and size indexed matrices.

Another very important example of an inductive family is the equality type. With the recently added support for inductive families to `Cubical Agda` we can work with equality without loosing the benefits of dependent pattern-matching on the reflexivity constructor. Furthermore, we can prove that the equality type is equivalent to the _≡_ type, making it possible to give computational meaning to functional extensionality and univalence expressed using the equality type.

2.2.1 *Vectors*

It is straightforward to define vectors the same way as in for example the `Agda` standard library. Here and in the following, we implicitly quantify over $m\ n\ k$: ℕ.

```
data Vec (A : Set ℓ) : ℕ → Set ℓ where
  []  : Vec A zero
  _::_ : (x : A) (xs : Vec A n) → Vec A (suc n)
```

We can also easily define some operations like vector concatenation:

```
_++_ : Vec A m → Vec A n → Vec A (m + n)
[]        ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

The fact that _++_ is associative can be conveniently expressed using PathP and +−assoc. The proof is just a direct use of pattern-matching and an inlined use of cong:

```
++−assoc : (xs : Vec A m) (ys : Vec A n) (zs : Vec A k) →
           PathP (λ i → Vec A (+−assoc m n k i))
                 (xs ++ (ys ++ zs)) ((xs ++ ys) ++ zs)
++−assoc {m = zero}  []        ys zs   = refl
++−assoc {m = suc m} (x :: xs) ys zs i = x :: ++−assoc xs ys zs i
```

2.2.2 *Matrices*

A common use of vectors is to define matrices; however it is quite difficult to prove properties about functions defined for matrices defined this way. Another representation that is better suited for proving properties is to use functions out of a finite set of indices. To define these, we need standard finite sets which is another example of an inductive family:

```
data Fin : ℕ → Set where
  zero : Fin (suc n)
  suc  : (i : Fin n) → Fin (suc n)
```

Using this we can define two different representations of matrices:

```
FinMatrix : (A : Set ℓ) (m n : ℕ) → Set ℓ
FinMatrix A m n = Fin m → Fin n → A

VecMatrix : (A : Set ℓ) (m n : ℕ) → Set ℓ
VecMatrix A m n = Vec (Vec A n) m
```

It is straightforward to define functions going between the two representations:

```
FinVec→Vec : (Fin n → A) → Vec A n
FinVec→Vec {n = zero}  xs = []
FinVec→Vec {n = suc _} xs = xs zero :: FinVec→Vec (λ x → xs (suc x))

lookup : Fin n → Vec A n → A
lookup zero    (x :: xs) = x
lookup (suc i) (x :: xs) = lookup i xs

Fin→VecMatrix : FinMatrix A m n → VecMatrix A m n
Fin→VecMatrix M = FinVec→Vec (λ i → FinVec→Vec (λ j → M i j))

Vec→FinMatrix : VecMatrix A m n → FinMatrix A m n
Vec→FinMatrix M i j = lookup j (lookup i M)
```

By using funExt we can prove that the functions between FinMatrix and VecMatrix cancel which gives us an equivalence of the two representations. This can then be transformed into a path by applying ua:

```
FinMatrix≡VecMatrix : (A : Set ℓ) (m n : ℕ) → FinMatrix A m n ≡ VecMatrix A m n
FinMatrix≡VecMatrix A m n = ua (FinMatrix≃VecMatrix A m n)
```

We can now do the same kind of transport of properties that we did for unary and binary numbers. Let us assume that we have a commutative ring R. We write + for the additive operation of R and the proof that it is commutative is called commring+−comm. It is then very easy to prove that addition of FinMatrix is commutative:

```
addFinMatrix : (M N : FinMatrix R m n) → FinMatrix R m n
addFinMatrix M N k l = M k l + N k l

addFinMatrixComm : (M N : FinMatrix R m n) →
                       addFinMatrix M N ≡ addFinMatrix N M
addFinMatrixComm M N i k l = +−comm (M k l) (N k l) i
```

Note the inlined use of function extensionality for binary functions in addFinMatrixComm. Following exactly the same recipe as for transporting addition of unary numbers and the fact that it is associative to binary numbers we can now transport addition of FinMatrix and the fact that it is commutative to VecMatrix:

```
T : Set ℓ → Set ℓ
T X = Σ[ _+_ ∈ (X → X → X) ] ((x y : X) → x + y ≡ y + x)

TVecMatrix : T (VecMatrix R m n)
TVecMatrix {m} {n} = subst T (FinMatrix≡VecMatrix R m n)
                             (addFinMatrix , addFinMatrixComm)
```

$$\mathsf{addVecMatrix} : (M\ N : \mathsf{VecMatrix}\ \mathsf{R}\ m\ n) \to \mathsf{VecMatrix}\ \mathsf{R}\ m\ n$$
$$\mathsf{addVecMatrix} = \mathsf{fst}\ \mathsf{TVecMatrix}$$

$$\mathsf{addVecMatrixComm} : (M\ N : \mathsf{VecMatrix}\ \mathsf{R}\ m\ n) \to \\ \mathsf{addVecMatrix}\ M\ N \equiv \mathsf{addVecMatrix}\ N\ M$$
$$\mathsf{addVecMatrixComm} = \mathsf{snd}\ \mathsf{TVecMatrix}$$

Note that there is really no added complexity here compared to the unary and binary numbers example. However, just like we get a naive addition function on binary numbers this way we get a naive one for VecMatrix as well. The addVecMatrix function transports the input matrices to FinMatrix, add them using addFinMatrix and then transport the result back. We can of course do better and define addition for VecMatrix more directly by:

$$\mathsf{addVec} : \mathsf{Vec}\ \mathsf{R}\ m \to \mathsf{Vec}\ \mathsf{R}\ m \to \mathsf{Vec}\ \mathsf{R}\ m$$
$$\mathsf{addVec}\ []\ \ []\ = []$$
$$\mathsf{addVec}\ (x :: xs)\ (y :: ys) = x + y :: \mathsf{addVec}\ xs\ ys$$

$$\mathsf{addVecMatrix}' : (M\ N : \mathsf{VecMatrix}\ \mathsf{R}\ m\ n) \to \mathsf{VecMatrix}\ \mathsf{R}\ m\ n$$
$$\mathsf{addVecMatrix}'\ []\ \ []\ = []$$
$$\mathsf{addVecMatrix}'\ (M :: MS)\ (N :: NS) = \mathsf{addVec}\ M\ N :: \mathsf{addVecMatrix}'\ MS\ NS$$

Using the fact that addVecMatrix is just addFinMatrix with transports back and forth we can prove that it is in fact equal to addVecMatrix′. The proof of this is a little bit more involved so we refer the interested reader to the formalization, however the proof is only about 10 lines of code so it is not that complex.

$$\mathsf{addVecMatrixPath} : (M\ N : \mathsf{VecMatrix}\ \mathsf{R}\ m\ n) \to \\ \mathsf{addVecMatrix}\ M\ N \equiv \mathsf{addVecMatrix}'\ M\ N$$

Combining this with addVecMatrixComm we easily get the fact that the direct definition of addition for vector matrices is commutative:

$$\mathsf{addVecMatrixComm}' : (M\ N : \mathsf{VecMatrix}\ \mathsf{R}\ m\ n) \to \\ \mathsf{addVecMatrix}'\ M\ N \equiv \mathsf{addVecMatrix}'\ N\ M$$

This example shows that the kind of reasoning that we did for unary and binary numbers also works for more complex datastructures like matrices. Furthermore, we can—without too much effort—relate proof-oriented definitions with computation-oriented ones. In proof assistants based on regular dependent theory, this is typically not as easy, since the user has to choose a representation and then stick to it. Even further, the fact that we can use paths to reason about equalities of FinMatrix makes it straightforward to use function extensionality which also simplifies many proofs. If one wants to develop this in standard `Agda` one instead has to resort to either using setoids or define a notion of finite functions represented by their graphs which complicates the definition considerably. For more details about various considerations when developing basic matrix operations in standard `Agda` we refer the interested reader to a blog post by Wood (2019).

2.2.3 Equality

A very important inductive family is the equality type. This is also written $\_\equiv\_$ in the `Agda` standard library, but in order to avoid confusion we call it Eq here.

$$\mathsf{data}\ \mathsf{Eq}\ \{A : \mathsf{Set}\ \ell\}\ (x : A) : A \to \mathsf{Set}\ \ell\ \mathsf{where}$$
$$\quad \mathsf{reflEq} : \mathsf{Eq}\ x\ x$$

With this definition we can define functions by pattern-matching on reflEq just like in regular `Agda`:

$$\mathsf{ap} : (f : A \to A')\ \{x\ y : A\} \to \mathsf{Eq}\ x\ y \to \mathsf{Eq}\ (f\ x)\ (f\ y)$$
$$\mathsf{ap}\ f\ \mathsf{reflEq} = \mathsf{reflEq}$$

More interestingly we can define functions between Eq and paths:

$$\mathsf{eqToPath} : \{x\ y : A\} \to \mathsf{Eq}\ x\ y \to x \equiv y$$
$$\mathsf{eqToPath}\ \mathsf{reflEq} = \mathsf{refl}$$

$$\mathsf{pathToEq} : \{x\ y : A\} \to x \equiv y \to \mathsf{Eq}\ x\ y$$
$$\mathsf{pathToEq}\ \{x = x\}\ p = \mathsf{transport}\ (\lambda\ i \to \mathsf{Eq}\ x\ (p\ i))\ \mathsf{reflEq}$$

It's straightforward to prove that these maps cancel so that we get a path between paths and equalities:

$$\mathsf{Path{\equiv}Eq} : \{x\ y : A\} \to (x \equiv y) \equiv (\mathsf{Eq}\ x\ y)$$
$$\mathsf{Path{\equiv}Eq} = \mathsf{ua}\ \mathsf{Path{\simeq}Eq}$$

This means that these types share all properties expressible in type theory. For example we can prove function extensionality for equality by going back and forth between paths:

$$\mathsf{funExtEq} : \{B : A \to \mathsf{Set}\ \ell'\}\ \{f\ g : (x : A) \to B\ x\} \to$$
$$\qquad ((x : A) \to \mathsf{Eq}\ (f\ x)\ (g\ x)) \to \mathsf{Eq}\ f\ g$$
$$\mathsf{funExtEq}\ p = \mathsf{pathToEq}\ (\lambda\ i\ x \to \mathsf{eqToPath}\ (p\ x)\ i)$$

Similarly, we can also prove univalence and define higher inductive types using equalities instead of paths. This way, we can replace the axioms from existing HoTT `Agda` libraries with concrete terms. The fact that `Cubical Agda` now has proper support for inductive families means that these developments should be able to compute closed terms properly. We are currently experimenting with this and have written some basic examples of computing winding numbers on the circle. For details see the `Cubical.Data.Equality` file in the `agda/cubical` repository. One should also be able to apply the techniques developed by Danielsson (2020).

2.3 Univalence for Coinductive Types

Coinductive types allow the direct manipulation of infinite structures without breaking the consistency of the language. However, in their treatment in `Coq` and `Agda`, reasoning about them was impeded by the inability to prove two elements equal whenever they have the same unfolding, rather than when they are the same by definition (McBride, 2009). Cubical Agda solves this by exploiting the interaction between path and projection copatterns (Abel *et al.*, 2013).

The prototypical example of a coinductive type are infinite streams, which can be declared in Agda as a coinductive record type with two fields: .head and .tail. A function returning a stream is then defined by explaining how it computes when applied to the projections. For example, here we define mapS which applies a function to every element of a stream.

```
record Stream (A : Set) : Set where
  coinductive; constructor _,_
  field
    head : A
    tail  : Stream A

mapS : (A → B) → Stream A → Stream B
mapS f xs .head = f (xs .head)
mapS f xs .tail  = mapS f (xs .tail)
```

The result is that mapS $f\, xs$ by itself will not unfold further. The termination checker is happy to accept this definition as productive since it always reaches a weak head normal form in finite time when applied to projections. As shown in the following proof of the identity law for mapS, Cubical Agda extends the notion of productivity by allowing the same recursion pattern also for paths between streams.

```
mapS−id : (xs : Stream A) → mapS (λ x → x) xs ≡ xs
mapS−id xs i .head = xs .head
mapS−id xs i .tail  = mapS−id (xs .tail) i
```

To define a path between (mapS $(\lambda\, x \to x)\, xs$) and xs we introduce an interval variable i and then are left to define (mapS−id $xs\, i$) of type Stream A, so we can proceed by copatterns and corecursion.

To convince ourselves that mapS−id defines the required path, we note that if mapS−id xs is supposed to be a path between mapS $(\lambda\, x \to x)\, xs$ and xs, then the type of $(\lambda\, i \to$ mapS−id $xs\, i$ .head) should be mapS $(\lambda\, x \to x)\, xs$ .head $\equiv xs$ .head. This type in turn reduces to xs .head $\equiv xs$ .head, by definition of mapS, and so the constant path suffices. A similar reasoning applies to the tail case, this time using the tail clause of mapS to realize that we need a path between mapS $(\lambda\, x \to x)$ (xs.tail) and xs.tail, which we provide with a corecursive call. We give a systematic description of how we compute such *boundary* constraints from the left hand sides of clauses in Sect. 4.

More generally, we can define bisimilarity as a coinductive record and show that two bisimilar streams are equal.

```
record _≈_ (xs ys : Stream A) : Set where
  coinductive
  field
    ≈head : xs .head ≡ ys .head
    ≈tail  : xs .tail  ≈ ys .tail
```

```
bisim : ∀ {xs ys : Stream A} → xs ≈ ys → xs ≡ ys
bisim xs≈ys i .head = xs≈ys .≈head i
bisim xs≈ys i .tail  = bisim (xs≈ys .≈tail) i
```

Finally we note that bisim is actually an equivalence, and so equality of streams is indeed bisimilarity.

```
path≡bisim : ∀ {xs ys : Stream A} → (xs ≡ ys) ≡ (xs ≈ ys)
```

The agda/cubical library contains the complete proof, as well as a proof of the universal property of indexed M-types (Ahrens *et al.*, 2015).

2.4 Quotient Types as Higher Inductive Types

Another major new addition in HoTT are *higher inductive types* (HITs). These are datatypes in which we can specify "higher" constructors representing non-trivial paths of the type (representing identifications of elements), in addition to the normal "point" constructors. These types enable many interesting constructions in type theory, in particular quotient types.

The addition of HITs in systems like Agda or Coq is usually done by postulating their existence, however this suffer from the same issues, in terms of computation, as postulating the univalence axiom. Cubical Agda extends the datatype declarations of Agda to also support a general schema of HITs so that it is not necessary to postulate their existence axiomatically.

In this section, we illustrate how we can use HITs to define quotient types in Cubical Agda. The first example of a quotient type is a very simple encoding of the integers—while this example might seem rather trivial it will help us showcase quite a few interesting possibilities of working with HITs. The second example is a general formulation of quotient types and set quotients.

2.4.1 Integers as a HIT

The integers are often represented as $\mathbb{N} + \mathbb{N}$, however this has the drawback that there are two zeroes (inl 0 and inr 0). This is usually resolved by shifting one of them by 1 (so that for example inl 0 represents -1, etc.), however this can easily lead to confusion and off-by-one errors. A better solution is to identify the two zeroes. This can be achieved with the following HIT.

```
data ℤ : Set where
  pos : (n : ℕ) → ℤ
  neg : (n : ℕ) → ℤ
  posneg : pos 0 ≡ neg 0

sucℤ : ℤ → ℤ
sucℤ (pos n)        = pos (suc n)
sucℤ (neg zero)     = pos 1
sucℤ (neg (suc n)) = neg n
sucℤ (posneg i)     = pos 1
```

This type is similar to $\mathbb{N}+\mathbb{N}$, except that there is also a *path* constructor posneg which identifies the two zeroes. For an element i of the interval type, $\mathsf{posneg}\ i$ is an integer which reduces to $\mathsf{pos}\ 0$ in case $i = \mathsf{i0}$ and $\mathsf{neg}\ 0$ in case $i = \mathsf{i1}$. These so-called *boundary conditions* of posneg have to be respected by any function on $\mathbb{Z}$. For example, the successor function on $\mathbb{Z}$ can be written as above. The final case maps the path constructor constantly to $\mathsf{pos}\ 1$ which is accepted by `Cubical Agda` as the following equations hold definitionally:

$$\mathsf{suc}\mathbb{Z}\ (\mathsf{pos}\ 0) = \mathsf{suc}\mathbb{Z}\ (\mathsf{neg}\ 0) = \mathsf{pos}\ 1.$$

It is direct to define an inverse to $\mathsf{suc}\mathbb{Z}$ (i.e., the predecessor function) and hence get an equivalence from $\mathbb{Z}$ to $\mathbb{Z}$ which, combined with ua, gives a non-trivial path from $\mathbb{Z}$ to $\mathbb{Z}$. Transporting along this path applies the successor function.

```
sucPathℤ : ℤ ≡ ℤ
sucPathℤ = isoToPath (iso sucℤ predℤ sucPredℤ predSucℤ)
```

We can also define addition and prove that addition with a fixed number is an equivalence, however this takes a bit of work as we need to define subtraction and prove that it is the inverse of addition. Using univalence we can take a shortcut and define an alternative addition function so that addition with a fixed number is automatically an equivalence. Consider the following path equality that composes $\mathsf{sucPath}\mathbb{Z}$ with itself n times.

```
addEq : ℕ → ℤ ≡ ℤ
addEq zero    = refl
addEq (suc n) = addEq n • sucPathℤ
```

Similarly we can define a path composing the predecessor path with itself n times. By transporting along these paths we get an addition function.

```
addℤ : ℤ → ℤ → ℤ
addℤ m (pos n)    = transport (addEq n) m
addℤ m (neg n)    = transport (subEq n) m
addℤ m (posneg _) = m
```

By using that transporting along a path is an equivalence we get that addition by a fixed number is an equivalence.

```
isEquivAddℤ : (m : ℤ) → isEquiv (λ n → addℤ n m)
isEquivAddℤ (pos n)    = isEquivTransport (addEq n)
isEquivAddℤ (neg n)    = isEquivTransport (subEq n)
isEquivAddℤ (posneg i) = isEquivTransport refl
```

2.4.2 *General Quotient Types and Set Quotients*

In `Agda` and other dependently typed programming languages, quotient type could so far only be defined axiomatically. Here we show how to define them as a HIT in Cubical Agda.

A first attempt is the following definition.

```
data _/_ (A : Set ℓ) (R : A → A → Set ℓ) : Set ℓ where
  [_] : A → A / R
  eq : (a b : A) → R a b → [ a ] ≡ [ b ]
```

This type has a constructor for mapping elements of A to the quotient by R and an equality identifying the image of each pair of related elements. However this is not exactly what we want because the resulting quotient type might have a too complex notion of equality. For example, if we use this construction to quotient Unit by the total relation, then we will get a type with a point [tt] and an identification of this point with itself. We would expect this to be equivalent to Unit, however it is in fact equivalent to the HIT circle that we will discuss in Sect. 2.5.1. As Unit and the circle do not satisfy the same properties, they are not equivalent; the loop space of Unit, here the type of paths from the only element to itself, is contractible, while the loop space of the circle is $\mathbb{Z}$ (Univalent Foundations Program, 2013).

We get the expected notion of quotients if we switch to *set* quotients. We add another higher constructor that eliminates all of the higher-dimensional structure from the quotient type, in other words, we *set truncate* the type.

```
data _/_ (A : Set ℓ) (R : A → A → Set ℓ) : Set ℓ where
  [_]   : A → A / R
  eq    : (a b : A) → (r : R a b) → [ a ] ≡ [ b ]
  trunc : (x y : A / R) → (p q : x ≡ y) → p ≡ q
```

This makes the quotient into a *recursive* HIT as the trunc constructor quantifies over elements of the type that we are constructing. It forces the quotient to be a set in the sense of satisfying the *uniqueness of identity proofs* (UIP) principle, in other words, any two proofs of equality of members of A / R are equal. Thanks to trunc, we can prove the universal property of set quotients:

```
setQuotUniversal : {A B : Set ℓ} {R : A → A → Set ℓ} → isSet B →
                   (A / R → B) ≃ (Σ[ f ∈ (A → B) ] (∀ a b → R a b → f a ≡ f b))
```

This says that maps out of the set quotient is the same as maps sending related elements to equal elements in the quotient (assuming that the image satisfies UIP). If we furthermore assume that R is a propositional equivalence relation then the set quotients are effective, in the sense that if $[\, a \,] \equiv [\, b \,]$ then also $R\, a\, b$. As an interesting application, we could for example define the positive fractions as a quotient of $\mathbb{N} \times \mathbb{N}$ by relating (n_1, d_1) and (n_2, d_2) if $(n_1 \cdot (1 + d_2) \equiv n_2 \cdot (1 + d_1))$.

2.5 Synthetic Homotopy Theory in Cubical Agda

One of the main applications of HITs in HoTT is the ability to reason *synthetically* about topological spaces inside type theory. This means that we can define topological spaces (like spheres, tori, etc.) as datatypes and reason about them using functional programming. The semantic justification for this is the standard model in Kan simplicial sets, a combinatorial representation of topological spaces (Kapulkin & Lumsdaine, 2012; Lumsdaine & Shulman, 2017). We will discuss how cubical type theory relates to Kan simplicial sets in Sect. 6. In this section, we illustrate how we can do synthetic homotopy theory in `Cubical Agda` by proving that the torus is equivalent to two circles.

2.5.1 *The Torus and Two Circles*

We can define circle and torus as the following higher inductive types.

```
data S¹ : Set where
  base : S¹
  loop : base ≡ base

data Torus : Set where
  point  : Torus
  line1  : point ≡ point
  line2  : point ≡ point
  square : PathP (λ i → line1 i ≡ line1 i) line2 line2
```

The idea is that the circle, S^1, is generated by a base point and a non-trivial path constructor loop connecting base to itself. The Torus on the other hand also has a base point with two non-trivial path constructors connecting it to itself and a square relating the two paths. This square can be illustrated by:

$$\begin{tikzcd}
\mathsf{point} \arrow[r, "\mathsf{line1}"] & \mathsf{point} \\
\mathsf{point} \arrow[u, "\mathsf{line2}"] \arrow[r, "\mathsf{line1}"'] \arrow[ur, phantom, "\mathsf{square}"] & \mathsf{point} \arrow[u, "\mathsf{line2}"']
\end{tikzcd}$$

The idea is that the square constructor identifies line2 with itself over an identification of line1 with itself. This has the effect of identifying the opposite sides of the square, making it into a torus (imagine the square being a sheet of soft paper that one folds so that the opposite sides match).

As we demonstrate in the following, a torus is equivalent to the product of two circles.

```
t2c : Torus → S¹ × S¹
t2c point        = (base   , base)
t2c (line1 i)    = (loop i , base)
t2c (line2 j)    = (base   , loop j)
t2c (square i j) = (loop i , loop j)

c2t : S¹ × S¹ → Torus
c2t (base   , base)   = point
c2t (loop i , base)   = line1 i
c2t (base   , loop j) = line2 j
c2t (loop i , loop j) = square i j
```

The functions back and forth are directly definable by pattern-matching. As a consequence, proving that they are mutually inverse is trivial, and we get an equality between the two types.

```
c2t−t2c : (t : Torus) → c2t (t2c t) ≡ t
c2t−t2c point          = refl
c2t−t2c (line1 _)      = refl
c2t−t2c (line2 _)      = refl
c2t−t2c (square _ _)   = refl

t2c−c2t : (p : S¹ × S¹) → t2c (c2t p) ≡ p
t2c−c2t (base   , base)   = refl
t2c−c2t (base   , loop _) = refl
t2c−c2t (loop _ , base)   = refl
t2c−c2t (loop _ , loop _) = refl
```

```
Torus≡S¹×S¹ : Torus ≡ S¹ × S¹
Torus≡S¹×S¹ = isoToPath (iso t2c c2t t2c−c2t c2t−t2c)
```

This is a rather elementary result in topology. However, it had a surprisingly non-trivial proof in HoTT because of the lack of definitional computation for higher constructors (Sojakova, 2016; Licata & Brunerie, 2015). With the additional definitional computation rules of `Cubical Agda` this proof is now almost entirely trivial.

2.5.2 Further Synthetic Homotopy Theory in Cubical Agda

The `agda/cubical` library contains several further results from synthetic homotopy theory. For instance, we have a direct proof that the fundamental group of the circle is $\mathbb{Z}$, inspired by Licata & Shulman (2013). Combined with the above characterization of the torus it proves that the fundamental group of the Torus is $\mathbb{Z} \times \mathbb{Z}$. The fact that univalence and HITs compute in `Cubical Agda` lets us then compute winding numbers of iterated loops around the circle and torus.

The library also features more substantial results: a proof that $\mathbb{S}^3$, i.e., the four dimensional sphere, is equivalent to the join of two circles, and a proof that the total space of the Hopf fibration is $\mathbb{S}^3$ (Mörtberg & Pujet, 2020). We also have a definition of the "Brunerie number": a number $n \in \mathbb{Z}$ such that $\pi_4(\mathbb{S}^3) \simeq \mathbb{Z}/n\mathbb{Z}$.[3] However, despite considerable efforts we have not been able to reduce n to a normal form yet; even though the absolute value of the expected result is just 2 as proved by Brunerie (2016).

3 Making Agda cubical

In the remainder of the paper we will describe how `Agda` was extended to become cubical. The key additions to `Agda` are:

1. The interval and path types (Sect. 3.1).
2. Generalized transport, transp (Sect. 3.2).
3. Partial elements (Sect. 3.3).
4. Homogeneous composition, hcomp (Sect. 3.4).
5. Higher inductive types (Sect. 4).
6. Glue types (Sect. 5).

[3] For details see `https://github.com/agda/cubical/blob/master/Cubical/Experiments/Brunerie.agda` .

We have already discussed the first two points in some detail in the examples, however, the implementation of the transp operation is especially interesting as it is what makes `Cubical Agda` compute. This operation is defined by cases on the type formers of `Agda`; a contribution of this paper is the extension of these to types that are present in `Agda` but not covered by Cohen *et al.* (2018), namely, record and coinductive types. Furthermore, the way the hcomp operation works in `Cubical Agda` differs from Coquand *et al.* (2018) in a subtle way which enables us to optimize the transp operation for Glue types. These types are what allows us to give computational content to univalence. By optimizing how their composition operation computes we obtain simpler and more efficient proofs of univalence. As discussed in Sect. 6, this also has metatheoretical consequences for the canonicity theorem for HITs.

3.1 The Interval and Path Types

The first thing `Cubical Agda` adds is an interval type I. Then, we add the PathP types that behave like function types out of the interval, but with fixed endpoints. Note that the interval in `Cubical Agda` is not inductively defined, so we cannot pattern match on it. This follows from the intuition that path types are *continuous* functions from the interval into a space so that they cannot provide arbitrarily different results for i0 and i1.

3.2 Generalized Transport

The next key thing that `Cubical Agda` adds is the generalized transport operation.

$$\mathsf{transp} : (A : \mathsf{I} \to \mathsf{Set}\ \ell) \to \mathsf{I} \to A\ \mathsf{i0} \to A\ \mathsf{i1}$$

Given a type line $A : \mathsf{I} \to \mathsf{Set}\ \ell$ and an element at end $\mathsf{A}\ \mathsf{i0}$, the transp operation gives an element at $\mathsf{A}\ \mathsf{i1}$, the other end of the line. This is generalized compared to regular transport in the sense that transp lets us specify where it should behave as the identity function. In particular there is an additional side condition to be satisfied for $\mathsf{transp}\ A\ r\ a$ to type check, which is that A should be a constant function whenever the constraint $r = \mathsf{i1}$ is satisfied. When r is equal to i1 the transp function will compute as the identity function,

$$\mathsf{transp}\ A\ \mathsf{i1}\ a = a.$$

and this would not be sound in general if A was allowed to be a more complex line that is non-constant when $r = \mathsf{i1}$. In case $r = \mathsf{i0}$ there is nothing to check, thus, $\mathsf{transp}\ A\ \mathsf{i0}\ a$ is well-formed for any A, as in the definition of $\mathsf{transport}\ A\ a$.

Internally, the transp operation computes differently for each of the type formers of `Agda`. We will show how this works in the special case of transport, but the general transp operation is not much more complicated. For a detailed type-theoretic presentation of these definitions see Huber (2017). This formulation of the computation rules for cubical type theory is based on a variation of the comp operation of Cohen *et al.* (2018) that was introduced in Coquand *et al.* (2018) in order to support HITs.

3.2.1 Function Types

Given two type lines $A\ B : \mathsf{I} \to \mathsf{Set}$ we seek to transport a function $f : A\ \mathsf{i0} \to B\ \mathsf{i0}$ to a function $A\ \mathsf{i1} \to B\ \mathsf{i1}$. To this end, we compose backward transport $A\ \mathsf{i1} \to A\ \mathsf{i0}$ along A, function f, and forward transport $B\ \mathsf{i0} \to B\ \mathsf{i1}$ along B.

```
transportFun : (A B : I → Set) → (A i0 → B i0) → (A i1 → B i1)
transportFun A B f = transport (λ i → B i) ∘ f ∘ transport (λ i → A (~ i))
```

By evaluating $\mathsf{transport}\ (\lambda\ i \to A\ i \to B\ i)\ f$ we can see that the definition of $\mathsf{transportFun}\ A\ B\ f$ is definitionally the same as the internal definition for how transp computes in `Cubical Agda`.

```
transportFunEq : (A B : I → Set) → (f : A i0 → B i0) →
                   transportFun A B f ≡ transport (λ i → A i → B i) f
transportFunEq A B f = refl
```

The definition for dependent functions is very similar, except that some extra work is required to correct the type in the outer $\mathsf{transport}$. This definition clarifies why we need to consider transp and not just the simpler $\mathsf{transport}$ operation.

```
transportPi : (A : I → Set) (B : (i : I) → A i → Set)
              → ((x : A i0) → B i0 x)
              → ((x : A i1) → B i1 x)
transportPi A B f = λ (x : A i1) →
  transport (λ j → B j (transp (λ i → A (j ∨ ~ i)) j x))
            (f (transport (λ i → A (~ i)) x))
```

If we would have used the same definition as for non-dependent functions the outer $\mathsf{transport}$ would have been ill-typed. The reason is that f has a dependent type, meaning that $f\ x'$ has type $B\ x'$ for $x' := \mathsf{transport}\ (\lambda\ i \to A\ (\sim i))\ x$. The first argument of the outer $\mathsf{transport}$ must hence be a line between $B\ \mathsf{i0}\ x'$ and $B\ \mathsf{i1}\ x$. This line is constructed by abstracting over j and considering $B\ j\ (\mathsf{transp}\ (\lambda\ i \to A\ (j \vee \sim i))\ j\ x)$. When j is $\mathsf{i0}$ this is indeed $B\ \mathsf{i0}\ x'$ and when j is $\mathsf{i1}$ this is $B\ \mathsf{i1}\ x$ by virtue of transp being the identity function when applied to $\mathsf{i1}$.

3.2.2 Records and Coinductive Types

For record types, the $\mathsf{transport}$ operation is computed pointwise, i.e. independently for every field. The only subtlety is when the record is dependent, in which case a similar type correction has to be done as for dependent functions.

As coinductive types in `Agda` are just record types these are handled the same way. In this case, however, we have to consider the issue of productivity, which is taken care of by how $\mathsf{transport}$ for record types unfolds only when projected from. Analogously to the Stream example from Sect. 2.3, we have that $\mathsf{transport}\ (\lambda\ i \to \mathsf{Stream}\ A)\ xs$ will not reduce further, while if we apply $.\mathsf{tail}$ to it we get $\mathsf{transport}\ (\lambda\ i \to \mathsf{Stream}\ A)\ (xs\ .\mathsf{tail})$.[4] Such controlled unfolding generally leads to smaller normal forms so `Cubical Agda` adopts it for record

[4] Productivity for the case of dependent records then relies on the type of later fields only being able to depend on earlier fields.

types in general. The same kind of controlled unfolding is also implemented for other "negative" types like function and path types.

3.2.3 *Datatypes*

The transport operation for inductive datatypes without parameters, for instance, the natural numbers, is trivial as they cannot vary along the interval.

$$\mathsf{transport}\ (\lambda i \to \mathbb{N})\ x = x$$

This would not work for inductive types with parameters like the disjoint union $A + B$ for which the transport operation would need to reduce to the transport operation in A or B depending on the argument.

$$\begin{aligned}
&\mathsf{transportSum} : (A\ B : \mathsf{I} \to \mathsf{Set}) \to A\ \mathsf{i0} + B\ \mathsf{i0} \to A\ \mathsf{i1} + B\ \mathsf{i1}\\
&\mathsf{transportSum}\ A\ B\ (\mathsf{inl}\ x) = \mathsf{inl}\ (\mathsf{transport}\ (\lambda\ i \to A\ i)\ x)\\
&\mathsf{transportSum}\ A\ B\ (\mathsf{inr}\ x) = \mathsf{inr}\ (\mathsf{transport}\ (\lambda\ i \to B\ i)\ x)
\end{aligned}$$

The transp operation for HITs is a bit more involved. It also computes by cases on the argument, but for the higher constructors some extra care has to be taken. In particular, complicated parameterized HITs, like pushouts, require additional endpoint corrections (Coquand *et al.*, 2018, Sect. 3.3.5).

3.2.4 *Inductive Families*

In the case of inductive families we have an extra complication: constructors only target specific indexes. For example it is not clear how to proceed when reducing $\mathsf{transport}\ (\lambda\ i \to \mathsf{Vec}\ A\ (p\ i))\ []$, as $[]$ might not fit the expected result type of $\mathsf{Vec}\ A\ (p\ \mathsf{i1})$. Moreover, we not only have to care about the endpoint, but we should also keep track of the path p, as it might contain computationally relevant information.

To address this problem we adapt the strategy of Cavallo & Harper (2019*a*), adding a constructor to each inductive family to represent a residual index transport. In the case of vectors we have a constructor[5] $\mathsf{transpX}_{\mathsf{Vec}}\ p\ r\ u_0 : \mathsf{Vec}\ A\ (p\ \mathsf{i1})$ for $p : \mathbb{I} \to \mathbb{N}$, $r : \mathbb{I}$ and $u_0 : \mathsf{Vec}\ A\ (p\ \mathsf{i0})$, such that p is constant when $r = \mathsf{i1}$. Like for transp, we have that $\mathsf{transpX}_{\mathsf{Vec}}\ p\ \mathsf{i1}\ u_0$ reduces to u_0. We can then transport constructors like so:

$$\begin{aligned}
\mathsf{transport}\ (\lambda i \to \mathsf{Vec}\ (A\ i)\ (p\ i))\ [] \quad &= \quad \mathsf{transpX}_{\mathsf{Vec}}\ (\lambda i \to p\ i)\ \mathsf{i0}\ []\\
\mathsf{transport}\ (\lambda i \to \mathsf{Vec}\ (A\ i)\ (p\ i))\ (x :: xs) \quad &= \quad \mathsf{transpX}_{\mathsf{Vec}}\ (\lambda i \to p\ i)\ \mathsf{i0}\\
&\quad (\mathsf{transport}\ (\lambda i \to A\ i)\ x :: \mathsf{transport}\ (\lambda i \to \mathsf{Vec}\ (A\ i)\ m)\ xs)
\end{aligned}$$

In the clause for $[]$ the typing lets us assume that $p\ \mathsf{i0}$ is equal to 0, while in the second clause the typing implies $p\ \mathsf{i0}$ is equal to $\mathsf{suc}\ m$ where m is the length of $xs : \mathsf{Vec}\ (A\ \mathsf{i0})\ m$. In both cases $\mathsf{transpX}_{\mathsf{Vec}}$ lets us produce a result at the desired index by storing the path p. The full reduction algorithm for transp on inductive families is an adaptation of the one for the coercion operator from Cavallo & Harper (2019*a*).

In Sect. 4 we will show how definitions by pattern matching can be extended to cover the extra transpX constructor, without the user needing to specify a clause for it.

[5] The X in transpX stands for "indeX".

3.2.5 Path Types

For path types we will need a new operation to provide the computation rules for transport as we need some way to record the endpoints of the path after transporting it. Indeed, consider the following naïve definition:

```
transportPath : (A : I → Set) (x y : (i : I) → A i) → x i0 ≡ y i0 → x i1 ≡ y i1
transportPath A x y p = λ i → transport (λ j → A j) (p i)
```

This might look plausible as a definition, but the resulting path does not have the correct boundary. When i is i0, for instance, the left boundary is transport $(\lambda\ j \to A\ j)\ (x$ i0$)$ and not just x i1. Note that these elements are equal up to a path (using $\lambda\ k \to$ transp $(\lambda\ j \to A\ (j \vee k))\ k\ (x\ k))$, so what we need is a way to compose the result with this path in order to correct the endpoints. To do this we introduce the homogeneous composition operation (hcomp) that generalizes binary composition of paths to n-ary composition of higher dimensional cubes.

3.3 Partial Elements

In order to describe the homogeneous composition operation we need to be able to write partially specified n-dimensional cubes, i.e., cubes where some faces are missing. Given an element of the interval r : I there is a new primitive predicate IsOne r which represents the constraint $r =$ i1. This comes with a proof 1=1 that i1 is in fact equal to i1, i.e., 1=1 : IsOne i1. The type IsOne $(i \vee \sim i)$ corresponds to the formula $(i =$ i1$) \vee (i =$ i0$)$ which represents the two endpoints of the line specified by i, so by considering formulas made out of more variables we can specify the boundary of cubes. The type IsOne r is also proof-irrelevant, meaning that any two of its elements are definitionally equal.

Building on IsOne we have extended `Cubical Agda` with partial cubical types, written Partial $r\ A$. The idea is that Partial $r\ A$ is the type of cubes in A that are only defined when IsOne r holds.[6] Concretely, Partial $r\ A$ is a special version of the function space IsOne $r \to A$ with a more extensional equality: two of its elements are considered judgmentally equal if they represent the same subcube of A. Concretely they are equal whenever they reduce to equal terms for all the possible assignment of variables that make r equal to i1. An example of where this much extensionality is useful is the definition of hfill in Sect. 3.4.

Elements of these partial cubical types are introduced using pattern matching lambdas. For this purpose `Cubical Agda` supports a new form of patterns, here $(i =$ i0$)$ and $(i =$ i1$)$, that specify the cases where IsOne $(i \vee \sim i)$ is true. Similarly to pattern matching on an inductive family, some variables from the context might get refined, in this case i, even if otherwise we would not be able to pattern match on them.

```
partialBool : ∀ i → Partial (i ∨ ~ i) Bool
partialBool i = λ { (i = i0) → true ; (i = i1) → false }
```

The term partialBool should be thought of as a boolean with different values when i is i0 and when i is i1. This is hence just the endpoints of a line and there is no way to connect

[6] Partial is somewhat analogous to constrained set $P \Rightarrow A = \{a \in A \mid P\}$ where $P =$ IsOne r, only that the proof of P matters.

them, since true is not equal to false. The pattern-matching cases must match the interval expression in the type (under the image of IsOne) and if there are overlapping cases then they must agree up to definitional equality. Furthermore, IsOne i0 is actually absurd and lets us define an empty partial element, also known as an "empty system" (Cohen *et al.*, 2018, Sect. 4.2).

```
empty : Partial i0 A
empty = λ { () }
```

`Cubical Agda` also has cubical subtypes as in Cohen *et al.* (2018); given A : Set ℓ and r : I and u : Partial $r\ A$ we can form the type $A\ [\ r \mapsto u\ ]$. A term v of this type is a term of type A that is definitionally equal to u when IsOne r is satisfied.[7] Any term $u : A$ can be seen as a term of type $A\ [\ r \mapsto u\ ]$ that agrees with itself when IsOne r. This observation is incarnated in the introduction principle inS:

```
inS : {r : I} (a : A) → A [ r ↦ (λ _ → a) ]
```

We can also forget that a partial element agrees with u when IsOne r holds. This insight is manifest in the subsumption principle outS:

```
outS : {r : I} {u : Partial r A} → A [ r ↦ u ] → A
```

We have that both outS (inS v) $= v$ and inS (outS v) $= v$ hold if well typed. Moreover, outS $\{r\}$ $\{u\}$ v will reduce to u 1=1 when $r =$ i1.

With all of this cubical infrastructure we can now describe the hcomp operation.

3.4 *Homogeneous Composition*

The homogeneous composition operation generalizes binary composition of paths so that we can compose multiple composable cubes.

```
hcomp : {r : I} (u : I → Partial r A) (u₀ : A [ r ↦ u i0 ]) → A
```

When calling hcomp $u\ u_0$, `Cubical Agda` makes sure that u_0 agrees with u i0 on r; this is specified in the type of u_0. The idea is that u_0 is the base and u specifies the sides of an open box where the side opposite of u_0 is missing. The hcomp operation then gives us the missing side opposite of u_0, which we refer to as the *lid* of the open box. For example binary composition of paths can be written as:

```
_•_ : {x y z : A} → x ≡ y → y ≡ z → x ≡ z
_•_ {x = x} p q i = hcomp (λ j → λ { (i = i0) → x ; (i = i1) → q j }) (inS (p i))
```

Pictorially we are given $p : x \equiv y$ and $q : y \equiv z$, and the composite of the two paths is obtained by computing the dashed lid at the top of the following square.

$$\begin{tikzcd} x \arrow[r, dashed] & z \\ x \arrow[u, "x"] \arrow[r, "\mathsf{inS}\ (p\ i)"'] & y \arrow[u, "q\ j"'] \end{tikzcd} \qquad \begin{tikzcd} {} \\ {} \arrow[u, "j"] \arrow[r, "i"'] & {} \end{tikzcd}$$

[7] In the set-theoretic analogy, $A\ [\ r \mapsto u\ ] = \{a \in A \mid \text{if } r \text{ then } (a = u)\} \subseteq A$, given $u \in \{a \in A \mid r\}$. We have $a \in A\ [\ r \mapsto \lambda\ \_ \to a\ ]$ always for $a \in A$.

As we are constructing a path from x to z along i we have i : I in context and put inS (p i) as bottom line. The direction j is abstracted in the first argument to hcomp and we use pattern-matching to specify the sides.

We can also define homogeneous filling of open boxes as

```
hfill : {r : I} (u : I → Partial r A) (u₀ : A [ r ↦ u i0 ]) → I → A
hfill {r = r} u u₀ i =
  hcomp (λ j → λ { (r = i1) → u (i ∧ j) 1=1 ; (i = i0) → outS u₀ })
        (inS (outS u₀))
```

When i is i0 this is just outS u_0 and when i is i1 it is hcomp ($\lambda\ j \to \lambda$ { (r = i1) → $u\ j$ 1=1 }) u_0 because the absurd face (i0 = i1) gets filtered out. By the extensionality of partial elements, this gives a line along i between outS u_0 and hcomp u u_0 which geometrically corresponds to the filling of an open box as it connects the base with the lid computed using hcomp. The elimination followed by introduction in inS (outS u_0) might look redundant, but it is necessary because the sides of this composition are defined on $r \vee {\sim}$i = i1, while u_0 belongs to a subtype specified on r. In the special case when q is refl the filler of the above square gives us a direct cubical proof that composing p with refl is p.

```
compPathRefl : {x y : A} (p : x ≡ y) → p • refl ≡ p
compPathRefl {x = x} {y = y} p j i =
  hfill (λ _ → λ { (i = i0) → x ; (i = i1) → y }) (inS (p i)) (~ j)
```

This way, we can do even more equality reasoning by directly working with higher dimensional cubes.

By combining hcomp and transp we can define the heterogeneous composition operation of Cohen *et al.* (2018).

```
comp : (A : I → Set ℓ) {r : I} (u : (i : I) → Partial r (A i)) (u0 : A i0 [ r ↦ u i0 ]) → A i1
comp A {r = r} u u0 =
  hcomp (λ i → λ { (r = i1) → transp (λ j → A (i ∨ j)) i (u _ 1=1) })
        (inS (transport (λ i → A i) (outS u0)))
```

With the comp operation we can then finally give the definition of transp for path types.

```
transportPath : (A : I → Set) (x y : (i : I) → A i) → x i0 ≡ y i0 → x i1 ≡ y i1
transportPath A x y p =
  λ i → comp A (λ j → λ { (i = i0) → x j ; (i = i1) → y j}) (inS (p i))
```

The computation rules for hcomp are also defined by cases on the type formers of `Agda`, just like for transp. These are all quite direct to define and we refer the interested reader to Huber (2017) for details. We will note, however, that for HITs and inductive families hcomp ($\lambda\ i \to \lambda$ { (r = i1) → u}) u_0 only reduces to u[i1/i] when r = i1, and is to be considered a canonical element otherwise. Therefore, functions defined by pattern matching on a HIT also have to make progress when provided an element built with hcomp. We will often refer to such an element as hcomp r u u_0 for ease of notation. The next section describes how this can be achieved, in the context of a core type theory for `Cubical Agda`.

4 Pattern Matching with Higher Inductive Types and Inductive Families

Our main technical contribution is an elaboration algorithm for (co)pattern matching definitions in the presence of HITs and path applications. Following Cockx & Abel (2018) we formulate our algorithm as a translation from (co)pattern matching clauses to case trees. The main challenges are generating the computational behavior on hcomp elements of HITs, $\mathsf{transpX}$ elements of inductive families, and making sure clauses for path constructors agree with what the function does at the endpoints of the path. We will consider the equality type as our only inductive family to simplify the presentation of the algorithm, while still being able to illustrate the relevant issues.

4.1 Elaboration by Example

Here we illustrate by example how we can handle the cases for hcomp and $\mathsf{transpX}$.

Consider the function $\mathsf{c2t}$ from Sect. 2.5.1, it is defined by four clauses, which pattern match on a pair of elements of the circle. Recalling that $\mathsf{hcomp}\, r\, u\, u_0$ is also a canonical element of the circle, we can see that we additionally have to cover the following cases:

$$\begin{array}{ll} \mathsf{c2t}\ (\mathsf{hcomp}\, r\, u\, u_0\ , y) & = ?0 \\ \mathsf{c2t}\ (\mathsf{base} \qquad\quad , \mathsf{hcomp}\, r\, u\, u_0) & = ?1 \\ \mathsf{c2t}\ (\mathsf{loop}\, i \qquad\quad , \mathsf{hcomp}\, r\, u\, u_0) & = ?2 \end{array}$$

We can cover the first by setting

$$?0 := \mathsf{hcomp}\ (\lambda\,\{j\ (r = \mathsf{i1}) \to \mathsf{c2t}\ (u\, j\ 1{=}1, y)\})\ (\mathsf{c2t}\ (u_0, y))$$

which not only produces an element of the right type, but also satisfies $?0 = \mathsf{c2t}\ (u\, i\ 1{=}1, y)$ when $r = \mathsf{i1}$, which is required to preserve the equality $\mathsf{hcomp}\ \mathsf{i1}\ u\, u_0 = u\, i\ 1{=}1$. The case ?1 can be solved analogously, while ?2, since it matches on loop, has additional constraints: ?2 should be equal to $\mathsf{c2t}\ (\mathsf{base}, \mathsf{hcomp}\, r\, u\, u_0)$ whenever $i = \mathsf{i0}$ or $i = \mathsf{i1}$. We can satisfy all of these constraints at once by including them in the composition, i.e. setting

$$?2 := \mathsf{hcomp}\ (\lambda\, j \to \lambda \left\{ \begin{array}{l} (r = \mathsf{i1}) \to \mathsf{c2t}\ (\mathsf{loop}\, i, u\, j\ 1{=}1) \\ (i = \mathsf{i0}) \to \mathsf{c2t}\ (\mathsf{base}, \mathsf{hcomp}\, r\, u\, u_0) \\ (i = \mathsf{i1}) \to \mathsf{c2t}\ (\mathsf{base}, \mathsf{hcomp}\, r\, u\, u_0) \end{array} \right\})\ (\mathsf{c2t}\ (\mathsf{loop}\, i, u_0))$$

where all the components of the partial element match up because they are all different specializations of $\mathsf{c2t}\ (\mathsf{loop}\, i, \mathsf{hcomp}\, r\, u\, u_0)$ under the different boundary conditions. In the general case the return type of the function can depend on the HIT argument, so a heterogeneous composition will be necessary.

To illustrate how to handle $\mathsf{transpX}$ elements, let us look at an artificially constrained proof of symmetry:

$$\begin{array}{l} \mathsf{sym0} : (x : \mathbb{N}) \to \mathsf{Eq}\ x\ 0 \to \mathsf{Eq}\ 0\ x \\ \mathsf{sym0}\ .0\ \mathsf{reflEq} = \mathsf{reflEq} \end{array}$$

Ideally we would only have to handle the following extra clause involving $\mathsf{transpX}$:

$$\mathsf{sym0}\ x\ (\mathsf{transpX}\ p\ r\ t) = \ ?0$$

However we find ourselves stuck, because t has type $\mathsf{Eq}\,x\,(p\ \mathsf{i0})$ so we cannot recurse with $\mathsf{sym0}\,x\,t$, as that is not well typed. To get around this, we can match with reflEq against t, which gives us this clause:

$$\mathsf{sym0}\,x\,(\mathsf{transpX}\ p\ r\ \mathsf{reflEq}) = \ ?0$$

Now we know from the typing that p is a path connecting x to $\mathsf{0}$, so we can use it to solve our goal with a transport:

$$?0 := \mathsf{transp}\,(\lambda i \to \mathsf{Eq}\ \mathsf{0}\ (p\ ({\sim}\ i)))\ r\ (\mathsf{sym0}\ \mathsf{0}\ \mathsf{reflEq})$$

It is not by chance that we were able to rely on the value of $\mathsf{sym0}$ at reflEq to solve this goal, but rather an instance of the specialization by unification technique which is used to translate, under certain conditions, definitions by clauses into functions defined solely by eliminators (Cockx & Devriese, 2018). We refer to the algorithm in Sect. 4.3.3 for the general case, including how to handle remaining patterns $\mathsf{transpX}\ p\ r\ (\mathsf{hcomp}\ s\ w\ w_0)$ and $\mathsf{transpX}\ p\ r\ (\mathsf{transpX}\ q\ s\ t_1)$.

4.2 Syntax of the Core Type Theory

We recall some definitions from Cockx & Abel (2018), extended to allow for the new cubical primitives.

Expressions (Fig. 1) are given in spine-normal form, so that the head symbol of an application is easily accessible. Rather than adding the cubical types and operations described in Sect. 3 as new expression formers, we subsume them under $\mathsf{f}\ \bar{e}$ and $\mathsf{c}\ \bar{e}_{\mathsf{c}}$. This happens also in the implementation of Agda, thanks to the pre-existing support for builtins and primitives.

We include a universe level ω for types like $\mathbb{I}$ which do not support transp or hcomp. Eliminations e include, beyond function application to u and projections $.\pi$, *path applications* $@_{u_0,u_1}\ v$. Path applications to interval element v are annotated by the endpoints u_0 and u_1 used for reduction, in case v becomes $\mathsf{i0}$ or $\mathsf{i1}$.

In contrast to ordinary applications, path applications of stuck terms can reduce; for instance, $x\ @_{u_0,u_1}\ \mathsf{i0}$ reduces to u_0. Thus, variable eliminations $x\,\bar{e}$ are not necessarily in weak head normal form. Thanks to HITs, this is not even the case for constructor applications; $\mathsf{c}\ \bar{e}$ might also reduce!

Binary application $\boxed{u\ e}$ is defined as a partial function on the syntax, by β reduction $(\lambda x.u)\ v = u[v/x]$ in case of abstractions, or by accumulating eliminations $(x\,\bar{e})\ e = x\ (\bar{e}, e)$, $(\mathsf{f}\ \bar{e})\ e = \mathsf{f}\ (\bar{e}, e)$, $(\mathsf{c}\ \bar{e})\ e_{\mathsf{c}} = \mathsf{c}\ (\bar{e}, e_{\mathsf{c}})$, and otherwise it is undefined.

Patterns are augmented with path application *copatterns*, also in the spine for constructors.

$$\begin{array}{rcll}
p & ::= & x & \text{variable pattern} \\
 & | & \mathsf{c}\ \bar{q}_{\mathsf{c}} & \text{fully applied constructor pattern} \\
 & | & \lfloor \mathsf{c} \rfloor\ \bar{q}_{\mathsf{c}} & \text{forced constructor pattern} \\
 & | & \lfloor u \rfloor & \text{forced argument} \\
 & | & \emptyset & \text{absurd pattern} \\
 & & & \\
q & ::= & q_{\mathsf{c}} & \text{copattern for constructors} \\
 & | & .\pi & \text{projection copattern} \\
q_{\mathsf{c}} & ::= & p & \text{application copattern} \\
 & | & @_{u_0,u_1}\ i & \text{path application copattern}
\end{array}$$

$$
\begin{array}{lcll}
x, i & & & \text{variables} \\
\ell & ::= & n \mid \omega & \text{universe levels} \\
A, B, u, v & ::= & w & \text{weak head normal form} \\
 & \mid & \mathsf{f}\ \bar{e} & \text{defined function or primitive applied to eliminations} \\
 & \mid & x\ \bar{e} & \text{variable applied to eliminations} \\
 & \mid & \mathsf{c}\ \bar{e}_{\mathsf{c}} & \text{constructor applied to eliminations} \\
W, w & ::= & (x : A) \to B & \text{dependent function type} \\
 & \mid & \mathsf{Set}_\ell & \text{universe } \ell \\
 & \mid & \mathsf{D}\ \bar{u} & \text{datatype fully applied to parameters} \\
 & \mid & \mathsf{R}\ \bar{u} & \text{record type fully applied to parameters} \\
 & \mid & \lambda x.\ u & \text{lambda abstraction} \\
e & ::= & e_{\mathsf{c}} & \text{elimination for constructors} \\
 & \mid & .\pi & \text{projection} \\
e_{\mathsf{c}} & ::= & u & \text{application} \\
 & \mid & @_{u_0,u_1}\ v & \text{path application}
\end{array}
$$

Fig. 1: Syntax of Terms.

Note that we retain the boundary annotations of path applications even in patterns, since we convert copatterns to eliminations, denoted as $\lceil q \rceil$, during type and coverage checking for case trees.

We write $\boxed{\mathrm{PV}(\bar{q})}$ for the set of variables appearing as variable patterns x or as i in a path application copattern. We will also often drop the subscript from e_{c} and q_{c}.

$$
\begin{array}{lcll}
s & ::= & \ominus & \text{status: unchecked} \\
 & \mid & \oplus & \text{status: checked} \\
decl^s & ::= & \mathsf{data}\ \mathsf{D}\ \Delta : \mathsf{Set}_n\ \mathsf{where}\ \overline{con} & \text{datatype declaration} \\
 & \mid & \mathsf{record}\ self : \mathsf{R}\ \Delta : \mathsf{Set}_n\ \mathsf{where}\ \overline{field} & \text{record declaration} \\
 & \mid & \mathsf{definition}\ \mathsf{f} : A\ \mathsf{where}\ \overline{cls^s} & \text{function declaration} \\
con & ::= & \mathsf{c}\ \Delta\ [\bar{i} \mid b] & \text{constructor declaration} \\
b & ::= & \varepsilon \mid (u_0, u_1)\ b & \text{boundary terms} \\
field & ::= & \pi : A & \text{field declaration} \\
cls^{\ominus} & ::= & \bar{q} \hookrightarrow rhs & \text{unchecked clause} \\
cls^{\oplus} & ::= & \Delta \vdash \bar{q} \hookrightarrow u : B & \text{checked clause} \\
rhs & ::= & u & \text{clause body: expression} \\
 & \mid & \mathsf{impossible} & \text{empty body for absurd pattern} \\
\Sigma & ::= & \overline{decl^{\oplus}} & \text{signature}
\end{array}
$$

Fig. 2: Declarations.

The theory is parameterized by a list of declarations Σ whose grammar is shown in Fig. 2. Declaration forms are due to Cockx & Abel (2018) except for datatype constructors $\mathsf{c}\,\Delta\,[\bar{i}\mid b]$. These take a telescope of arguments Δ, i.e. a list of variable typings $(x:A)$, but now also a *boundary* $[\bar{i}\mid b]$, which specifies the dimensions $\bar{i}$ and endpoints b of path constructors. For example, posneg from Sect. 2.4.1 would be specified by $\mathsf{posneg}\ \varepsilon\ [i \mid (\mathsf{pos}\ 0, \mathsf{neg}\ 0)]$. We will write $\boxed{\Delta[\bar{i}\mid b]\to A}$ for the iterated function and path type defined by the following equations:

$$
\begin{aligned}
[\,] \to A &= A\\
[i\,\bar{i}\mid (u_0,u_1)\,b] \to A &= \mathsf{PathP}\ (\lambda i.\,[\bar{i}\mid b]\to A)\ (\lambda \bar{i}.\,u_0)\ (\lambda \bar{i}.\,u_1)\\
(x:B)\Delta[\bar{i}\mid b]\to A &= (x:B)\to \Delta[\bar{i}\mid b]\to A
\end{aligned}
$$

Further, we write $\hat{\Delta}[\bar{i}\mid b]$ for the appropriate sequence of function and path applications. A constructor $\mathsf{c}\,\Delta'\,[\bar{i}\mid b]$ for a datatype $\mathsf{D}\,\Delta$ will then have type $\Delta\Delta'[\bar{i}\mid b]\to \mathsf{D}\,\hat{\Delta}$.

The core definition of Cockx & Abel (2018) is the elaboration judgment for function definitions $\Sigma;\Gamma \vdash P \mid \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma'$ which performs type and coverage checking for the user supplied clauses of f given to the judgment as P. It further computes the corresponding case tree Q and checked clauses $cls^{\oplus}$ in Σ'. Case trees Q are previously specified by a typing judgment $\Sigma;\Gamma \vdash \mathsf{f}\,\bar{q} := Q : C \rightsquigarrow \Sigma'$ which follows the same structure but takes the case tree as input. Whenever the elaboration judgment succeeds, the typing judgment will also hold. In particular, given a signature Σ, a function declaration $\mathsf{definition}\ \mathsf{f}:C\ \mathsf{where}\ \overline{\bar{q}' \hookrightarrow rhs}$ is elaborated by a call to the elaboration judgment where $\Gamma=\varepsilon$, $\bar{q}=\varepsilon$ and $P=\{\bar{q}'_i \hookrightarrow rhs_i \mid i=1\ldots k\}$. In the following, we only present the rules for the case tree typing judgement and refer to the supplemental material for the elaboration judgement.

4.3 Case Trees

Figures 3 and 5 describe the case tree typing judgment $\boxed{\Sigma;\Gamma \vdash \mathsf{f}\,\bar{q} := Q : C \mid \Theta \rightsquigarrow \Sigma'}$. In our version, the judgment takes an extra input Θ which is a possibly empty list of *boundary assignments* α, which in turn are lists of assignments of interval variables to either $\mathsf{i0}$ or $\mathsf{i1}$. We denote with $\boxed{[\alpha]}$ the substitution implied by the equalities in α itself. The list Θ is used to keep track of which faces of the current definition have accumulated some boundary constraints, due to the rules to introduce a path (CtIntroPath), a partial element (CtSplitPartial), or to pattern match on an higher inductive type (CtSplitConHIT).

In the following, we comment on the individual rules of Figure 3:

CtDone A leaf of a case tree consists of a term v of the expected type C. Moreover v has to fulfill the boundary constraints on the faces specified by Θ: for every α_i we require that $\mathsf{f}\,\bar{q}$ and v agree when substituted with $[\alpha_i]$, i.e., when restricted to the face in question. Note that we impose the boundary constraints in the signature Σ where we have not added the clause $\mathsf{f}\,\bar{q}\hookrightarrow v$ yet, so they are non-trivial to satisfy. We use $\boxed{\Gamma_\alpha}$ to denote the context obtained by removing the variables in α from Γ and substituting their occurrences with the specified values.

$\boxed{\Sigma;\Gamma \vdash \mathsf{f}\,\bar{q} := Q : C \mid \Theta \rightsquigarrow \Sigma'}$ Presupposes: $\Sigma;\Gamma \vdash \mathsf{f}\,\lceil \bar{q} \rceil : C$ and $\mathrm{dom}(\Gamma) = \mathrm{PV}(\bar{q})$ and $\Theta = \alpha_1; \ldots; \alpha_n$ where $\alpha ::= \varepsilon \mid (i = \mathsf{i0})\ \alpha \mid (i = \mathsf{i1})\ \alpha$ such that $\Sigma;\Gamma \vdash i : \mathbb{I}$. Checks case tree Q and outputs an extension Σ' of Σ by the clauses represented by "$\mathsf{f}\,\bar{q} \hookrightarrow Q$".

$$\frac{\Sigma;\Gamma \vdash v : C \qquad \Theta = \alpha_1; \ldots; \alpha_n \\ (\Sigma;\Gamma_{\alpha_i} \vdash \mathsf{f}\,\bar{q}[\alpha_i] = v[\alpha_i] : C[\alpha_i])_{i=1\ldots n}}{\Sigma;\Gamma \vdash \mathsf{f}\,\bar{q} := v : C \mid \Theta \rightsquigarrow \Sigma, (\mathsf{clause}\,\Gamma \vdash \mathsf{f}\,\bar{q} \hookrightarrow v : C)}\ \textsc{CtDone}$$

$$\frac{\Sigma;\Gamma \vdash C = (x : A) \to B : \mathsf{Set}_\ell \qquad \Sigma;\Gamma(x : A) \vdash \mathsf{f}\,\bar{q}\,x := Q : B \mid \Theta \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash \mathsf{f}\,\bar{q} := \lambda x.\ Q : C \mid \Theta \rightsquigarrow \Sigma'}\ \textsc{CtIntro}$$

$$\frac{\Sigma;\Gamma \vdash C = \mathsf{PathP}\,B\,u_0\,u_1 : \mathsf{Set}_n \qquad \Theta' = (i = 0); (i = 1); \Theta \\ \Sigma;\Gamma(i : \mathbb{I}) \vdash \mathsf{f}\,\bar{q}\ @_{u_0,u_1}\ i := Q : B\,i \mid \Theta' \rightsquigarrow \Sigma'}{\Sigma;\Gamma \vdash \mathsf{f}\,\bar{q} := \lambda i.\ Q : C \mid \Theta \rightsquigarrow \Sigma'}\ \textsc{CtIntroPath}$$

$$\frac{\Sigma_0;\Gamma \vdash C = \mathsf{PartialP}\,r\,A : \mathsf{Set}_\omega \qquad \Sigma_0;\Gamma \vdash r = \bigvee_i \bigwedge \alpha_i : \mathbb{I} \\ (\Sigma_{i-1};\Gamma_{\alpha_i} \vdash (\mathsf{f}\,\bar{q}[\alpha_i]\ \lfloor 1{=}1 \rfloor) := Q_i : (A\ 1{=}1) \mid (\alpha_1; \ldots; \alpha_{i-1}; \Theta)[\alpha_i] \rightsquigarrow \Sigma_i)_{i=1\ldots n}}{\Sigma_0;\Gamma \vdash \mathsf{f}\,\bar{q} := \mathsf{split}\{\alpha_1 \mapsto Q_1; \ldots; \alpha_n \mapsto Q_n\} : C \mid \Theta \rightsquigarrow \Sigma_n}\ \textsc{CtSplitPartial}$$

$$\frac{\Sigma_0;\Gamma \vdash C = \mathsf{R}\,\bar{v} : \mathsf{Set}_n \qquad \mathsf{record}\ \mathit{self} : \mathsf{R}\,\Delta : \mathsf{Set}_n\ \mathsf{where}\ \overline{\pi_{\mathsf{i}} : A_i} \in \Sigma_0 \\ \sigma = [\bar{v} \,/\, \Delta, \mathsf{f}\,\lceil \bar{q} \rceil \,/\, \mathit{self}] \qquad (\Sigma_{i-1};\Gamma \vdash \mathsf{f}\,\bar{q}\,.\pi_{\mathsf{i}} := Q_i : A_i\sigma \mid \Theta \rightsquigarrow \Sigma_i)_{i=1\ldots n}}{\Sigma_0;\Gamma \vdash \mathsf{f}\,\bar{q} := \mathsf{record}\{\pi_1 \mapsto Q_1; \ldots; \pi_{\mathsf{n}} \mapsto Q_n\} : C \mid \Theta \rightsquigarrow \Sigma_n}\ \textsc{CtCosplit}$$

$$\frac{\Sigma_0;\Gamma_1 \vdash A = \mathsf{D}\,\bar{v} : \mathsf{Set}_n \qquad \mathsf{data}\ \mathsf{D}\,\Delta : \mathsf{Set}_n\ \mathsf{where}\ \overline{\mathsf{c_i}\,\Delta_i\,[]} \in \Sigma_0 \\ (\Delta_i' = \Delta_i[\bar{v} \,/\, \Delta])_{i=1\ldots n} \qquad (\rho_i = \mathbb{1}_{\Gamma_1} \uplus [\mathsf{c_i}\,\hat{\Delta}_i' \,/\, x] \quad \rho_i' = \rho_i \uplus \mathbb{1}_{\Gamma_2})_{i=1\ldots n} \\ (\Sigma_{i-1};\Gamma_1\Delta_i'(\Gamma_2\rho_i) \vdash \mathsf{f}\,\bar{q}\rho_i' := Q_i : C\rho_i' \mid \Theta \rightsquigarrow \Sigma_i)_{i=1\ldots n}}{\Sigma_0;\Gamma_1(x : A)\Gamma_2 \vdash \mathsf{f}\,\bar{q} := \mathsf{case}_x\{\mathsf{c_1}\,\hat{\Delta}_1' \mapsto Q_1; \ldots; \mathsf{c_n}\,\hat{\Delta}_n' \mapsto Q_n\} : C \mid \Theta \rightsquigarrow \Sigma_n}\ \textsc{CtSplitCon}$$

$$\frac{\begin{array}{c} \Sigma_0;\Gamma_1 \vdash A = \mathsf{D}\,\bar{v} : \mathsf{Set}_n \qquad \Gamma = \Gamma_1(x : A)\Gamma_2 \\ \mathsf{data}\ \mathsf{D}\,\Delta : \mathsf{Set}_n\ \mathsf{where}\ \overline{\mathsf{c_i}\,\Delta_i\,[\bar{j}_i \mid b_i]} \in \Sigma_0 \qquad \exists k.\ [\bar{j}_k \mid b_k] \neq [] \\ \left(\begin{array}{c} \Delta_i' = \Delta_i(\bar{j}_i : \mathbb{I})[\bar{v} \,/\, \Delta] \qquad \bar{q}_i = \hat{\Delta}_i[\bar{j}_i \mid b_i][\bar{v} \,/\, \Delta] \\ \rho_i = \mathbb{1}_{\Gamma_1} \uplus [\mathsf{c_i}\,\bar{q}_i \,/\, x] \qquad \rho_i' = \rho_i \uplus \mathbb{1}_{\Gamma_2} \\ \Theta_i = \textsc{boundary}(\bar{j}_i); \Theta \\ \Sigma_{i-1};\Gamma_1\Delta_i'(\Gamma_2\rho_i) \vdash \mathsf{f}\,\bar{q}\rho_i' := Q_i : C\rho_i' \mid \Theta_i \rightsquigarrow \Sigma_i \end{array}\right)_{i=1\ldots n} \\ \Sigma_n;\Gamma_1(x : \mathsf{D}\,\bar{v})\Gamma_2 \vdash \mathsf{f}\,\bar{q} := \mathsf{case}_x\{\mathsf{hcomp}\,r\,u\,u_0 \mapsto Q_{\mathsf{hc}}\} : C \mid \Theta \rightsquigarrow \Sigma_{n+1} \end{array}}{\Sigma_0;\Gamma \vdash \mathsf{f}\,\bar{q} := \mathsf{case}_x\left\{\begin{array}{l} \mathsf{c_1}\,\bar{q}_1 \mapsto Q_1; \ldots; \mathsf{c_n}\,\bar{q}_n \mapsto Q_n \\ \mathsf{hcomp}\,r\,u\,u_0 \mapsto Q_{\mathsf{hc}} \end{array}\right\} : C \mid \Theta \rightsquigarrow \Sigma_{n+1}}\ \textsc{CtSplitConHIT}$$

Fig. 3: Typing rules for case trees (excluding Eq).

$\boxed{\Sigma;\Gamma \vdash \mathsf{f}\,\bar{q} := \mathsf{case}_x\{\mathsf{hcomp}\,r\,u\,u_0 \mapsto Q\} : C \mid \Theta \rightsquigarrow \Sigma'}$ Presupposes: $(x : A) \in \Gamma$ where A is a type supporting hcomp, and the presuppositions made by the typing of case trees judgment (Figure 3).

Checks case tree Q can be used for the hcomp case of a split on x.

$$\frac{\begin{array}{c}\Delta_{\mathsf{hc}} = (r : \mathbb{I})(u : \mathbb{I} \to \mathsf{Partial}\,r\,A)(u_0 : A\,[\,r \mapsto u\,\mathsf{i0}\,]) \\ \rho_{\mathsf{hc}} = \mathbb{1}_{\Gamma_1} \uplus [\mathsf{hcomp}\,r\,u\,u_0\,/\,x] \qquad \rho'_{\mathsf{hc}} = \rho_{\mathsf{hc}} \uplus \mathbb{1}_{\Gamma_2} \\ \Sigma;\Gamma_1\Delta_{\mathsf{hc}}(\Gamma_2\rho_{\mathsf{hc}}) \vdash \mathsf{f}\,\bar{q}\rho'_{\mathsf{hc}} := Q : C\rho'_{\mathsf{hc}} \mid (r = \mathsf{i1});\Theta \rightsquigarrow \Sigma'\end{array}}{\Sigma;\Gamma_1(x : A)\Gamma_2 \vdash \mathsf{f}\,\bar{q} := \mathsf{case}_x\{\mathsf{hcomp}\,r\,u\,u_0 \mapsto Q\} : C \mid \Theta \rightsquigarrow \Sigma'}$$

Fig. 4: Typing a match against hcomp.

$$\frac{\begin{array}{c}\Sigma;\Gamma_1 \vdash A = \mathsf{Eq}_B\,u\,v : \mathsf{Set}_\ell \qquad \Sigma;\Gamma_1 \vdash^r_p u =^? v : B \Rightarrow \mathrm{YES}(\Gamma'_1, \rho, \tau, \_) \\ \\ \rho' = (\rho \uplus [\mathsf{refl}\,/\,x]) \uplus \mathbb{1}_{\Gamma_2} \quad \tau' = \tau \uplus \mathbb{1}_{\Gamma_2} \\ \Sigma;\Gamma'_1(\Gamma_2(\rho \uplus [\mathsf{refl}\,/\,x])) \vdash \mathsf{f}\,\bar{q}\rho' := Q_{\mathsf{refl}} : C\rho' \rightsquigarrow \Sigma_1 \\ \\ \Sigma_1;\Gamma_1(x : \mathsf{Eq}_B\,u\,v)\Gamma_2 \vdash \mathsf{f}\,\bar{q} := \mathsf{case}_x\{\mathsf{hcomp}\,r\,t\,t_0 \mapsto Q_{\mathsf{hc}}\} : C \rightsquigarrow \Sigma_2 \\ \\ \Delta_{\mathsf{tX}} = (b : B)(r : \mathbb{I})(p : \mathsf{Path}^r\,B\,b\,v)(t_0 : \mathsf{Eq}_B\,u\,b) \\ \rho_{\mathsf{tX}} = \mathbb{1}_{\Gamma_1} \uplus [\mathsf{transpX}\,p\,r\,t_0\,/\,x] \qquad \rho'_{\mathsf{tX}} = \rho_{\mathsf{tX}} \uplus \mathbb{1}_{\Gamma_2} \\ \Sigma_2;\Gamma_1\Delta_{\mathsf{tX}}(\Gamma_2\rho_{\mathsf{tX}}) \vdash \mathsf{f}\,\bar{q}\rho'_{\mathsf{tX}} := Q_{\mathsf{tX}} : C\rho'_{\mathsf{tX}} \mid (r = \mathsf{i1});\Theta \rightsquigarrow \Sigma'\end{array}}{\Sigma;\Gamma_1(x : A)\Gamma_2 \vdash \mathsf{f}\,\bar{q} := \mathsf{case}_x\left\{\begin{array}{l}\mathsf{refl} \mapsto^{\tau'} Q_{\mathsf{refl}} \\ \mathsf{hcomp}\,r\,t\,t_0 \mapsto Q_{\mathsf{hc}} \\ \mathsf{transpX}_b\,p\,r\,t_0 \mapsto Q_{\mathsf{tX}}\end{array}\right\} : C \rightsquigarrow \Sigma'}\ \textsc{CtSplitEq}$$

$$\frac{\Sigma;\Gamma_1 \vdash A = \mathsf{Eq}_B\,u\,v : \mathsf{Set}_\ell \qquad \Sigma;\Gamma_1 \vdash^r_p u =^? v : B \Rightarrow \mathrm{NO}}{\Sigma;\Gamma_1(x : A)\Gamma_2 \vdash \mathsf{f}\,\bar{q} := \mathsf{case}_x\{\} : C \rightsquigarrow \Sigma}\ \textsc{CtSplitAbsurdEq}$$

Fig. 5: Typing rules for case trees involving Eq.

CtIntroPath If the expected type C is a path type $\mathsf{PathP}\,B\,u_0\,u_1$ then we can extend the left hand side to $\mathsf{f}\,\bar{q}\ @_{u_0,u_1}\ i$. We also extend the list of boundary assignments to include the two faces $(i = \mathsf{i0})$ and $(i = \mathsf{i1})$, which will in CtDone ensure that Q produces an element that connects u_0 and u_1. To see this, note that u_0 is judgmentally equal to expression $\mathsf{f}\,\bar{q}\ @_{u_0,u_1}\ \mathsf{i0}$ and u_1 to $\mathsf{f}\,\bar{q}\ @_{u_0,u_1}\ \mathsf{i1}$, because of equality for path applications.

CtSplitPartial If the expected type is equal to $\mathsf{PartialP}\,r\,A$,[8] then we can proceed by splitting on the faces $\alpha_1, \ldots, \alpha_n$ as long as they together *cover* all the ways in which we can have $r = \mathsf{i1}$. This is ensured by the premise $r = \bigvee_i \bigwedge \alpha_i$, where $\boxed{\bigwedge \alpha}$ is defined by mapping $(i = \mathsf{i1})$ to i and $(i = \mathsf{i0})$ to $\sim i$, and combining the resulting elements with $\wedge$. For each face, the left hand side is first refined to $\mathsf{f}\,\bar{q}[\alpha_i]$

[8] PartialP is a dependent version of Partial where A is a partial element as well, i.e., $A : \mathsf{Partial}\,r\,\mathsf{Set}_n$.

so that the variables in the copatterns $\bar{q}$ match their assignment, and then extended by $\lfloor 1{=}1 \rfloor$ which is the right elimination for $\mathsf{PartialP}\ r\, A$ as we have $r = \mathsf{i1}$ in Γ_{α_i}. Additionally, since the faces α_i can have overlaps we need to make sure that the different case trees Q_i agree on the intersections, this is accomplished by extending Θ with the assignments of the previous cases. Finally when substituting into a list of assignments, as in $\boxed{\Theta[\alpha]}$, any trivial equalities get removed and any contradictory ones cause the whole assignment in which they appear to be removed.

CtSplitConHIT If the left-hand side $\mathsf{f}\ \bar{q}$ contains a variable x of a data type $\mathsf{D}\ \bar{v}$, then we can pattern match on x, building a $\mathsf{case}_x\{\ldots\}$ node that covers all the alternatives. Note that here, as well as CtSplitCon, the datatype is not an indexed family, so we do not require a clause for $\mathsf{transpX}$. In this rule we deal with the case in which $\mathsf{D}\ \bar{v}$ is an HIT, as at least one of the $\mathsf{c_i}$ has a non-empty boundary. For each of the constructors $\mathsf{c_i}$ we check the case tree Q_i. To do so we (1) refine $\mathsf{f}\ \bar{q}$ by replacing x with $\mathsf{c_i}$ fully applied to variable and path application copatterns according to its type; and (2) expand the list of assignments with both $(j = \mathsf{i0})$ and $(j = \mathsf{i1})$ for each j in $\bar{j}_i$, which is what $\boxed{\textsc{boundary}(\bar{j}_i)}$ denotes. Moreover we need to consider the case for $\mathsf{hcomp}\ r\ u\ u_0$ (see Figure 4): we check Q_{hc} by (1) replacing x by the pattern $\mathsf{hcomp}\ r\ u\ u_0$ in the left-hand side and (2) extending Θ with $(r = \mathsf{i1})$ since that is the face where $\mathsf{hcomp}\ r\ u\ u_0$ computes to $u\ \mathsf{i1}\ 1{=}1$.

We discuss rule CtSplitEq in the next section, while the remaining rules do not directly involve any of the new features of `Cubical Agda`, so we ask the reader to refer back to the corresponding judgment in Cockx & Abel (2018).

4.3.1 *Unification: splitting on the inductive equality type.*

The rule CtSplitEq allows splitting on $\mathsf{Eq}_B\ u\ v$ only when u and v can be unified by some substitution ρ, so that refl will be typeable at $\mathsf{Eq}_{B\rho}\ u\rho\ v\rho$. Cockx & Devriese (2018) define a proof relevant notion of unification, where most general unifiers are equivalences of the form $\Gamma(x : \mathsf{Id}_B\ u\ v) \simeq \Gamma'$. Moreover they also define a notion of *strong unifier* which requires additional definitional equalities to be satisfied by the equivalence, these guarantee that reductions are preserved when a case tree is translated to eliminators. Cockx & Abel (2018) adapt the notion of strong unifier to their specific setting, but they only preserve the substitutions between the two contexts, as the proofs are guaranteed to specialize to reflexivity when the element of $\mathsf{Eq}_B\ u\ v$ is refl, and they have no other canonical elements of that type. In our context, the arguments r and p of $\mathsf{transpX}\ p\ r\ t$, when paired together, correspond to the canonical elements of Swan's Id type (Cohen *et al.*, 2018), which supports the same interface of the identity type used in Cockx & Devriese (2018) to define their unifiers, as shown in Section 9.1 of Cohen *et al.* (2018). Accordingly, we define $\boxed{\mathsf{Path}^r\ A\ a_0\ a_1}$ to be the type of paths $a_0 \equiv_A a_1$ that are refl when $r = \mathsf{i1}$, so that we can use a telescope like $(r : \mathbb{I})(p : \mathsf{Path}^r\ A\ a_0\ a_1)$ to represent unification problems, and the inputs to $\mathsf{transpX}$. The notion of $\mathsf{Path}^r\ A\ a_0\ a_1$ has an extensional flavour, because an assumption $(p : \mathsf{Path}^{\mathsf{i1}}\ A\ a_0\ a_1)$ implies the definitional equality of a_0 and a_1. For this reason we only introduce it as a bookkeeping notation in the typing context of case trees, and merely regard it as an aid to express the metatheory developed here. To keep track of paths between substitutions we define a *substitution path* between substitutions $\Gamma \vdash \sigma_0, \sigma_1 : \Delta$ to be a substitution $\Gamma(i : \mathbb{I}) \vdash \eta : \Delta$ such that $\Gamma \vdash \eta[\mathsf{i0}/i] = \sigma_0 : \Delta$ and $\Gamma \vdash \eta[\mathsf{i1}/i] = \sigma_1 : \Delta$. We also say that a substitution path $\Gamma(i : \mathbb{I}) \vdash \eta : \Gamma$ is *constant* if it is equal to $\mathbb{1}_\Gamma$ weakened by $(i : \mathbb{I})$. We now have everything in place to give our definitions of strong unifier and disunifier.

Definition 1 (Strong unifier)**.** Let Γ be a well-formed context and u and v be terms such that $\Gamma \vdash u, v : A$. A strong unifier $(\Gamma', \sigma, \tau, \eta)$ of u and v consists of a context Γ' and substitutions $\Gamma' \vdash \sigma : \Gamma(r : \mathbb{I})(p : \mathsf{Path}^r\, A\, u\, v)$ and $\Gamma(r : \mathbb{I})(p : \mathsf{Path}^r\, A\, u\, v) \vdash \tau : \Gamma'$ and a substitution path η between $\sigma; \tau$ and $\mathbb{1}_{\Gamma(r:\mathbb{I})(p:\mathsf{Path}^r\, A\, u\, v)}$ such that:

1. $\Gamma' \vdash r\sigma = \mathsf{i1} : \mathbb{I}$ and $\Gamma' \vdash p\sigma = \mathsf{refl} : \mathsf{Path}^{\mathsf{i1}}\, A\sigma\, u\sigma\, v\sigma$ (these imply the definitional equality $\Gamma' \vdash u\sigma = v\sigma : A\sigma$).
2. $\Gamma' \vdash \tau; \sigma = \mathbb{1}_{\Gamma'} : \Gamma'$
3. For any $\Gamma_0 \vdash \sigma_0 : \Gamma(r : \mathbb{I})(p : \mathsf{Path}^r\, A\, u\, v)$ such that $\Gamma_0 \vdash r\sigma_0 = \mathsf{i1} : \mathbb{I}$ and $\Gamma_0 \vdash p\sigma_0 = \mathsf{refl} : \mathsf{Path}^{\mathsf{i1}}\, A\sigma_0\, u\sigma_0\, v\sigma_0$, we have that $\Gamma_0 \vdash \sigma; \tau; \sigma_0 = \sigma_0 : \Gamma(r : \mathbb{I})(p : \mathsf{Path}^r\, A\, u\, v)$ and that $\eta; (\sigma_0 \uplus \mathbb{1}_{i:\mathbb{I}})$ is a constant substitution path.

The last condition about $\eta; (\sigma_0 \uplus \mathbb{1}_{i:\mathbb{I}})$ being a constant substitution path makes sure that transporting along η will be the identity whenever r an p are solved by $\mathsf{i1}$ and refl. It corresponds to the analogous condition about isLinv in Definition 53 of Cockx & Devriese (2018). Note that while η is not used in the CTSPLITEQ typing rule, it will be necessary in Sect. 4.3.3.

Definition 2 (Disunifier)**.** Let Γ be a well-formed context and $\Gamma \vdash u, v : A$. A disunifier of u and v is a function $\Gamma \vdash f : (u \equiv_A v) \to \bot$ where $\bot$ is the empty type.

Finally we can assume the existence of a proof relevant unification algorithm which we specify through the following judgments:

- A positive success $\Sigma; \Gamma \vdash^r_p u =^? v : B \Rightarrow \textsc{yes}(\Gamma', \rho, \tau, \eta)$ ensures that the tuple $(\Gamma', \rho \uplus [\mathsf{i1}\ /\ r, \mathsf{refl}\ /\ p], \tau, \eta)$ is a strong unifier.
- A negative success $\Sigma; \Gamma \vdash^r_p u =^? v : B \Rightarrow \textsc{no}$ ensures that there exist a disunifier of u and v.

Note that ρ is a *pattern substitution*, i.e., it contains only variables and forced arguments, so that it can be applied to patterns. In general the algorithm might also fail to provide a definitive answer, in which case no split on $\mathsf{Eq}_B\, u\, v$ is allowed.

Remark 3. Note that the unification rules from Cockx & Devriese (2018) that are specific to datatypes will not apply to datatypes with path constructors, as properties like injectivity and distinctness of constructors cannot be guaranteed to hold in that case. In principle we could ask the user to provide suitable proofs of these properties, but there is no such interface at the moment.

$$\boxed{\Sigma;\Gamma(x:\mathsf{D}\ \bar{v})\Delta\vdash \mathsf{f}\ \bar{q}:C\mid\Theta\Rightarrow^{x}\textsc{hc-rhs}(rhs)}$$

$$\frac{\begin{array}{rl}\Theta = \alpha_1;\ldots;\alpha_n & \Delta_{\mathsf{hc}} = (r:\mathbb{I})(u:\mathbb{I}\to\mathsf{Partial}\ r\ (\mathsf{D}\ \bar{v}))(u_0:\mathsf{D}\ \bar{v}\ [\,r\mapsto u\ \mathsf{i0}\,]) \\ \Delta^i = \Delta[\mathsf{hfill}\ u\ u_0\ i\ /\ x] & \\ \delta^i:\Delta^i = \textsc{transp-tel}\ (j.\,\Delta^{\sim j\vee i})\ i\ \hat{\Delta}^{\mathsf{i1}} & \\ sys = \lambda i.\left\{\begin{array}{ll}(r=\mathsf{i1}) & \to \mathsf{f}\ \bar{q}[u\ i\ 1{=}1\ /\ x,\ \delta^i\ /\ \Delta^i] \\ \alpha_1 & \to \mathsf{f}\ \bar{q}[\mathsf{hfill}\ u\ u_0\ i\ /\ x,\ \delta^i\ /\ \Delta^i][\alpha_1] \\ \vdots & \\ \alpha_n & \to \mathsf{f}\ \bar{q}[\mathsf{hfill}\ u\ u_0\ i\ /\ x,\ \delta^i\ /\ \Delta^i][\alpha_n]\end{array}\right\} & \\ rhs = \mathsf{comp}\ (\lambda i.\,C[\mathsf{hfill}\ u\ u_0\ i\ /\ x,\ \delta^i\ /\ \Delta^i])\ sys\ (\mathsf{inS}\ (\mathsf{f}\ \bar{q}[\mathsf{outS}\ u_0\ /\ x,\ \delta^{\mathsf{i0}}\ /\ \Delta])) & \\ \text{Derivable typing:}\quad \Gamma\Delta_{\mathsf{hc}}\Delta[\mathsf{hcomp}\ u\ u_0\ /\ x]\vdash rhs:C[\mathsf{hcomp}\ u\ u_0\ /\ x] & \end{array}}{\Sigma;\Gamma(x:\mathsf{D}\ \bar{v})\Delta\vdash \mathsf{f}\ \bar{q}:C\mid\Theta\Rightarrow^{x}\textsc{hc-rhs}(rhs)}$$

Fig. 6: Computing the right hand side of a hcomp match.

4.3.2 Inferring Right-Hand Side of a hcomp $r\ u\ u_0$ *match*

In the examples given in Sect. 2 we have never given a clause for the hcomp constructor when pattern matching on an element of a higher inductive type. That is because `Cubical Agda` generates a suitable clause for us during elaboration. How to deal with the hcomp case was already explained in Cohen *et al.* (2018), but only for the respective induction principle, while in our case we have to deal with user clauses that include multiple-argument and nested pattern matching.

Fortunately in the context of the rule CtSplitConHIT we have the right information available to construct a term that would be suitable for Q_{hc}. This is accomplished in Fig. 6 by the only rule of the judgment $\Sigma;\Gamma(x:\mathsf{D}\ \bar{v})\Delta\vdash \mathsf{f}\ \bar{q}:C\mid\Theta\Rightarrow^{x}\textsc{hc-rhs}(rhs)$. The term *rhs* is supposed to be typable as

$$\Gamma\Delta_{\mathsf{hc}}\Delta[\mathsf{hcomp}\ u\ u_0\ /\ x]\vdash rhs:C[\mathsf{hcomp}\ u\ u_0\ /\ x]$$

while also satisfying the constraints implied by $r=\mathsf{i1};\Theta$. The construction of *rhs* is fairly involved, so we will build up to it with simpler cases.

First let us assume both Δ and Θ are empty and that $C=T$ where T does not depend on x, which means we want $\Gamma\Delta_{\mathsf{hc}}\vdash rhs:T$. We already have $\Gamma(x:\mathsf{D}\ v)\vdash \mathsf{f}\ \bar{q}:T$, which we can use with x replaced by elements of type $\mathsf{D}\ v$ obtained by u and u_0, and build a composition in T. Writing $g(d)$ for $\mathsf{f}\ \bar{q}[d\ /\ x]$, we define *rhs* as

$$\Gamma\Delta_{\mathsf{hc}}\vdash rhs:=\mathsf{hcomp}\ (\lambda\{i\ (r=\mathsf{i1})\to g(u\ i\ 1{=}1)\})\ (\mathsf{inS}(g(\mathsf{outS}\ u_0))):T$$

Note that $\Gamma\Delta_{\mathsf{hc}},r=\mathsf{i1}\vdash rhs=\mathsf{f}\ \bar{q}[u\ \mathsf{i1}\ 1{=}1\ /\ x]$ as expected, while defining *rhs* as just $g(\mathsf{outS}\ u_0))$ would not have satisfied this equality.

Slightly more complex is the case where T does depend on x, so that $g(u\ i\ 1{=}1)$ has type $T[u\ i\ 1{=}1\ /\ x]$ and $g(\mathsf{outS}\ u_0)$ has type $T[\mathsf{outS}\ u_0\ /\ x]$, while *rhs* will have type $T[\mathsf{hcomp}\ u\ u_0/x]$. What we need then is to use heterogeneous composition, comp, along with a suitable family $A:\mathbb{I}\to\mathsf{Set}_\ell$ which matches the three types above in the respective cases. We have already seen that $\mathsf{hfill}\ u\ u_0$ is a filler for the open box specified by u and u_0 and with lid $\mathsf{hcomp}\ u\ u_0$, so taking $A = \lambda i.\ T[\mathsf{hfill}\ u\ u_0\ i\ /\ x]$ will do the job

$$\Gamma\Delta_{\mathsf{hc}} \vdash rhs := \mathsf{comp}\ A\ (\lambda\{i\ (r = \mathsf{i1}) \to g(u\ i\ 1{=}1)\})\ (\mathsf{inS}(g(\mathsf{outS}\ u_0))) : T[\mathsf{hcomp}\ u\ u_0\ /\ x]$$

this definition also matches the behaviour of the eliminator for spheres in Sect. 9.2 of Cohen *et al.* (2018).

Let us now consider that we have a single y of type $Y : \mathsf{Set}_\ell$ in Δ:

$$\Gamma\Delta_{\mathsf{hc}}(y : Y[\mathsf{hcomp}\ u\ u_0\ /\ x]) \vdash rhs : T[\mathsf{hcomp}\ u\ u_0\ /\ x]$$

This case could be seen as a special case of the previous one, by replacing T with $(y : Y) \to T$ and having $\lambda y.g(d)$ in place of $g(d)$. However, for the sake of consistency with other cases, we write out explicitly how such a composition can be obtained. Let $g(d, v) = \mathsf{f}\ \bar{q}[d\ /\ x, v\ /\ y]$ and $y^i = \mathsf{transp}\ (\lambda j.\ Y[\mathsf{hfill}\ u\ u_0\ ({\sim}\ j \vee i)\ /\ x])\ i\ y$ and $A = \lambda i.\ T[\mathsf{hfill}\ u\ u_0\ i\ /\ x, y^i\ /\ y]$, then

$$rhs = \mathsf{comp}\ A\ (\lambda\{i\ (r = \mathsf{i1}) \to g(u\ i\ 1{=}1, y^i)\})\ (\mathsf{inS}(g(\mathsf{outS}\ u_0), y^{\mathsf{i0}})) : T[\mathsf{hcomp}\ u\ u_0\ /\ x].$$

Compare this composition with the definition of $\mathsf{transportPi}$ in Sect. 3.2.1.

Alternatively, Δ could be $(k : \mathbb{I})$, with $\Theta = (k = \mathsf{i0}); (k = \mathsf{i1})$, as might happen when $\bar{q}$ contains a path application. Then *rhs* should have typing

$$\Gamma\Delta_{\mathsf{hc}}(k : \mathbb{I}) \vdash rhs : T[\mathsf{hcomp}\ u\ u_0\ /\ x]$$

and be equal to $\mathsf{f}\ \bar{q}[\mathsf{hcomp}\ u\ u_0\ /\ x, \mathsf{i0}\ /\ k]$ whenever $k = \mathsf{i0}$, and likewise for $\mathsf{i1}$. As in the implementation of $\mathsf{transporPath}$ we can address these constraints by adding to the sides of the composition. Let $g(d, s) = \mathsf{f}\ \bar{q}[d\ /\ x, s\ /\ k]$, and A be as in the empty Δ case, then we define *rhs* as

$$sys = \lambda i. \left\{ \begin{array}{l} (r = \mathsf{i1}) \to g(u\ i\ 1{=}1, k) \\ (k = \mathsf{i0}) \to g(\mathsf{hcomp}\ u\ u_0, \mathsf{i0}) \\ (k = \mathsf{i1}) \to g(\mathsf{hcomp}\ u\ u_0, \mathsf{i1}) \end{array} \right\}$$

$$rhs = \mathsf{comp}\ A\ sys\ (\mathsf{inS}(g(\mathsf{outS}\ u_0, k))) : T[\mathsf{hcomp}\ u\ u_0\ /\ x]$$

Note that it does not matter from which part of the context k comes from for the definition above to be well-typed, so the same strategy applies also when Θ refers to variables in Γ.

Finally, when we let Δ and Θ contain multiple variables and equations we end up with the definition given in Fig. 6. There we use a version of transp generalized to telescopes, $\boxed{\text{TRANSP-TEL}}$, which covers both the cases like $(k : \mathbb{I})$ where nothing is to be done, and $(y : Y)$ where transp for Y is used. It can also fail if the type does not support transp but it is not closed, in which case elaboration fails.

During elaboration we can then run the algorithm expressed by this judgment and generate internally a clause for $\mathsf{hcomp}\ r\ u\ u_0$.

4.3.3 Inferring case tree of a $\mathsf{transpX}\ p\ r\ t_0$ *match*

What we discussed in the previous section about a hcomp match also holds if we are splitting on an inductive family like $\mathsf{Eq}_B\ u\ v$. Additionally, however, we have to handle the possbility that we are matching against an element built with $\mathsf{transpX}\ p\ r\ t_0$. In this case we will produce a case tree that performs a further split on t_0 so that we have to produce

$$\boxed{\Sigma;\Gamma(x:\mathsf{Eq}_A\,u\,v)\Delta\mid(\Gamma',\rho,\tau,\eta)\vdash \mathsf{f}\,\bar{q}:C\mid\Theta\Rightarrow\textsc{trxrefl}(\mathit{rhs})}$$

$$\frac{\begin{array}{rl} & \Theta=\alpha_1;\ldots;\alpha_n \quad \Delta_{\mathsf{refl}}=(r:\mathbb{I})(p:\mathsf{Path}^r\,B\,u\,v)\\ \Delta^i &= \Delta[\mathsf{transpX}\,p\,r\,\mathsf{refl}\,/\,x][\eta]\\ \delta^i:\Delta^i &= \textsc{transp-tel}\,(j.\,\Delta^{\sim\,j\vee i})\,(i\vee r)\,\hat{\Delta}^{\mathsf{i1}}\\ \mathit{sys} &= \lambda i.\left\{\begin{array}{ll}(r=\mathsf{i1}) & \to \mathsf{f}\,\bar{q}[(\rho\uplus\mathsf{refl}\,/\,x)[\tau]\uplus\delta^i\,/\,\Delta]\\ \alpha_1 & \to \mathsf{f}\,\bar{q}[\mathsf{transpX}\,p\,r\,\mathsf{refl}\,/\,x][\eta\uplus\delta^i\,/\,\Delta^i][\alpha_1]\\ \vdots & \\ \alpha_n & \to \mathsf{f}\,\bar{q}[\mathsf{transpX}\,p\,r\,\mathsf{refl}\,/\,x][\eta\uplus\delta^i\,/\,\Delta^i][\alpha_n]\end{array}\right\}\\ \mathit{base} &= \mathsf{inS}\,(\mathsf{f}\,\bar{q}[(\rho\uplus\mathsf{refl}\,/\,x)[\tau]\uplus\delta^{\mathsf{i0}}\,/\,\Delta])\\ \mathit{rhs} &= \mathsf{comp}\,(\lambda i.\,C[\mathsf{transpX}\,p\,r\,\mathsf{refl}\,/\,x][\eta\uplus\delta^i\,/\,\Delta^i])\,\mathit{sys}\,\mathit{base}\\ \text{Derivable typing:} & \Gamma\Delta_{\mathsf{refl}}\Delta[\mathsf{transpX}\,p\,r\,\mathsf{refl}\,/\,x]\vdash\mathit{rhs}:C[\mathsf{transpX}\,p\,r\,\mathsf{refl}\,/\,x]\end{array}}{\Sigma;\Gamma(x:\mathsf{Eq}_A\,u\,v)\Delta\mid(\Gamma',\rho,\tau,\eta)\vdash\mathsf{f}\,\bar{q}:C\mid\Theta\Rightarrow\textsc{trxrefl}(\mathit{rhs})}$$

Fig. 7: Computing the right hand side of a $\mathsf{transpX}\;p\;r\;\mathsf{refl}$ match.

right hand sides that fit the following clauses for the function f,

$$\begin{aligned}\mathsf{f}\,\bar{q}[\mathsf{transpX}\,p\,r\,\mathsf{refl}/x] &= \mathit{rhs}_{\mathsf{refl}}\\ \mathsf{f}\,\bar{q}[\mathsf{transpX}\,p\,r\,(\mathsf{transpX}\,q\,s\,t_1)/x] &= \mathit{rhs}_{\mathsf{tX}}\\ \mathsf{f}\,\bar{q}[\mathsf{transpX}\,p\,r\,(\mathsf{hcomp}\,s\,w\,w_0)/x] &= \mathit{rhs}_{\mathsf{hc}}\end{aligned}$$

the term $\mathit{rhs}_{\mathsf{hc}}$ can be computed using the HC-RHS judgment from the previous section, we only need to specialize the copatterns $\bar{q}$ to $\bar{q}[\mathsf{transpX}\,p\,r\,x\,/\,x]$ and update the other arguments accordingly. Specifically, using the definitions from rule CTSPLITEQ,

$$\Sigma;\Gamma_1\Delta_{\mathsf{tX}}\Gamma_2\rho_{\mathsf{tX}}\vdash\mathsf{f}\,\bar{q}\rho'_{\mathsf{tX}}:C\rho'_{\mathsf{tX}}\mid(r=\mathsf{i1});\Theta\Rightarrow^{t_0}\textsc{hc-rhs}(\mathit{rhs}_{\mathsf{hc}}).$$

The other two terms, $\mathit{rhs}_{\mathsf{refl}}$ and $\mathit{rhs}_{\mathsf{tX}}$, are obtained using the judgments in figures 7 and 8.

The judgment $\Sigma;\Gamma(x:\mathsf{Eq}_A\,u\,v)\Delta\mid(\Gamma',\rho,\tau,\eta)\vdash\mathsf{f}\,\bar{q}:C\mid\Theta\Rightarrow\textsc{trxrefl}(\mathit{rhs})$ in Figure 7 is where we make use of the η component of the strong unifier obtained from the judgment $\Sigma;\Gamma\vdash^r_p u=^? v:A\Rightarrow\textsc{yes}(\Gamma',\rho,\tau,\eta)$. [Andrea: TODO: explain specialization by match on refl?] Let us focus on the case where Δ and Θ are empty, then we want to construct a term rhs with typing

$$\Gamma(r:\mathbb{I})(p:\mathsf{Path}^r\,u\,v)\vdash\mathit{rhs}:C[\mathsf{transpX}\,p\,r\,\mathsf{refl}/x].$$

By the definition of strong unifier we have that $\sigma:=\rho\uplus[\mathsf{i1}/r,\mathsf{refl}/p]$ is an equivalence between $\Gamma(r:\mathbb{I})(p:\mathsf{Path}^r\,u\,v)$ and Γ', so we can use it to rewrite our goal to

$$\Gamma'\vdash ?0:C[\mathsf{transpX}\,p\,r\,\mathsf{refl}/x]\sigma.$$

By simplifying substitutions and reducing $\mathsf{transpX}$, the type of $?0$ is equal to $C[\rho\uplus\mathsf{refl}/x]$, which is the type of $\mathsf{f}\,\bar{q}[\rho\uplus\mathsf{refl}/x]$, so we can use that to conclude. Writing an explicit term

$$\boxed{\Sigma; \Gamma(x : \mathsf{Eq}_A\, u\, v)\Delta \vdash \mathsf{f}\; \bar{q} : C \mid \Theta \Rightarrow \text{TRXTRX}(rhs)}$$

$$\begin{array}{l}
\Theta = \alpha_1; \ldots; \alpha_n \quad \Delta_{\mathsf{tX}} = (b : B)(r : \mathbb{I})(p : \mathsf{Path}^r\, B\, b\, v) \\
\qquad\qquad\qquad\qquad\qquad (b' : B)(s : \mathbb{I})(q : \mathsf{Path}^s\, B\, b'\, b)(t : \mathsf{Eq}_B\, u\, b') \\
eq_0 = \mathsf{transpX}\; (q \bullet^{s,r} p)\; (r \wedge s)\; t \quad eq_1 = \mathsf{transpX}\; p\; r\; (\mathsf{transpX}\; q\; s\; t) \\
\text{Let } eq : eq_0 \equiv eq_1 \text{ and } eq_{s,r} : \mathsf{PartialP}\; (s \vee r)\; (\lambda\, (s \vee r = \mathsf{i1}) \to eq \equiv \mathsf{refl}) \\
\text{both constant when } r \wedge s = \mathsf{i1}. \\
\qquad \Delta' = \Delta[eq_1 \,/\, x] \\
\delta(r', eq', j) = \text{TRANSP-TEL}\; (i.\, \Delta[eq'({\sim}i \vee j) \,/\, x])\; ((s \wedge r) \vee r' \vee j)\; \hat{\Delta}' \\
f(r', eq', j) = \mathsf{f}\; \bar{q}[eq'(j) \,/\, x, \delta(r', eq', j) \,/\, \Delta] \\
C'(r', eq') = \lambda\, j \to C[eq'(j) \,/\, x, \delta(r', eq', j) \,/\, \Delta] \\
sys(r', eq') = \lambda\, j \to \left\{ \begin{array}{ll}
\alpha_1 & \to f(r', eq', j)[\alpha_1] \\
\vdots & \\
\alpha_n & \to f(r', eq', j)[\alpha_n] \\
(r \wedge s = \mathsf{i1}) & \to f(r', eq', j) \\
(r' = \mathsf{i1}) & \to f(r', eq', j)
\end{array} \right\} \\
c(r', eq') = \mathsf{comp}\; C'(r', eq')\; sys(r', eq')\; (\mathsf{inS}\; (f(r', eq', \mathsf{i0}))) \\
sys' = \lambda\, i \to \left\{ \begin{array}{ll}
\alpha_1 & \to \mathsf{f}\; \bar{q}[eq_0 \,/\, x][\alpha_1] \\
\vdots & \\
\alpha_n & \to \mathsf{f}\; \bar{q}[eq_0 \,/\, x][\alpha_1] \\
(r = \mathsf{i1}) & \to c(i, eq_{s,r}\; 1{=}1\; i) \\
(s = \mathsf{i1}) & \to c(i, eq_{s,r}\; 1{=}1\; i)
\end{array} \right\} \\
rhs = \mathsf{hcomp}\; sys'\; c(\mathsf{i0}, eq) \\
\qquad \text{Derivable typing:} \qquad \Gamma\Delta_{\mathsf{tX}}\Delta' \vdash rhs : C[eq_1 \,/\, x]
\end{array}$$

$$\overline{\Sigma; \Gamma(x : \mathsf{Eq}_A\, u\, v)\Delta \vdash \mathsf{f}\; \bar{q} : C \mid \Theta \Rightarrow \text{TRXTRX}(rhs)}$$

Fig. 8: Computing the right hand side of a $\mathsf{transpX}\; p\; r\; (\mathsf{transpX}\; q\; s\; t)$ match.

for the reasoning above we get

$$\begin{array}{l}
rhs = \mathsf{comp}\; (\lambda i.\, C[\mathsf{transpX}\; p\; r\; \mathsf{refl}/x]\eta) \\
\qquad\qquad (\lambda\{i\; (r = \mathsf{i1}) \to \mathsf{f}\; \bar{q}[\rho \uplus \mathsf{refl}/x]\tau\}) \\
\qquad\qquad (\mathsf{inS}(\mathsf{f}\; \bar{q}[\rho \uplus \mathsf{refl}/x]\tau)).
\end{array}$$

Finally, the judgment $\Sigma; \Gamma(x : \mathsf{Eq}_A\, u\, v)\Delta \vdash \mathsf{f}\; \bar{q} : C \mid \Theta \Rightarrow \text{TRXTRX}(rhs)$ in Figure 8 is where we take care of the case $\mathsf{f}\; \bar{q}[\mathsf{transpX}\; p\; r\; (\mathsf{transpX}\; q\; s\; t) \,/\, x]$ by making use of the case $\mathsf{f}\; \bar{q}[\mathsf{transpX}\; (q \bullet^{s,r} p)\; (r \wedge s)\; t]$, which has one fewer $\mathsf{transpX}$ and so gets us closer to the $\mathsf{transpX}\; \_\; \_\; \mathsf{refl}$ base case. The expression $(q \bullet^{s,r} p)$ is built with a transitivity operator that makes use of the s and r argument to reduce to q when $r = \mathsf{i1}$ and reduce to p when $s = \mathsf{i1}$, making eq_0 and eq_1 definitionally equal under either condition. Using connections and transports we can define both eq and $eq_{s,r}$ as specified, and then we proceed to define the required term *rhs* by nesting compositions inside an homogeneous composition. The term obtained with $c(\mathsf{i0}, eq)$ would have the right type and satisfy the boundary conditions from Θ, but it would not satisfy the ones imposed by $(s = \mathsf{i1}); (r = \mathsf{i1})$, the matching cases of sys' take care of that.

5 Glue Types in Cubical Agda

Glue types are the key contribution of Cohen *et al.* (2018) for equipping the univalence principle with computational content. Given that a type in cubical type theory stands for a higher dimensional cube, Glue types let us construct a cube where some faces have been replaced by equivalent types. This is analogous to how hcomp lets us replace some faces of a cube by composing it with other cubes, however for Glue types we can compose with equivalences instead of paths. This implies the univalence principle and it is what lets us transport along paths built out of equivalences.

5.1 Glue Types and Univalence

As everything in `Cubical Agda` has to work up to higher dimensions, the Glue types take a partial family of types A that are equivalent to the base type B. The idea is then that these types get glued onto B so that the equivalence data gets packaged up into a new datatype.

$$\mathsf{Glue} : (B : \mathsf{Set}\ \ell)\ \{r : \mathsf{I}\} \to \mathsf{Partial}\ r\ (\Sigma[\ A \in \mathsf{Set}\ \ell\ ]\ (A \simeq B)) \to \mathsf{Set}\ \ell$$

When r is $\mathsf{i1}$ the type $\mathsf{Glue}\ \{r\}\ B\ Ae$ reduces to $Ae\ \mathsf{1{=}1}\ .\mathsf{fst}$.

Using Glue types we can turn an equivalence of types into a path and hence define ua.

$$\mathsf{ua} : \{A\ B : \mathsf{Set}\ \ell\} \to A \simeq B \to A \equiv B$$
$$\mathsf{ua}\ \{A = A\}\ \{B = B\}\ e\ i = \mathsf{Glue}\ B\ (\lambda\ \{\ (i = \mathsf{i0}) \to (A\ ,\ e)\ ;\ (i = \mathsf{i1}) \to (B\ ,\ \mathsf{idEquiv}\ B)\ \})$$

The idea is that we glue A onto B when i is $\mathsf{i0}$ using e and B onto itself when i is $\mathsf{i1}$ using the identity equivalence. The term $\mathsf{ua}\ e$ is a path from A to B as the Glue type reduces when the face conditions are satisfied, so when i is $\mathsf{i0}$ this reduces to A and when i is $\mathsf{i1}$ it reduces to B. Pictorially we can describe $\mathsf{ua}\ e$ as the dashed line in:

$$\begin{tikzcd}
A \arrow[r, dashed] \arrow[d, "e"', "\wr"] & B \arrow[d, "\wr"', "\mathsf{idEquiv}\ B"] \\
B \arrow[r, "B"'] & B
\end{tikzcd}$$

The transp operation for Glue types is the most complicated part of the internals of `Cubical Agda`. The algorithm closely follows Huber (2017, Sect. 3.6), which is a variation of the original algorithm from Cohen *et al.* (2018, Sect. 6.2). We will focus on the special case of $\mathsf{transport}\ (\lambda\ i \to \mathsf{ua}\ e\ i\ )\ a$ for simplicity. This will transport a from A to B by going through the three fully filled lines in the above picture.

Unfolding ua gives

$$\mathsf{transport}\ (\lambda\ i \to \mathsf{Glue}\ B\ (\lambda\ \{\ (i = \mathsf{i0}) \to (A\ ,\ e)\ ;\ \ (i = \mathsf{i1}) \to (B\ ,\ \mathsf{idEquiv}\ B)\ \})\ a$$

By the boundary equations for Glue types we get that $a : A$ (as it is in the i = i0 face of the Glue type). The algorithm then applies the function of e (i.e., e .fst : $A \to B$) to a giving an element in B. As B is constant along i we could now be done, however for the general algorithm there is no reason for the base to be constant along i; it could for example be another Glue type! We must hence transport along $(\lambda\ i \to B)$ to get an element in the bottom-right B in the diagram. In order to go up to the top-right corner we then use the inverse of the identity equivalence.[9] Since this is the identity function we end up with:

$$\mathsf{transport}\ (\lambda\ i \to B)\ (e\ .\mathsf{fst}\ a)$$

Using the same path as in the definition of transport for path types we can prove that this is equal to e .fst a up to a path:

$$\begin{aligned}&\mathsf{ua}\beta : \{A\ B : \mathsf{Set}\ \ell\}\ (e : A \simeq B)\ (a : A) \to \mathsf{transport}\ (\mathsf{ua}\ e)\ a \equiv e\ .\mathsf{fst}\ a\\&\mathsf{ua}\beta\ \{B = B\}\ e\ a = \lambda\ i \to \mathsf{transp}\ (\lambda\ \_ \to B)\ i\ (e\ .\mathsf{fst}\ a)\end{aligned}$$

Transporting along the path that we get from applying ua to an equivalence is, thus, the same as applying the equivalence. This makes it possible to use the univalence axiom computationally in `Cubical Agda`: we can package up equivalences as paths, do equality reasoning using these paths, and in the end transport along the paths to compute with the equivalences. Furthermore, the combination of ua and uaβ is sufficient to prove that ua is an equivalence which gives the full univalence theorem, i.e., an equivalence between paths and equivalences.[10]

$$\mathsf{univalence} : \forall \{\ell\}\{A\ B : \mathsf{Set}\ \ell\} \to (A \equiv B) \simeq (A \simeq B)$$

5.2 General Case of transp for Glue Types and the ghcomp Operation

While the special case of transp for Glue types above is quite simple the general case is a lot more complex. The reason is that the input might depend on many more variables than just i. When considering

$$\mathsf{transport}\ (\lambda\ i \to \mathsf{Glue}\ B\ (\lambda\ \{\ (r = \mathsf{i1}) \to (A\ ,\ e)\ \})\ a$$

the interval element r might be quite complex and its disjunctive normal form might contain clauses that do not involve i. On these parts the transp function should compute like the transp function for A by the boundary rules for Glue types. This in turn means that additional corrections have to be made compared to the ua case. In Cohen *et al.* (2018) the part of r that does not mention i is written $\forall i.r$ (as this operation corresponds to universal quantification on the interval).[11]

One of the modifications we have to do in the general case of transp for Glue types is that the simple transport in B has to be a comp with suitable corrections for the $\forall i.r$ faces. While this is easily achieved it has some unfortunate consequences in the case of transporting along ua. In this particular case r is $i \vee \sim i$ so that $(\forall i.\mathrm{r})$ = i0 as there is no

[9] In general this might not be the identity function, thus, this step might actually do something.
[10] https://github.com/agda/cubical/blob/master/Cubical/Foundations/Univalence.agda#L63.
[11] Technically speaking the $\forall$ operation in Cohen *et al.* (2018) is not an operation on the interval, but rather on the face lattice $\mathbb{F}$. However it is direct to define an analogous operation on the interval and it is this one we use here.

part that does not mention i. This means that the comp correction will introduce an empty system which implies that our simple proof of uaβ does not work anymore. In order to fix this we have to extend the proof of uaβ with a suitable hfill in order to compensate for the additional empty system.

Luckily there is a simple trick in `Cubical Agda` that lets us adapt the correction to eliminate the empty system. The problem with the above sketched definition is that the comp does not reduce when $\forall i.r$ is i0, however if we add a clause mapping to the base for this case the issue with the empty system goes away. This relies on a subtle difference between the hcomp operation in `Cubical Agda` and the one in Coquand *et al.* (2018). In the latter the boundary constraints were elements of the face lattice $\mathbb{F}$ generated by formal generators $(i = \mathsf{i0})$ and $(i = \mathsf{i1})$ subject to the relation $(i = \mathsf{i0}) \wedge (i = \mathsf{i1}) = \bot$. In `Cubical Agda` on the other hand the hcomp operation takes a family of partial elements that are specified by some $r : \mathsf{I}$. This means that we in `Cubical Agda` can add a face when $(r = \mathsf{i0})$ which was not possible in Coquand *et al.* (2018) as there is no corresponding operation for $\mathbb{F}$.

The reason that $\mathbb{F}$ in Coquand *et al.* (2018) does not admit such an operation is that while every $\varphi : \mathbb{F}$ is expressible as $r = \mathsf{i1}$ the choice of r is not unique. In particular for $\varphi = 0_{\mathbb{F}}$ we can choose either i0 or $i \wedge \neg i$ which would give different results when equated to i0. Using $r : \mathsf{I}$ to specify boundaries in `Cubical Agda` avoids the need to make such a choice, and in particular $(r = \mathsf{i0})$ is represented by $\neg r$. It would be tempting to instead extend $\mathbb{F}$ with a negation operation, however that would allow us to represent new kinds of boundaries, like the open interval $(0, 1]$ as $\neg(i = \mathsf{i0})$, and it is not clear how they would impact decidability of typechecking. Modifying hcomp and transp to take a $r : \mathsf{I}$ is semantically justified by the fact that it is not necessary for boundaries to be specified by a subobject of the subobject classifier Ω in the presheaf topos of cubical sets in order to obtain a model of univalent type theory.[12]

Inspired by Angiuli *et al.* (2017, Page 53) we call the homogeneous version of this operation *generalized homogeneous composition*, ghcomp. The heterogeneous version used above can be implemented by using ghcomp in the definition of comp. We can write the ghcomp operation as:

```
ghcomp : {r : I} (u : I → Partial r A) (u₀ : A [ r ↦ u i0 ]) → A
ghcomp {r = r} u u₀ =
  hcomp (λ j → λ { (r = i1) → u j 1=1 ; (r = i0) → outS u₀ })
        (inS (outS u₀))
```

By using this in all of the places where the $\forall$ correction has to be made in the general algorithm for transp for Glue we obtain a better algorithm which does not produce any new empty systems. This way the proof of uaβ can stay as simple as above and no additional corrections has to be made. This is an improvement compared to the algorithm in Cohen *et al.* (2018) (that is implemented in `cubicaltt`) which produced a surprisingly large number of empty systems even in simple cases.

[12] This generalization has been formally verified in `Agda` in https://github.com/mortberg/gen-cart/.

6 Metatheory of Cubical Type Theory and Cubical Agda

The original formulation of cubical type theory as in Cohen *et al.* (2018) has a model in Kan cubical sets with connections and reversals, that is, presheaves on a suitable cube category where types have structure corresponding to the comp operation. This model has been formally verified in both the `NuPRL` proof assistant by Bickford (2018) and using `Agda` as the internal language of the presheaf topos of cubical sets by Orton & Pitts (2016) and Licata *et al.* (2018). This hence provides semantic consistency proofs for the cubical type theory that `Cubical Agda` is based on. Applying Tait's computability method, a syntactic consistency proof for this cubical type theory was given in Huber (2016) by defining an operational semantics and proving that any term of type $\mathbb{N}$ computes to a numeral.

A crucial property for synthetic mathematics, as in Sect. 2.5, is the existence of interesting models of the theory. Ideally we would like to be able to interpret these results in topological spaces or even any (Grothendieck) ∞-topos. Currently these questions have not been fully resolved for the various cubical type theories that have been considered. In fact, Sattler (2018) has shown that the standard model of `Cubical Agda` is *not* equivalent[13] to topological spaces. However, if one drops the reversal operation ($\sim\_$) from `Cubical Agda` any internal result about homotopy groups of spheres corresponds to a result about the homotopy groups of spheres in spaces.[14] Furthermore, there has been recent progress on an "equivariant" cubical set model that is equivalent to spaces (Riehl, 2019). We are hence very optimistic that these issues will be resolved in the near future. Furthermore, as soon as a satisfactory cubical type theory with a model in spaces has been developed we expect it to be straightforward to adapt `Cubical Agda` and its library to that theory. Indeed, the main features that we rely on—computational univalence and higher inductive types with definitional computation rules for all constructors—should also be satisfied by that cubical type theory.

The syntax and semantics of HITs in cubical type theory were studied in Coquand *et al.* (2018). The canonicity proof has been shown to extend to the circle and propositional truncation in Huber (2016, Sect. 5). One technical consequence of the way the system in Coquand *et al.* (2018) is designed is that there are closed terms of the circle in an empty context that are not base, for example hcomp ($\lambda\ i \rightarrow$ empty) base. These degenerate elements were a serious problem in `cubicaltt` as they complicated both programming and proving, affecting the efficiency of the system.

These elements arose from the way comp reduces for Glue types in Cohen *et al.* (2018), but with the optimization discussed in Sect. 5.2 using ghcomp we can eliminate them. This requires us to impose a "validity" constraint on partial elements—following Angiuli *et al.* (2018, Def. 12)—which says that Partial $r\ A$ is *valid* if it cannot become empty from a dimension substitution (a concrete condition is that r is a classical tautology). Validity combined with ghcomp eliminates all of the ways that a partial element can become empty in the system. As `Cubical Agda` implement the ghcomp optimization we expect it to be possible to prove a refinement of the canonicity theorem stating that the point constructors are the only elements of HITs in the empty context.

[13] By "equivalent" we mean that the notion of fibration in the cubical set model gives rise to a model structure that is Quillen equivalent to the classical Quillen model structure on spaces.

[14] For further details and discussions about this result see: https://groups.google.com/forum/#!topic/homotopytypetheory/imPb56IqxOI

While the comp operation is complicated a recent result by Coquand *et al.* (2019) shows that for the Cohen *et al.* (2018) cubical type theory any implementation of the comp operation yield the same result for natural numbers up to a path. As Cubical Agda is based on this cubical type theory the result also applies, so even though the implementation of comp differs from the way comp was defined in Cohen *et al.* (2018), the result for closed terms of type natural numbers will be the same up to a path.

7 Conclusion

In this paper we presented Cubical Agda, an extension of Agda with features from cubical type theory. This brings to a proof assistant both a fully computational univalence principle and HITs. Moreover, induction on HITs and construction of paths are integrated into Agda's very expressive pattern matching, providing support for more idiomatic definitions than direct use of eliminators. We expect that such a development environment will lead to more widespread use and experimentation not only of cubical type theory but also of HoTT/UF, in particular for programming applications.

7.1 Related Work

This work is based on the work on cubical type theory of Cohen *et al.* (2018) and Coquand *et al.* (2018) and the cubicaltt prototype implementation (Cohen *et al.*, 2015). However, that implementation did not have support for many of the features of a modern proof assistant (implicit arguments, type inference, powerful pattern-matching, etc.) so Cubical Agda can be seen as its successor. Additionally, the transport structure for inductive families is based on the schema presented in Cavallo & Harper (2019*b*).

The most closely related cubical proof assistant to Cubical Agda is redtt (The RedPRL Development Team, 2018), which also supports computable univalence and HITs. It is based on a variation of cubical type theory called *cartesian* cubical type theory. This has models in cartesian cubical sets (Angiuli *et al.*, 2019) and cartesian cubical computational type theory (Angiuli *et al.*, 2018; Cavallo & Harper, 2019*b*). The redtt system has been developed from scratch in order to be a proof assistant for cubical type theory and it has some features that are not in Cubical Agda yet, like pre-type universes and extension types inspired by Riehl & Shulman (2017).

The work of Tabareau *et al.* (2018, 2020) extends Coq with the ability to transport programs and properties along equivalences using what the authors call *univalent parametricity*. While this achieves some consequences of constructive univalence it does not provide computational content to the full univalence axiom, in particular to neither function nor propositional extensionality. There is also no support for HITs.

The computation rules for equality are also defined by cases on the type in *Observational Type Theory* (OTT) (Altenkirch & McBride, 2006; Altenkirch *et al.*, 2007). This type theory also proves function and propositional extensionality without sacrificing typechecking and constructivity, however it satisfies UIP. Recently, the XTT type theory has been developed (Sterling *et al.*, 2019) to reconstruct OTT's exact equality using cubical methods, satisfying UIP rather than univalence. Languages like XTT and OTT can be used as an extensional substrate for a two-level type theory (Voevodsky, 2013; Annenkov *et al.*, 2017), which would have both equality and path types.

Examples of ideas from HoTT/UF in computer science include Angiuli *et al.* (2016) where the authors use univalence and HITs to model `Darcs` style patch theory. This work envisioned what could be done if these notions were computing, but at the time it was unknown how to make this happen. However, now that `Cubical Agda` supports this, it would be interesting to redo the examples as the implementation would now compute. Another example is `HoTTSQL` (Chu *et al.*, 2017) which defines a formal `SQL` style language. The use of HoTT/UF is restricted to reasoning about cardinal numbers and it is not clear how much would be gained from doing this cubically.

Since the conference version (Vezzosi *et al.*, 2019) of this article was published some interesting formalizations have been performed using `Cubical Agda`. Forsberg *et al.* (2020) implemented three equivalent ordinal notations systems and transported programs and proofs between them. Altenkirch & Scoccola (2020) considered a higher inductive version of the integers which differs from the one in Sect. 2.4.1. Veltri & Vezzosi (2020) formalize the π-calculus using a *guarded* version of `Cubical Agda` (Birkedal *et al.*, 2019). Various results from synthetic homotopy theory, extending Sect. 2.5, were developed by Mörtberg & Pujet (2020). Finally, Angiuli *et al.* (2020) explored consequences of cubical type theory and `Cubical Agda` to traditional computer science applications like program/proof transfer and representation independence.

7.2 Future Work

Interesting further directions would be to study meta-theoretical properties of cubical type theory, including a proof of decidability of type-checking and a complete correctness proof of the conversion checking algorithm with respect to a declarative specification of equality. We believe this can be done by extending the canonicity proof of Huber (2016) using ideas from Abel *et al.* (2017).

We would also like to extend `Cubical Agda` with more cubical features, like cubical extension types inspired by Riehl & Shulman (2017). An important open problem in the area of constructive synthetic homotopy theory is to compute the Brunerie number (Brunerie, 2016) which so far has proved to be infeasible using `cubicaltt` and `Cubical Agda`. It would hence be interesting to study compilation and efficient closed term evaluators of cubical languages in order to be able to do this kind of computations.

References

Abel, A., Pientka, B., Thibodeau, D. and Setzer, A. (2013) Copatterns: Programming infinite structures by observations. *SIGPLAN Not.* **48**(1):27–38.

Abel, A., Öhman, J. and Vezzosi, A. (2017) Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.* **2**(POPL):23:1–23:29.

Agda development team. (2018) *Agda 2.5.4.2 documentation*.

Ahrens, B., Capriotti, P. and Spadotti, R. (2015) Non-wellfounded trees in homotopy type theory. *CoRR* **abs/1504.02949**.

Altenkirch, T. and McBride, C. (2006) *Towards Observational Type Theory*. Unpublished draft.

Altenkirch, T. and Scoccola, L. (2020) The Integers as a Higher Inductive Type. *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '20, pp. 67–73. Association for Computing Machinery.

Altenkirch, T., McBride, C. and Swierstra, W. (2007) Observational equality, now! *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification* pp. 57–68. ACM.

Angiuli, C., Morehouse, E., Licata, D. R. and Harper, R. (2016) Homotopical patch theory. *Journal of Functional Programming* **26**.

Angiuli, C., (Favonia), K.-B. H. and Harper, R. (2017) *Computational Higher Type Theory III: Univalent Universes and Exact Equality*. Preprint arXiv:1712.01800v1.

Angiuli, C., Hou (Favonia), K.-B. and Harper, R. (2018) Cartesian Cubical Computational Type Theory: Constructive Reasoning with Paths and Equalities. Ghica, D. and Jung, A. (eds), *27th EACSL Annual Conference on Computer Science Logic (CSL 2018)*. Leibniz International Proceedings in Informatics (LIPIcs) 119, pp. 6:1–6:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

Angiuli, C., Brunerie, G., Coquand, T., Hou (Favonia), K.-B., Harper, R. and Licata, D. R. (2019) *Syntax and Models of Cartesian Cubical Type Theory*. Preprint.

Angiuli, C., Cavallo, E., Mörtberg, A. and Zeuner, M. (2020) *Internalizing Representation Independence with Univalence*. Preprint arXiv:2009.05547v2 [cs.PL].

Annenkov, D., Capriotti, P. and Kraus, N. (2017) *Two-Level Type Theory and Applications*.

Bickford, M. (2018) Formalizing Category Theory and Presheaf Models of Type Theory in Nuprl. *CoRR* **abs/1806.06114**.

Birkedal, L., Bizjak, A., Clouston, R., Grathwohl, H. B., Spitters, B. and Vezzosi, A. (2019) Guarded Cubical Type Theory. *Journal of Automated Reasoning* **63**(2):211–253.

Brady, E. (2013) Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* **23**(5):552–593.

Brunerie, G. (2016) *On the homotopy groups of spheres in homotopy type theory*. PhD thesis, Université de Nice.

Cavallo, E. and Harper, R. (2019a) Higher inductive types in cubical computational type theory. *Proc. ACM Program. Lang.* **3**(POPL).

Cavallo, E. and Harper, R. (2019b) Higher inductive types in cubical computational type theory. *Proc. ACM Program. Lang.* **3**(POPL):1:1–1:27.

Chu, S., Weitz, K., Cheung, A. and Suciu, D. (2017) Hottsql: Proving query rewrites with univalent sql semantics. *SIGPLAN Not.* **52**(6):510–524.

Cockx, J. and Abel, A. (2018) Elaborating dependent (co)pattern matching. *Proc. ACM Program. Lang.* **2**(ICFP):75:1–75:30.

Cockx, J. and Devriese, D. (2018) Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *Journal of Functional Programming* **28**:e12.

Cohen, C., Dénès, M. and Mörtberg, A. (2013) Refinements for Free! Gonthier, G. and Norrish, M. (eds), *Certified Programs and Proofs*. Lecture Notes in Computer Science 8307, pp. 147–162. Springer International Publishing.

Cohen, C., Coquand, T., Huber, S. and Mörtberg, A. (2015) *Cubicaltt*. https://github.com/mortberg/cubicaltt.

Cohen, C., Coquand, T., Huber, S. and Mörtberg, A. (2018) Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. *Types for Proofs and Programs (TYPES 2015)*. LIPIcs 69, pp. 5:1–5:34.

Coquand, T. and Danielsson, N. A. (2013) Isomorphism is equality. *Indagationes Mathematicae* **24**(4):1105–1120.

Coquand, T., Huber, S. and Mörtberg, A. (2018) On higher inductive types in cubical type theory. *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '18, pp. 255–264. ACM.

Coquand, T., Huber, S. and Sattler, C. (2019) Homotopy canonicity for cubical type theory. Preprint available at http://www.cse.chalmers.se/ simonhu/papers/can.pdf.

Danielsson, N. A. (2020) *Higher Inductive Type Eliminators Without Paths*. http://www.cse.chalmers.se/~nad/publications/danielsson-hits-without-paths.html.

de Moura, L., Kong, S., Avigad, J., van Doorn, F. and von Raumer, J. (2015) The lean theorem prover. *Automated Deduction - CADE-25, 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*.

Escardó, M. H. (2019) *Introduction to univalent foundations of mathematics with Agda*.

Forsberg, F. N., Xu, C. and Ghani, N. (2020) Three Equivalent Ordinal Notation Systems in Cubical Agda. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020, pp. 172–185. Association for Computing Machinery.

Huber, S. (2016) *Canonicity for Cubical Type Theory*. Preprint arXiv:1607.04156.

Huber, S. (2017) *A Cubical Type Theory for Higher Inductive Types*.

Kapulkin, C. and Lumsdaine, P. L. (2012) *The Simplicial Model of Univalent Foundations (after Voevodsky)*. Preprint arXiv:1211.2851v4.

Licata, D. R. and Brunerie, G. (2015) A cubical approach to synthetic homotopy theory. *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS'15* pp. 92–103. ACM.

Licata, D. R. and Shulman, M. (2013) Calculating the Fundamental Group of the Circle in Homotopy Type Theory. *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS'13, pp. 223–232.

Licata, D. R., Orton, I., Pitts, A. M. and Spitters, B. (2018) Internal universes in models of homotopy type theory. Kirchner, H. (ed), *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*. LIPIcs 108, pp. 22:1–22:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Lumsdaine, P. L. and Shulman, M. (2017) *Semantics of higher inductive types*. Preprint arXiv:1705.07088.

Martin-Löf, P. (1975) An intuitionistic theory of types: Predicative part. Rose, H. E. and Shepherdson, J. (eds), *Logic Colloquium '73*, pp. 73–118. North–Holland, Amsterdam.

McBride, C. (2009) Let's see how things unfold: Reconciling the infinite with the intensional. *Proceedings of the 3rd International Conference on Algebra and Coalgebra in Computer Science*. CALCO'09, pp. 113–126. Springer-Verlag.

Mörtberg, A. and Pujet, L. (2020) Cubical Synthetic Homotopy Theory. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020, pp. 158–171. Association for Computing Machinery.

Orton, I. and Pitts, A. M. (2016) Axioms for modelling cubical type theory in a topos. *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*. LIPIcs 62, pp. 24:1–24:19.

Riehl, E. (2019) *The equivariant uniform Kan fibration model of cubical homotopy type theory*. Talk given at *The International Conference on Homotopy Type Theory (HoTT 2019)* at Carnegie Mellon University.

Riehl, E. and Shulman, M. (2017) A type theory for synthetic ∞-categories. *Higher Structures* **1**(1):147–224.

Sattler, C. (2018) *Do cubical models of type theory also model homotopy types?* Talk given at *Types, Homotopy Type theory, and Verification* at the Hausdorff Center for Mathematics in Bonn.

Sojakova, K. (2016) The Equivalence of the Torus and the Product of Two Circles in Homotopy Type Theory. *ACM Transactions on Computational Logic* **17**(4):29:1–29:19.

Sterling, J., Angiuli, C. and Gratzer, D. (2019) Cubical syntax for reflection-free extensional equality. Geuvers, H. (ed), *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019*. Leibniz International Proceedings in Informatics, LIPIcs. Schloss Dagstuhl- Leibniz-Zentrum fur Informatik GmbH, Dagstuhl Publishing.

Tabareau, N., Tanter, E. and Sozeau, M. (2018) Equivalences for free: Univalent parametricity for effective transport. *Proc. ACM Program. Lang.* **2**(ICFP):92:1–92:29.

Tabareau, N., Éric Tanter and Sozeau, M. (2020) *The Marriage of Univalence and Parametricity*. Preprint arXiv:1909.05027 [cs.PL].

Team, T. C. D. (2019) *The Coq Proof Assistant, version 8.9.0*.

The RedPRL Development Team. (2018) *The* `redtt` *Proof Assistant*.

Univalent Foundations Program, T. (2013) *Homotopy Type Theory: Univalent Foundations of Mathematics*.

Veltri, N. and Vezzosi, A. (2020) Formalizing π-Calculus in Guarded Cubical Agda. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2020, pp. 270–283. Association for Computing Machinery.

Vezzosi, A. (2017) *Streams for Cubical Type Theory*. http://saizan.github.io/streams-ctt.pdf.

Vezzosi, A., Mörtberg, A. and Abel, A. (2019) Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages* **3**(ICFP):87:1–87:29.

Voevodsky, V. (2013) *A simple type system with two identity types*.

Voevodsky, V. (2015) An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science* **25**:1278–1294.

Wood, J. (2019) *Vectors and Matrices in Agda*. Blog post at https://personal.cis.strath.ac.uk/james.wood.100/blog/html/VecMat.html.