Heq: a Coq library for Heterogeneous Equality

Chung-Kil Hur*

PPS, Université Paris Diderot

Abstract. We give an introduction to the library Heq, which provides a set of tactics to manipulate heterogeneous equality and explicit coercion, such as rewriting of heterogeneous equality and elimination and relocation of explicit coercions.

1 Introduction

One of the main strengths of Coq is that it fully supports dependent types. However, it is unfortunate that sometimes explicit coercion is needed for dependent terms to type check and dealing with such coercion is quite a big overhead. This is due to the intensional type theory of Coq that uses convertibility as an internal notion of equality, rather than Leibniz equality, for the sake of decidability of type checking.

The purpose of the library Heq is to reduce this overhead to deal with explicit coercion. We achieve this goal mainly in two ways. First, we use a notion of heterogeneous equality, which hides explicit coercions occurring in equality propositions. Second, more importantly, we provide a set of tactics that helps to manipulate explicit coercions: mainly elimination and relocation of them.

This abstract is an introduction to the library Heq. The library and a more detailed tutorial are available at [4]. It is compatible with Coq 8.2pl1.

2 Motivating example

We take `vector` as a running example and consider the vector append operation `vapp`.

```
Set Implicit Arguments.

Inductive vector (A:Type) : nat → Type :=
| vnil : vector A 0
| vcons (hd:A) n (tl:vector A n): vector A (S n).
Implicit Arguments vnil [A].

Infix ":::" := vcons (at level 60, right associativity).

Fixpoint vapp A n (v: vector A n) n' (v':vector A n') : vector A (n+n') :=
  match v with
    | vnil ⇒ v'
    | vcons hd _ tl ⇒ hd ::: vapp tl v'
  end.

Infix "+++" := vapp (right associativity, at level 61).
```

To see the problem, let us try to write a lemma saying that appending `vnil` to a vector yields the same vector.

* Research supported by Digiteo/Ile-de-France project COLLODI (2009-28HD)

```
Lemma vapp_nil_eq : ∀ A n (v:vector A n), v +++ vnil = v.
```

This code seems right, but Coq complains that it does not type check because `v +++ vnil` and `v` have inconvertible types `vector A (n+0)` and `vector A n`. One way to solve this problem is to use an explicit cast.

```
Lemma vapp_nil_cast : ∀ A n (v:vector A n), v +++ vnil = eq_rect _ (vector A) v _ (plus_n_O n).
```

Here we coerce the term `v` to `vector A (n+0)` using the cast operator `eq_rect` and the proof `plus_n_O n : n = n + 0`. Though this idea works, it is quite inconvenient to write explicit proof terms like `plus_n_O n`.

3 Heterogeneous equality

A well-known idea to hide explicit coercions in equality propositions is to use heterogeneous equality such as JMeq [5] and dependent equality `eq_dep`.

```
Inductive JMeq (A : Type) (x : A) : ∀ B : Type, B → Prop :=
  JMeq_refl : JMeq x x.

Lemma vapp_nil_jmeq : ∀ A n (v:vector A n), JMeq (v +++ vnil) v.

Inductive eq_dep (U : Type) (Sig : U → Type) (p : U) (x : Sig p) : ∀ q : U, Sig q → Prop :=
  eq_dep_intro : eq_dep Sig p x p x.
Implicit Arguments eq_dep [U p q].

Lemma vapp_nil_eqdep : ∀ A n (v:vector A n), eq_dep (vector A) (v +++ vnil) v.
```

`JMeq` and `eq_dep` relate terms of different types, but such an equality has a proof only when the corresponding types are provably equal and the corresponding terms are related by appropriate coercions.

`JMeq` looks simpler because it does not require any signature like `vector A`, but there is a price to pay: `p = q` is derivable from `eq_dep Sig x y` for `x : Sig p` and `y : Sig q`; while only `Sig p = Sig q` from `JMeq x y`. As the signature `Sig` may not be injective in general[1], `eq_dep` is stronger than `JMeq`. Since the former property plays a crucial role in our tactics for dealing with heterogeneous equality, JMeq is not appropriate for our purposes.

Dependent equality `eq_dep` has an alternative representation as equality on dependent pairs (see Eqdep.v in the standard library of Coq).

```
Lemma equiv_eqex_eqdep: ∀ (U : Type) (Sig : U → Type) (p q : U) (x : Sig p) (y : Sig q),
  existT Sig p x = existT Sig q y ↔ eq_dep U P p x q y.
```

We use equality on dependent pairs instead of `eq_dep` as the former has some technical advantages, which will be discussed in Section 6.

4 Library Heq

We introduce the library Heq and show how it deals with heterogeneous equality and explicit casts.

[1] The author has shown that assuming the injectivity of inductive type constructors is inconsistent with the law of excluded middle (see `https://lists.chalmers.se/pipermail/agda/2010/001522.html` for the discussion).

Though explicit casts can be avoided in equality propositions using heterogeneous equality, they may still be needed in other forms of propositions. More importantly, explicit casts are useful gadgets for manipulating heterogeneous equality. Thus Heq provides simple notations for the cast operator and heterogeneous equality.

```
Notation "<< Sig # C >> x" := (eq_rect _ Sig x _ C) (at level 65).
Notation "{{ Sig # x == y }}" := (existT Sig _ x = existT Sig _ y) (at level 50).
```

As a first example, we prove the lemma `vapp_nil` by induction on the vector `v`.

```
Require Import Heq.

Definition Svec {A} (n:nat) : Type := vector A n.

Lemma vapp_nil : ∀ A n (v:vector A n), {{ Svec # v +++ vnil == v }}.
Proof. induction v. reflexivity. simpl. Hrewritec IHv. reflexivity. Qed.
```

The base case is done by `reflexivity` as both sides of the equality are `vnil`. The subgoal for induction step is as follows.

```
...
IHv : {{Svec # v +++ vnil == v}}
============================
{{Svec # hd ::: (v +++ vnil) == hd ::: v}}
```

To discharge this goal, one needs to rewrite `v +++ vnil` to `v` using the hypothesis `IHv`. However, the tactic `rewrite` does not apply because the types of the two terms are inconvertible. Instead we apply `Hrewritec IHv` and it successfully performs the rewriting. Though the tactic `dependent rewrite IHv` also works in this particular example, it fails in many other cases where `Hrewritec` succeeds.

We now examine how the tactic `Hrewritec` works. First, the equalities `Hty IHv : n = n + 0` and `v +++ vnil = <<Svec # Hty IHv>> v` are derived from the heterogeneous equality `IHv`. Then `v +++ vnil` is rewritten to `<<Svec # Hty IHv>> v` according to the latter equality and the equality proof `Hty IHv` is generalized.

```
============================
∀ C : n = n + 0, {{Svec # hd ::: (<< Svec # C >> v) == hd ::: v}}
```

Then rewrite `n+0` to `n` using a copy of `C`.

```
============================
∀ C : n = n, {{Svec # hd ::: (<< Svec # C >> v) == hd ::: v}}
```

Note that without the generalization of `Hty IHv` to `C`, the above rewriting will fail as `Hty IHv` cannot have type `n = n`. Finally, the term `C` is rewritten to the reflexivity proof `refl : n = n` by the uniqueness of identity proof (UIP), which is equivalent to Streicher's Axiom K [3] that is the only axiom Heq assumes. Then `<<Svec # refl>> v` simplifies to `v` by unfolding `eq_rect`.

As an exercise, we prove the associativity of `vapp`.

```
Lemma vapp_ass : ∀ A n (v:vector A n) n' (v':vector A n') n'' (v'':vector A n''),
  {{ Svec # (v +++ v') +++ v'' == v +++ v' +++ v'' }}.
Proof. intros. induction v. reflexivity. simpl. Hrewritec IHv. reflexivity. Qed.
```

5 Further examples

The vector reverse function `vrev` is an example where explicit coercion is unavoidable.

```
Program Fixpoint vrev A n (v:vector A n) : vector A n :=
  match v in vector _ n return vector A n with
    | vnil ⇒ vnil
    | vcons hd _ tl ⇒ << Svec # _ >> vrev tl +++ hd ::: vnil
  end.
Next Obligation. clear tl; induction Anonymous; simpl; auto. Defined.
```

See Appendix A for a lemma proving that `{{Svec # vrev (v +++ v') == vrev v' +++ vrev v}}`.

We give a more elaborate example. Consider the function `vplus` that adds two vectors of the same size componentwise. The functions `vhd` and `vtl` below are the usual notion of head and tail operations (see Appendix A for the definitions).

```
Fixpoint vplus n : vector nat n → vector nat n → vector nat n :=
  match n with
    | 0 ⇒ fun v v' ⇒ vnil
    | S n' ⇒ fun v v' ⇒ (vhd v + vhd v') ::: (vplus (vtl v) (vtl v'))
  end.
```

Now we consider the following goal.

```
Goal ∀ n (v: vector nat n) m (v': vector nat m) a b C0 C1,
  {{ Svec # vplus (a ::: (v +++ v')) (<<Svec # C0>> b ::: (v' +++ v)) ==
     a+b ::: vplus (v +++ v') (<<Svec # C1>> v' +++ v) }}.
Proof. intros. Hrefl (v'+++v : Svec _) at 1. Helimc → C0. Hunify. reflexivity. Qed.
```

In order to unfold `vplus` in the left hand side, the cast `<<Svec # C0>>` should be removed. However, if you just take it out, the resulting term would not type check. The solution is to move the cast elsewhere: rewriting `<< Svec # C0 >> b ::: (v' +++ v)` to `b ::: << Svec # C >> v' +++ v` for some `C : m + n = n + m`. Heq supports such an operation. First attach the reflexivity cast in front of the first occurrence of `v' +++ v` by `Hrefl (v'+++v : Svec _) at 1`.

```
...
============================
 ∀ (C1 : m + n = n + m) (C0 : S (m + n) = S (n + m)) (C : m + n = m + n),
 {{ Svec # vplus (a ::: (v +++ v')) (<< Svec # C0 >> b ::: (<< Svec # C >> v' +++ v)) ==
    a + b ::: vplus (v +++ v') (<< Svec # C1 >> v' +++ v) }}
```

Then eliminate the cast `C0` by `Helimc → C0`.

```
============================
 ∀ C0 C1 : m + n = n + m,
 {{ Svec # vplus (a ::: (v +++ v')) (b ::: (<< Svec # C1 >> v' +++ v)) ==
    a + b ::: vplus (v +++ v') (<< Svec # C0 >> v' +++ v) }}
```

Then unify the casts `C0` and `C1` of the same type by `Hunify` and discharge the goal by `reflexivity`.

6 Equality on iterated dependent pairs

Heq also deals with equality on iterated dependent pairs, by which we mean dependent pairs whose first components are also dependent pairs. Heterogeneous vectors are good examples.

```
Inductive vectorH : ∀ n, vector Set n → Type :=
  | vnilH : vectorH vnil
  | vconsH (T:Set) (hd:T) n (Ts:vector Set n) (tl:vectorH Ts) : vectorH (T ::: Ts).

Infix ":::::" := vconsH (at level 60, right associativity).
```

A heterogeneous vector is a vector that contains heterogeneous values, *i.e.*, elements of different sets. For instance, the term `3 :::: true :::: vnilH` has type `vectorH (nat ::: bool ::: vnil)`.

A heterogeneous vector `v : vectorH Ts` for `Ts : vector Set n` forms an iterated dependent pair `existT SvecH (existT Svec n Ts) v` for the signature `SvecH` defined as follows.

```
Definition SvecH (Ts : {n:nat & vector Set n}) := vectorH (projT2 Ts).
Notation  PvecH := ({{Svec # _,_}}) (only parsing).
```

Thus, equality between heterogeneous vectors can be represented as Heq with the signature `SvecH`. In order to make the Coq type checker happy, we need to define the pattern `PvecH` and use the following notations that come with the library Heq.

```
Notation "<< Sig , Pat # C >> x" := (eq_rect Pat Sig x Pat C) (at level 65, only parsing).
Notation "{{ Sig , Pat # x == y }}" := (existT Sig Pat x = existT Sig Pat y)
  (at level 50, only parsing).
```

The append operation for heterogeneous vectors `vappH` is defined similarly and denoted by the infix operator `++++` (see Appendix A for the definition). Then the following lemmas are proved exactly the same way as before.

```
Lemma vapp_nilH: ∀ n (Ts: vector _ n) (v:vectorH Ts), {{ SvecH,PvecH # v ++++ vnilH == v }}.
Proof. induction v. reflexivity. simpl. Hrewritec IHv. reflexivity. Qed.

Lemma vapp_assH : ∀ n (Ts:vector _ n) (v:vectorH Ts) n' (Ts':vector _ n') (v':vectorH Ts')
  n'' (Ts'':vector _ n'') (v'':vectorH Ts''),
  {{ SvecH,PvecH # (v ++++ v') ++++ v'' == v ++++ v' ++++ v'' }}.
Proof. intros. induction v. reflexivity. simpl. Hrewritec IHv. reflexivity. Qed.
```

We finally remark that Heq tactics are applied recursively as casts for iterated dependent pairs also form equalities on dependent pairs. However, the dependent equality `eq_dep` does not have this property. That is why Heq uses equality on dependent pairs rather than `eq_dep`.

7 Tactic hpattern

One of the key algorithms used in the library Heq is the new tactic `hpattern` that generalizes the existing tactic `pattern`. To see what it does, consider the following goal.

```
Goal ∀ (P: ∀ n m l, vector nat n → vector nat m → Prop) n m,
  S n = m → ∀ (v : vector nat (S n)) (w: vector nat n),
  P (S n) (S n) (S n) v (0:::w).
Proof. intros P n m H. hpattern (S n). rewrite H. Abort.
```

After `intros P n m H`, we have the following.

```
H : S n = m
============================
 ∀ (v : vector nat (S n)) (w : vector nat n), P (S n) (S n) (S n) v (0 ::: w)
```

Now we try to rewrite `S n` to `m`. The tactic `rewrite H` or `pattern (S n)` will fail because the term `S n` cannot be abstracted. However, if you try to abstract only the first and second occurrences of `S n` by `pattern (S n) at 1 2`, then it will succeed. Finding such occurrences is what `hpattern` does for us. By `hpattern (S n); rewrite H`, we get the following.

```
============================
 ∀ (v : vector nat m) (w : vector nat n), P m (S n) (S n) v (0 ::: w)
```

The tactic `hpattern t at k` finds the minimal set of occurrences of the term `t` including the k-th occurrence such that those occurrences can be abstracted, whenever such a set exists. Moreover, the algorithm is linear in the number of occurrences of `t`. When applied without the argument `k`, it tries `hpattern t at 1`, `hpattern t at 2`, ... until it succeeds or reaches the last occurrence.

The algorithm is simple. We see how `hpattern (S n) at 1` works in the above example. First it replaces the first occurrence of `S n` with a fresh abstract variable `k : nat` and the all other occurrences of `S n` with a hole `_` to get the following term.

```
∀ (v : vector nat k) (w : vector nat n), P _ _ _ v (0 ::: w)
```

Then, apply the Coq type checker to this incomplete term. The type checker will not fail to infer the first hole because it is connected to the position where the abstract variable `k` is located and thus the unification will find the unique instance `k` due to the completeness of first-order unification. However, the type checker may try to infer the second and third holes first and may fail. Assume that the type checker failed saying that it cannot infer the second hole. Then we replace the second hole with the original term `S n` and apply the type checker again. It may fail to infer the third hole as well. Then we do the same job again. Then finally the only uninstantiated hole is the first one, which will be inferred correctly as discussed above.

```
∀ (v : vector nat k) (w : vector nat n), P k (S n) (S n) v (0 ::: w)
```

The collection of all occurrences of the abstract variable `k` is the solution we seek.

8 Conclusion

We have demonstrated how Heq deals with heterogeneous equality and explicit coercion. To show scalability of Heq, we have given more elaborate examples, which are available at [4]. First, we mechanically converted the Coq standard library for list (about 2000 lines) into a library for vector in a few hours without understanding the proofs. We just changed the syntax appropriately and added Heq tactics to manipulate casts. In other words, the definitions and proof structures remained the same during this transformation. As another example, we have simplified the strongly-typed formalization of system F [1] using the library Heq, and formalized the proof of strong normalization of system F given in [2].

In personal communication with Matthieu Sozeau, he has shown that the signature such as `Svec` can be inferred using the type classes in Coq [6], so that one can write `{{ x == y }}` instead of `{{ Sig # x == y }}`. Also, we expect that the pattern such as `PvecH` will be unnecessary in a later version of Coq, as it is not difficult to improve the type checker to infer such patterns.

Acknowledgements. We are grateful to Hugo Herbelin for useful discussions.

References

1. Nick Benton, Chung-Kil Hur, Andrew Kennedy, and Conor McBride. Strongly typed term representations in Coq. Submitted, 2009.
2. Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1989.
3. Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Proceedings of the meeting Twenty-five years of constructive type theory*. Oxford University Press, 1998.
4. Chung-Kil Hur. Heq. `http://www.pps.jussieu.fr/~gil/Heq/`, 2010.

5. Conor McBride. Elimination with a motive. In *Proceedings of the International Workshop on Types for Proofs and Programs (TYPES'00)*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer-Verlag, 2002.
6. Matthieu Sozeau and Nicolas Oury. First-class type classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 278–293. Springer-Verlag, 2008.

A Complete Coq script of the examples

```
Require Import Program.
Set Implicit Arguments.

(**
   Section 2. Motivating example
*)

Inductive vector (A:Type) : nat → Type :=
| vnil : vector A 0
| vcons (hd:A) n (tl:vector A n): vector A (S n).
Implicit Arguments vnil [A].

Infix ":::" := vcons (at level 60, right associativity).

Fixpoint vapp A n (v: vector A n) n' (v':vector A n') : vector A (n+n') :=
  match v with
    | vnil ⇒ v'
    | vcons hd _ tl ⇒ hd ::: vapp tl v'
  end.

Infix "+++" := vapp (right associativity, at level 61).

(** The following does not type check.

  Lemma vapp_nil_eq : ∀ A n (v:vector A n), v +++ vnil = v.
*)

Lemma vapp_nil_cast : ∀ A n (v:vector A n), v +++ vnil = eq_rect _ (vector A) v _ (plus_n_O n).
Abort.

(**
   Section 3. Heterogeneous equality
*)

Inductive JMeq (A : Type) (x : A) : ∀ B : Type, B → Prop :=
  JMeq_refl : JMeq x x.

Lemma vapp_nil_jmeq : ∀ A n (v:vector A n), JMeq (v +++ vnil) v.
Abort.

Inductive eq_dep (U : Type) (Sig : U → Type) (p : U) (x : Sig p) : ∀ q : U, Sig q → Prop :=
  eq_dep_intro : eq_dep Sig p x p x.
Implicit Arguments eq_dep [U p q].

Lemma vapp_nil_eqdep : ∀ A n (v:vector A n), eq_dep (vector A) (v +++ vnil) v.
Abort.
```

```
(**
   Section 4. Library Heq
*)

Require Import Heq.

Definition Svec {A} (n:nat) : Type := vector A n.

Lemma vapp_nil : ∀ A n (v:vector A n), {{ Svec # v +++ vnil == v }}.
Proof. induction v. reflexivity. simpl. Hrewritec IHv. reflexivity. Qed.

Lemma vapp_ass : ∀ A n (v:vector A n) n' (v':vector A n') n'' (v'':vector A n''),
  {{ Svec # (v +++ v') +++ v'' == v +++ v' +++ v'' }}.
Proof. intros. induction v. reflexivity. simpl. Hrewritec IHv. reflexivity. Qed.

(**
   Section 5. Further examples
*)

Program Fixpoint vrev A n (v:vector A n) : vector A n :=
  match v in vector _ n return vector A n with
    | vnil ⇒ vnil
    | vcons hd _ tl ⇒ << Svec # _ >> vrev tl +++ hd ::: vnil
  end.
Next Obligation. clear tl; induction Anonymous; simpl; auto. Defined.

Lemma vdistr_rev : ∀ A n (v:vector A n) m (v':vector A m),
  {{ Svec # vrev (v +++ v') == vrev v' +++ vrev v }}.
Proof.
intros; induction v as [| a n v IHv].
  destruct v'. reflexivity. simpl. rewrite (Hrw (vapp_nil _)); Hsimpl. reflexivity.
  simpl. Hsimplc. Hrewritec IHv. rewrite (Hrw (vapp_ass _ _ _)); Hsimpl. reflexivity.
Qed.

Program Definition vhd A n (v:vector A (S n)) : A :=
  match v with | vnil ⇒ _ | vcons hd _ _ ⇒ hd end.

Program Definition vtl A n (v:vector A (S n)) : vector A n :=
  match v with | vnil ⇒ _ | vcons _ _ tl ⇒ tl end.

Fixpoint vplus n : vector nat n → vector nat n → vector nat n :=
  match n with
    | 0 ⇒ fun v v' ⇒ vnil
    | S n' ⇒ fun v v' ⇒ (vhd v + vhd v') ::: (vplus (vtl v) (vtl v'))
  end.

Goal ∀ n (v: vector nat n) m (v': vector nat m) a b C0 C1,
  {{ Svec # vplus (a ::: (v +++ v')) (<<Svec # C0>> b ::: (v' +++ v)) ==
     a+b ::: vplus (v +++ v') (<<Svec # C1>> v' +++ v) }}.
Proof. intros. Hrefl (v'+++v : Svec _) at 1. Helimc → C0. Hunify. reflexivity. Qed.

(**
   Section 6. Equality on iterated dependent pairs
*)

Inductive vectorH : ∀ n, vector Set n → Type :=
  | vnilH : vectorH vnil
```

```
  | vconsH (T:Set) (hd:T) n (Ts:vector Set n) (tl:vectorH Ts) : vectorH (T ::: Ts).

Infix "::::" := vconsH (at level 60, right associativity).

Check (3 :::: true :::: vnilH).

Definition SvecH (Ts : {n:nat & vector Set n}) := vectorH (projT2 Ts).
Notation  PvecH := ({{Svec # _,_}}) (only parsing).

Fixpoint vappH n (Ts: vector Set n) (v: vectorH Ts)
  n' (Ts': vector Set n') (v':vectorH Ts') : vectorH (Ts+++Ts') :=
  match v in vectorH Ts return vectorH (Ts+++Ts') with
    | vnilH ⇒ v'
    | vconsH _ hd _ _ tl ⇒ hd :::: vappH tl v'
  end.

Infix "++++" := vappH (right associativity, at level 61).

Lemma vapp_nilH: ∀ n (Ts: vector _ n) (v:vectorH Ts),
  {{ SvecH,PvecH # v ++++ vnilH == v }}.
Proof. induction v. reflexivity. simpl. Hrewritec IHv. reflexivity. Qed.

Lemma vapp_assH : ∀ n (Ts:vector _ n) (v:vectorH Ts)
  n' (Ts':vector _ n') (v':vectorH Ts') n'' (Ts'':vector _ n'') (v'':vectorH Ts''),
  {{ SvecH,PvecH # (v ++++ v') ++++ v'' == v ++++ v' ++++ v'' }}.
Proof. intros. induction v. reflexivity. simpl. Hrewritec IHv. reflexivity. Qed.

(**
   Section 7. Tactic hpattern
*)

Goal ∀ (P: ∀ n m l, vector nat n → vector nat m → Prop) n m,
  S n = m → ∀ (v : vector nat (S n)) (w: vector nat n),
  P (S n) (S n) (S n) v (0:::w).
Proof. intros P n m H. hpattern (S n). rewrite H. Abort.
```