Internalizing Representation Independence with Univalence: Background & Notation

Bram Pellen & Aron van Hof

Institute for Computing and Information Sciences
Radboud University Nijmegen

20th December 2022

Outline

Introduction

Equalities as paths

Univalence

Higher inductive types

Introduction

- Representation independence in programming languages: display certain behaviour regardless of implementation
- Example: Binary & unary numbers for proof reuse
- Problem: current results within dependently-typed languages are either meta-theoretic (rather than internal) or restricted to isomorphisms

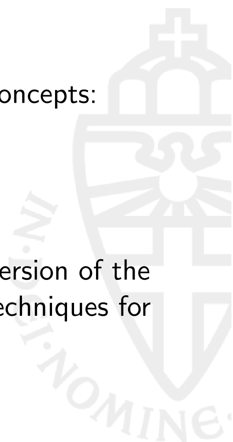
Example: Queues (1)

- Consider a queue interface with enqueue and dequeue functions
- A simple implementation may use a single list
- For greater performance use a batched queue, a pair of two lists
- You can then define a function AppendReverse: AppendReverse :: BatchedQueue A $\rightarrow$ ListQueue A
- Note that this function is *structure-preserving*: it commutes with enqueue and dequeue

Example: Queues (2)

- By representation independence, this means the implementations are contextually equivalent
- One can try to also obtain this result internally through univalence, specifically through the *Structure Identity Principle* (SIP)
- However, this principle relies on AppendReverse being an isomorphism
- Unfortunately, it is not, as it e.g. sends both ([], [1, 0]) and ([0], [1]) to [0, 1]

Outline



In the rest or our presentation we will review several concepts:

- Paths
- Univalence
- Higher inductive types

The rest of the paper uses these concepts to state a version of the SIP and how to apply this principle, thereby gaining techniques for establishing internal representation independence.

Notions of Equality

- Most languages with full-spectrum dependent types, such as:
  - Agda
  - Coq
  - Idris
  - Lean

  support two notions of equality:
  - Definitional equality
  - Equality types

Definitional equality and Equality types

- *Definitional (Judgmental) equality* is a notion of equality that can use computation rules such as β-reduction to determine equality. For example:

Example and counterexample of definitional equality

$1 + 2 = 2 + 1$ is a definitional equality, since $1 + 2$ and $2 + 1$ are both definitionally equal to 3. However, $\forall n \in \mathbb{N}$, $1 + n = n + 1$ is *not* a definitional equality.

Definitional equality and Equality types

- *Definitional (Judgmental) equality* is a notion of equality that can use computation rules such as β-reduction to determine equality. For example:

Example and counterexample of definitional equality

$1 + 2 = 2 + 1$ is a definitional equality, since $1 + 2$ and $2 + 1$ are both definitionally equal to 3. However, $\forall n \in \mathbb{N}$, $1 + n = n + 1$ is *not* a definitional equality.

- *Equality types (propositional equality)* can be used to prove equalities that computation rules such as β-reduction cannot deal with on their own. For example:

Example equality that can be proven with equality types

$\forall n \in \mathbb{N}$, $1 + n = n + 1$ *can* be proven with equality types.

Using Equality types

- Equality types lack several mathematical properties of equality, among which:
    - Function extensionality.
    - Propositional extensionality.
    - The ability to define quotients.
- Principles such as function extensionality, univalence and the law of excluded middle can be added via axioms.
- But axioms in type theory lack computational content. Axioms can be thought of as function types without corresponding definitions, causing proofs that use them to become "stuck".

Using Paths for equalities

- Using paths for equalities, we can obtain principles of equalities such as:
  - Function extensionality.
  - Propositional extensionality.
  - The ability to define quotients.

  in a way that does have computational content.

Intervals

- In Cubical Type Theory, paths are defined as maps out of the interval I.
- The elements $\mathsf{i0} : \mathsf{I}$ and $\mathsf{i1} : \mathsf{I}$ of I are thus used to specify the endpoints of paths.
- Since paths are used for equalities, maps $\mathsf{f} : \mathsf{I} \to \mathsf{A}$ serve as "evidence" that $\mathsf{f}(\mathsf{i0})$ and $\mathsf{f}(\mathsf{i1})$ are equal in A.
- The interval I has three operations:
  - minimum $(_\wedge_ : \mathsf{I} \to \mathsf{I} \to \mathsf{I})$
  - maximum $(_\vee_ : \mathsf{I} \to \mathsf{I} \to \mathsf{I})$
  - reversal $(\sim_ : \mathsf{I} \to \mathsf{I})$

  which satisfy the laws of a De Morgan Algebra, i.e. $(\mathsf{i0}, \mathsf{i1}, _\wedge_, _\vee_)$ is a bounded distributive lattice with a De Morgan involution $\sim_$.

PathP types

- PathP types are defined as:

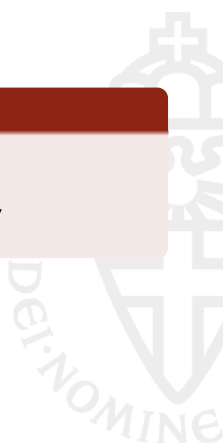
PathP types

```
PathP : (A : I → Type ℓ) → A i0 → A i1 → Type ℓ
```

- PathP types are a special form of dependent function type (i : I) → A i that specify the output of their elements (i.e. functions (i : I) → A i) for i0 and i1.
- For a path p : PathP A a0 a1, we have the definitional equalities p i0 = a0, and p i1 = a1, where a0 : A i0 and a1 : A i1.
- PathP types represent heterogeneous equalities, since their "endpoints" have types A i0 and A i1, which can be different.
- Note that Set is now renamed to Type, to avoid confusion with homotopy sets.

Homogeneous paths

- PathP types can be used to obtain homogeneous (non-dependent) paths:



Homogeneous (non-dependent) path equality

```
_≡_ :  {A:  Type  ℓ} → A → A → Type  ℓ
_≡_  {A = A}  x  y = PathP  (λ  \_ → A)  x  y
```

Homogeneous paths

- PathP types can be used to obtain homogeneous (non-dependent) paths:

Homogeneous (non-dependent) path equality

```
_≡_: {A: Type ℓ} → A → A → Type ℓ
_≡_ {A = A} x y = PathP (λ \_ → A) x y
```

- Homogeneous paths can be used to obtain reflexivity:

Reflexivity

```
refl: {x: A} → x ≡ x
refl {x = x} = λ \_ → x
```

Function extensionality

- Homogeneous paths can be used to obtain function extensionality:

Function extensionality

```
funExt : {f g: A → B} → ((x: A) → f x ≡ g x) →
    f ≡ g
funExt p i x = p x i
```

- Since function extensionality is defined, we get computational content for it, which axiomatic versions of function extensionality do not have.

Congruence

- Homogeneous paths can be used to define congruence.
- Congruence in this context means that if for elements x y:A, x ≡ y, then, for all dependent functions f: (a:A) → B a, f(x) ≡ f(y).
- Congruence can be defined as follows:

Congruence

```
cong: {B: A → Type} (f: (a: A) → B a) {x y: A}
    (p: x ≡ y)
   → PathP (λ i → B (p i)) (f x) (f y)
cong f p i = f (p i)
```

Transport and Path induction

- Homogeneous paths can be used to define transport:

Transport

```
transport: A ≡ B → A → B
transport p a = transp (λ i → p i) i0 a
```

Transport and Path induction

- Homogeneous paths can be used to define transport:

Transport

```
transport : A ≡ B → A → B
transport p a = transp (λ i → p i) i0 a
```

- Transport and homogeneous paths can be used to define path induction:

Path induction

```
J : {x : A} (P : ∀ y → x ≡ y → Type)
    (d : P x refl) {y : A} (p : x ≡ y) → P y p
J P d p = transport
    (λ i → P (p i) (λ j → p (i ∧ j))) d
```

Path induction (1/2)

Path induction

```
J: {x: A} (P: ∀ y → x ≡ y → Type)
     (d: P x refl)
     {y: A} (p: x ≡ y) → P y p
J P d p = transport (λ i → P (p i)
     (λ j → p (i ∧ j))) d
```

Path induction modified to resemble the HoTT book

```
J' : {A : Type ℓ} (P : ∀ x y → x ≡ y → Type ℓ)
     (d : (x : A) → P x x refl)
     {x : A} {y : A} (p : x ≡ y) → P x y p
J' P d p = transport (λ i → P (p i0) (p i)
     (λ j → p (i ∧ j))) (d (p i0))
```

Path induction (2/2)

- Given a family:

$$C : \prod_{x,y:A} (x =_A y) \to \mathcal{U}$$

```
(P : ∀ x y → x ≡ y → Type ℓ)
```

- and a function:

$$c : \prod_{x:A} C(x, x, \mathsf{refl}_x)$$

```
(d : (x : A) → P x x refl)
```

- there is a function:

$$f : \prod_{x,y:A} \prod_{p:x=_A y} C(x, y, p)$$

```
{x : A} {y : A} (p : x ≡ y)
  → P x y p
```

- such that:

$$f(x, x, \mathsf{refl}_x) :\equiv c(x)$$

```
transport (λ i → P (p i0) (p i)
    (λ j → p (i ∧ j))) (d (p i0))
```

Substitution

- Transport and homogeneous paths can be used to obtain substitution:

Substitution

```
subst : (B: A → Type) {x y: A} → x ≡ y → B x
    → B y
subst B p b = transport (λ i → B (p i)) b
```

The univalence principle

Equivalence implies equality

Any equivalence of types yields an equality of types:

$$\mathsf{ua} : \{A\ B : \mathsf{Type}\} \to A \simeq B \to A \equiv B$$

The univalence principle

Equivalence implies equality

Any equivalence of types yields an equality of types:

```
ua : {A B : Type} → A ≃ B → A ≡ B
```

Transporting along an equivalence applies that equivalence

For a given equivalence e, transporting along ua e applies the function underlying e:

```
uaβ : {A B : Type} (e : A ≃ B) (x : A) → transport
      (ua e) x ≡ (equivFun e) x
```

Here, equivFun: A ≃ B → A → B returns the function underlying an equivalence.

Example instance of *uaβ*

Transporting along an equivalence applies that equivalence

For a given equivalence e, transporting along ua e applies the function underlying e:

```
uaβ: {A B: Type} (e: A ≃ B) (x: A) → transport
    (ua e) x ≡ (equivFun e) x
```

Here, equivFun: A ≃ B → A → B returns the function underlying an equivalence.

Example instance of *uaβ*

For an equivalence not≃ with underlying function not : Bool → Bool:

```
_ : transport (ua not≃) false ≡ true
_ = refl
```

Uniqueness of Identity Proofs (UIP) (1/2)

- Univalence refutes the Uniqueness of Identity Proofs (UIP).
- UIP is the principle that all equality proofs between the same two elements are equal.
- Types that satisfy UIP can be called *h*-sets (or simply sets).
- The HoTT book defines *h*-sets as:

_h_-sets as defined in the HoTT book

$$\mathsf{isSet}(A) :\equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q)$$

Uniqueness of Identity Proofs (UIP)

- UIP is the principle that all equality proofs between the same two elements are equal.
- The universe of types does not satisfy UIP, because univalence can produce unequal proofs of equality between types.
- The paper provides a proof:

Counterexample

We can see via transport that ua not≃ and refl are unequal proofs of Bool ≡ Bool:

```
not≢refl : ¬ (ua not≃ ≡ refl)
not≢refl e = true≢false
    (transport
    (λ i → transport (e i) true ≡ false) refl)
```

h-sets

- Types that satisfy UIP are called *h*-sets (or simply sets).
- The HoTT book defines *h*-sets as:

h-sets as defined in the HoTT book

$$\mathsf{isSet}(A) :\equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q)$$

- This paper defines *h*-sets as:

h-sets as defined in this paper

```
isSet : Type → Type
isSet A = (x y : A) → isProp (x ≡ y)
```

h-propositions

- The paper defines *h*-sets in terms of *h*-propositions.
- An *h*-proposition (or mere proposition) is a type of which all the elements are equal.
- The HoTT book defines *h*-propositions as:

h-propositions as defined in the HoTT book

$$\mathrm{isProp}(P) :\equiv \prod_{x,y:P} (x = y)$$

- The paper defines *h*-propositions as:

h-propositions as defined in the paper

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

h-sets

- Shown together, *h*-sets and *h*-propositions are defined as:

_h_-sets and _h_-propositions as defined in the HoTT book

$$\mathsf{isProp}(P) :\equiv \prod_{x,y:P} (x = y)$$

$$\mathsf{isSet}(A) :\equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q)$$

_h_-sets as defined in the paper

```
isSet: Type → Type
isSet A = (x y: A) → isProp (x ≡ y)

isSet': Type → Type
isSet' A = (x y: A) → (p q: x ≡ y) → p ≡ q
```

h-propositions and Type formers

- The paper states that *h*-propositions are closed under many type formers.
- As an example, the paper provides the following definition, which shows that dependent functions that yield propositions (and are therefore type formers) are themselves propositions:

Dependent functions that yield propositions are propositions

```
isPropΠ: {B:A → Type} (h: (x:A) → isProp(B x))
    → isProp((x:A) → B x)
isPropΠ h p q i x = h x (p x) (q x) i
```

Equivalence

- An equivalence $A \simeq B$ is a function $A \to B$ such that the preimage of every point in B is contractible:

Equivalence

```
isContr : Type → Type
isContr A = Σ[ x ∈ A ] ((y: A) → x ≡ y)
```

Equivalence

- An equivalence $A \simeq B$ is a function $A \to B$ such that the preimage of every point in B is contractible:

Equivalence

```
isContr : Type → Type
isContr A = Σ[ x ∈ A ] ((y: A) → x ≡ y)

preim : {A B: Type} (f: A → B) (y: B) → Type
preim {A = A} f y = Σ[ x ∈ A ] (f x ≡ y)
```

Equivalence

- An equivalence $A \simeq B$ is a function $A \to B$ such that the preimage of every point in B is contractible:

Equivalence

```
isContr: Type → Type
isContr A = Σ[ x ∈ A ] ((y: A) → x ≡ y)

preim: {A B: Type} (f: A → B) (y: B) → Type
preim {A = A} f y = Σ[ x ∈ A ] (f x ≡ y)

isEquiv: {A B: Type} (f: A → B) → Type
isEquiv {B = B} f= (y: B) → isContr(preim f y)
```

Equivalence

- An equivalence $A \simeq B$ is a function $A \to B$ such that the preimage of every point in B is contractible:

Equivalence

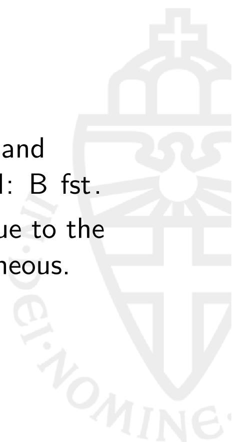
```
isContr: Type → Type
isContr A = Σ[ x ∈ A ] ((y: A) → x ≡ y)

preim: {A B: Type} (f: A → B) (y: B) → Type
preim {A = A} f y = Σ[ x ∈ A ] (f x ≡ y)

isEquiv: {A B: Type} (f: A → B) → Type
isEquiv {B = B} f= (y: B) → isContr(preim f y)

_≃_: (A B: Type) → Type
A ≃ B = Σ[ f ∈ (A → B) ] (isEquiv f)
```

Equality in Σ-types (1/2)



- Σ-types are dependent pairs of types $(A : \text{Type})$ and $(B : A \to \text{Type})$ with projections $\text{fst} : A$ and $\text{snd} : B\ \text{fst}$.
- Equality in Σ-types can be difficult to manage, due to the equality of the second projections being heterogeneous.

Equality in Σ-types (2/2)

- Σ-types are dependent pairs of types (A : Type) and (B : A → Type) with projections fst : A and snd: B fst.
- Paths between instances of Σ-types are equivalent to paths between their first and second projections:

Equality in Σ-types

```
ΣPath≃PathΣ: {B: A → Type} (x y: Σ A B) →
  (Σ[ p ∈ (fst x ≡ fst y) ]
  (PathP (λ i → B (p i)) (snd x) (snd y)))
    ≃
  (x ≡ y)
ΣPath≃PathΣ x y = isoToEquiv (iso
  (λ { (p, q) i → (p i, q i) })
  (λ p → ((λ i → fst(p i)), (λ i → snd(p i))))
  (λ _ → refl) (λ _ → refl))
```

Propositional extensionality

- Propositional extensionality: Propositions that imply each other ($A \leftrightarrow B$) are equivalent ($A \simeq B$), and thus equal by the ua map of univalence:

Propositional extensionality

```
prop≃: isProp A → isProp B → (A → B) → (B → A)
    → A ≃ B
prop≃ pA pB f g = isoToEquiv
    (iso f g (λ _ → pB _ _) (λ _ → pA _ _))
```

Higher inductive types

- HIT: generalizes inductive types by allowing equality types in constructors
- Here we use HIT's to take quotients of types by equivalence relations
- In other words, we quotient a type into several equivalence classes via an equivalence relation. So two related elements are in the same equivalence class

Propositional truncation

- Propositional truncation: quotient a type by the total relation, yielding a proposition
- So a relation where every pair of elements is related.
- In other words, you can truncate a type to a mere proposition
- This can be defined by:

Propositional truncation

```
data ||_|| (A : Type) : Type where
|_| : A → ||A||
squash : (x y : ||A||) → x ≡ y
```

Propositional truncation: Map

- You can define functions on this type by pattern matching:

Map function

```
map : (A → B) → ||A|| → ||B||
map f |x| = |f x|
map f (squash x y i)
    = squash (map f x) (map f y) i
```

Example: Cost monad

- Cost function that pairs an element of a given type to a 'hidden' integer:

Cost function

```
Cost : (A : Type) → Type
Cost A = A × ||ℕ||

Cost≡ : (x y : Cost A) → fst x ≡ fst y → x ≡ y
Cost≡ (x , cx) (y , cy) h i =
    (h i , squash cx cy i)
```

Example: Cost monad

- Now we can define the Cost monad itself, it models cost by incrementing whenever the bind is used:

Defining the monad

```
_>>=_  :  Cost A → (A → Cost B) → Cost B
(x , m) >>= g with g x
... | (y , n) = (y , map suc (map2 _+_ m n))

return : A → Cost A
return x = (x , |0|)
```

Example: Using Cost monad for Fibonacci

simple fibonacci implementation

```
fib : ℕ → Cost ℕ
fib 0 = return 0
fib 1 = return 1
fib (suc (suc x)) = do
    y ← fib (suc x)
    z ← fib x
    return (y + z)

_ : fib 20 ≡ (6765 , |21890|)
_ = refl
```

Example: Using Cost monad for Fibonacci

tail-recursive fibonacci implementation

```
fibAux : ℕ → ℕ → ℕ → Cost ℕ
fibAux 0 res _ = return res
fibAux (suc x) res prev = do
    r ← fibAux x (res + prev) res
    return r

fibTail : ℕ → Cost ℕ
fibTail 0 = return 0
fibTail (suc x) = fibAux x 1 0

_ : fibTail 20 ≡ (6765 , |19|)
_ = refl
```

Proving equality

Note that you can use the Cost≡ function to prove these implementations are equal despite the difference in runtime behaviour!

Set quotients

- Set quotient: quotient a type by an arbitrary given relation, yielding a set
- This can be defined as:

Set quotient

```
data _/_ (A : Type) (R : A → A → Type) : Type
  where
[_] : (a : A) → A / R
eq/ : (a b : A) → (r : R a b) → [ a ] ≡ [ b ]
squash/ : isSet (A / R)
```

Universal set quotient property

SetQuotUniversal

```
setQuotUniversal :
    {R : A → A → Type}
    {B : Type} →
    isSet B →
(A / R → B)
        ≃
(Σ[ f ∈ (A → B) ]
((a b : A) → R a b → f a ≡ f b))
```

Example: Rational numbers

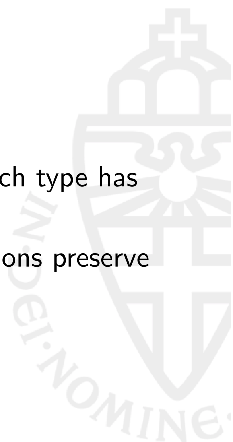
Rational number multiplication

```
_∼_ : ℤ× ℕ+→ ℤ× ℕ+→ Type
(a , b) ∼ (c , d) = a *ℕ+ d ≡ c *ℕ+ b

ℚ : Type
ℚ = (ℤ× ℕ+) / _∼_

ℚ' : Type
ℚ' = Σ[(a , b) ∈ ℤ× ℕ+] (coprime (abs a) b)
```

Setoids



- Use setoids rather than HIT's for quotients
- This means an explicit equivalence relation for each type has to be given
- However, this requires manual proofs that operations preserve this relation

Q&A