Custom definitional equalities in Agda

Guillaume Brunerie

Université de Nice/Institute for Advanced Study

July 14, 2015
ICMS 2016, Berlin

Definitional equalities in Agda

The collection of definitional equalities in Agda is fixed:

- application of a lambda-abstraction
- pattern matching on a constructor
- projection on a record constructor
- η-conversion

But in HoTT we want new definitional equalities, in particular inspired by cubical type theory.

We present here rewriting, a new experimental feature of Agda developed by Jesper Cockx, which allows the user to add new reduction rules.

Rewriting in Agda

We enable rewriting:

```
{-# OPTIONS --rewriting #-}
```

We declare the rewriting relation:

postulate $\_\mapsto\_ : \forall \{i\} \{A : \mathsf{Type}\ i\} \to A \to A \to \mathsf{Type}\ i$
```
{-# BUILTIN REWRITE _↦_ #-}
```

We declare rewrite rules:

postulate rew : (arguments) $\to$ (lhs $\mapsto$ rhs)
```
{-# REWRITE rew #-}
```

The left-hand side must be a symbol applied to some arguments and all the arguments of rew must be bound in lhs.

The circle with definitional reduction rule for base

```
postulate
  Circle : Type₀
  base : Circle
  loop : base == base

module _ {i} {P : Circle → Type i}
  (base* : P base)
  (loop* : PathOver P loop base* base*) where
  postulate
    Circle-elim : (x : Circle) → P x
    Circle-base-β : Circle-elim base ↦ base*
    {-# REWRITE Circle-base-β #-}
    Circle-loop-β : apd Circle-elim loop == loop*
```

+ recursor (non-dependent elimination rule)

Cubical type theory

- We get a much cleaner implementation of higher inductive types

- What if we try to add even more reduction rules to try to simulate cubical type theory?
- We will prove that the torus is equivalent to the product of two circles, code available at `https://github.com/guillaumebrunerie/TorusRewriting`

Examples

```
postulate
  _==_ : ∀ {i} {A : Type i} → A → A → Type i
  idp : ∀ {i} {A : Type i} {a : A} → a == a

  PathOver : ∀ {i j} {A : Type i} (B : A → Type j)
    {x y : A} (p : x == y) (u : B x) (v : B y) → Type j
  PathOver-cst : ∀ {i j} {A : Type i} {B : Type j} {x y : A} (p : x == y)
    (u v : B) → (PathOver (λ _ → B) p u v) ↦ (u == v)
  {-# REWRITE PathOver-cst #-}

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

  PathOver↦idp : ∀ {i j} {A : Type i} (B : A → Type j) {x : A} (u v : B x)
    → PathOver B idp u v ↦ (u == v)
  {-# REWRITE PathOver↦idp #-}
  ap↦idp : ∀ {i j} {A : Type i} {B : A → Type j} (f : (a : A) → B a) {x : A}
    → ap f (idp {a = x}) ↦ idp
  {-# REWRITE ap↦idp #-}
```

Ap of a composition and ap-cur

For $f : A \to B$ and $g : B \to C$, we have the rewrite rule:

```
postulate
  ∘ap : {x y : A} (p : x == y) →
    ap g (ap f p) ↦ ap (λ x → g (f x)) p
{-# REWRITE ∘ap #-}
```

But sometimes we need it in the other direction...
A (non-ideal) solution is to define all reverse instances whenever we need them.

A similar phenomenon happens for `ap-cur`: applying an uncurrified function to a pair of paths is the same as applying the currified function successively.

The torus

```
postulate
  Torus : Type₀
  baseT : Torus
  loopT1 : baseT == baseT
  loopT2 : baseT == baseT
  surfT : Square loopT1 loopT1 loopT2 loopT2
```

The torus

```
module _ {i} {P : Torus → Type i}
    (baseT* : P baseT)
    (loopT1* : PathOver P loopT1 baseT* baseT*)
    (loopT2* : PathOver P loopT2 baseT* baseT*)
    (surfT* : SquareOver P surfT
      loopT1* loopT1* loopT2* loopT2*) where
  postulate
    Torus-elim : (x : Torus) → P x
    Torus-baseT-β : Torus-elim baseT ↦ baseT*
    {-# REWRITE Torus-baseT-β #-}
    Torus-loopT1-β : ap Torus-elim loopT1 ↦ loopT1*
    {-# REWRITE Torus-loopT1-β #-}
    Torus-loopT2-β : ap Torus-elim loopT2 ↦ loopT2*
    {-# REWRITE Torus-loopT2-β #-}
    Torus-surfT-β : ap□ Torus-elim surfT ↦ surfT*
    {-# REWRITE Torus-surfT-β #-}
```

The two maps

```
{- Map from the torus to the product of two circles -}
to : Torus → Circle × Circle
to = Torus-rec (base , base) (loop ,= idp) (idp ,= loop) (idh ,'□ idv)

{- Map from the product of two circles to the torus -}
from : Circle × Circle → Torus
from (u , v) = from-curry u v where

  from-curry : Circle → Circle → Torus
  from-curry = Circle-rec loopT2-map (funext from-aux) where

    loopT2-map : Circle → Torus
    loopT2-map = Circle-rec baseT loopT2

    from-aux : (x : Circle) → loopT2-map x == loopT2-map x
    from-aux = Circle-elim loopT1
      (↓-='-in loopT2-map loopT2-map loop surfT)
```

Rewrite rules for the first composite

rew1 : ap from (ap to loopT1) ↦ loopT1
rew1 = ap-cur' from loop (idp :> base == base)
```
{-# REWRITE rew1 #-}
```

rew2 : ap from (ap to loopT2) ↦ loopT2
rew2 = ap-cur' from (idp :> base == base) loop
```
{-# REWRITE rew2 #-}
```

rew3 : ap□ from (ap□ to surfT) ↦ surfT
rew3 = ap□-cur' from {l = loop} (idh {p = loop}) (idv {p = loop})
```
{-# REWRITE rew3 #-}
```

loopT1-β : ap (λ z → from (to z)) loopT1 ↦ loopT1
loopT1-β = ap∘ from to loopT1
```
{-# REWRITE loopT1-β #-}
```

loopT2-β : ap (λ z → from (to z)) loopT2 ↦ loopT2
loopT2-β = ap∘ from to loopT2
```
{-# REWRITE loopT2-β #-}
```

surfT-β : ap□ (λ z → from (to z)) surfT ↦ surfT
surfT-β = ap□∘ from to surfT
```
{-# REWRITE surfT-β #-}
```

The first composite

```
from-to : (x : Torus) → from (to x) == x
from-to = Torus-elim
  idp
  (↓-='-in (from ∘ to) (λ x → x) loopT1 idv)
  (↓-='-in (from ∘ to) (λ x → x) loopT2 idv)
  (↓-□'-in (from ∘ to) (λ x → x) surfT idhc)
```

Rewriting rules for the second composite

```
red1 : ap (λ z → to (from-curry base z)) loop ↦ (idp ,= loop)
red1 = ap∘ to (from-curry base) loop
{-# REWRITE red1 #-}

module _ (z : Circle) where
  red2 : ap (λ x → from-curry x z) loop ↦ from-aux z
  red2 = ap∘ (λ h → h z) from-curry loop
  {-# REWRITE red2 #-}

  red3 : ap (λ x → to (from-curry x z)) loop ↦ ap to (from-aux z)
  red3 = ap∘ to (λ x → from-curry x z) loop
  {-# REWRITE red3 #-}

red4 : ap (λ z → ap (λ x → to (from-curry x z)) loop) loop ↦ _
red4 = ap↓∘ (ap to) from-aux loop
{-# REWRITE red4 #-}

red5 : ap↓ (λ {a} → ap to {loopT2-map a} {loopT2-map a})
         {p = loop} (↓-==-eq-in idp surfT) ↦ _
red5 = ap↓-ap-↓-='-in to {p = loop} surfT
{-# REWRITE red5 #-}
```

The second composite

```
to-from : (u : Circle × Circle) → to (from u) == u
to-from (x , y) = to-from-curry y x where

  to-from-curry : (y x : Circle) → to (from-curry x y) == (x , y)
  to-from-curry y =
    Circle-elim (to-from-curry-base y)
      (↓-='-in [...] (to-from-curry-loop y))  where

    to-from-curry-base : (y : Circle) → [...]
    to-from-curry-base = Circle-elim idp (↓-='-in [...] idv)

    to-from-curry-loop : (y : Circle) → [...]
    to-from-curry-loop = Circle-elim idv (↓-□='-in [...] idhc)
```

Conclusion

- Gives a cleaner implementation of higher inductive types definitional-for-point-constructors
- Can be used to simulate cubical type theory, but is not a replacement of it
- Easy to mess things up, to make Agda loop or to break basic features of type theory