HIT Me with Your Best STLC

Forming Quotients over the Simply Typed
Lambda Calculus in Cubical Agda

Noah Goodman

Submitted in partial fulfillment of the
requirements for honors in the degree of
Bachelor of Science in Computer Science
at the University of Chicago

Chicago, IL
May 2021

Contents

1 Introduction

Proof assistants such as Agda and Coq are often used in the fields of programming language research and type theory to formalize the syntax and semantics of the various type system that are the objects of study in these fields. Using such tools is greatly beneficial, as often enough, carrying out proofs about properties of type systems by hand can be rather tedious and error prone. And while there are countless variations in the different sorts of type systems being studied and formalized via proof assistants, there are also many different techniques that one can use within these tools to study these systems.

Accordingly, when one is formalizing a type system, one must make several choices about which techniques to use or not. For example, some choices may include whether or not to use a deep or shallow embedding, have extrinsically or intrinsically typed terms, use De Bruijn indices or not, and more. In almost all cases, each option will have its pros and cons. For example, taking the example of whether to use De Bruijn indices or, say, meta-level strings for variables, there are arguments in either direction. De Bruijn indices are nice because they make it impossible for one to even refer to an out-of-scope variable, they alleviate worries about renaming when defining substitution, and they make the question of α-convertibility a moot point. However, they can be harder to read and write, and while we don't need to worry about variable capture or having to rename anything during substitutions, we do have to explicitly work with various weakening operations instead.

In the case of De Bruijn indices, there's an essential trade-off occurring. We gain the assurance that certain invariants or properties (in this case, proper scoping and trivial α-conversion) are upheld by the type checker in exchange for some added complexity.

Another property that may be desirable in some type systems is that the various operations on the term structure (like substitution, weakening, evaluations, etc.) respect β- and η-equivalence on terms. Usually, it's not too difficult to provide such facts as lemmas or theorems about the operations as defined on normal terms, but it turns out that it's also possible to define a type of terms in such a way that any operation defined on these terms automatically respects $\beta\eta$-equivalence. This is possible using *quotient types*, which are types that represent the mathematical operation of taking a quotient on a set. That is, quotient types allow us to define a base type (in our case, terms) and then identify certain elements of this type with one another (usually via an equivalence relation, for us $\beta\eta$-equivalence), yielding a new type.

In this paper, we will investigate different approaches towards and techniques for quotienting the term grammar of the Simply Typed Lambda Calculus (STLC) by $\beta\eta$-equivalence using Higher Inductive Types in Cubical Agda. Cubical Agda [VMA21] is a relatively new extension for the proof assistant Agda, which implements various concepts from Homotopy Type Theory, including Higher Inductive Types, and Higher Inductive Types, in turn, will allow us to define quotient types in a fairly elegant manner. Additionally, this paper will also serve as a brief introduction to Homotopy Type Theory and Cubical Agda, while also demonstrating various proof techniques and features of the latter.

Section 5 of this paper is fully formalized in Agda, development version 2.6.2. This development, along with most of the work discussed in sections 4 and 6 can be found in [Goo].

2 Background

In this section, we'll cover some background information on the Agda proof assistant, along with Homotopy Type Theory and Cubical Agda.

2.1 Agda

The reader should be generally familiar with the Agda [Nor07] (or similar) proof assistant, but I'll cover a few specific details about it here that are especially relevant.

2.1.1 Propositional equality

Due to the Curry-Howard isomorphism, dependent type theories such as Martin-Löf Type Theory (abbr. MLTT) [MLS84] or Agda's type system can be seen as formal systems for doing constructive mathematics, and as such, these systems contain analogues to all the familiar connectives from first order logic. The equality relation, of course, is no exception, and in Agda, this relation is formalized[1] in a similar[2] manner as follows:

```
data _≡_ {A : Set} : A → A → Set where
  refl : ∀ {x : A} → x ≡ x
```

That is, a ≡ b is the type of proofs that a is the same as b, and the only way we can construct evidence of "sameness" is by applying reflexivity via the data constructor refl. Note that this equality type is often called "propositional equality", in contrast with "judgmental" or "definitional equality", which is the meta-level equality judgment of the type system.

We can see that the only way to introduce a proof of equality is through refl, but how can one eliminate a proof of equality in Agda? Or rather, given a proof of equality, how can we use this to prove other stuff? Generally speaking, in Agda, inductively defined data types are simply eliminated by dependent pattern-matching, and propositional equality is no exception. Assuming we have a proof p : a ≡ b, then pattern matching on p will unify a and b, and p will be unified with refl, causing any mentions of p in the goal or other dependent arguments to be replaced with refl.

[1]Note that implicit arguments are contained in braces and that Set refers to Agda's universe of small types.

[2]the true definition is slightly more general, but this gets the idea across and is easier to read

Dependent pattern matching, however, is really just a user-level convenience. Internally, dependent pattern matching gets elaborated to case trees in the core language, which then could be translated to primitive eliminators (but this doesn't happen in Agda) [CA18]. Hence, we can expect that any proof that follows by dependent pattern matching on some evidence of the form p : a ≡ b for distinct a and b[3] will also work using the induction principle for _≡_, which is known as J [Str93, agd]:

```
J : {A : Set} (P : (x y : A) → x ≡ y → Set) →
  ((x : A) → P x x refl) → (x y : A) (x≡y : x ≡ y) → P x y x≡y
```

J basically says that if we're trying to prove some proposition P that depends on some arbitrary proof of equality p between some elements x and y, it suffices to simply prove that P holds on refl. Similarly, if we pattern match on a proof of equality p when trying to prove something, we unify p with refl, and the resulting proof obligation is essentially the base case required by the J rule.

J is rather general, and indeed, with only J (or the form of dependent pattern matching we outlined above) one can prove most things of interest [4], but Agda's pattern matching mechanism is actually even more general than what we described. Not only can we eliminate evidence of the form p : a ≡ b via dependent pattern matching, but this also works on evidence of the form p : a ≡ a. Now, it might seem strange that these two forms of pattern matching constitute distinct cases to begin with, since the second case seems to just be a special case of the first, but it's fairly easy to demonstrate a situation in which J fails, but Agda's pattern matching has no problem.

Suppose we want to prove that any proof that x ≡ x is itself equal to refl; that is, that identity proofs are unique (this is called Uniqueness of Identity Proofs, or UIP, for short). We start with some proof p : x ≡ x, but if we try to prove that p ≡ refl using J, this is as far as we get:

```
uip : ∀ {A : Set} (x : A) → (p : x ≡ x) → p ≡ refl
uip x p = J (λ y z q → q ≡ {! !}) (λ _ → refl) x x p
```

The problem is that we'd like to fill in the hole (denoted {!!}) in the motive with refl, leading us to the exact property we want to prove, but refl can't fill this hole because q has type y ≡ z, so the contents of the hole must also have the same type. refl, however, can only have type y ≡ y or z ≡ z. In general, the problem is that the motive for eliminating propositional equalities is generalized to consider equalities between two distinct elements, and while applications of the motive can be specified to one element (such as P x x refl), the motive itself must be general (otherwise we wouldn't be able to prove *most* of the things we'd like to).

On the other hand, Agda's pattern matching facilities make short work of this proof:

[3] We'll see momentarily why we need to make this stipulation

[4] For example, this is the extent of equality elimination in Coq

```
uip′ : ∀ {A : Set} (x : A) → (p : x ≡ x) → p ≡ refl
uip′ x refl = refl
```

So we can see that Agda's pattern matching also can eliminate equalities of the form p : x ≡ x, and doing so will, in practice, unify any occurrences of p with refl. As with the first form of pattern matching we saw above (which was equivalent to applications of J), this form of pattern matching also has an equivalent elimination principle, known as K[5] [Str93, agd]:

```
K : {A : Set} {x : A} (P : x ≡ x → Set) →
  P refl → (x≡x : x ≡ x) → P x≡x
```

Indeed, we can prove Uniqueness of Identity Proofs using an application of axiom K:

```
uip′′ : ∀ {A : Set} (x : A) → (p : x ≡ x) → p ≡ refl
uip′′ x p = K (λ q → q ≡ refl) refl p
```

So we can see why there is a need to make a distinction between different cases of pattern matching on proofs of equality in Agda from the point of view of giving these matches semantics in identity eliminators, but aside from that, why do we actually care? As it turns out, the K rule is actually incompatible with some forms of type theory (such as Homotopy Type Theory), so if we want to develop proofs compatible with such systems, we will need to restrict our pattern matching capabilities so that the type checker does not need to assume axiom K's validity in order to type check any pattern matches. One result of this restriction is that we will only be able to eliminate propositional equalities of the form x ≡ y. In order to facilitate such developments where we cannot assume K, Agda provides an option called `--without-K` that restricts pattern matching in such a way. For a more in depth discussion of how `--without-K` actually works, see [CDP14].

2.2 Homotopy Type Theory

This introduction to Homotopy Type Theory is based loosely off of the introduction from the HoTT/UF book. For more details, see [Uni13].

Homotopy Type Theory (commonly abbreviated HoTT) is a fairly recent field of research that combines ideas from type theory, homotopy theory, and category theory, all with the goal of formulating an alternative foundations for mathematics to the standard set theoretic foundations used by most mathematicians. This proposed alternative foundations for mathematics, called Univalent Foundations, takes the form of something that resembles Martin Lof̈ Type

[5]Note that the actual criteria that the unification algorithm behind dependent pattern matching uses are more general and technical than the basic distinction we're making here, and recall, again, that Agda does not actually translate pattern matches into applications of J or K, but rather, it must assume these principles at different times to type check certain matches. For more details, see [CA18, CDP14]

Theory plus a few additional, key axioms, along with a perspective rooted in homotopy theory.

Before addressing the specific details that set Homotopy Type Theory apart from standard type theory, we first need to address the issue of how type theory and homotopy theory are even related in the first place.

Homotopy theory studies abstract spaces (consisting of points and paths between them) and maps between spaces, while mainly considering such spaces and maps "up to homotopy". Given two spaces and two maps from one to the other, a homotopy between these maps is essentially a way that they can be "continuously deformed" into one another. Furthermore, this relationship between maps induces an equivalence between spaces called "homotopy equivalence". It's not really important to get into the details of what a homotopy equivalence between two spaces actually is, but the upshot is that in homotopy theory, homotopy equivalent spaces are considered more-or-less "the same", so the object of study isn't really individual spaces, but rather equivalence classes of spaces under homotopy equivalence. Such a class is called a *homotopy type*.

This brings us to Homotopy Type Theory. HoTT resembles type theory syntactically and carries over many concepts from traditional type theory, but instead of thinking of a type as a kind of set consisting of similar elements, we now think of a type as a space, and the inhabitants of a type can be thought of as points in this space. This on its own, perhaps, is not so interesting, as this is just a renaming of objects we already have in plain type theory, but this perspective leads to a new interpretation of propositional equality, which in a sense, is the crux of the entire theory.

In HoTT, the type of propositional equality between two elements $a \equiv_A b$ is now viewed as the space of paths between the points a and b in A. That is, a proof $p : a \equiv_A b$ is essentially *one* specific path between a and b. Indeed, in HoTT, identity proofs are no longer unique. This is topologically justified, since if proofs of equality are viewed as paths, then a proof that two identity proofs are the same would consist of a "path" from one path to the other, but this is just a homotopy, and it's certainly not the case that distinct paths in a space are homotopic, in general.

2.2.1 Higher Inductive Types

To summarize so far, proofs of equality in HoTT are viewed as paths between points in a space, and as such, there may be multiple non-trivial, distinct paths between two given points. In addition, when defining a space, we have the ability to specify when such paths exist between certain points. Recall that in standard type theory with user-defined inductive data types, one can inductively define arbitrary types by giving different (possibly recursive) constructors for inhabitants of the type one is defining. Or in HoTT terminology, we can inductively specify a space by giving rules for which points exist in this space. HoTT extends this idea with Higher Inductive Types, which are just inductively defined types in which the user can also specify non-trivial *paths* that exist between points (or paths, or paths between paths, etc.) in the space they are defining.

For example, one might start by specifying the discrete two point space (a.k.a. the type of Booleans) via two constructors $\mathsf{true} : \mathsf{Bool}$ and $\mathsf{false} : \mathsf{Bool}$, but then specify an extra *path constructor*:

$$\mathsf{t\text{-}eq\text{-}f} : \mathsf{true} \equiv \mathsf{false}$$

While it's certainly not the case that the two points in the discrete two point space are the same (or that true equals false), by adding the path constructor t-eq-f, we've essentially identified these two points. As a result, we now get that the two inhabitants of this type are effectively indistinguishable (because we explicitly said that they're the same).

This previous example is not incredibly interesting, but it demonstrates the general principle of HITs. In practice, the ability to specify non-trivial higher inductive paths between elements of a type is incredibly useful. When it comes to formalizing mathematics, this ability allows one to define various topological spaces in a very direct fashion. And when it comes to programming, HITs allow one to define quotient types and other related constructions (eg. multisets, $\mathbb{Z}$, etc.) in a direct manner[6].

2.2.2 Univalence

Another one of HoTT's major innovations is the Univalence Axiom. The Univalence Axiom (UA for short), states that an equivalence between two types is equivalent to an equality between the types. In HoTT, "equivalence" is a defined notion, which corresponds roughly to homotopy equivalence under the interpretation of types as spaces, so what this axiom is really doing is formalizing the common informal notion that two equivalent or isomorphic objects are effectively "the same".

An important practical consequence of Univalence is that it allows us to transport proofs and definitions between isomorphic structures in a principled way. A classic example of this principle is the case of how natural numbers are represented. The natural numbers can be represented by a unary or a binary encoding in type theory, and both of these encodings have their pros and cons. For example, it's often easier to perform induction over the unary version of the natural numbers, but the binary encoding is more space efficient and also generally lends itself better to problems that deal with doubling of natural numbers. In the end, though, our choice of encoding does not affect the fact that the natural numbers are the natural numbers. That is to say, these two encodings are isomorphic [7]. Then, what univalence gives us is a way to easily port functions defined and results proven on one encoding to the other encoding. For more details, and an example of this principle in action, see [VMA21, Chapter 2.1].

[6]For examples of both of these applications in Cubical Agda, see `Cubical.HITs` in [cub].

[7]this fact is a common beginning exercise for learning proof assistants, found in both Software Foundations and PLFA.

2.2.3 h-sets/h-levels

As we mentioned earlier, in HoTT, we can now have multiple distinct paths (or equivalently, proofs of equality) between two elements of a type, but not all types have this higher-dimensional path structure. Some types just resemble normal types in MLTT in that there is at most one path between any two elements. Such types are known as h-sets (because their structure resembles the interpretation of types as sets, ie. a bunch of discrete points in space without any extra paths between them).

More generally, h-sets are just a single level of a hierarchy of h-levels indexed by the natural numbers. A type of h-level 0 is defined to be a contractible type, which is a type that has a single element with paths to every other element of this type. In practice, such a type is equivalent to the unit type. For the inductive case, a type A has h-level $n + 1$ if for all $x, y \in A$, the identity type $x \equiv y$ has h-level n. Filling in the holes, this implies that types of h-level 1 are those that are "trivial/contractible if inhabited". Recalling our definition of an h-set, we see that an h-set is a type whose identity types have h-level 1 (the only path between any two points is the trivial/reflexive path, if it exists), so h-sets are just types with h-level 2. For more on this, see chapters 3 and 7 in [Uni13].

2.3 Cubical Agda

Cubical Agda [VMA21] is an extension for Agda based off of Cubical Type Theory, which serves as a model for Homotopy Type Theory in which the Univalence Axiom has a computational interpretation. This means that not only does Cubical Agda implement various key features of HoTT such as Univalence, Path types, Higher Inductive Types, and much more, but these constructs also carry computational significance. Most notably, this allows for the user to transport elements and operators along non-trivial equalities/paths.

In addition to the core Cubical Agda language extension, the associated Cubical library [cub] provides a large number of features aimed at carrying out proofs in HoTT, along with more general utilities. We will focus on just a few of the basics of Cubical Agda and the associated library, along with some discussion of certain features that will prove useful later on.

2.3.1 Paths

First, and most importantly, we must consider propositional equality. As we saw earlier, propositional equality in standard Agda is simply defined as an inductive data type, and the only way to construct a proof of this equality was via reflexivity. Cubical Agda, however, deals with equality in a manner that is closer to the perspective taken in Homotopy Type Theory. As in HoTT, equality in Cubical Agda is thought of as a `Path` between two elements of a given type; however, in Cubical Agda, this intuition is taken quite literally, as a path in some type A is represented as a function from the unit interval to A[8]. The way this is accomplished is via a unit interval type I with two distinguished endpoints i0 and i1. So now, if we want to prove that 2 equals 2, then we just need to provide a path from 2 to 2 as follows:

```
two≡two : Path ℕ 2 2
two≡two = λ i → 2
```

When constructing a path in this way, the type checker makes sure that the path's endpoints match what they're supposed to be. Since we view the path two≡two as a function I → ℕ, this just amounts to checking that the left endpoint, two≡two i0 is 2 and that the right endpoint two≡two i1 is, as well. This is indeed the case (the path we've given is 2 everywhere, in fact), so we see that we have given a valid proof that 2 equals 2.

In many cases, reasoning about equalities by thinking of their proofs as functions out of the unit interval can lead to rather elegant proofs. For example, the congruence rule cong can be proven in Cubical Agda as follows:

```
cong : ∀ {A B : Type} (f : A → B) {x y : A}
  → Path A x y → Path B (f x) (f y)
cong f p i = f (p i)
```

Here, we note that since p is a path from x to y, p i0 = x and p i1 = y, so f (p i0) = f x and f (p i1) = f y, which is exactly what we want. Also note note that Cubical Agda renames Set to Type to prevent confusion with the HoTT notion of h-set, and that Path A x y can also be written as x ≡ y, which we will do from now on.

Sometimes, however, it is convenient to stick with the more traditional style of manipulating proofs of equality via various utility functions such as cong, subst, trans, etc. And accordingly, all of the standard functions for reasoning about and manipulating propositional equality have fairly direct analogs on path equality, all of which are provided in the Cubical library.

One technique for reasoning about propositional equality that has no direct analog in Cubical Agda, however, is the technique of eliminating proofs of equality via dependent pattern matching. Since a proof of equality in Cubical Agda is really a path, which is kind of like a function (but is actually a primitive), there's no meaningful way to pattern match on one. In many (but not all) cases, however, a proof that previously used pattern matching on evidence of propositional equality can be carried through in Cubical Agda by manually applying the path induction principle J (which is defined analogously to J in standard Agda) one or more times. One limitation of J in Cubical Agda, however, is that it does not always compute definitionally as one might expect. For example, when transporting along equalities using path induction [9], we would expect that

[8]Internally a Path between two elements in A is not really a function from the unit interval to A, but the syntax for introducing and eliminating such a path is practically the same as for functions, allowing us to reason about them as if this were the case.

[9]which is really just the same thing as using Cubical Agda's transport function, as J is defined via transport

transporting along the constant path (aka. refl) would simply compute as the identity function, allowing us to carry out a proof such as:

```
transportExample : ∀ {A : Type} → (a : A) → transport refl a ≡ a
transportExample a = {! refl !}
```

But trying to fill in the hole with refl, as we see here doesn't work. Instead we get the error:

```
transp (λ i → refl i) i0 a != a of type A
```

Which essentially means that Agda can't unify transport refl a and a, because the former does not reduce, as we might expect, to the latter. All hope is not lost, however, as our desired computation rule does hold "up to a path". That is, while the thing we want to prove doesn't follow definitionally, Cubical Agda does allow us to construct a path between transport refl a and a, using the function transportRefl, allowing us to complete the proof as follows:

```
transportExample a = transportRefl a
```

In cases where we're using J as opposed to transport, we also have JRefl, which is really just wrapper for transportRefl that takes an extra parameter for the path induction motive.

This previous example showed how we can adapt proofs that used pattern matching on equality in a way that is equivalent to J into Cubical Agda, but the same technique does not apply to situations in which we try to eliminate a path of the form x ≡ x. As we mentioned in our discussion of propositional equality, performing such an elimination requires us to assume axiom K, which in turn implies UIP, but in HoTT, UIP is simply false! Indeed, the existence of Higher Inductive Types means that we could inductively define a type with non-trivial proofs of identity for some elements[10]. Hence, theorems whose proofs previously relied on K can no longer be proved in Cubical Agda. In many cases, though, such theorem statements can be generalized into provable forms that can be specialized as needed later (see section 6.1),

2.3.2 Higher Inductive Types

Cubical Agda also provides support for user-defined Higher Inductive Types. These are declared in the same way that normal inductive types are, plus extra path constructors that specify extra non-trivial paths that exist between elements (or paths between elements, or path between paths, etc.) of this type. For example, the Cubical library defines the HIT for (h-)set quotients as follows:

```
data _/_ (A : Type) (R : A → A → Type) : Type where
  [_] : (a : A) → A / R
```

[10]For example, the circle, S^1 can be defined as have some point base : S^1 along with a path constructor loop : base ≡ base. In this case loop is a non-trivial identity proof for base.

```
  eq/ : (a b : A) → (r : R a b) → [ a ] ≡ [ b ]
  squash/ : (x y : A / R) → (p q : x ≡ y) → p ≡ q
```

Here, we're using Agda's infix notation to declare the type constructor as an infix binary operator, allowing a set quotient type to be written as A / R. Now, let's walk through what's going on in this definition. To begin with, we're quotienting the underlying type A by some binary relation over A called R[11]. The first constructor [_] is a normal data constructor that allows us to simply embed an element of A into A / R. The second constructor eq/ is an example of a path constructor. Here, we're specifying that for all a and b in A that are related by R, we identify the elements [a] and [b] in A / R. Finally, we'd like this quotient type to be an h-set, so we need to provide the path constructor squash/, which "set-truncates" A / R, essentially collapsing any higher-dimensional structure by identifying identity proofs on base elements of A / R.

Now, if we want to eliminate an element on an HIT, then just like for any inductive type in Agda, we can use pattern matching. In the case of pattern matching on path constructors, we essentially need to provide a proof for each path constructor that the previous cases of pattern matching do not distinguish between elements of our type that we've identified via the path constructor in question. More concretely, if we want to define a function A / R → C for some type C via pattern matching, we need to specify a function f from A to C, along with a proof that for all a b : A, R a b implies that f a ≡ f b. In addition we need to show that C is an h-set in order to define the case for squash/. Indeed, the recursion principle for set quotients is defined in the Cubical Agda library via pattern matching as follows:

```
rec : {B : Type ℓ}
  (Bset : isSet B)
  (f : A → B)
  (feq : (a b : A) (r : R a b) → f a ≡ f b)
  → A / R → B
rec Bset f feq [ a ] = f a
rec Bset f feq (eq/ a b r i) = feq a b r i
rec Bset f feq (squash/ x y p q i j) =
  Bset (g x) (g y) (cong g p) (cong g q) i j
  where
  g = rec Bset f feq
```

And from the type signature of the recursor, we see that this is simply how one typically defines a function on a quotient set in mathematics (minus the set-truncation condition).

[11]Note that R itself does not need to be an equivalence relation, as the resulting quotient really is formed by the reflexive, symmetric, transitive closure of R, due to the fact that eq/ lifts relations given by R to equalities (which do have these properties).

2.3.3 Univalence and isomorphism

Cubical Agda also provides an interpretation of the Univalence Axiom from Homotopy Type Theory, which takes the form of the function ua. This function takes an equivalence between two types and returns a path between them, and have an equivalence between two types if there is a map between them with contractible fibers. That is, if our map goes from A to B, then for each y in B, the set of x in A such that f maps x to y is contractible.

A fact that is often very useful for demonstrating equivalences is that any two isomorphic types are equivalent. In (Cubical) Agda, an isomorphism between two types A and B is given by a record type consisting of a map fun : $A \to B$ and an inverse map inv : $B \to A$, along with proofs that inv is both a right and left inverse to fun. So if one can provide these two maps, along with the two proofs that inv is indeed an inverse to fun, then by turning this isomorphism into an equivalence (using isoToEquiv defined in the Cubical library) we can apply univalence to arrive at a path $A \equiv B$. As we'll see later, such a path between types can be used to easily transport functions and proofs in one type to the other.

2.3.4 Moving between propositional and Cubical equality

When working with Cubical Agda, the main notion of equality that we deal with is the path equality that we described in the previous sections. In many cases, path equality gives rise to very simple and direct ways of reasoning about equality, but occasionally it can be convenient to use normal propositional equality alongside path equality. To this end, the Cubical library[12] provides us with utilities ptoc (read, Propositional TO Cubical) and ctop for converting between propositional and cubical equality types.

ptoc works by pattern matching on the provided propositional equality and then returning the constant path, and ctop uses transport along the provided path equality to take the propositional proof of reflexivity reflp to a propositional equality of the appropriate type.

These utilities can be useful if we already have a proof using propositional equality from some previous work, and we'd like to use it in a cubical development. One option is to rewrite the entire proof using path equality, but it is much faster and simpler in most cases to simply wrap the propositional proof in ptoc.

While path equality is a rather powerful and flexible notion of equality, it is often slightly more cumbersome to use, as we can't simply pattern match on proofs of path equality as we would with propositional equality. In some cases, though, we can recover this ability locally in a cubical proof by pattern matching on a path equality that we've wrapped in ctop. For example, while it's very easy to prove cong directly using path equality, we can use ctop to recover a proof that resembles the one in standard Agda, which follows by pattern matching on the provided proof of equality:

[12]in `Cubical.Data.Equality`

```
cong f p with ctop p
... | reflp = refl
```

Note that here, refl is the cubical proof of reflexivity. Or even slicker:

```
cong f p rewrite ctop p = refl
```

While this example doesn't really improve over the direct interval-based proof in any way, there are some cases when using this "ctop trick" can greatly simplify a proof that would otherwise require an unwieldy use of subst or other equality combinators.

3 Motivation

The main motivation behind this paper is Thorsten Altenkirch and Ambrus Kaposi's paper, "Type Theory in Type Theory using Quotient Inductive Types" [AK16], in which the authors formalize a small dependent type theory in Agda using a slightly more limited version of Higher Inductive Types known as Quotient Inductive Types (QITs), which are essentially set-truncated HITs. Their approach follows the one in Internal Type Theory [Dyb95], in that it makes use of an explicit substitution calculus to handle term substitutions. This choice is particularly amenable to defining the term structure as a QIT, since β-equivalence gets essentially broken up into a bunch of smaller equivalences involving explicit substitutions. In particular, the use of explicit substitution obviates the need to define a separate substitution function at the same time as one defines the term structure (as we will see, this can lead to problems).

As discussed above, the approach taken in "Type Theory in Type Theory" is a rather natural and elegant way to formalize a type system while quotienting by $\beta\eta$-equivalence, but it may not be ideal in situations where an explicit substitution calculus is undesirable (for example, if explicit substitutions don't interact nicely with another feature of the type theory). In particular, if one's goal is to adapt *pre-existing code* to use Quotient/Higher Inductive Types to quotient the term structure by a notion of $\beta\eta$-equivalence, then this approach may force one to put in a too much work converting all the code one has already written to work with a substitution calculus.

My goals in this paper are twofold: first, I would like to investigate the feasibility of encoding terms modulo $\beta\eta$-equivalence in an HIT in a more traditional style that avoids explicit substitutions, and I would also like to investigate techniques involved in adapting pre-existing Agda developments to be compatible with Cubical Agda. To these ends, I selected the source code [KA10a] from the paper "Hereditary Substitutions for Simple Types, Formalized" [KA10b] by Chantal Keller and Thorsten Altenkirch as a playground for experimenting with different techniques in the pursuit of these two goals. I chose this particular development for several reasons. The first factor that informed this choice was the fact that the results of this paper make essential use of an inductively defined $\beta\eta$-equivalence relation, naturally leading to the question of how this relation (and the results proved about it) can be lifted to a series of higher inductive paths in an HIT. Second, this paper deals with a very simple type system, the Simply Typed Lambda Calculus, which would allow me to focus more on the techniques involved in the translation, as opposed to the difficulties inherent in formalizing more complex type systems. Finally, this formalization makes use of a standalone substitution function (as opposed to one explicitly embedded in the term grammar), which speaks to my first goal of investigating a quotiented type theory that *does not* use explicit substitutions.

4 The direct approach

The first approach I took towards formalizing the Simply Typed Lambda Calculus modulo $\beta\eta$-equivalence was the fairly direct strategy of simply defining an HIT consisting of the term constructors for the STLC plus path constructors corresponding to the required components of $\beta\eta$-equivalence on the term constructors. This approach differs from the one taken in "Type Theory in Type Theory" both in the sense that we seek to directly manipulate term elements of the HIT (as opposed to interpreting syntax in some other model for which we prove the path constructors hold) and in the fact that we do not make use of explicit substitutions in our syntax. We will now outline some different ways to go about this approach and discuss the different considerations one should make when attempting similar endeavors.

4.1 Trying induction-recursion

The first strategy I took towards implementing the approach described above consisted of defining an HIT of terms and equivalences concurrently with term substitution and weakening using a mutual inductive-recursive definition. Since we need a notion of term substitution in order to define our higher inductive path constructor corresponding to β-equivalence and substitution is defined by pattern matching on our term structure, it would make sense to define these two notions at the same time, in a manner roughly akin[13] to Figure 1.

This may seem like a sensible place to start; however, it is not long before we discover some serious issues. First off, the HIT `Tm Γ σ` that we define for our term grammar is not strictly positive! Generally speaking, an inductively defined data type is strictly positive if the type does not appear to the right of an arrow in the types of all arguments to all of its constructors [agd]. Agda checks all **data** declarations for strict positivity, as non-strictly positive data types can allow one to define non-terminating functions, compromising the soundness of the system. It's not exactly easy to see here how our definition of `Tm Γ σ` fails the positivity check, but it essentially boils down to the fact that `Tm Γ σ` appears to the right of an arrow in the type of `subst`, which appears in the path

[13]Note that we leave out implicit parameters in type signatures when they can be inferred from context - this will be the general convention from now on

```
wkTm : (x : Var Γ σ) → Tm (Γ - x) τ → Tm Γ τ
subst : Tm Γ τ → (x : Var Γ σ) → Tm (Γ - x) σ → Tm (Γ - x) τ

data Tm where
  var : Var Γ σ → Tm Γ σ
  Λ : Tm (Γ , σ) τ → Tm Γ (σ ⇒ τ)
  app : Tm Γ (σ ⇒ τ) → Tm Γ σ → Tm Γ τ
  beta : (t : Tm (Γ , σ) τ) (u : Tm Γ σ) → app (Λ t) u ≡ subst t vz u
  eta : (t : Tm Γ (σ ⇒ τ)) → Λ (app (wkTm vz t) (var vz)) ≡ t
    -- We set-truncate our HIT
  trunc : isSet (Tm Γ σ)

wkTm x t = ...
subst t x u = ...
```

Figure 1: General approach for defining terms and equations via induction-recursion

constructor beta. But for the time being, our definition seems sensible enough, so we can ignore the positivity checker using the `NO_POSITIVITY_CHECK` pragma.

We get into even more serious trouble, however, when faced with actually defining our functions subst and wkTm. The problem here arises when defining the weakening function wkTm. This function recurses on an element of our HIT Tm Γ σ, so we need to show that weakening preserves both β and η equivalence. In order to show that wkTm preserves β equivalence, we need to basically prove that wkTm and subst commute in a particular way. But this means that our definition of weakening depends on our definition of substitution, which in turn depends on our definition of weakening. At this point, it's clear that any definition of substitution and weakening will either be impossible due to fundamental coherence issues arising from the dependency between the two, or the solution will involve a series of delicate, mutually defined proofs that are rather difficult to reason about. This seemed needlessly complicated, so I turned towards an approach that avoided induction-recursion on terms and substitution altogether.

4.2 Working directly with quotiented terms

The next approach I adapted abandoned the idea of using mutual induction-recursion to define the term syntax, equivalences, and substitution in favor of an approach that more closely resembles taking the quotient of an already defined term grammar by the $\beta\eta$-equivalence relation. The canonical way to do this, perhaps, would be to literally use a quotient type, defining the type of terms modulo $\beta\eta$-equivalence as simply `Tm Γ σ / _βη-≡_`, where `_/_` is the infix type constructor for set-quotients, and _βη-≡_ is the inductively defined $\beta\eta$-equivalence relation. And indeed, it would be perfectly reasonable to form this quotient and then proceed to define various operations like substitution and weakening on elements of this quotient type, but I settled for a slightly different (albeit isomorphic) approach. Rather than using the predefined set quotient constructor, I instead chose to define a custom HIT consisting of a constructor for embedded terms, [_], along with path constructors for $\beta\eta$-equalities and a few congruences, as in Figure 2.

```
data QTm : Con → Ty → Set where
  [_] : Tm Γ σ → QTm Γ σ
  congΛ : {t t₁ : Tm (Γ , σ) τ}
    → [ t ] ≡ [ t₁ ]
    → [ Λ t ] ≡ [ Λ t₁ ]
  congApp : {t t₁ : Tm Γ (σ ⇒ τ)} {u u₁ : Tm Γ σ}
    → [ t ] ≡ [ t₁ ]
    → [ u ] ≡ [ u₁ ]
    → [ app t u ] ≡ [ app t₁ u₁ ]
  beta : (t : Tm (Γ , σ) τ) (u : Tm Γ σ)
    → [ app (Λ t) u ] ≡ [ subst t vz u ]
  eta : (t : Tm Γ (σ ⇒ τ))
    → [ Λ (app (wkTm vz t) (var vz)) ] ≡ [ t ]
  trunc : isSet (QTm Γ σ)
```

Figure 2: Encoding of terms modulo $\beta\eta$-equivalence into HIT, where the equivalence is broken up into separate path constructors

The advantages of this encoding as opposed to using _/_ are mostly matters of convenience: namely, it allows us to work directly with normal propositional equality _≡_ when proving coherences for our operations that eliminate elements of QTm Γ σ as opposed to proving properties of _βη-≡_ wrapped in eq/[14]. We do, however, lose the convenience of being able to make use of results about _/_ from the Cubical standard library.

Using this approach, we avoid the earlier issues with mutual recursion by defining terms, weakening, and substitution normally (without any higher inductive paths) and then defining our HIT as above with higher inductive paths between embedded terms, using our already defined notion of substitution. However, after forming this type of quotiented terms QTm Γ σ, we would like to lift our normal weakening and substitution functions on Tm's to work on QTm's. In the case of weakening, we can accomplish this fairly straightforwardly by

[14]This is also what enables us to leave out path constructors for reflexivity, symmetry, and transitivity (even though they're defined as part of _βη-≡_), because _≡_ already has these properties

pattern matching on the structure of QTm Γ σ:

```
wkQTm : (x : Var Γ σ) → QTm (Γ - x) τ → QTm Γ τ
wkQTm x [ t ] = [ wkTm x t ]
wkQTm x (congΛ p i) = ...
wkQTm x (congApp p q i) = ...
wkQTm x (beta t u i) = ...
wkQTm x (eta t i) = ...
wkQTm x (trunc t u p q i j) = ...
```

We've elided the details, but all the cases corresponding to path constructors correspond to proofs that wkQTm as we've defined it on [t] respects these equalities. For example in the case of wkQTm x (congΛ p i), we need to show that if we have terms t and u such that [t] ≡ [u], then we have that [wkTm x (Λ t)] ≡ [wkTm x (Λ u)]. In all cases, the proofs follow directly either from structural recursion on terms or from lemmas proved by Altenkirch and Keller in their original development (indeed, they also prove the lemma congWkTm, which proves the equivalent well-definedness result for wkTm with respect to $\beta\eta$-equivalence).

Defining substitution on QTm Γ σ's turns out to be fairly more involved, due to the fact that we need to recurse over both term arguments in order to define this function, leading to a significantly larger number of coherences to prove. This is a situation in which using the standard set quotient construction might be advantageous, as doing so would allow us to use the binary recursor for set quotients from the Cubical library:

```
rec2 : {B : Type ℓ} (Bset : isSet B)
  (f : A → A → B)
  (feql : (a b c : A) (r : R a b) → f a c ≡ f b c)
  (feqr : (a b c : A) (r : R b c) → f a b ≡ f a c)
  → A / R → A / R → B
```

While using a recursor in this manner does not fundamentally change what the programmer must prove, it does help to organize the proofs and ensure that the user does not end up proving more than what is necessary (as some of the cases generated by pattern matching on both arguments are somewhat redundant or follow from other results).

Whether using some sort of recursor or just pattern matching on the constructors of QTm Γ σ, we find ourselves undertaking a fairly feasible, yet tedious task.

5 Univalence to the rescue

In the previous section, we saw that it makes plenty of sense to think about terms modulo $\beta\eta$-equivalence as a concept, but actually defining computations like weakening and substitution on these terms is rather labor-intensive and occasionally quite difficult. One of the major benefits of univalence in Homotopy

Type Theory, however, is that it allows us to transfer proofs and definitions between isomorphic types. For example, in the paper "Three Equivalent Ordinal Notation Systems in Cubical Agda" [FXG20], we see how one can define equivalent notations for ordinals in Cubical Agda, define different arithmetic operations in whichever notation is most convenient, and then transport these operations and proofs about them to apply to the other ordinal notations via univalence.

Similarly, if we can define a type that is isomorphic to our type of terms modulo $\beta\eta$-equivalence on which it is easier to define various notions like weakening and substitution, then we could use univalence to get these operations on quotiented terms for free. And indeed, the type of normal terms `Nf Γ σ` gives us exactly that. This makes sense because a normal term can be thought of as the canonical representative of an equivalence class of STLC terms modulo $\beta\eta$-equivalence. Now, to make this statement formal, we need to construct an isomorphism between the type of normal forms `Nf Γ σ` as defined in "Hereditary Substitution" and the quotient type `Tm Γ σ / _βη-≡_`. The isomorphism map and its inverse are simply given by injecting a normal form into a term and them embedding this term in the quotient and by normalization of the representative of a quotient term, respectively. Normalization of quotient terms is well-defined, as proven by Altenkirch and Keller with their proof `soundness`, and the facts that normalization serves as both a right and left inverse are given by `completeness` and `stability` of normalization[15], respectively. With these results in hand, the isomorphism is defined as follows[16]:

```
quotientIso : Iso (Nf Γ σ) (Tm Γ σ / _βη-≡_)
Iso.fun quotientIso nt = [ ⌈ nt ⌉ ]
Iso.inv quotientIso t = rec isSetNf nf (λ t u r → ptoc (soundness r)) t
Iso.rightInv quotientIso t =
  elim {B = λ u → [ ⌈ Iso.inv quotientIso u ⌉ ] ≡ u}
    (λ u → isOfHLevelPath 2 squash/ [ ⌈ Iso.inv quotientIso u ⌉ ] u)
    (λ u → eq/ _ _ (completeness u))
    rightInvLem
    t
Iso.leftInv quotientIso nt = ptoc (stability nt)
```

As we just described, most of the actual "content" of the isomorphism proof consists of results already demonstrated about normalization and $\beta\eta$-equivalence in the paper "Hereditary Substitution"; however, in the cases of defining the inverse map and proving that it is a right inverse, we need to eliminate an element of our set quotient type, leading to several non-trivial proof obligations ensuring that these operations are well-defined on the quotient type. In both cases, we have to show that the type we're eliminating into is also an h-set and that our operation on the underlying terms respects $\beta\eta$-equivalence. In

[15] also both proved in "Hereditary Substitution"

[16] Note that we make use of Agda's copattern syntax for defining the different fields of the isomorphism record type

the case of defining the inverse map, this means that we must show that the type of normal forms Nf Γ σ is an h-set (isSetNf), and the second condition, as we mentioned, is proved by soundness. In the case of proving the right inverse portion of the isomorphism, we have to show that the identity type Iso.fun quotientIso (Iso.inv quotientIso t) ≡ t is an h-set for all t of type Tm Γ σ / _βη-≡_ and that the proofs we construct of this fact for $\beta\eta$-equivalent t and u are themselves equal (this last fact is given by rightInvLem). We'll now give a brief sketch of each of these proofs.

5.1 Normal forms are an h-set

In order to be able to eliminate an element of our quotient set to begin with, we need to show that the type we're eliminating into is also a set. In the case of defining the inverse map, we are taking elements of the quotient type to normal terms, so we have to show that the type of normal terms is a set. A common and fairly straightforward way to show that an inductively defined type is a set is by using Hedberg's theorem, which states that types with decidable equality are h-sets [Uni13, Chapter 7.2]. So all we have to do is define a decision procedure that decides equality between normal terms.

In most cases, this is fairly straightforward; however, there are a few cases that require a bit of cleverness. In general, we proceed by recursively invoking the decision procedure on smaller subterms and using these intermediate results to construct our final proof for the whole term. A common situation is that we're trying to decide if two terms t and u are equal or not, where both of these terms consists of two subterms: t = (t_1, t_2) and u = (u_1, u_2). Let's say that we recursively call the decision procedure to decide equality between t_1 and u_1, and we get a proof that ¬ t_1 ≡ u_1. We now just need to show that ¬ (t_1 , t_2) ≡ (u_1 , u_2). To do this, we suppose for contradiction that we have a proof of this equality, then project out the first component using congruence, which contradicts the fact that ¬ t_1 ≡ u_1, or in code:

```
¬q : ¬ (t₁ ≡ u₁)

π₁ : A × B → A
π₁ (a , _) = a

contr : ¬ (t₁ , t₂) ≡ (u₁ , u₂)
contr = λ p → ¬q (cong π₁ p)
```

This works in most cases, however, a problem arises when trying to project out of neutral terms (denoted Ne Γ σ). Looking at the definition of neutral terms,

```
data Ne : Con → Ty → Set where
  _,_ : ∀ {Γ σ τ} → Var Γ σ → Sp Γ σ τ → Ne Γ τ
```

We see that the constructor _,_ is indexed by an extra variable σ that doesn't appear in the actual type of the resulting neutral term. This means that there's no general way to write a function that projects out the inner variable or spine from a neutral term (writing the type signature for such a function, we get about as far as: π₁ : Ne Γ τ → Sp Γ ?? τ).

But we don't necessarily need a projection function that works in general. Suppose that we have two neutral terms of type Ne Γ τ, and we have a proof that ¬ sp₁ ≡ sp₂, where these are the spine components of the two neutral terms, and they have type Sp Γ σ τ. Then we know what σ is *in this specific case*, so we can write a locally-scoped projection function that only projects out of neutral terms whose spines have the right type:

```
π₂ : ∀ {Δ τ} → Ne Δ τ → Maybe (Sp Δ σ τ)
π₂ {Δ} {τ} (_,_ {σ = σ′} _ s) with discreteTy σ σ′
... | no ¬p = nothing
... | yes p = just (subst (λ z → Sp Δ z τ) (sym p) s)
```

Note that σ is bound outside of the definition of π₂, as opposed to Δ and τ, which are arbitrary. Also note that discreteTy is a decision procedure for equality on Tys (and hence Ty is also an h-set). We've essentially defined a partial function here, but we know that we'll only use it on neutral terms whose spines have type Sp Γ σ τ, so it will always return a spine in these cases. Having defined this specialized projection function, the proof of contradiction follows in roughly the manner we described earlier, modulo some reasoning about the substitution used in the projection function, and along with the help of another lemma:

```
discreteTyRefl′ : ∀ σ → discreteTy σ σ ≡ yes refl
```

that allows our projection function to compute properly.

5.2 Proofs of right inversion are h-sets and respect $\beta\eta$-equivalence

In the case of proving the right inverse property for our isomorphism, we are again eliminating from an element of our quotient set, but this time, we're eliminating into an equality (concretely, we want to send a term [t] to a proof that [⌈ nf t ⌉] ≡ [t]). But as before, we need to show that the path space we eliminate into is an h-set and that this map (given by completeness) is well-defined.

Showing that the identity type we're eliminating into is an h-set is simple, because this type is formed from elements of our quotient set, and h-level is closed under forming identity types[17]. Recalling our proof of this fact:

```
λ u → isOfHLevelPath 2 squash/ [ ⌈ Iso.inv quotientIso u ⌉ ] u
```

Here we see the Cubical library's machinery for writing h-level generic code, so to speak, as demonstrated in the type signature of isOfHLevelPath:

[17]By inductive definition of h-level, plus the fact that h-levels are closed upwards [HoTT Book 3.1/7.1]

```
isOfHLevelPath : (n : HLevel)
  → isOfHLevel n A
  → (x y : A) → isOfHLevel n (x ≡ y)
```

In our case, isOfHLevel 2 computes to isSet, which is what we want.

The second thing we have to show is that our construction of the proof of right inversion is sound with respect to $\beta\eta$-equivalence. Conceptually, this is pretty straightforward, as the equalities we're constructing should be equal due to the set-truncation of our quotients, but the difficulty arises in the fact that we're faced with two equalities between distinct pairs of elements. That is, instead of a simple path between equalities (which we could collapse with squash/), we have a dependent path, as shown by the type signature of the relevant proof:

```
rightInvLem : ∀ {Γ σ} (a b : Tm Γ σ) (r : a βη-≡ b)
  → PathP (λ i → [ ⌈ ptoc (soundness r) i ⌉ ] ≡ eq/ a b r i)
    (eq/ ⌈ nf a ⌉ a (completeness a))
    (eq/ ⌈ nf b ⌉ b (completeness b))
```

Note that the first constructor to PathP gives a path between the types of the two endpoints of the dependent path. This is a path between paths, so using the intuition underlying Cubical Type Theory that paths are just faces/edges of higher dimensional cubes, it may help to draw a picture illustrating this path:

$$\begin{tikzcd}
{[\ \lceil \mathsf{nf}\ b \rceil\ ]} \arrow[rr, "\text{eq/ ⌈ nf b ⌉ b (completeness b)}"] & {} & {[\ b\ ]} \\
{[\ \lceil \mathsf{nf}\ a \rceil\ ]} \arrow[u, "\text{λ i → [⌈ ptoc (soundness r) i ⌉]}"] \arrow[rr, "\text{eq/ ⌈ nf a ⌉ a (completeness a)}"'] & {} \arrow[u, Rightarrow, "\text{rightInvLem}"] & {[\ a\ ]} \arrow[u, "\text{eq/ a b r}"']
\end{tikzcd}$$

We can interpret this diagram as saying that we need to provide the double arrow going from the bottom to the top edge, and the sides of the box will give us a way to turn the type of the bottom path into the type of the top path[18] Fortunately, there is a function compPathL→PathP in the Cubical library that will give us this path, so long as we are able show that if we invert the direction of the left edge, then the two resulting paths from the top-left to the top-right corners are equal. In pictures:

$$\begin{tikzcd}
C \arrow[rrr, "s"] & & & D & & C \arrow[r, "s"] & D \\
C \arrow[u] \arrow[r, "p^{-1}"] & A \arrow[r, "r"] & B \arrow[r, "q"] & D \arrow[u] & \Rightarrow & A \arrow[u, "p"] \arrow[r, "r"] & B \arrow[u, "q"]
\end{tikzcd}$$

[18]Alternatively, if we ignore the double arrow, we can just think of the dependent path as simply being the entire box itself. This corresponds with the intuition that a path between two n-cubes is an $n + 1$-cube.

However, this first path is no longer a dependent path, meaning that we can use set truncation on our quotient set to prove this condition, as follows:

```
rightInvLem a b r =
  let
    bot = eq/ ⌈ nf a ⌉ a (completeness a)
    top = eq/ ⌈ nf b ⌉ b (completeness b)
    right = eq/ a b r
    left = (λ i → [ ⌈ ptoc (soundness r) i ⌉ ])
  in
    compPathL→PathP (squash/ _ _ (sym left • bot • right) top)
```

5.3 Applying univalence

Now that we've shown that normal forms and terms modulo $\beta\eta$-equivalence are isomorphic, we can apply univalence to arrive at the result we're after

```
Nf≡Tm/βη : ∀ {Γ σ} → Nf Γ σ ≡ (Tm Γ σ / _βη-≡_)
Nf≡Tm/βη = ua (isoToEquiv quotientIso)
```

And in a manner similar to that outlined in "Three Equivalent Ordinal Notation Systems", we can easily transport weakening and hereditary substitution defined on normal forms to our quotiented term type, as we see in Figure 3 (where we now denote Tm Γ σ / _βη-≡_ as QTm Γ σ for brevity)

```
quotWk : ∀ {Γ σ τ} → (x : Var Γ σ) → QTm (Γ - x) τ → QTm Γ τ
quotWk {Γ} {σ} {τ} =
  transport
    (λ i → (x : Var Γ σ)
      → Nf≡Tm/βη {Γ - x} {τ} i
      → Nf≡Tm/βη {Γ} {τ} i)
    (wkNf {σ} {Γ} {τ})

quotSubst : ∀ {Γ σ τ} → QTm Γ σ → (x : Var Γ τ) → QTm (Γ - x) τ
  → QTm (Γ - x) σ
quotSubst {Γ} {σ} {τ} =
  transport
    (λ i → Nf≡Tm/βη {Γ} {σ} i
      → (x : Var Γ τ)
      → Nf≡Tm/βη {Γ - x} {τ} i
      → Nf≡Tm/βη {Γ - x} {σ} i)
    (_[_:=_] {τ} {Γ} {σ})
```

Figure 3: Weakening and substitution, defined on quotiented terms

This method, as we can see, is much cleaner than defining weakening and substitution on quotiented terms directly, and it is far more scalable. However, we do lose the benefit of these definitions having any easy to reason about computational properties, as the computational content of the underlying operations on normal forms is stuck behind a wall of transports and other plumbing. We can, however, easily recover properties of these operations on quotients that follow from their well-definedness (which we would normally have to prove by hand). For example, weakening respects $\beta\eta$-equivalence:

```
wkResp : (x : Var Γ σ) (t u : Tm (Γ - x) τ)
  → t βη-≡ u → quotWk x [ t ] ≡ quotWk x [ u ]
wkResp x t u p i = quotWk x (eq/ t u p i)
```

6 Adapting proofs to Cubical Agda

In the previous sections, we focused mostly on the new things we had to prove in order to work with quotient sets of terms, often relying on results proved in the original "Hereditary Substitution" paper as key lemmas, but in doing so, we actually glossed over the fact that some of the results proven in standard Agda do not necessarily work out of the box when using Cubical Agda. As such, in order to prove the equivalence result in the previous section, it was necessary to adapt the entirety of the source code from "Hereditary Substitution" so that it compiled under the `--cubical` option, which in a few cases involved some fairly interesting techniques.

Generally speaking, when adapting proofs from previous Agda developments to work with Cubical Agda, it's not actually necessary to switch from normal propositional equality to cubical/path equality in most cases. As we saw earlier, any proof of propositional equality can be transported to a cubical path with ptoc. However, even if using this approach, we may need to adapt some proofs to be compatible with the `--without-K` option, which is enabled by default in Cubical Agda, since, as we discussed earlier, axiom K is incompatible with HoTT. It is also possible (although more labor intensive) to adapt many proofs using propositional equality to proofs that use path equality directly.

We adapted the entirety of the "Hereditary Substitution" source code to compile without K in order to transport these results to the cubical setting using ptoc. We also translated several lemmas via more direct cubical reasoning as an experiment. Below, we'll briefly discuss some of the more interesting points from both of these endeavors.

6.1 Translating proofs to avoid K

The general problem that arises when adapting Agda code to compile without K is that there are some cases where pattern matching fails without K, when it had previously succeed with K. This does not necessarily mean that axiom K is necessary to define/prove the thing in question; rather, that the unification algorithm used in checking the original pattern match needed to assume K in order to perform certain steps of the unification.

The typical situation in which this issue manifests itself is when we are trying to pattern match on a variable whose type contains a variable that appears more than once and also has a constructor where the indices corresponding to these multiple occurrences are the same. For example, if we have a variable p with type y ≡ y, then we see that the variable y appears twice in p's type, and recalling our definition of refl:

```
refl : ∀ {x : A} → x ≡ x
```

We see that the two y's correspond to the same index variable in the definition of refl. The problem is that trying to pattern match on p will lead to a unification problem of the form y = y, but the unification algorithm with K disabled is not able to discard such an equation, leading it to fail[19].

In adapting the code from "Hereditary Substitution" there were only two instances in which we ran into such a problem. The solution was basically the same in both cases, so we'll just detail the first instance. The problem occurred when defining the lemma wkvInj (which states that variable weakening is injective). This lemma essentially follows by a series of pattern matches on different variables, but one case requires pattern matching on a variable j of type Var ((Γ - k) , τ) τ. The problem was that in trying to match this variable with the constructor vz (the zero De Bruijn index), we get a unification problem of the form τ = τ, for basically the same reason we described above.

Normally, the solution to a problem like this would just be to generalize our variable j to have some type like Var ((Γ - k) , τ) ρ, but this is not possible here, because we're also constrained by the equality vz ≡ wkv (vs k) j, which forces j to have the same type as the constructor vz, defined as:

```
vz : ∀ {Γ σ} → Var (Γ , σ) σ
```

But if the definition of vz were itself generalized, *then* we could generalize j, as we'd like to. To this end, we define a type of *generalized variables* as follows:

```
data GenVar : Con → Ty → Set where
  gvz : ∀ {Γ σ τ} → (σ ≡ τ) → GenVar (Γ , τ) σ
  gvs : ∀ {τ Γ σ} → GenVar Γ σ → GenVar (Γ , τ) σ
```

We can see that gvz, as compared to vz, allows the type of the variable to differ syntactically from the type of the last variable in the context, but we now need to provide a proof that these two types are indeed the same. It's fairly straightforward to show that Vars and GVars are isomorphic and to define variable weakening on GVars. All of this then lets us easily prove the relevant case for wkvInj on GVars, where we don't have issues with K. Finally, we can bring this result back over to normal Vars, using the isomorphism we proved plus a few commutation lemmas.

[19]For more details see [Pattern Matching Without K]

6.2 Manually translating proofs to use path equality

In this section, we detail some of the effort involved in translating proofs to work with cubical path equality directly (as opposed to using ptoc). Our efforts focused mainly on several of the functions and proofs defined in the file lemmas1.agda from [KA10a].

When making such a translation, the first major difference is that pattern matching on evidence of equality is no longer possible in the same way that it is with propositional equality, due to the fact that path equality is not inductively defined. We can use the ctop trick to recover pattern matching in situations where we only need pattern matching to unify the variables on either side of the equality we're matching on, but in other cases, this trick does not suffice. For instance, the ctop trick will not help us if the proof of equality itself appears in the goal.

For example, many of the lemmas used in "Hereditary Substitution" concern themselves with several coercion operators that take variables, terms, etc. in one context to ones in a propositionally equal context. For example, we define the coercion of a variable along a path between contexts as follows:

```
!_>₀_ : Γ ≡ Δ → Var Γ σ → Var Δ σ
!_>₀_ {σ = σ} p x = transport (λ i → Var (p i) σ) x
```

Then, we want to prove the lemma !vz, which states that the variable vz is still vz after we change the context:

```
!vz : (p : Γ ≡ Δ) → (! (λ i → (p i) , σ) >₀ vz) ≡ vz
```

This is a situation where the ctop trick doesn't really help us. Pattern matching on ctop p will unify Γ and Δ, but it doesn't unify p with refl, which is really what we would need to prove this. We can, however, use the induction principal for path equality, J:

```
!vz {σ = σ} p = J (λ _ q → ! (λ i → (q i) , σ) >₀ vz ≡ vz) (!refl₀ vz) p
```

Note that the first argument to J is the motive, the second argument is a proof of the motive in the special case where the path (q, here) is replaced with refl, and the third argument is the path we're inducting on. Note that if we are working with propositional equality, the second argument would just need to be refl, because ! refl $>_0$ vz would compute to vz definitionally. With path equality, however, this is not true definitionally, but we can prove it with the lemma !refl$_0$:

```
!refl₀ : (x : Var Γ σ) → (! refl >₀ x) ≡ x
!refl₀ x = transportRefl x
```

Well really, there's not much to prove here, as we just use transportRefl, but this is a good example of a situation where transport's lack of computation on refl affects the proof that we need to give.

7 Conclusion

Looking back at our various efforts to define a type of STLC terms modulo $\beta\eta$-equivalence along with weakening and substitution on these terms, we've learned a few things. For one, we've developed several examples and techniques for reasoning about Higher Inductive Types and set quotients in particular, and we've also developed some useful techniques for adapting code in standard Agda to work with Cubical Agda. Perhaps more saliently, though, we've gained a better understanding of some of the trade-offs and considerations involved in using Higher Inductive Types to model terms modulo $\beta\eta$-equivalence. In particular, it should be noted that while using HITs in this way provides us with a fairly elegant way of expressing the quotient structure we're interested in and ensures that any operations we define on said terms are sound with respect to our relation, working with elements of said HITs can be rather involved. Indeed, the more "direct approach" we first described, in which operations on terms are defined directly via case work (or induction principles) on HITs, required us to tackle a large number of soundness lemmas in order to even define a notion as basic as substitution on quotiented terms. Coming from this perspective, the main strength of working with quotiented terms across the isomorphism with normal forms is that this approach essentially contains and minimizes the amount of work we have to do directly with HITs. That is, we're able to define more computational notions like weakening and substitution on normal forms, where soundness is enforced by the structure of normal forms themselves, and then transport these to our quotiented term type, where we can take advantage of the more general properties of set quotients.

Indeed, an interesting area of further research in this topic might be how we can exploit the opposite direction of the isomorphism to prove things about normal forms using quotiented terms. Or perhaps more broadly, it could be interesting to investigate how univalence and other features of HoTT can be used to formalize equivalences between different type systems and how these equivalences can then be used to transport results between systems in beneficial ways. In any case, the intersection between HoTT (in particular cubical type theory) and programming language research is an area with much future potential.

Acknowledgments

First and foremost, I'd like to thank my thesis adviser Professor Stuart A. Kurtz for the many helpful insights, explanations, suggestions, words of encouragement, and funny anecdotes with which he has provided me throughout this past year. I'd also like to thank Thorsten Altenkirch and Ambrus Kaposi for writing the paper that inspired this one, and I'd like to thank Chantal Keller (and Thorsten Altenkirch, again) for developing the code that I used as a starting point for the work in this paper.

References

[agd] The agda user manual. `https://agda.readthedocs.io/en/v2.6.1.3/`.

[AK16] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices*, 51(1):18–29, 2016.

[CA18] Jesper Cockx and Andreas Abel. Elaborating dependent (co) pattern matching. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018.

[CDP14] Jesper Cockx, Dominique Devriese, and Frank Piessens. Pattern matching without k. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*, pages 257–268, 2014.

[cub] An experimental library for cubical agda. `https://github.com/agda/cubical`.

[Dyb95] Peter Dybjer. Internal type theory. In *International Workshop on Types for Proofs and Programs*, pages 120–134. Springer, 1995.

[FXG20] Fredrik Nordvall Forsberg, Chuangjie Xu, and Neil Ghani. Three equivalent ordinal notation systems in cubical agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 172–185, 2020.

[Goo] Noah Goodman. HIT me with your best STLC, source code. `https://github.com/buonuomo/bs-thesis`.

[KA10a] Chantal Keller and Thorsten Altenkirch. The decidability of the $\beta\eta$-equivalence using hereditary substitutions. `https://prosecco.gforge.inria.fr/personal/ckeller/Recherche/hsubst.html`, 2010.

[KA10b] Chantal Keller and Thorsten Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, pages 3–10, 2010.

[MLS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 9. Bibliopolis Naples, 1984.

[Nor07] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Citeseer, 2007.

[Str93] Thomas Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, 1993.

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[VMA21] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: a dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31, 2021.