Towards quotient inductive-inductive-recursive types

Ambrus Kaposi

Eötvös Loránd University, Budapest

TYPES
Valencia
13 June 2023

Contents

Induction-recursion

```
data A : Set
f : A → B
```

Induction-recursion

```
data U : Set
El : U → Set

data U where
  bool : U
  pi   : (a : U) →
         (El a → U) → U

El bool      = Bool
El (pi a b) = (x : El a) →
               El (b x)
```

Reducing induction-recursion

```
data A : Set
f : A → B
```

```
data A* : B → Set
```

Reducing induction-recursion

```
data U : Set
El : U → Set

data U where
  bool : U
  pi   : (a : U) →
    (El a → U) → U

El bool     = Bool
El (pi a b) =
  (x : El a) → El (b x)
```

```
data U* : Set → Set₁ where
  bool* : U* Bool
  pi*   :
    (a* : U* A) →
    ((x : A) → U* (B x)) →
    U* ((x : A) → B x)
```

Something we cannot reduce

```
data A : Set
f : A → B
```

```
data A* : B → Set
```

Something we cannot reduce

```
data A : Set                          data A* : B → Set
f : A → B


data A : Set                          data A* : A* → Set ????
f : A → A
```

An example of a weird IR type

```
Ty  : Set
Con : Set


data Tm  : Con → Ty → Set
data Sub : Con → Con → Set
_[_] : Tm Γ A → Sub Δ Γ → Tm Δ A
```

An example of a weird IR type

```
data Tm  : Con → Ty → Set
  lam    : Tm (Γ , A) B → Tm Γ (A ⇒ B)
  app    : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B


data Sub : Con → Con → Set
  ⟨_⟩    : Tm Γ A → Sub Γ (Γ , A)
  _⁺     : Sub Δ Γ → Sub (Δ ⁺) (Γ ⁺)

_[_] : Tm Γ A → Sub Δ Γ → Tm Δ A
lam t   [ σ ] := lam (t [ σ ⁺])
app t u [ σ ] := app (t [ σ ]) (u [ σ ])
```

An example of a weird QIR type

```
data Tm  : Con → Ty → Set
  lam    : Tm (Γ , A) B → Tm Γ (A ⇒ B)
  app    : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B
  β      : app (lam t) u ≡ t [ ⟨ u ⟩ ]

data Sub : Con → Con → Set
  ⟨_⟩    : Tm Γ A → Sub Γ (Γ , A)
  _⁺     : Sub Δ Γ → Sub (Δ ⁺) (Γ ⁺)

_[_] : Tm Γ A → Sub Δ Γ → Tm Δ A
lam t   [ σ ] := lam (t [ σ ⁺])
app t u [ σ ] := app (t [ σ ]) (u [ σ ])
β       [ σ ] : app (lam t) u [ σ ] ≡
                t [ ⟨ u ⟩ ] [ σ ]
```

What is a QIIRT?

- Quotients: equality constructors
- Inductive-inductive (very mutual)
- Some operations defined recursively

What is a QIIRT?

- Quotients: equality constructors
- Inductive-inductive (very mutual)
- Some operations defined recursively

What is the elimination principle?

What is a QIIRT?

- Quotients: equality constructors
- Inductive-inductive (very mutual)
- Some operations defined recursively

What is the elimination principle?

What is an algebra?

What is a QIIRT? A partial answer

This is the QIIRT:

```
data Tm  : Con → Ty → Set
  lam    : Tm (Γ , A) B → Tm Γ (A ⇒ B)
  app    : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B
  β      : app (lam t) u ≡ t [ ⟨ u ⟩ ]

data Sub : Con → Con → Set
  ⟨_⟩    : Tm Γ A → Sub Γ (Γ , A)
  _⁺     : Sub Δ Γ → Sub (Δ ⁺) (Γ ⁺)

_[_] : Tm Γ A → Sub Δ Γ → Tm Δ A
lam t   [ σ ] := lam (t [ σ ⁺])
app t u [ σ ] := app (t [ σ ]) (u [ σ ])
```

What is a QIIRT? A partial answer

There is a corresponding QIIT.

```
data Tm  : Con → Ty → Set
  lam    : Tm (Γ , A) B → Tm Γ (A ⇒ B)
  app    : Tm Γ (A ⇒ B) → Tm Γ A → Tm Γ B
  β      : app (lam t) u ≡ t [ ⟨ u ⟩ ]
  _[_]   : Tm Γ A → Sub Δ Γ → Tm Δ A
  lam[]  : lam t   [ σ ] ≡ lam (t [ σ ⁺ ])
  app[]  : app t u [ σ ] ≡ app (t [ σ ]) (u [ σ ])

data Sub : Con → Con → Set
  ⟨_⟩    : Tm Γ A → Sub Γ (Γ , A)
  _⁺     : Sub Δ Γ → Sub (Δ ⁺) (Γ ⁺)
```

What is a QIIRT? A partial answer

We define a new substitution operation.

```
data Tm  : Con → Ty → Set
  _[_]   : Tm Γ A → Sub Δ Γ → Tm Δ A

data Sub : Con → Con → Set

_[_]*   : Tm Γ A → Sub Δ Γ → Tm Δ A
correct : t [ σ ]* ≡ t [ σ ]
```

What is a QIIRT? A partial answer

We define a new syntax.

```
data Tm  : Con → Ty → Set
  _[_]   : Tm Γ A → Sub Δ Γ → Tm Δ A

data Sub : Con → Con → Set

_[_]*   : Tm Γ A → Sub Δ Γ → Tm Δ A
correct : t [ σ ]* ≡ t [ σ ]

Tm* : Con → Ty → Set
Tm* := Tm

lam*    := lam
app*    := app
β*      := β and correct
app[]* := refl
```

What is a QIIRT? A partial answer

We prove the elimination principle for the new syntax.

```
data Tm  : Con → Ty → Set
  _[_]   : Tm Γ A → Sub Δ Γ → Tm Δ A

_[_]*   : Tm Γ A → Sub Δ Γ → Tm Δ A
correct : t [ σ ]* ≡ t [ σ ]

Tm* : Con → Ty → Set

newElim : (P    : Tm* Γ A → Set)
          (lamP : P t → P (lam* t))
          (appP : P t → P u → P (app* t u))
          ...
          (t : Tm* Γ A) → P t
```

What is a QIIRT? A partial answer

We prove the elimination principle for the new syntax.

```
data Tm  : Con → Ty → Set
  _[_]   : Tm Γ A → Sub Δ Γ → Tm Δ A

_[_]*   : Tm Γ A → Sub Δ Γ → Tm Δ A
correct : t [ σ ]* ≡ t [ σ ]

Tm* : Con → Ty → Set

newElim : (P    : Tm* Γ A → Set)
          (lamP : P t → P (lam* t))
          (appP : P t → P u → P (app* t u))
          ...
          (t : Tm* Γ A) → P t
```

Why would you do this?

Why would you do this?

```
oldElim :
  (P      : Tm Γ A → Set)
  (lamP   : P t → P (lam t))
  (appP   : P t → P u → P (app t u))
  (_[_]P  : P t → P σ → P (t [ σ ]))
  (appP[] :
    ((appP tP uP) [ σP ]P)
    ≡
    appP (tP[σP]P) (uP[σP]P))
  ...
  (t : Tm Γ A) → P t
```

Why would you do this?

```
oldElim :
  (P      : Tm Γ A → Set)
  (lamP   : P t → P (lam t))
  (appP   : P t → P u → P (app t u))
  (_[_]P  : P t → P σ → P (t [ σ ]))
  (appP[] :
    ((appP tP uP) [ σP ]P)     : P(app t u[σ])
    ≡
    appP (tP[σP]P) (uP[σP]P)) : P(app(t[σ])(u[σ]))
  ...
  (t : Tm Γ A) → P t
```

Why would you do this?

```
oldElim :
  (P       : Tm Γ A → Set)
  (lamP    : P t → P (lam t))
  (appP    : P t → P u → P (app t u))
  (_[_]P   : P t → P σ → P (t [ σ ]))
  (appP[] : subst P app[]
    ((appP tP uP) [ σP ]P)
    ≡
    appP (tP[σP]P) (uP[σP]P))
  ...
  (t : Tm Γ A) → P t
```

Why would you do this?

```
newElim :
  (P       : Tm* Γ A → Set)
  (lamP    : P t → P (lam* t))
  (appP    : P t → P u → P (app* t u))
  (_[_]P   : P t → P σ → P (t [ σ ]*))
  (appP[] :
    ((appP tP uP) [ σP ]P)
    ≡
    appP (tP[σP]P) (uP[σP]P))
  ...
  (t : Tm* Γ A) → P t
```

Strictification of QIITs

Recipe:

1. Take a QIIT.
2. Redefine some constructors recursively and show that they are equal to the original ones.
3. Define a new algebra using old constructors and the new recursive functions.
4. Show that the new algebra has an induction principle.

Decreasing transport hell

Canonicity for simple type theory:

- 190 lines for the weak syntax
- 160 lines for the strict syntax

Decreasing transport hell

Canonicity for simple type theory:

- 190 lines for the weak syntax
- 160 lines for the strict syntax
- 44 lines for the syntax with equality reflection

Decreasing transport hell

Canonicity for simple type theory:

- 190 lines for the weak syntax
- 160 lines for the strict syntax
- 44 lines for the syntax with equality reflection

Can we strictify more equations?

```
β       : app (lam t) u ≡ t [ ⟨ u ⟩ ]
[∘]     : t [ σ ∘ ρ ] ≡ t [ σ ] [ ρ ]
```

Application to the syntax of type theory

Spectrum:

1. Untyped preterms, typing relation, conversion relation
   (Abel–Öhman–Vezzosi POPL 2018)
2. Categories with families (CwF)
3. Locally cartesian closed categories

Application to the syntax of type theory

Spectrum:

1. Untyped preterms, typing relation, conversion relation
   (Abel–Öhman–Vezzosi POPL 2018)
2. Categories with families (CwF)
3. Locally cartesian closed categories

Algebraic view:

- A language is a generalised algebraic theory (GAT)
- A model is an algebra of the GAT
- The syntax is the initial model definable as a QIIT

Application to the syntax of type theory

Spectrum:

1. Untyped preterms, typing relation, conversion relation
   (Abel–Öhman–Vezzosi POPL 2018)
2. Categories with families (CwF)
3. Locally cartesian closed categories

Algebraic view:

- A language is a generalised algebraic theory (GAT)
- A model is an algebra of the GAT
- The syntax is the initial model definable as a QIIT
- Sometimes (parts of) the syntax are definable, e.g.
  - Substitution normal forms
  - First order logic as a Cw2F
  - Simple type theory using hereditary substitutions

Second-order GATs

Lambda calculus as a SOGAT:

```
Tm  : Set⁺
lam : (Tm → Tm) → Tm
app : Tm → Tm → Tm
β   : app (lam t) u = t u
```

Second-order GATs

Lambda calculus as a SOGAT:

```
Tm  : Set⁺
lam : (Tm → Tm) → Tm
app : Tm → Tm → Tm
β   : app (lam t) u = t u
```

Conjecture:
For any SOGAT, there is an initial first order model where all substitution laws are definitional.