Abstract

The following is a formalisation of many of the proofs of Part [multisets-in-type-theory] and Part [iterative-sets] in Agda. Agda is a proof-checker and programming language based on dependent type theory and pattern matching, which allow a very close correspondence between the style of proofs in the article and the formalised, machine-readable proof.

Some proofs are present both in the article and the formalisation, while others are only present in one of them. Some proofs in the article are mere sketches, while the details are in the formalised proof.

The original motivation for starting this formalisation was to make the details of Lemma [fibre-equiv] in Part [multisets-in-type-theory] clear. In fact, the informal proof in the article only reached its current form after the formal proof was completed. After the formalised proof was done it was a lot easier to pick out the important elements, which a human might be interested in, hiding the tedious details behind the phrase "…can be verified by -induction…".

A strength of Agda is that it gives easy access to term-normalisation. Being able to normalise terms quickly opens up for more experimentation to simplify proofs than if each normalisation would be carried out by hand. In our development this has manifested as simplifications of the proof of Lemma [w-id] and Lemma [fibre-fun] of Part [multisets-in-type-theory]. Both of which were first proved by more complicated path computations, but once formalised, inserting the reflexivity proof various places, `idp`, and allowing Agda to attempt unification, showed that there were simplifications to be made.

Prerequisites

Proof-checkers are pieces of software, undergoing development for long periods of time and therefore existing in many versions. Agda is no exception, and it is a matter of fact that a proof formalised and checked by one version may fail to check by the next version of Agda. This may be due to introduction of errors in the new version, removal of errors from the previous version, or due to intended changes in syntax. It is therefore of essence to specify along with any formalisation what version of the proof-checker the code has been checked with. If the formalisation depend on any library, such must also be specified with versions. In practise, locating and installing old versions of all dependencies – and all dependencies of dependencies, etc. – on a new system, may prove difficult.

An alternative route, which requires some dedication, is to keep the formalisation up to date with the latest changes in the proof-checker. At the very least, one should aim to write the formalisation in such a way that it is not only machine-readable, but also human-readable, so that in case the specific version of the proof-checker should disappear, and the code cease to verify on any available versions, the intention behind the code should be clear so that any able reader can themself perform the changes needed for the code to verify on a later version.

Our formalisation depends on the HoTT-Agda library, which is a development of Homotopy Type Theory in Agda.

Prerequisite	Tested versions	URL
Agda	2.4.2.4	
HoTT-agda	eb24ea20e1a28de31	

Assumptions of the formalisation

Agda implements dependent type theory with pattern matching. Traditionally, the way pattern matching has been implemented allows proofs of the uniqueness of identity proofs (UIP) to type check[1], which is inconsistent with Univalence. However, recent version includes an option to tighten the restraints on pattern matching so that one can no-longer prove UIP. This allows HoTT-agda, upon which our development depends, to postulate the Univalence Axiom.

Agda allows defining new recursive data types and functions. To ensure consistency of these it applies positivity checks and termination checks. However, all types used in our development are well-known from type theory literature, and thus we believe that the proofs should be transferable to any proof-checker with a type system able to represent the following type constructs:

- Π-types
- Σ-types
- Binary sum types
- W-types
- Id-types
- A univalent universe
- (-1)-truncation (for set theoretical axioms on V)
- Mere set quotients (for set theoretical axioms on V)

The proofs concerning the identity type of W-types uses, η-reduction for functions (i.e. $(\lambda x.fx) \equiv f$). One may speculate that the proofs could be modified to hold even if η-reduction only holds up to identity, but most dependent type systems have some function type with η-reduction, so we do not pursue that road here.

[1]The principle states that for $a, b : A$ and any $\alpha, \beta : \mathrm{Id}_A\ a\, b$ we have that $\mathrm{Id}\ \alpha\, \beta$.

Case study

To demonstrate how the human readable proofs and the formal proofs relate, we will have a closer look at a single theorem, and its incarnation in both article and code. The theorem chosen is the theorem which shows that our chosen notion of equality for multisets is equivalent to the identity type – given the univalence axiom. This is found in Part [multisets-in-type-theory] as Theorem [id-eq-theorem]. The code formalising the theorem is quoted below.

```
Id-is-Eq' : (i : ULevel) (x y : M i) → (x == y) ≃ Eq' x y
Id-is-Eq' i (sup A f) (sup B g) = IH ∘e EXT ∘e UA ∘e WLEM where

   IH : Σ (A ≃ B) (\α → ((x : A) → (g (apply α x)) == (f x)))
      ≃ Eq' (sup A f) (sup B g)
   IH = equiv-Σ-snd (\α → equiv-Π (ide _)
                                    (\x → Id-is-Eq' i (g (apply α x)) (f x)))

   EXT : Σ (A ≃ B) (\α → (g ∘ (apply α) == f))
       ≃ Σ (A ≃ B) (\α → ((x : A) → (g (apply α x)) == (f x)))
   EXT = equiv-Σ-snd (\α → app=-equiv)

   lem0 : {A : Type i} → (ua (ide A)) == idp
   lem0 = ua-η idp
   lem = equiv-induction
          (\α → transport (T i) (fst ua-equiv α) == apply α)
          (\A → ap (\f → coe (ap (T i) f)) lem0)
   UA : (Σ (A == B) (\α → (g ∘ (transport (T i) α) == f)))
      ≃  Σ (A ≃ B)  (\α → (g ∘ (apply α) == f))
   UA = (equiv-Σ-snd (\α → coe-equiv (ap (\h → g ∘ h == f) (lem α))))
     ∘e (equiv-Σ-fst (\α → g ∘ (transport (T i) α) == f)(snd ua-equiv)) ⁻¹
   WLEM : ((sup A f) == (sup B g))
        ≃ Σ (A == B) (\α → (g ∘ (transport (T i) α) == f))
   WLEM = W-id-≃ (T i)


Id≃Eq : (i : ULevel) (x y : M i) → (x == y) ≃ Eq x y
Id≃Eq = \i x y → Eq'-is-Eq x y ∘e Id-is-Eq' i x y
```

Starting at the bottom, the term `Id≃Eq` is the proof of the theorem we want. We can see the statement of the theorem on the line above. The type `Eq` is the name in the formalisation of $=_M$, and '==' is the identity type.

While M in our paper was defined with an implicit dependency of a universe U, the Agda formalisation defines `M i` having an explicit dependency of a universe level `i`. That's why the type of `Id≃Eq` starts with the quantification `(i : ULevel)`.

Looking at the definition of `Id≃Eq` one quickly sees mention of something called `Eq'`, which doesn't appear in the proof of the article. This is because in its natural formulation `Eq` lies one universe level below the identity type of `M`, and it turned out that the cleanest way to handle this in Agda was to define an auxiliary type with the same inductive definition as `Eq` on a higher level.

Thus, the essence of the proof is in the term `Id-is-Eq'`, and it closely corresponds to the informal proof, which is a chain of equivalences. In the formalisation each step is given short name: `IH`, `EXT`, `UA` and `WLEM`.

The informal proof has an addition step helping the reader recognise the definition of a homotopy between two function. Agda usually needs no help recognising definitional equalities. However, each step in the proof is easy enough that the reader will understand without further explanation, Agda needs to be presented the exact terms. Each step therefore is given a definition in the formalisation – in each case a simple adaption of some already known term. Perhaps the only step which seems unduly complicated is the definition of `UA`. This might be due to the fact that the univalence axiom is a kind of artificial addition to Agda, which is simply postulated and doesn't come with many convenient definitional equalities, forcing a bit of transport yoga.

In conclusion, we see that there is as close a correspondence between the informal proof and the Agda formalisation as we could hope for. However, they differ in which details we chose to leave to the reader/proof-checker. The human can be trusted to see how to apply already known theorems in the relevant way, while Agda excel at computing definitional equalities.

Selected excerps from HoTT-agda

The formalisation of our results depends on the HoTT-Agda library[2] – mostly for standard definitions of Homotopy Type Theory. In stead of quoting lengthy files of code from the library, we include here the types of some of the terms we use in our formalisation.

```
PathOver : ∀ {i j} {A : Type i} (B : A → Type j)
  {x y : A} (p : x == y) (u : B x) (v : B y) → Type j

syntax PathOver B p u v =
  u == v [ B ↓ p ]

ap : ∀ {i j} {A : Type i} {B : Type j} (f : A → B) {x y : A}
  → (x == y → f x == f y)
```

[2]The HoTT-Agda library is © 2013 Guillaume Brunerie, Evan Cavallo, Favonia, Nicolai Kraus, Dan Licata, Christian Sattler

```
transport : ∀ {i j} {A : Type i} (B : A → Type j) {x y : A} (p : x == y)
  → (B x → B y)

Π : ∀ {i j} (A : Type i) (P : A → Type j) → Type (lmax i j)
Σ : ∀ {i j} (A : Type i) (B : A → Type j) → Type (lmax i j)

_•_ : {x y z : A}
    → (x == y → y == z → x == z)

_•'_ : {x y z : A}
     → (x == y → y == z → x == z)

•'=• : {x y z : A} (p : x == y) (q : y == z)
     → p •' q == p • q

•'-assoc : {x y z t : A} (p : x == y) (q : y == z) (r : z == t)
    → (p •' q) •' r == p •' (q •' r)

•-unit-r : {x y : A} (q : x == y) → q • idp == q

•'-unit-l : {x y : A} (q : x == y) → idp •' q == q

! : {x y : A} → (x == y → y == x)

!-inv-l : {x y : A} (p : x == y) → (! p) • p == idp
!-inv-r : {x y : A} (p : x == y) → p • (! p) == idp

 _•ᵈ_ : {x y z : A} {p : x == y} {p' : y == z}
        {u : B x} {v : B y} {w : B z}
      → (u == v [ B ↓ p ]
      → v == w [ B ↓ p' ]
      → u == w [ B ↓ (p • p') ])
```