The Grammar of Proof [Preliminary Draft]

Warrick Macmillan

February 14, 2021

1 Introduction

The central concern of this thesis is the syntax of mathematics, programming languages, and their respective mutual influence, as conceived and practiced by mathematicians and computer scientists. From one vantage point, the role of syntax in mathematics may be regarded as a 2nd order concern, a topic for discussion during a Fika, an artifact of ad hoc development by the working mathematician whose real goals are producing genuine mathematical knowledge. For the programmers and computer scientists, syntax may be regarding as a matter of taste, with friendly debates recurring regarding the use of semicolons, brackets, and white space. Yet, when viewed through the lens of the propositions-as-types paradigm, these discussions intersect in new and interesting ways. When one introduces a third paradigm through which to analyze the use of syntax in mathematics and programming, namely Linguistics, I propose what some may regard as superficial detail, indeed becomes a central paradigm, with many interesting and important questions.

To get a feel for this syntactic paradigm, let us look at a basic mathematical example: that of a group homomorphism, as expressed in a variety of sources.

Definition 1 *In mathematics, given two groups,* $(G, *)$ *and* $(H, \cdot)$*, a group homomorphism from* $(G, *)$ *to* $(H, \cdot)$ *is a function* $h : G \to H$ *such that for all* u *and* v *in* G *it holds that*

$$h(u * v) = h(u) \cdot h(v)$$

Definition 2 *Let* $G = (G, \cdot)$ *and* $G' = (G', *)$ *be groups, and let* $\phi : G \to G'$ *be a map between them. We call* ϕ *a* ***homomorphism*** *if for every pair of elements* $g, h \in G$*, we have*

$$\phi(g * h) = \phi(g) \cdot \phi(h)$$

Definition 3 *Let* G*,* H*, be groups. A map* $\phi : G \to H$ *is called a* group homomorphism *if*

$$\phi(xy) = \phi(x)\phi(y) \text{ for all } x, y \in G$$

(Note that xy *on the left is formed using the group operation in* G*, whilst the product* $\phi(x)\phi(y)$ *is formed using the group operation* H*.)*

Definition 4 *Classically, a group is a monoid in which every element has an inverse (necessarily unique).*

We inquire the reader to pay attention to nuance and difference in presentation that is normally ignored or taken for granted by the fluent mathematician.

If one want to distill the meaning of each of these presentations, there is a significant amount of subliminal interpretation happening very much analagous to our innate lingusitic ussage. The inverse and identity are discarded, even though they are necessary data when

defning a group. The order of presentation of information is incostent, as well as the choice to use symbolic or natural language information. In (3), the group operation is used implicitly, and its clarification a side remark.

Details aside, these all mean the same thing–don't they? This thesis seeks to provide an abstract framework to determine whether two lingusitically nuanced presenations mean the same thing via their syntactic transformations.

These syntactic transformations come in two flavors : parsing and linearization, and are natively handled by a Logical Framework (LF) for specifying grammars : Grammatical Framework (GF).

We now show yet another definition of a group homomorphism formalized in the Agda programming language:

[TODO: replace monoidhom with grouphom]

```
monoidHom : {ℓ : Level}
            → ((monoid' a _ _ _ _ _ _) (monoid' a' _ _ _ _ _ _) : Monoid' {ℓ} )
            → (a → a') → Type ℓ
monoidHom
  (monoid' A ε _•_ left-unit right-unit assoc carrier-set)
  (monoid' A₁ ε₁ _•₁_ left-unit₁ right-unit₁ assoc₁ carrier-set₁)
  f
  = (m1 m2 : A) → f (m1 • m2) ≡ (f m1) •₁ (f m2)
```

While the first three definitions above are should be linguistically comprehensible to a non-mathematician, this last definition is most certainly not. While may carry degree of comprehension to a programmer or mathematician not exposed to Agda, it is certainly comprehensible to a computer : that is, it typechecks on a computer where Cubical Agda is installed. While GF is designed for multilingual syntactic transformations and is targeted for natural language translation, it's underlying theory is largely based on ideas from the compiler communities. A cousin of the BNF Converter (BNFC), GF is fully capable of parsing progamming languages like Agda! And while the above definition is just another concrete syntactic presentation of a group homomorphism, it is distinct from the natural language presentations above in that the colors indicate it has indeed type checked.

While this example may not exemplify the power of Agda's type checker, it is of considerable interest to many. The typechecker has merely assured us that monoidHom, is a well-formed type. The type-checker is much more useful than is immediately evident: it delegates the work of verifying that a proof is correct, that is, the work of judging whether a term has a type, to the computer. While it's of practical concern is immediate to any exploited grad student grading papers late on a Sunday night, its theoretical concern has led to many recent developments in modern mathematics. Thomas Hales solution to the Kepler Conjecture was seen as unverifiable by those reviewing it. This led to Hales outsourcing the verification to Interactive Theorem provers HOL Light and Isabelle, during which led to many minor corrections in the original proof which were never spotted due to human oversight.

Fields Medalist Vladimir Voevodsky, had the experience of being told one day his proof of the Milnor conjecture was fatally flawed. Although the leak in the proof was patched, this experience of temporarily believing much of his life's work invalid led him to investigate proof assintants as a tool for future thought. Indeed, this proof verification error was a key event that led to the Univalent Foundations Project [1].

While Agda and other programming languages are capable of encoding definitions, theorems, and proofs, they have so far seen little adoption, and in some cases treated with suspicion and scorn by many mathematicians. This isn't entirely unfounded : it's a lot of work to learn how to use Agda or Coq, software updates may cause proofs to break, and the inevitable errors we humans are instilled in these Theorem Provers. And that's not to mention

that Martin-Löf Type Theory, the constructive foundational project which underlies these proof assistants, is rejected by those who dogmatically accept the law of the excluded middle and ZFC as the word of God.

It should be noted, the constructivist rejects neither the law of the excluded middle nor ZFC. She merely observes them, and admits their handiness in certain cituations. Excluded middle is indeed a helpful tool, as many mathematicians may attest. The contention is that it should be avoided whenever possible - proofs which don't rely on it, or it's corallary of proof by contradction, are much more ameanable to formalization in systems with decideable type checking. And ZFC, while serving the mathematicians of the early 20th century, is lacking when it comes to the higher dimensional structure of n-categories and infinity groupoids.

What these theorem provers give the mathematician is confidence that her work is correct, and even more importantly, that the work which she takes for granted and references in her work is also correct. The task before us is then one of religious conversion. And one doesn't undertake a conversion by simply by preaching. Foundational details aside, this thesis is meant to provide a blueprint for the syntactic reformation that must take place.

It doesn't ask the mathematician to relinquish the beautiful language she has come to love in expressing her ideas. Rather, it asks her to make a compromise for the time being, and use a Controlled Natural Language (CNL) to develop her work. In exchange she'll get the confidence that Agda provides. Not only that, she'll be able to search through a library, to see who else has possibly already postulated and proved her conjecture. A version of this grandiose vision is explored in The Formal Abstracts Project.

2 Preliminaries

[Todo : Give overview of tools and theory in the making of this thesis]

2.1 Grammatical Framework

...

2.2 Martin-Löf Type Theory

...

2.3 Agda

...

2.4 Natural Language and Mathematics

...

3 HoTT Proofs

3.1 Why HoTT for natural language?

We note that all natural language definitions, theorems, and proofs are copied here verbatim from the HoTT book. This decision is admittedly arbitrary, but does have some benefits. We list some here :

- As the HoTT book was a collaborative effort, it mixes the language of many individuals and editors, and can be seen as more "linguistically neutral"
- By its very nature HoTT is interdiscplinary, conceived and constructed by mathematicians, logicians, and computer scientists. It therefore is meant to interface with all these discplines, and much of the book was indeed formalized before it was written
- It has become canonical reference in the field, and therefore benefits from wide familiarity
- It is open source, with publically available Latex files free for modification and distribution

The genisis of higher type theory is a somewhat elementary observation : that the identity type, parameterized by an arbitrary type A and indexed by elements of A, can actually be built iteratively from previous identities. That is, A may actually already be an identity defined over another type A', namely $A :\equiv x =_{A'} y$ where $x, y : A'$. The key idea is that this iterating identities admits a homotpical interpretation :

- Types are topological spaces
- Terms are points in these space
- Equality types $x =_A y$ are paths in A with endpoints x and y in A
- Iterated equality types are paths between paths, or continuous path deformations in some higher path space. This is, intuitively, what mathematicians call a homotopy.

To be explicit, given a type A, we can form the homotopy $p =_{x=_A y} q$ with endpoints p and q inhabiting the path space $x =_A y$.

Let's start out by examining the inductive definition of the identity type. We present this definition as it appears in section 1.12 of the HoTT book.

Definition 5 *The formation rule says that given a type $A : \mathcal{U}$ and two elements $a, b : A$, we can form the type $(a =_A b) : \mathcal{U}$ in the same universe. The basic way to construct an element of $a = b$ is to know that a and b are the same. Thus, the introduction rule is a dependent function*

$$\mathsf{refl} : \prod_{a:A} (a =_A a)$$

called ***reflexivity****, which says that every element of A is equal to itself (in a specified way). We regard* refl_a *as being the constant path at the point a.*

We recapitulate this definition in Agda, and treat :

```
data _≡'_ {A : Set} : (a b : A) → Set where
  r : (a : A) → a ≡' a
```

3.2 An introduction to equality

There is already some tension brewing : most mathematicains have an intuition for equality, that of an identitfication between two pieces of information which intuitively must be the same thing, i.e. $2 + 2 = 4$. They might ask, what does it mean to "construct an element of $a = b$"? For the mathematician use to thinking in terms of sets $\{a = b \mid a, b \in \mathbb{N}\}$ isn't a well-defined notion. Due to its use of the axiom of extensionality, the set theoretic notion of equality is, no suprise, extensional. This means that sets are identified when they have the same elements, and equality is therefore external to the notion of set. To inhabit a type means to provide

evidence for that inhabitation. The reflexivity constructor is therefore a means of providing evidence of an equality. This evidence approach is disctinctly constructive, and a big reason why classical and constructive mathematics, especially when treated in an intuitionistic type theory suitable for a programming language implementation, are such different beasts.

In Martin-Löf Type Theory, there are two fundamental notions of equality, propositional and definitional. While propositional equality is inductively defined (as above) as a type which may have possibly more than one inhabitant, definitional equality, denoted $- \equiv -$ and perhaps more aptly named computational equality, is familiarly what most people think of as equality. Namely, two terms which compute to the same canonical form are computationally equal. In intensional type theory, propositional equality is a weaker notion than computational equality : all propositionally equal terms are computationally equal. However, computational equality does not imply propistional equality - if it does, then one enters into the space of extensional type theory.

Prior to the homotopical interpretation of identity types, debates about extensional and intensional type theories centred around two features or bugs : extensional type theory sacrificed decideable type checking, while intensional type theories required extra beauracracy when dealing with equality in proofs. One approach in intensional type theories treated types as setoids, therefore leading to so-called "Setoid Hell". These debates reflected Martin-Löf's flip-flopping on the issue. His seminal 1979 Constructive Mathematics and Computer Programming, which took an extensional view, was soon betrayed by lectures he gave soon thereafter in Padova in 1980. Martin-Löf was a born again intensional type theorist. These Padova lectures were later published in the "Bibliopolis Book", and went on to inspire the European (and Gothenburg in particular) approach to implementing proof assitants, whereas the extensionalists were primarily eminating from Robert Constable's group at Cornell.

This tension has now been at least partially resolved, or at the very least clarified, by an insight Voevodsky was apparently most proud of : the introduction of h-levels. We'll delegate these details for a later section, it is mentioned here to indicate that extensional type theory was really "set theory" in disguise, in that it collapses the higher path structure of identity types. The work over the past 10 years has elucidated the intensional and extensional positions. HoTT, by allowing higher paths, is unashamedly intentional, and admits a collapse into the extensional universe if so desired. We now the examine the structure induced by this propositional equality.

3.3 All about Identity

We start with a slight reformulation of the identity type, where the element determining the equality is treated as a parameter rather than an index. This is a matter of convenience more than taste, as it delegates work for Agda's typechecker that the programmer may find a distraction. The reflexivity terms can generally have their endpoints inferred, and therefore cuts down on the beauracry which often obscures code.

```
data _≡_ {A : Set} (a : A) : A → Set where
  r : a ≡ a

infix 20 _≡_
```

It is of particular concern in this thesis, because it hightlights a fundamental difference between the lingusitic and the formal approach to proof presentation. While the mathematician can whimsically choose to include the reflexivity arguement or ignore it if she believes it can be inferred, the programmer can't afford such a laxidasical attitude. Once the type has

been defined, the arguement strcuture is fixed, all future references to the definition carefully adhere to its specification. The advantage that the programmer does gain however, that of Agda's powerful inferential abilities, allows for the insides to be seen via interaction windown.

Perhaps not of much interest up front, this is incredibly important detail which the mathematician never has to deal with explicity, but can easily make type and term translation infeasible due to the fast and loose nature of the mathematician's writing. Conversely, it may make Natural Language Generation (NLG) incredibly clunky, adhering to strict rules when created sentences out of programs.

[ToDo, give a GF example]

A prime source of beauty in constructive mathematics arises from Gentzen's recognition of a natural duality in the rules for introducing and using logical connectives. The mutually coherence between introduction and elmination rules form the basis of what has since been labeled harmony in a deductive system. This harmony isn't just an artifact of beauty, it forms the basis for cuts in proof normalization, and correspondingly, evaluation of terms in a programming langauge.

The idea is simple, each new connective, or type former, needs a means of constructing its terms from its constiutuent parts, yielding introduction rules. This however, isn't enough - we need a way of dissecting and using the terms we construct. This yields an elimantion rule which can be uniquely derived from an inductively defined type. These elimination forms yield induction principles, or a general notion of proof by induction, when given an interpration in mathematics. In the non-depedent case, this is known as a recursion principle, and corresponds to recursion known by programmers far and wide. The proof by induction over natural numbers familiar to mathematicians is just one special case of this induction principle at work–the power of induction has been recognized and brought to the fore by computer scientists.

We now elaborate the most important induction principle in HoTT, namely, the induction of an identity type.

Definition 6 (Version 1) *Moreover, one of the amazing things about homotopy type theory is that all of the basic constructions and axioms—all of the higher groupoid structure—arises automatically from the induction principle for identity types. Recall from [section 1.12] that this says that if*

- *for every* $x, y : A$ *and every* $p : x =_A y$ *we have a type* $D(x, y, p)$*, and*
- *for every* $a : A$ *we have an element* $d(a) : D(a, a, \mathsf{refl}_a)$*,*

then

- *there exists an element* $\mathsf{ind}_{=_A}(D, d, x, y, p) : D(x, y, p)$ *for* every *two elements* $x, y : A$ *and* $p : x =_A y$*, such that* $\mathsf{ind}_{=_A}(D, d, a, a, \mathsf{refl}_a) \equiv d(a)$*.*

The book then reiterates this definition, with basically no natural language, essentially in the raw logical framework devoid of anything but dependent function types.

Definition 7 (Version 2) *In other words, given dependent functions*

$$D : \prod_{(x,y:A)} (x = y) \to \mathcal{U}$$
$$d : \prod_{a:A} D(a, a, \mathsf{refl}_a)$$

there is a dependent function

$$\mathsf{ind}_{=_A}(D, d) : \prod_{(x,y:A)} \prod_{(p:x=y)} D(x, y, p)$$

such that

$$\mathsf{ind}_{=_A}(D, d, a, a, \mathsf{refl}_a) \equiv d(a) \tag{1}$$

for every $a : A$. Usually, every time we apply this induction rule we will either not care about the specific function being defined, or we will immediately give it a different name.

Again, we define this, in Agda, staying as true to the syntax as possible.

```
J : {A : Set}
  → (D : (x y : A) → (x ≡ y) → Set)
  → ((a : A) → (D a a r )) -- → (d : (a : A) → (D a a r ))
  → (x y : A)
  → (p : x ≡ y)
  ------------------------------------
  → D x y p
J D d x .x r = d x
```

It should be noted that, for instance, we can choose to leave out the d label on the third line. Indeed minimizing the amount of dependent typing and using vanilla function types when dependency is not necessary, is generally considered "best practice" Agda, because it will get desugared by the time it typechecks anyways. For the writer of the text; however, it was convenient to define d once, as there are not the same constraints on a mathematician writing in latex. It will again, serve as a nontrivial exercise to deal with when specifying the grammar, and will be dealt with later [ToDo add section]. It is also of note that we choose to include Martin-Löf's original name J, as this is more common in the computer science literature.

Once the identity type has been defined, it is natural to develop an "equality calculus", so that we can actually use it in proof's, as well as develop the higher groupoid structure of types. The first fact, that propositional equality is an equivalence relation, is well motivated by needs of practical theorem proving in Agda and the more homotopically minded mathematician. First, we show the symmetry of equality–that paths are reversible.

Lemma 1 *For every type A and every $x, y : A$ there is a function*

$$(x = y) \to (y = x)$$

*denoted $p \mapsto p^{-1}$, such that $\mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x$ for each $x : A$. We call p^{-1} the **inverse** of p.*

Proof 1 (First proof) *Assume given $A : \mathcal{U}$, and let $D : \prod_{(x,y:A)}(x = y) \to \mathcal{U}$ be the type family defined by $D(x, y, p) :\equiv (y = x)$. In other words, D is a function assigning to any $x, y : A$ and $p : x = y$ a type, namely the type $y = x$. Then we have an element*

$$d :\equiv \lambda x.\, \mathsf{refl}_x : \prod_{x:A} D(x, x, \mathsf{refl}_x).$$

Thus, the induction principle for identity types gives us an element $\mathsf{ind}_{=_A}(D, d, x, y, p) : (y = x)$ for each $p : (x = y)$. We can now define the desired function $(-)^{-1}$ to be $\lambda p.\, \mathsf{ind}_{=_A}(D, d, x, y, p)$, i.e. we set $p^{-1} :\equiv \mathsf{ind}_{=_A}(D, d, x, y, p)$. The conversion rule [missing reference] gives $\mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x$, as required.

The Agda code is certainly more brief:

```
_⁻¹ : {A : Set} {x y : A} → x ≡ y → y ≡ x
_⁻¹ {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = y ≡ x
    d : (a : A) → D a a r
    d a = r

infixr 50 _⁻¹
```

While first encountering induction principles can be scary, they are actually more mechanical than one may think. This is due to the the fact that they uniquely compliment the introduction rules of an an inductive type, and are simply a means of showing one can "map out", or derive an arbitrary type dependent on the type which has been inductively defined. The mechanical nature is what allows for Coq's induction tactic, and perhaps even more elegantly, Agda's pattern matching capabilities. It is always easier to use pattern matching for the novice Agda programmer, which almost feels like magic. Nonetheless, for completeness sake, the book uses the induction principle for much of Chapter 2. And pattern matching is unique to programming languages, its elegance isn't matched in the mathematicians' lexicon.

Here is the same proof via "natural language pattern matching" and Agda pattern matching:

Proof 2 (Second proof) *We want to construct, for each $x, y : A$ and $p : x = y$, an element $p^{-1} : y = x$. By induction, it suffices to do this in the case when y is x and p is* refl_x*. But in this case, the type $x = y$ of p and the type $y = x$ in which we are trying to construct p^{-1} are both simply $x = x$. Thus, in the "reflexivity case", we can define* refl_x^{-1} *to be simply* refl_x*. The general case then follows by the induction principle, and the conversion rule* $\mathsf{refl}_x^{-1} \equiv \mathsf{refl}_x$ *is precisely the proof in the reflexivity case that we gave.*

```
_⁻¹' : {A : Set} {x y : A} → x ≡ y → y ≡ x
_⁻¹' {A} {x} {y} r = r
```

Next is trasitivity–concatenation of paths–and we omit the natural language presentation, which is a slightly more sophisticated arguement than for symmetry.

```
_•_ : {A : Set} → {x y : A} → (p : x ≡ y) → {z : A} → (q : y ≡ z) → x ≡ z
_•_ {A} {x} {y} p {z} q = J D d x y p z q
    where
    D : (x₁ y₁ : A) → x₁ ≡ y₁ → Set
    D x y p = (z : A) → (q : y ≡ z) → x ≡ z
    d : (z₁ : A) → D z₁ z₁ r
    d = λ v z q → q

infixl 40 _•_
```

Putting on our spectacles, the reflexivity term serves as evidence of a constant path for any given point of any given type. To the category theorist, this makes up the data of an identity map. Likewise, conctanation of paths starts to look like function composition. This,

along with the identity laws and associativity as proven below, gives us the data of a category. And we have not only have a category, but the symmetry allows us to prove all paths are isomorphisms, giving us a groupoid. This isn't a coincedence, it's a very deep and fascinating articulation of power of the machinery we've so far built. The fact the path space over a type naturally must satisfies coherence laws in an even higher path space gives leads to this notion of types as higher groupoids.

As regards the natural language–at this point, the bookkeeping starts to get hairy. Paths between paths, and paths between paths between paths, these ideas start to lose geometric inutiotion. And the mathematician often fails to express, when writing $p = q$, that she is already reasoning in a path space. While clever, our brains aren't wired to do too much bookkeeping. Fortunately Agda does this for us, and we can use implicit arguements to avoid our code getting too messy. [ToDo, add example]

We now proceed to show that we have a groupoid, where the objects are points, the morphisms are paths. The isomorphisms arise from the path reversal. Many of the proofs beyond this point are either routinely made via the induction principle, or even more routinely by just pattern matching on equality paths, we omit the details which can be found in the HoTT book, but it is expected that the GF parser will soon cover such examples.

```
iₗ : {A : Set} {x y : A} (p : x ≡ y) → p ≡ r • p
iₗ {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p ≡ r • p
    d : (a : A) → D a a r
    d a = r

iᵣ : {A : Set} {x y : A} (p : x ≡ y) → p ≡ p • r
iᵣ {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p ≡ p • r
    d : (a : A) → D a a r
    d a = r

leftInverse : {A : Set} {x y : A} (p : x ≡ y) → p ⁻¹ • p ≡ r
leftInverse {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p ⁻¹ • p ≡ r
    d : (x : A) → D x x r
    d x = r

rightInverse : {A : Set} {x y : A} (p : x ≡ y) → p • p ⁻¹ ≡ r
rightInverse {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p • p ⁻¹ ≡ r
    d : (a : A) → D a a r
    d a = r

doubleInv : {A : Set} {x y : A} (p : x ≡ y) → p ⁻¹ ⁻¹ ≡ p
doubleInv {A} {x} {y} p = J D d x y p
  where
```

```
    D : (x y : A) → x ≡ y → Set
    D x y p = p ⁻¹ ⁻¹ ≡ p
    d : (a : A) → D a a r
    d a = r

associativity :{A : Set} {x y z w : A} (p : x ≡ y) (q : y ≡ z) (r' : z ≡ w ) → p • (q • r') ≡ p • q • r'
associativity {A} {x} {y} {z} {w} p q r' = J D₁ d₁ x y p z w q r'
  where
    D₁ : (x y : A) → x ≡ y → Set
    D₁ x y p = (z w : A) (q : y ≡ z) (r' : z ≡ w ) → p • (q • r') ≡ p • q • r'
    -- d₁ : (x : A) → D₁ x x r
    -- d₁ x z w q r' = r -- why can it infer this
    D₂ : (x z : A) → x ≡ z → Set
    D₂ x z q = (w : A) (r' : z ≡ w ) → r • (q • r') ≡ r • q • r'
    D₃ : (x w : A) → x ≡ w → Set
    D₃ x w r' = r • (r • r') ≡ r • r • r'
    d₃ : (x : A) → D₃ x x r
    d₃ x = r
    d₂ : (x : A) → D₂ x x r
    d₂ x w r' = J D₃ d₃ x w r'
    d₁ : (x : A) → D₁ x x r
    d₁ x z w q r' = J D₂ d₂ x z q w r'
```

When one starts to look at structure above the groupoid level, i.e., the paths between paths between paths level, some interesting and nonintuitive results emerge. If one defines a path space that is seemingly trivial, namely, taking the same starting and end points, the higherdimensional structure yields non-trivial structure. We now arrive at the first "interesting" result in this book, the Eckmann-Hilton Arguement. It says that composition on the loop space of a loop space, the second loop space, is commutitive.

Definition 8 *Thus, given a type A with a point $a : A$, we define its **loop space** $\Omega(A, a)$ to be the type $a =_A a$. We may sometimes write simply ΩA if the point a is understood from context.*

Definition 9 *It can also be useful to consider the loop space* of *the loop space of A, which is the space of 2-dimensional loops on the identity loop at a. This is written $\Omega^2(A, a)$ and represented in type theory by the type* $\mathsf{refl}_a =_{(a=_A a)} \mathsf{refl}_a$.

Theorem 1 (Eckmann–Hilton) *The composition operation on the second loop space*

$$\Omega^2(A) \times \Omega^2(A) \to \Omega^2(A)$$

is commutative: $\alpha \cdot \beta = \beta \cdot \alpha$, for any $\alpha, \beta : \Omega^2(A)$.

Proof 3 *First, observe that the composition of 1-loops $\Omega A \times \Omega A \to \Omega A$ induces an operation*

$$\star : \Omega^2(A) \times \Omega^2(A) \to \Omega^2(A)$$

as follows: consider elements $a, b, c : A$ and 1- and 2-paths,

$$\begin{aligned} p &: a = b, & r &: b = c \\ q &: a = b, & s &: b = c \\ \alpha &: p = q, & \beta &: r = s \end{aligned}$$

as depicted in the following diagram (with paths drawn as arrows).

[TODO Finish Eckmann Hilton Arguement]

[Todo, clean up code so that its more tightly correspondent to the book proof] The corresponding agda code is below :

```
-- whiskering
_•ᵣ_ : {A : Set} → {b c : A} {a : A} {p q : a ≡ b} (α : p ≡ q) (r′ : b ≡ c) → p • r′ ≡ q • r′
_•ᵣ_ {A} {b} {c} {a} {p} {q} α r′ = J D d b c r′ a α
  where
    D : (b c : A) → b ≡ c → Set
    D b c r′ = (a : A) {p q : a ≡ b} (α : p ≡ q) → p • r′ ≡ q • r′
    d : (a : A) → D a a r
    d a a′ {p} {q} α = iᵣ p ⁻¹ • α • iᵣ q

-- iᵣ == ruₚ

_•ₗ_ : {A : Set} → {a b : A} (q : a ≡ b) {c : A} {r′ s : b ≡ c} (β : r′ ≡ s) → q • r′ ≡ q • s
_•ₗ_ {A} {a} {b} q {c} {r′} {s} β = J D d a b q c β
  where
    D : (a b : A) → a ≡ b → Set
    D a b q = (c : A) {r′ s : b ≡ c} (β : r′ ≡ s) → q • r′ ≡ q • s
    d : (a : A) → D a a r
    d a a′ {r′} {s} β = iₗ r′ ⁻¹ • β • iₗ s

_⋆_ : {A : Set} → {a b c : A} {p q : a ≡ b} {r′ s : b ≡ c} (α : p ≡ q) (β : r′ ≡ s) → p • r′ ≡ q • s
_⋆_ {A} {q = q} {r′ = r′} α β = (α •ᵣ r′) • (q •ₗ β)

_⋆′_ : {A : Set} → {a b c : A} {p q : a ≡ b} {r′ s : b ≡ c} (α : p ≡ q) (β : r′ ≡ s) → p • r′ ≡ q • s
_⋆′_ {A} {p = p} {s = s} α β = (p •ₗ β) • (α •ᵣ s)

Ω : {A : Set} (a : A) → Set
Ω {A} a = a ≡ a

Ω² : {A : Set} (a : A) → Set
Ω² {A} a = _≡_ {a ≡ a} r r

lem1 : {A : Set} → (a : A) → (α β : Ω² {A} a) → (α ⋆ β) ≡ (iᵣ r ⁻¹ • α • iᵣ r) • (iₗ r ⁻¹ • β • iₗ r)
lem1 a α β = r

lem1′ : {A : Set} → (a : A) → (α β : Ω² {A} a) → (α ⋆′ β) ≡ (iₗ r ⁻¹ • β • iₗ r) • (iᵣ r ⁻¹ • α • iᵣ r)
lem1′ a α β = r

-- ap\_
apf : {A B : Set} → {x y : A} → (f : A → B) → (x ≡ y) → f x ≡ f y
apf {A} {B} {x} {y} f p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = {f : A → B} → f x ≡ f y
    d : (x : A) → D x x r
    d = λ x → r

ap : {A B : Set} → {x y : A} → (f : A → B) → (x ≡ y) → f x ≡ f y
ap f r = r

lem20 : {A : Set} → {a : A} → (α : Ω² {A} a) → (iᵣ r ⁻¹ • α • iᵣ r) ≡ α
```

```
lem20 α = iᵣ (α) ⁻¹

lem21 : {A : Set} → {a : A} → (β : Ω² {A} a) → (iₗ r ⁻¹ • β • iₗ r) ≡ β
lem21 β = iᵣ (β) ⁻¹

lem2 : {A : Set} → (a : A) → (α β : Ω² {A} a) → (iᵣ r ⁻¹ • α • iᵣ r) • (iₗ r ⁻¹ • β • iₗ r) ≡ (α • β)
lem2 {A} a α β = apf (λ - → - • (iₗ r ⁻¹ • β • iₗ r) ) (lem20 α) • apf (λ - → α • -) (lem21 β)

lem2' : {A : Set} → (a : A) → (α β : Ω² {A} a) → (iₗ r ⁻¹ • β • iₗ r) • (iᵣ r ⁻¹ • α • iᵣ r) ≡ (β • α )
lem2' {A} a α β = apf (λ - → - • (iᵣ r ⁻¹ • α • iᵣ r)) (lem21 β) • apf (λ - → β • -) (lem20 α)
-- apf (λ - → - • (iᵢ r ⁻¹ • β • iᵢ r) ) (lem20 α) • apf (λ - → α • -) (lem21 β)

⋆≡• : {A : Set} → (a : A) → (α β : Ω² {A} a) → (α ⋆ β) ≡ (α • β)
⋆≡• a α β = lem1 a α β • lem2 a α β

-- proven similairly to above
⋆'≡• : {A : Set} → (a : A) → (α β : Ω² {A} a) → (α ⋆' β) ≡ (β • α)
⋆'≡• a α β = lem1' a α β • lem2' a α β


--eckmannHilton : {A : Set} → (a : A) → (α β : Ω² {A} a) → α • β ≡ β • α
--eckmannHilton a r r = r
```

[TODO, fix without k errors]

4 GF Grammar for types

We now discuss the GF implementation, capable of parsing both natural language and Agda syntax. The parser was appropriated from the cubicaltt BNFC parser, de-cubified and then gf-ified. The languages are tightly coupled, so the translation is actually quite simple. Some main differences are:

- GF treats abstract and concrete syntax seperately. This allows GF to support many concrete syntax implementation of a given grammar
- Fixity is dealt with at the concrete syntax layer in GF. This allows for more refined control of fixity, but also results in difficulties : during linearization their can be the insertion of extra parens.
- GF supports dependent hypes and higher order abstract syntax, which makes it suitable to typecheck at the parsing stage. It would very interesting to see if this is interoperable with the current version of this work in later iterations [Todo - add github link referncing work I've done in this direction]
- GF also is enhanced by a PGF back-end, allowing an embedding of grammars into, among other languages, Haskell.

While GF is targeted towards natural language translation, there's nothing stopping it from being used as a PL tool as well, like, for instance, the front-end of a compiler. The innovation of this thesis is to combine both uses, thereby allowing translation between Controlled Natural Languages and programming languages.

Example expressions the grammar can parse are seen below, which have been verified by hand to be isomorphic to the corresponding cubicaltt BNFC trees:

```
data bool : Set where true | false
data nat : Set where zero | suc ( n : nat )
caseBool ( x : Set ) ( y z : x ) : bool -> Set = split false -> y || true -> z
indBool ( x : bool -> Set ) ( y : x false ) ( z : x true ) : ( b : bool ) -
> x b = split false -> y || true  -> z
funExt  ( a : Set )  ( b : a -> Set )  ( f g :  ( x : a )  -> b x )  ( p :  ( x : a )  -
> ( b x )   ( f x ) == ( g x )  )  : (  ( y : a )  -> b y )  f == g = undefined
foo ( b : bool ) : bool = b
```

[Todo] add use cases

5 Goals and Challenges

The parser is still quite primitive, and needs to be extended extensively to support natural language ambiguity in mathematics as well as other linguistic nuance that GF captures well, like tense and aspect. This can follow a method expored in Aarne's paper : "Translating between Language and Logic: What Is Easy and What Is Difficult" where one develops a denotational semantics for translating between natural language expressions with the desired AST. The bulk of this work will be writing a Haskell back-end implementing this AST transformation. The extended syntax, designed for linguistic nuance, will be filtered into the core syntax, which is essentially what I have done.

The Resource Grammar Library (RGL) is designed for out-of-the box grammar writing, and therefore much of the linearization nuance can be outsourced to this robust and well-studied library. Nonetheless, each application grammar brings its own unique challenges, and the RGL will only get one so far. My linearization may require extensive tweaking.

Thus far, our parser is only able to parse non-cubical fragments of the cubicalTT standard library. Dealing with Agda pattern matching, it was realized, is outside the theoretical boundaries of GF (at least, if one were to do it in a non ad-hoc way) due to its inability to pass arbitrary strings down the syntax tree nodes during linearization. Pattern matching therefore needs to be dealt with via pre and post processing. Additionally, cubicaltt is weaker at dealing with telescopes than Agda, and so a full generalization to Agda is not yet possible. Universes are another feature future iterations of this Grammar would need to deal with, but as they aren't present in most mathematician's vernacular, it is not seen as relevant for the state of this project.

Records should also be added, but because this grammar supports sigma types, there is no rush. The Identity type is so far deeply embedded in our grammar, so the first code fragment may just be for explanatory purposes. The degree to which the library is extended to cover domain specific information is up to debate, but for now the grammar is meant to be kept as minimal as possible.

One interesting extension, time dependnet, would be to allow for a bidirectional feedback between GF and Agda : thereby allowing ad hoc extensions to GF's ASTs to allow for newly defined Agda functions to be treated with more care, i.e. have an arguement structure rather than just treating everything as variables. This may be too ambitious for the time being.

6 Code

6.1 GF Parser

Here is the abstract syntax specification.

```
abstract Exp = {

flags startcat = Decl ;

    -- note, cubical tt doesn't support inductive families, and therefore the datatype (& label

cat
  Comment ;
  Module  ;
  AIdent ;
  Imp ; --imports, add later
  Decl ;
  Exp ;
  ExpWhere ;
  Tele ;
  Branch ;
  PTele ;
  Label ;
  [AIdent]{0} ; -- "x y z"
  [Decl]{1} ;
  [Tele]{0} ;
  [Branch]{1} ;
  -- [Branch]{0} ;
  [Label]{1} ;
  [PTele]{1} ;
  -- [Exp]{1};

  --cat [C] {n}
  -- =
  --cat ListC ;
  --fun BaseC : C -> ...-> C -> ListC ; -- n C 's
  --fun ConsC : C -> ListC -> ListC

fun

  DeclDef : AIdent -> [Tele] -> Exp -> ExpWhere -> Decl ;
  -- foo ( b : bool ) : bool = b
  DeclData : AIdent -> [Tele] -> [Label] -> Decl ;
  -- data nat : Set where zero | suc ( n : nat )
  DeclSplit : AIdent -> [Tele] -> Exp -> [Branch] -> Decl ;
  -- caseBool ( x : Set ) ( y z : x ) : bool -> Set = split false -> y || true -> z
  DeclUndef : AIdent -> [Tele] -> Exp -> Decl ;
  -- funExt ( a : Set ) ( b : a -> Set ) ( f g : ( x : a ) -> b x ) ( p : ( x : a ) -
> ( b x ) ( f x ) == ( g x ) ) : ( ( y : a ) -> b y ) f == g = undefined
```

```
Where : Exp -> [Decl] -> ExpWhere ;
-- foo ( b : bool ) : bool =
-- f b where f : bool -> bool = negb
NoWhere : Exp -> ExpWhere ;
-- foo ( b : bool ) : bool =
-- b

Split : Exp -> [Branch] -> Exp ;
--split@ ( nat -> bool ) with zero  -> true || suc n -> false
Let : [Decl] -> Exp -> Exp ;
-- foo ( b : bool ) : bool =
-- let f : bool -> bool = negb in f b
Lam : [PTele] -> Exp -> Exp ;
-- \\ ( x : bool ) -> negb x
-- todo, allow implicit typing
Fun : Exp -> Exp -> Exp ;
-- Set -> Set
-- Set -> ( b : bool ) -> ( x : Set ) -> ( f x )
Pi    : [PTele] -> Exp -> Exp ;
--( f : bool -> Set ) -> ( b : bool ) -> ( x : Set ) -> ( f x )
-- ( f : bool -> Set ) ( b : bool ) ( x : Set ) -> ( f x )
Sigma : [PTele] -> Exp -> Exp ;
-- ( f : bool -> Set ) ( b : bool ) ( x : Set ) * ( f x )
App : Exp -> Exp -> Exp ;
-- proj1 ( x , y )
Id : Exp -> Exp -> Exp -> Exp ;
-- Set bool == nat
IdJ : Exp -> Exp -> Exp -> Exp -> Exp -> Exp ;
-- J e d x y p
Fst : Exp -> Exp ; -- "proj1 x"
Snd : Exp -> Exp ; -- "proj2 x"
-- Pair : Exp -> [Exp] -> Exp ;
Pair : Exp -> Exp -> Exp ;
-- ( x , y )
Var : AIdent -> Exp ;
-- x
Univ : Exp ;
-- Set
Refl : Exp ;
-- refl
--Hole : HoleIdent -> Exp ; -- need to add holes

OBranch :  AIdent -> [AIdent] -> ExpWhere -> Branch ;
-- suc m -> split@ ( nat -> bool ) with zero -> false || suc n -> equalNat m n
-- for splits

OLabel : AIdent -> [Tele] -> Label ;
-- suc ( n : nat )
-- fora data types

-- construct telescope
```

```
  TeleC : AIdent -> [AIdent] -> Exp -> Tele ;
  -- "( f g : ( x : a ) -> b x )"
    --  (  a  :  Set  )  (  b  :  (  a  )  ->  (  Set  )  )  (  f  g  :  (  x  :  a  )  -
> ( ( b ) ( x ) ) ) ( p : ( x : a ) -> ( ( ( b ) ( x ) ) ( ( f ) ( x ) ) == ( ( g ) ( x ) ) ) )

  -- why does gr with this fail so epically?
  PTeleC : Exp -> Exp -> PTele ; -- ( x : Set ) -- ( y : x -> Set )" -- ( x : f y z )"

  --everything below this is just strings

  Foo : AIdent ;

 A , B , C , D , E , F , G , H , I , J , K , L , M , N , O , P , Q , R , S , T , U , V , W , X , Y , Z : AI

  True , False , Bool : AIdent ;
  NegB : AIdent ;
  CaseBool : AIdent ;
  IndBool : AIdent ;

  FunExt : AIdent ;

  Nat : AIdent ;
  Zero : AIdent ;
  Suc : AIdent ;
  EqualNat : AIdent ;

  Unit : AIdent ;
  Top : AIdent ;
}
```

And here is the concrete specification.

```
concrete ExpCubicalTT of Exp = open Prelude, FormalTwo in {

lincat
  Comment,
  Module ,
  AIdent,
  Imp,
  Decl ,
  ExpWhere,
  Tele,
  Branch ,
  PTele,
  Label,
    -- = Str ;
  [AIdent],
  [Decl] ,
  -- [Exp],
  [Tele],
```

```
  [Branch] ,
  [PTele],
  [Label]
    -- = {hd,tl : Str} ;
    = Str ;
  Exp = TermPrec ;

lin

  DeclDef a lt e ew = a ++ lt ++ ":" ++ usePrec 0 e ++ "=" ++ ew ;
  DeclData a t d = "data" ++ a ++ t ++ ": Set where" ++ d ;
  DeclSplit ai lt e lb = ai ++ lt ++ ":" ++ usePrec 0 e ++ "= split" ++ lb ;
   DeclUndef a lt e = a ++ lt ++ ":" ++ usePrec 0 e ++ "= undefined" ; --
postulate in agda

  Where e ld = usePrec 0 e ++ "where" ++ ld ;
  NoWhere e = usePrec 0 e ;

  Let ld e = mkPrec 0 ("let" ++ ld ++ "in" ++ (usePrec 0 e)) ;
  Split e lb = mkPrec 0 ("split@" ++ usePrec 0 e ++ "with" ++ lb) ;
  Lam pt e = mkPrec 0 ("\\" ++ pt ++ "->" ++ usePrec 0 e) ;
  Fun = infixr 1 "->" ; -- A -> Set
  Pi pt e = mkPrec 1 (pt ++ "->" ++ usePrec 1 e) ;
  Sigma pt e = mkPrec 1 (pt ++ "*" ++ usePrec 1 e) ;
  App = infixl 2 "" ;
  Id e1 e2 e3 = mkPrec 3 (usePrec 4 e1 ++ usePrec 4 e2 ++ "==" ++ usePrec 3 e3) ;
-- for an explicit vs implicit use of parameters, may have to use expressions as records, with a
  IdJ e1 e2 e3 e4 e5 = mkPrec 3 ("J" ++ usePrec 4 e1 ++ usePrec 4 e2 ++ usePrec 4 e3 ++ usePrec 4 e
  Fst e = mkPrec 4 ("proj1" ++ usePrec 4 e) ;
  Snd e = mkPrec 4 ("proj2" ++ usePrec 4 e) ;
  Pair e1 e2 = mkPrec 5 ("(" ++ usePrec 0 e1 ++ "," ++ usePrec 0 e2 ++ ")") ;
  Var a = constant a ;
  Univ = constant "Set" ;
  Refl = constant "refl" ;

  BaseAIdent = "" ;
  ConsAIdent x xs = x ++ xs ;

  -- [Decl] only used in ExpWhere
  BaseDecl x = x ;
  ConsDecl x xs = x ++ "\n" ++ xs ;

  -- maybe accomodate so split on empty type just gives ()
  -- BaseBranch = "" ;
  BaseBranch x = x ;
  -- ConsBranch x xs = x ++ "\n" ++ xs ;
  ConsBranch x xs = x ++ "||" ++ xs ;

  -- for data constructors
  BaseLabel x = x ;
  ConsLabel x xs = x ++ "|" ++ xs ;
```

```
BasePTele x = x ;
ConsPTele x xs = x ++ xs ;

BaseTele = "" ;
ConsTele x xs = x ++ xs ;

OBranch a la ew = a ++ la ++ "->" ++ ew ;
TeleC a la e = "(" ++ a ++ la ++ ":" ++ usePrec 0 e ++ ")" ;
PTeleC e1 e2 = "(" ++ top e1 ++ ":" ++ top e2 ++ ")" ;

OLabel a lt = a ++ lt ;

--object language syntax, all variables for now

Bool = "bool" ;
True = "true" ;
False = "false" ;
CaseBool = "caseBool" ;
IndBool = "indBool" ;
FunExt = "funExt" ;

Nat = "nat" ;
Zero = "zero" ;
Suc = "suc" ;
EqualNat = "equalNat" ;

Unit = "unit" ;
Top = "top" ;

Foo = "foo" ;

A = "a" ;
B = "b" ;
C = "c" ;
D = "d" ;
E = "e" ;
F = "f" ;
G = "g" ;
H = "h" ;
I = "i" ;
J = "j" ;
K = "k" ;
L = "l" ;
M = "m" ;
N = "n" ;
O = "o" ;
P = "p" ;
Q = "q" ;
R = "r" ;
S = "s" ;
```

```
  T = "t" ;
  U = "u" ;
  V = "v" ;
  W = "w" ;
  X = "x" ;
  Y = "y" ;
  Z = "z" ;

  NegB = "negb" ;

  -- p "foo ( b : bool ) : bool = f b where f : bool -> bool = negb"
}
```

6.2 Additional Agda Hott Code

[ToDo, clean this up, a lot!]

```
open import Agda.Builtin.Sigma public

apfHom : {A B : Set} {x y z : A} (p : x ≡ y) (f : A → B) (q : y ≡ z) → apf f (p • q) ≡ (apf f p) • (apf f q)
apfHom {A} {B} {x} {y} {z} p f q = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = {f : A → B} {q : y ≡ z} → apf f (p • q) ≡ (apf f p) • (apf f q)
    d : (x : A) → D x x r
    d x = r

apfInv : {A B : Set} {x y : A} (p : x ≡ y) (f : A → B) → apf f (p ⁻¹) ≡ (apf f p) ⁻¹
apfInv {A} {B} {x} {y} p f = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = {f : A → B} → apf f (p ⁻¹) ≡ (apf f p) ⁻¹
    d : (x : A) → D x x r
    d x = r

-- compostion
infixl 40 _∘_
_∘_ : {A B C : Set} → (B → C) → (A → B) → (A → C)
(g ∘ f) x = g (f x)

apfComp : {A B C : Set} {x y : A} (p : x ≡ y) (f : A → B) (g : B → C) → apf g (apf f p) ≡ apf (g ∘ f) p
apfComp {A} {B} {C} {x} {y} p f g = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = {f : A → B} {g : B → C} → apf g (apf f p) ≡ apf (g ∘ f) p
    d : (x : A) → D x x r
    d x = r

id : {A : Set} → A → A
id = λ z → z
```

```
-- apfId : {A B : Set} {x y : A} (p : x ≡ y) (f : _≡_ {A}) → apf f p ≡ p

apfId : {A : Set} {x y : A} (p : x ≡ y) → apf id p ≡ p
apfId {A} {x} {y} p = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = apf id p ≡ p
    d : (x : A) → D x x r
    d = λ x → r
-- D x y p = {f : A → B} → apf f (p ⁻¹) ≡ (apf f p) ⁻¹

-- apfHom : {A} {x y z} (p q) : apf (p • q) ≡ apf p • apf q


transport : ∀ {A : Set} {P : A → Set} {x y : A} (p : x ≡ y) → P x → P y
transport {A} {P} {x} {y} = J D d x y
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = P x → P y
    d : (x : A) → D x x r
    d = λ x → id

-- all from escardo
-- trans' : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
-- trans' {x = x} p q = transport {P = λ - → x ≡ - } q p

-- trans' : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
-- trans' {x = x} {y = y} {z = z} p q = transport {P = λ - → - ≡ z } (p ⁻¹) q

-- i think this is the solution escardo's after
-- trans' : {A : Set} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
-- trans' {x = x} {y = y} {z = z} p q = transport {P = λ _ → x ≡ z } p (transport {P =

-- inv : {A : Set} {x y : A} → x ≡ y → y ≡ x
-- inv {x = x} p = transport {P = λ - → - ≡ x} p r

-- ap : {A B : Set} (f : A → B) (x y : A) → x ≡ y → f x ≡ f y
-- ap f x y p = transport {P = λ - → f x ≡ f - } p r


-- transport : ∀ {A : Set} (P : A → Set) {x y : A} (p : x ≡ y) → P x → P y
-- transport {A} P {x} {y} = J D d x y

-- transport' : ∀ {A : Set} {P : A → Set} {x y : A} (p : x ≡ y) → P x → P y
-- transport' r u = u


p* : {A : Set} {P : A → Set} {x : A} {y : A} {p : x ≡ y} → P x → P y
-- p* {P = P} {p = p} u = transport P p u
p* {P = P} {p = p} u = transport p u

_* : {A : Set} {P : A → Set} {x : A} {y : A} (p : x ≡ y) → P x → P y
(p *) u = transport p u
-- p * u = transport p u
```

```
_×_ : Set → Set → Set
A × B = Σ A (λ _ → B)

_->'_ : Set → Set → Set
-- _->'_ : ?
A ->' B = ((x : A) → B)

arrow : Set₁
arrow = (A B : Set) {b : B} → ((x : A) → B)

constDepType : (A B : Set) → (A → Set)
constDepType A B = λ _ → B

-- transportArrow : {A B : Set} → {x y : A} (p : x ≡ y) → B → B
-- transportArrow {A} {B} p = transport (constDepType A B) p

lift : {A : Set} {P : A → Set} {x y : A} (u : P x) (p : x ≡ y) → (x , u) ≡ (y , p* {P = P} {p = p} u)
lift {P} u r = r --could use J, but we'll skip the effort for now


-- the type inference needs p below
apd : {A : Set} {P : A → Set} (f : (x : A) → P x) {x y : A} {p : x ≡ y}
  → p* {P = P} {p = p} (f x) ≡ f y
apd {A} {P} f {x} {y} {p} = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = p* {P = P} {p = p} (f x) ≡ f y
    d : (x : A) → D x x r
    d = λ x → r

-- the type inference needs p below


-- transportconst : {A B : Set} {x y : A} {p : x ≡ y} (b : B) → transport (constDepT
-- where P(x) :≡ B
transportconst : {A B : Set} {x y : A} {p : x ≡ y} (b : B) → transport {P = λ _ → B} p b ≡ b
transportconst {A} {B} {x} {y} {p} b = J D d x y p
  where
    D : (x y : A) → x ≡ y → Set
    D x y p = transport {P = constDepType A B} p b ≡ b
    d : (x : A) → D x x r
    d = λ x → r

-- 2.3.9

-- problem is we can't avoid keeping the P around to keep the type inference happy
twothreenine : {A : Set} {P : A → Set} {x y z : A} (p : x ≡ y) (q : y ≡ z ) {u : P x} → ((q *) (_* {P = P}
-- twothreenine : {A : Set} {P : A → Set} {x y z : A} (p : x ≡ y) (q : y ≡ z ) {u :
twothreenine r r = r

twothreeten : {A B : Set} {f : A → B} {P : B → Set} {x y : A} (p : x ≡ y) {u : P (f x) } → transport p u
twothreeten r = r

twothreeeleven : {A : Set} {P Q : A → Set} {f : (x : A) → P x → Q x} {x y : A} (p : x ≡ y) {u : P x} → t
```

```
twothreeeleven r = r

-- 2.4

infixl 20 _~_
-- defn of homotopy
_~_ : {A : Set} {P : A → Set} (f g : (x : A) → P x) → Set
-- _~_ {A} f g = (x : A) → f x ≡ g x
f ~ g = (x : _) → f x ≡ g x


-- equiv relation
refl~ : {A : Set} {P : A → Set} → ((f : (x : A) → P x) → f ~ f)
refl~ f x = r

sym~ : {A : Set} {P : A → Set} → (f g : (x : A) → P x) → f ~ g → g ~ f
sym~ f g fg = λ x → fg x ⁻¹


-- composite homotopy
trans~ : {A : Set} {P : A → Set} → (f g h : (x : A) → P x) → f ~ g → g ~ h → f ~ h
trans~ f g h fg gh = λ x → (fg x) • (gh x)


-- transrightidentity, note not defitionally equal
translemma : {A : Set} {x y : A} (p : x ≡ y) → p • r ≡ p
translemma r = r

-- first use of implicit non-definitional equality (oudside of the eckmann hilton argu
hmtpyNatural : {A B : Set} {f g : A → B} {x y : A} (p : x ≡ y) → ((H : f ~ g) → H x • apf g p ≡ apf f p •
hmtpyNatural {x = x} r H = translemma (H x)

-- via wadler's presentation
module ≡-Reasoning {A : Set} where

  infix  1 begin_
  infixr 2 _≡⟨⟩_ _≡⟨_⟩_
  infix  3 _∎

  begin_ : ∀ {x y : A}
    → x ≡ y
    -----
    → x ≡ y
  begin x≡y = x≡y

  _≡⟨⟩_ : ∀ (x : A) {y : A}
    → x ≡ y
    -----
    → x ≡ y
  x ≡⟨⟩ x≡y = x≡y

  _≡⟨_⟩_ : ∀ (x : A) {y z : A}
    → x ≡ y
    → y ≡ z
    -----
```

```
    → x ≡ z
  x ≡⟨ x≡y ⟩ y≡z = x≡y • y≡z

  _∎ : ∀ (x : A)
    -----
    → x ≡ x
  x ∎ = r

open ≡-Reasoning

-- 2.4.4
coroll : {A B : Set} {f : A → A} {x y : A} (p : x ≡ y) → ((H : f ~ (id {A})) → H (f x) ≡ apf f (H x) )
coroll {A} {f = f} {x = x} p H =
  begin
    H (f x)
  ≡⟨ translemma (H (f x)) ⁻¹ ⟩
    H (f x) • r
  ≡⟨ apf (λ - → H (f x) • -) ll51 ⟩
    H (f x) • (apf (λ z → z) (H x) • H x ⁻¹ )
  ≡⟨ associativity (H (f x)) (apf (λ z → z) (H x)) ((H x ⁻¹)) ⟩
    H (f x) • apf (λ z → z) (H x) • H x ⁻¹
  ≡⟨ whisk ⟩
    apf f (H x) • H (x) • H x ⁻¹
  ≡⟨ associativity (apf f (H x)) (H (x)) (H x ⁻¹) ⁻¹ ⟩
    apf f (H x) • (H (x) • H x ⁻¹)
  ≡⟨ apf (λ - → apf f (H x) • -) locallem ⟩
    apf f (H x) • r
  ≡⟨ translemma (apf f (H x)) ⟩
    apf f (H x) ∎
  where
    thatis : H (f x) • apf (λ z → z) (H x) ≡ apf f (H x) • H (x)
    thatis = hmtpyNatural (H x) H
    whisk : H (f x) • apf (λ z → z) (H x) • H x ⁻¹ ≡ apf f (H x) • H (x) • H x ⁻¹
    whisk = thatis •ᵣ (H x ⁻¹)
    locallem :  H x • H x ⁻¹ ≡ r
    locallem = rightInverse (H x)
    ll51 : r ≡ apf (λ z → z) (H x) • H x ⁻¹
    ll51 = locallem ⁻¹ • (apf (λ - → - • H x ⁻¹) (apfId (H x))) ⁻¹

-- Defn. 2.4.6
-- has-inverse in Reijke
qinv : {A B : Set} → (f : A → B) → Set
qinv {A} {B} f = Σ (B → A) λ g → (f ∘ g ~ id {B}) × (g ∘ f ~ id {A})

-- examples

-- 2.4.7

qinvid : {A : Set} → qinv {A} {A} id
qinvid = id , (λ x → r) , λ x → r

-- 2.4.8

p• : {A : Set} {x y z : A} (p : x ≡ y) → ((y ≡ z) → (x ≡ z))
```

```
p• p = λ - → p • -

qinvcomp : {A : Set} {x y z : A} (p : x ≡ y) → qinv (p• {A} {x} {y} {z} p)
qinvcomp p = (λ - → p ⁻¹ • -) , sec , retr
  where
    sec : (λ x → p• p (p ⁻¹ • x)) ~ (λ z → z)
    sec x =
      begin
        p• p (p ⁻¹ • x)
      ≡⟨ associativity p (p ⁻¹) x ⟩
        (p • p ⁻¹) • x
      ≡⟨ apf (λ - → - • x) (rightInverse p) ⟩
        r • x
      ≡⟨ idl x ⁻¹ ⟩
        x ∎
    retr : (λ x → p ⁻¹ • p• p x) ~ (λ z → z)
    retr x =
      begin
        p ⁻¹ • p• p x
      ≡⟨ associativity (p ⁻¹) p x ⟩
        (p ⁻¹ • p) • x
      ≡⟨ apf (λ - → - • x) (leftInverse p) ⟩
        x ∎


-- 2.4.9
sec' : {A : Set} {P : A → Set} {x y : A} (p : x ≡ y) → (λ x₁ → transport {P = P} p (transport (p ⁻¹) x₁)) ~
sec' r x = r

qinvtransp : {A : Set} {P : A → Set} {x y : A} (p : x ≡ y) → qinv (transport {P = P} p)
qinvtransp {A} {P} {x} {y} p = transport (p ⁻¹) , sec , retr p
  where
    sec : (λ x₁ → transport p (transport (p ⁻¹) x₁)) ~ (λ z → z)
    sec z = sec' p z
            -- type inference not working, ugh.
            -- begin transport p (transport (p ⁻¹) z)
            -- ≡⟨ twothreenine (p ⁻¹) p ⟩
            -- ((p ⁻¹ • p) *) z
            -- -- ≡⟨ apf {A = _ ≡ _} {B = P _} {x = p ⁻¹ • p} {y = r} (λ - → (- *) z) (l
            -- ≡⟨ apf (λ - → (- *) z) (leftInverse p) ⟩
            -- -- ≡⟨ apf ? (leftInverse p) ⟩
            -- (r *) z
            -- ≡⟨ {!!} ⟩
            -- z ∎
    retr : (p : x ≡ y) → (λ x₁ → transport (p ⁻¹) (transport p x₁)) ~ (λ z → z)
    retr r z = r

isequiv : {A B : Set} → (f : A → B) → Set
isequiv {A} {B} f = Σ (B → A) λ g → (f ∘ g ~ id {B}) × Σ (B → A) λ g → (g ∘ f ~ id {A})

qinv->isequiv : {A B : Set} → (f : A → B) → qinv f → isequiv f
qinv->isequiv f (g , α , β) = g , α , g , β
```

```
-- not the same is as the book, but I can't understand what the book is doing.  maybe t
isequiv->qinv : {A B : Set} → (f : A → B) → isequiv f → qinv f
isequiv->qinv f (g , α , g' , β ) = (g' ∘ f ∘ g) , sec , retr
  where
    sec : (λ x → f (g' (f (g x)))) ~ (λ z → z)
    sec x = apf f (β (g x)) • α x
      -- begin f (g' (f (g x)))
      -- ≡⟨ apf f (β (g x)) ⟩
      -- f (g x)
      -- ≡⟨ α x ⟩
      -- x ∎
    retr : (λ x → g' (f (g (f x)))) ~ (λ z → z)
    retr x = apf g' (α (f x)) • β x
      -- begin g' (f (g (f x)))
      -- ≡⟨ apf g' (α (f x)) ⟩
      -- g' (f x)
      -- ≡⟨ β x ⟩
      -- x ∎

-- book defn, confusing because of the "let this be the composite homotopy" which mixe
isequiv->qinv' : {A B : Set} → (f : A → B) →  isequiv f → qinv f
isequiv->qinv' f (g , α , h , β ) = g , α , β'
  where
    -- γ : λ x → (trans~ ? ? ? ? ?) x -- trans~ g (g' ∘ f ∘ g) g' ? ? -- {!trans~ g
    γ : (λ x → g x) ~ λ x → h x
    γ x = β (g x) ⁻¹ • apf h (α x)
    β' : (λ x → g (f x)) ~ (λ z → z)
    β' x = (γ (f x)) • β x

-- \simeq ≃

_≃_ : (A B : Set) → Set
A ≃ B = Σ (A → B) λ f → isequiv f


≃refl : {A : Set} → A ≃ A
≃refl = (id) , (qi qinvid)
  where
    qi : qinv (λ z → z) → isequiv (λ z → z)
    qi = qinv->isequiv (id )
-- type equivalence is an equivalence relation, 2.4.12
-- qinv->isequiv

-- how to find this in agda automatically?
comm× : {A B : Set} → A × B → B × A
comm× (a , b) = (b , a)

≃sym : {A B : Set} → A ≃ B → B ≃ A
≃sym (f , eqf) = f-1 , ef (f , comm× sndqf)
  where
    qf : isequiv f → qinv f
    qf = isequiv->qinv f
    f-1 : _ → _
```

```
    f-1 = fst (qf eqf)
    sndqf : ((λ x → f (fst (isequiv->qinv f eqf) x)) ~ (λ z → z)) ×
              ((λ x → fst (isequiv->qinv f eqf) (f x)) ~ (λ z → z))
    sndqf = snd (qf eqf)
    ef : qinv f-1 → isequiv f-1
    ef = qinv->isequiv f-1

-- is there any way to make this pattern matching easier
≃trans : {A B C : Set} → A ≃ B → B ≃ C → A ≃ C
≃trans (f , eqf) (g , eqg) = (g ∘ f) , qinv->isequiv (λ z → g (f z)) ((f-1 ∘ g-1) , sec , retr) -- qgf
  where
    qf : isequiv f → qinv f
    qf = isequiv->qinv f
    f-1 = fst (qf eqf)
    qg : isequiv g → qinv g
    qg = isequiv->qinv g
    g-1 = fst (qg eqg)
    sec : (λ x → g (f (f-1 (g-1 x)))) ~ (λ z → z)
    sec x =
            begin g (f (f-1 (g-1 x)))
            ≡⟨ apf g (fst (snd (qf eqf)) (g-1 x)) ⟩
            g (g-1 x)
            ≡⟨ fst (snd (qg eqg)) x ⟩
            x ∎
    retr : (λ x → f-1 (g-1 (g (f x)))) ~ (λ z → z)
    retr x =
             begin f-1 (g-1 (g (f x)))
             ≡⟨ apf f-1 ((snd (snd (qg eqg)) (f x))) ⟩
             f-1 (f x)
             ≡⟨ snd (snd (qf eqf)) x ⟩
             x ∎


-- 2.6.1
fprodId : {A B : Set} {x y : A × B} → _≡_ {A × B} x y → ((fst x) ≡ (fst y)) × ((snd x) ≡ (snd y))
fprodId p = (apf fst p) , (apf snd p)
-- fprodId r = r , r

-- 2.6.2
equivfprod : {A B : Set} (x y : A × B) → isequiv (fprodId {x = x} {y = y} )
equivfprod (x1 , y1) (x2 , y2) = qinv->isequiv fprodId (sn , h1 , h2)
  where
    sn : (x1 ≡ x2) × (y1 ≡ y2) → (x1 , y1) ≡ (x2 , y2)
    sn (r , r) = r
    h1 : (λ x → fprodId (sn x)) ~ (λ z → z)
    h1 (r , r) = r
    -- h1 (r , r) = r
    h2 : (λ x → sn (fprodId x)) ~ (λ z → z)
    h2 r = r

-- 2.6.4
-- alternative name consistent with book, A×B
×fam : {Z : Set} {A B : Z → Set} → (Z → Set)
```

```
×fam {A = A} {B = B} z = A z × B z

transport× : {Z : Set} {A B : Z → Set} {z w : Z} (p : z ≡ w) (x : ×fam {Z} {A} {B} z) → (transport p x
transport× r s = r

fprod : {A B A' B' : Set} (g : A → A') (h : B → B') → (A × B → A' × B')
fprod g h x = g (fst x) , h (snd x)

-- inverse of fprodId
pair= : {A B : Set} {x y : A × B} → (fst x ≡ fst y) × (snd x ≡ snd y) → x ≡ y
pair= (r , r) = r

-- 2.6.5
functorProdEq : {A B A' B' : Set} (g : A → A') (h : B → B') (x y : A × B) (p : fst x ≡ fst y) (q : snd x ≡ sn
functorProdEq g h (a , b) (.a , .b) r r = r


-- 2.7.2
-- rename f to g to be consistent with book
equivfDprod : {A : Set} {P : A → Set} (w w' : Σ A (λ x → P x)) → (w ≡ w') ≃ Σ (fst w ≡ fst w') λ p → p*
equivfDprod (w1 , w2) (w1' , w2') = f , qinv->isequiv f (f-1 , ff-1 , f-1f)
  where
    f : (w1 , w2) ≡ (w1' , w2') → Σ (w1 ≡ w1') (λ p → p* {p = p} w2 ≡ w2')
    f r = r , r
    f-1 : Σ (w1 ≡ w1') (λ p → p* {p = p} w2 ≡ w2') → (w1 , w2) ≡ (w1' , w2')
    -- f-1 (r , psndw) = apf (λ - → (w1 , -)) psndw
    f-1 (r , r) = r
    ff-1 : (λ x → f (f-1 x)) ~ (λ z → z)
    ff-1 (r , r) = r
    f-1f : (λ x → f-1 (f x)) ~ (λ z → z)
    f-1f r = r

-- 2.7.3
etaDprod : {A : Set} {P : A → Set} (z : Σ A (λ x → P x)) → z ≡ (fst z , snd z)
etaDprod z = r

-- 2.7.4
-- Σfam : {A : Set} {P : A → Set} (Q : Σ A (λ x → P x) → Set) → ((x : A) → Σ (P x)
Σfam : {A : Set} {P : A → Set} (Q : Σ A (λ x → P x) → Set) → (A → Set)
Σfam {P = P} Q x = Σ (P x) λ u → Q (x , u)

dpair= : {A : Set} {P : A → Set} {w1 w1' : A} {w2 : P w1 } {w2' : P w1'} → (p : Σ (w1 ≡ w1') (λ p →
dpair= (r , r) = r

transportΣ : {A : Set} {P : A → Set} (Q : Σ A (λ x → P x) → Set) (x y : A) (p : x ≡ y) ((u , z) : Σfam Q x)
             → _* {P = λ - → Σfam Q - } p (u , z) ≡ ((p *) u , _* {P = λ - → Q ((fst -) , (snd -))} (dpair= (p
transportΣ Q x .x r (u , z) = r -- some agda bug here.  try ctrl-c ctrl-a

fDprod : {A A' : Set} {P : A → Set} {Q : A' → Set} (g : A → A') (h : (a : A) → P a → Q (g a)) → (Σ A λ a →
fDprod g h (a , pa) = g a , h a pa

-- ap2 : {A B C : Set} (x x' : A) (y y' : B) (f : A → B → C)
-- → (x ≡ x') → (y ≡ y') → (f x y ≡ f x' y')
-- ap2 x x' y y' f r q = ap (λ - → f x -) y y' q
```

```
ap2 : {A B C : Set} {x x' : A} {y y' : B} (f : A → B → C)
      → (x ≡ x') → (y ≡ y') → (f x y ≡ f x' y')
ap2 f r r = r

-- ap2d : {A : Set} {x x' : A} {P : A → Set} {y : P x} {y' : P x'} {C : (x : A) → P
-- → (p : x ≡ x') → (q : (p *) y ≡ y') →
-- p* {p = q} (p* {p = p} (f x)) y ≡ {!f x' y'!}
-- -- (f x y ≡ f x' y')
-- ap2d = {!!}

transportd : {X : Set } (A : X → Set ) (B : (x : X) → A x → Set )
  {x : X} ((a , b) : Σ (A x) λ a → B x a) {y : X} (p : x ≡ y)
  → B x a → B y (transport {P = A} p a)
transportd A B (a , b) r = id

-- ap2d : {A : Set} {x x' : A} {P : A → Set} {y : P x} {y' : P x'} {C : (x : A)
-- → P x → Set} (f : (x : A) → (y : P x) → C x y )
-- → (p : x ≡ x') → (q : p* {P = P} {p = p} y ≡ y') →
-- p* {P = C x'} {p = q} (transportd P C (y , f x y) p (f x y)) ≡ f x' y'
-- ap2d f r r = {!!}

-- functorDProdEq : {A A' : Set} {P : A → Set} {Q : A' → Set} (g : A → A')
-- (h : (a : A) → P a → Q (g a))
-- → (x y : Σ A λ a → P a)
-- → (p : fst x ≡ fst y) (q : p* {p = p} (snd x) ≡ snd y)
-- → apf (λ - → fDprod {P = P} {Q = Q} g h -) (dpair= (p , q))
-- ≡ dpair= (apf g p , ap2d h p {!q!} )
-- functorDProdEq = {!!}


-- 2.8
data Unit : Set where
  ⋆ : Unit

-- in which the composite is referencing the type, not the function applications expli
path1 : (x y : Unit) → (x ≡ y) ≃ Unit
path1 x y = (λ p → ⋆) , qinv->isequiv (λ p → ⋆) (f-1 x y , ff-1 , f-1f x y)
  where
    f-1 : (x y : Unit) → Unit → x ≡ y
    f-1 ⋆ ⋆ ⋆ = r
    ff-1 : (λ x₁ → ⋆) ~ (λ z → z)
    ff-1 ⋆ = r
    f-1f : (x y : Unit) → (λ _ → f-1 x y ⋆) ~ (λ z → z)
    f-1f ⋆ .□ r = r

-- 2.9

happly : {A : Set} {B : A → Set} {f g : (x : A) → B x} → f ≡ g → ((x : A) → f x ≡ g x )
happly r x = r

postulate
  -- funext : ∀ {A : Set} {B : A → Set} {f g : (x : A) → B x} → isequiv (happly {f =
  funext : {A : Set} {B : A → Set} {f g : (x : A) → B x} → ((x : A) → f x ≡ g x ) → f ≡ g
```

```
-- betaPi : {A : Set} {B : A → Set} {f g : (x : A) → B x} (h : (x : A) → f x ≡ g x
-- betaPi h x = {!!}

-- etaPi : {A : Set} {B : A → Set} {f g : (x : A) → B x} (p : f ≡ g) → p ≡ funext λ
-- etaPi p = {!!}

-- -- is this the first example where explict refl arguements are needed
-- reflf : {A : Set} {B : A → Set} {f : (x : A) → B x} → _≡_ {A = (x : A) → B x} r
-- reflf = ?

-- -- is this the first example where explict refl arguements are needed
-- α-1 : {A : Set} {B : A → Set} {f g : (x : A) → B x} {α : f ≡ g} → α ⁻¹ ≡ funext
-- α-1 {A} {B} {f} {.f} {r} = {!r!}

->fam : {X : Set} (A B : X → Set) → X → Set
->fam A B x = A x → B x

-- 2.9.4
transportF : {X : Set} {A B : X → Set} {x1 x2 : X} {p : x1 ≡ x2} {f : A x1 → B x1} →
             transport {P = ->fam A B} p f ≡ λ x → transport {P = B} p (f (transport {P = A} (p ⁻¹) x))
             -- (transport {P = A} p a)
transportF {X} {A} {B} {x1} {.x1} {r} {f} = funext (λ x → r)

-- begin {!!}
-- ≡⟨ {!!} ⟩
-- {!!}
-- ≡⟨ {!!} ⟩
-- {!!} ∎

data List (A : Set) : Set where
  [] : List A
  cons : A → List A → List A

-- exercises Reijke ch 4.

inv_assoc : {A : Set} {x y z : A} (p : x ≡ y) (q : y ≡ z) → (p • q) ⁻¹ ≡ q ⁻¹ • p ⁻¹
inv_assoc r q = ir (q ⁻¹)
-- -- inv_assoc r r = r


iscontr : (A : Set) → Set
iscontr A =  Σ A λ a → (x : A) → (a ≡ x)

-- singind : {A : Set} → (Σ A λ a → {!(B : A → Set) → !})
-- singind {A} = {!!}

data ⊤ : Set where
  star  : ⊤

data ⊥ : Set where

abort : {A : Set} → ⊥ → A
abort ()
```

```
¬ : Set → Set
¬ A = A → ⊥

ind-unit :  {P : ⊤ → Set} → P star → ((x : ⊤) → P x)
ind-unit p star = p

const : (A B : Set) → (b : B) → A → B
const A B b a = b

ex51 :  {A : Set} {a : A} → ind-unit a ~ const ⊤ A a
ex51 star = r

-- -- -- apf (const A B b) z \
-- helper : {A B : Set} (b : B) → (x y : A) (z : x ≡ y) → apf {x = x} {y = y} (cons
-- helper b x .x r = r

ex52 :  {A B : Set} (b : B) → (x y : A) → apf {x = x} {y = y} (const A B b) ~ const (x ≡ y) (b ≡ b) r
ex52 b x .x r = r

invisequiv : {A : Set} (x y : A) → isequiv (_⁻¹ {x = x} {y = y})
invisequiv x y = qinv->isequiv _⁻¹ (_⁻¹ , doubleInv , doubleInv )
-- invisequiv x y = _⁻¹ , doubleInv , _⁻¹ , doubleInv

-- p• : {A : Set} {x y z : A} (p : x ≡ y) → ((y ≡ z) → (x ≡ z))
-- p• p = λ - → p • -

-- qinvcomp : {A : Set} {x y z : A} (p : x ≡ y) → qinv (p• {A} {x} {y} {z} p)
-- qinvcomp p = (λ - → p ⁻¹ • -) , sec , retr

compisequiv : {A : Set} (x y z : A) (p : x ≡ y) → isequiv (_•_ {x = x} {y = y} p {z = z})
compisequiv x y z p = qinv->isequiv (p• p) (qinvcomp p)

-- 5.4
hmtpyinducedEqv : {A B : Set} (f g : A → B) → (H : f ~ g) → isequiv f → isequiv g
hmtpyinducedEqv f g H (f' , ff' , g' , g'f) = f' , trans~ (λ x → g (f' x)) (λ x → f (f' x)) (λ x → x) (λ x → H (
                                              , g' , (trans~ (λ x → g' (g x)) (λ x → g' (f x)) (λ x → x) (λ x → ap

hasSection : {A B : Set} (f : A → B) → Set
hasSection {A} {B} f = Σ (B → A) λ g → (f ∘ g ~ id {B})

hasRetraction : {A B : Set} (f : A → B) → Set
hasRetraction {A} {B} f = Σ (B → A) λ g → (g ∘ f ~ id {A})

-- 5.5
-- {A B X : Set} (f : A → X) (h : A → X) (g : B → X) → (H : f ~ g ∘ h)

eqvSec : {A B : Set} (f : A → B) → isequiv f → hasSection f
eqvSec f (f1 , ff1 , g) = f1 , ff1

eqvRetr : {A B : Set} (f : A → B) → isequiv f → hasRetraction f
eqvRetr f (f1 , ff1 , g1 , gg1) = g1 , gg1

secRetrEqv : {A B : Set} (f : A → B) → hasSection f → hasRetraction f → isequiv f
secRetrEqv f (f-1 , ff-1) (f-1' , f-1'f) = (f-1 , ff-1 , f-1' , f-1'f)
```

```
55a1 : {A B X : Set} (f : A → X) (h : A → B) (g : B → X) → (H : f ~ g ∘ h) → hasSection h → hasSection
55a1 f h g H (h-1 , hh-1) (f-1 , ff-1) = h ∘ f-1 , trans~ (λ x → g (h (f-1 x))) (f ∘ f-1) (λ x → x) (λ x → H (f-1

55a2 : {A B X : Set} (f : A → X) (h : A → B) (g : B → X) → (H : f ~ g ∘ h) → hasSection h → hasSection
55a2 f h g H (h-1 , hh-1) (g-1 , gg-1) = h-1 ∘ g-1 , trans~ (λ x → f (h-1 (g-1 x))) (g ∘ h ∘ h-1 ∘ g-1) (λ x →
  (trans~ (λ x → g (h (h-1 (g-1 x)))) (g ∘ g-1) (λ x → x) (λ x → apf g (hh-1 (g-1 x))) gg-1)

55b1 : {A B X : Set} (f : A → X) (h : A → B) (g : B → X) → (H : f ~ g ∘ h) → hasRetraction g → hasRetra
55b1 f h g H (g-1 , g-1g) (f-1 , f-1f) = f-1 ∘ g , trans~ (λ x → f-1 (g (h x))) (f-1 ∘ f) (λ x → x) (λ x → apf f-1

55b2 : {A B X : Set} (f : A → X) (h : A → B) (g : B → X) → (H : f ~ g ∘ h) → hasRetraction g → hasRetra
55b2 f h g H (g-1 , g-1g) (h-1 , h-1h) = (λ z → h-1 (g-1 z)) , (trans~ (λ x → h-1 (g-1 (f x))) (h-1 ∘ g-1 ∘ g
  (trans~ (λ x → h-1 (g-1 (g (h x)))) (h-1 ∘ h) (λ x → x) (λ x → apf h-1 (g-1g (h x))) h-1h))

dunno' : {A B X : Set} (f : A → X) (h : A → B) (g : B → X) → (H : f ~ g ∘ h) → isequiv f → isequiv g → is
dunno' f h g H (f-1 , ff-1 , f'-1 , f'-1f) (g-1 , gg-1 , g'-1 , g'-1g) = secRetrEqv h (55a2 h f g'-1 (trans~ h (g

3for2a : {A B X : Set} (f : A → X) (g : X → B) → isequiv f → isequiv g → isequiv (g ∘ f)
3for2a f g ef eg = secRetrEqv (λ x → g (f x)) (55a2 (g ∘ f) f g (λ x → r) (eqvSec f ef) (eqvSec g eg))
                                                (55b2 (g ∘ f) f g (λ x → r) (eqvRetr g eg) (eqvRetr f ef))

dunno : {A B X : Set} (f : A → X) (h : A → B) (g : B → X) → (H : f ~ g ∘ h) → isequiv f → isequiv h → ise
dunno f h g H (f-1 , ff-1 , f'-1 , f'-1f) (h-1 , hh-1 , h'-1 , h'-1h) = secRetrEqv g (55a1 f h g H (h-1 , hh-1) (
  (55b2 g h-1 f (trans~ g (g ∘ h ∘ h-1) (λ x → f (h-1 x)) (λ x → apf g (hh-1 x ⁻¹)) λ x → (H (h-1 x) ⁻¹)) (f'-

3for2b : {A B X : Set} (f : A → X) (g : X → B) → isequiv f → isequiv (g ∘ f) → isequiv g
3for2b f g ef egf = secRetrEqv g (55a1 (g ∘ f) f g (λ x → r) (eqvSec f ef) (eqvSec (λ z → g (f z)) egf)) (eq

3for2c : {A B X : Set} (f : A → X) (g : X → B) → isequiv g → isequiv (g ∘ f) → isequiv f
3for2c f g eg egf = secRetrEqv f (eqvSec f (dunno' (g ∘ f) f g (λ x → r) egf eg)) (55b1 (g ∘ f) f g (λ x → r
```

References

[1] The Univalent foundations program and N.J.) Institute for advanced study (Princeton. *Homotopy Type Theory: Univalent Foundations of Mathematics*. 2013.