Freedom for Proofs!

Representation Independence is More than Parametricity

Irene Yoon

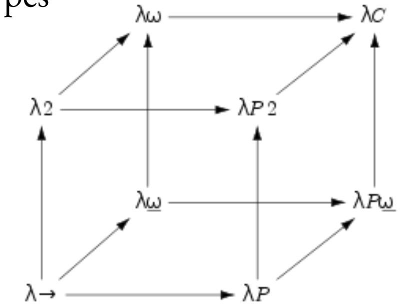
Modularity in programming

- Software should have correct **abstractions** that can **compose**
- *"Type structure is a syntactic discipline for enforcing levels of abstraction" - John Reynolds*

- ***Representation Independence***
    - Programmers can give <u>different implementations</u> for the same abstract interface
    - e.g. Two different implementations of a queue can be interchangeable

Parametricity $\forall\alpha.\tau$

- Parametrically polymorphic functions behave uniformly in their type arguments
  - Strachey (1967) / Lambek(1972) "generality"
- Reynold's *relational parametricity* (1983)
  - System F (polymorphic lambda-calculus)
  - **Logical relations**: related inputs lead to related outputs
- Mitchell's *representation independence and data abstraction (1986)*
  - Applies parametricity to prove representation independence for existential types
- Wadler's *free theorems* (1989)
  - "Every function of the same type satisfies the same theorem"



..beyond System F!

Dependently-Typed Programming

Agda

Nuprl

TWELF

LEAN

Idris

- *Good*: Rich program specifications
- *Not so good*: Notoriously labor-intensive

How can we bring about representation independence to dependently-typed programming?

Bird's Eye View and **expectations** $^{\lambda}$

Krishnaswami & Dreyer 2013	**Internalizing Relational Parametricity in the Extensional Calculus of Constructions**
Tabareau et al. 2019	**Marriage of Univalence and Parametricity**
Angiuli et al. 2021	**Internalizing Representation Independence with Univalence**

Internalizing Relational Parametricity in the Extensional Calculus of Constructions

Bird's Eye View

	Main Technique	Type theory	Result
Krishnaswami & Dreyer 2013	Internalized parametricity with *realizability semantics*	Extensional Calculus of Constructions	1. Relationally parametric model 2. Adding **semantically well-typed terms** as axioms with computational content
Tabareau et al. 2019	Marriage of Univalence and Parametricity		
Angiuli et al. 2021	Internalizing Representation Independence with Univalence		

Parametric Type Theories

Bernardy et al. [2010, 2012a, 2012b, 2013, 2015]

- **Abstraction Theorem**

$$If\ \Gamma \vdash t : A\ \ then\ [\![\Gamma]\!]\ \vDash [\![t]\!]\ : [\![A]\!]\ t\ t$$

- *Internalized parametricity*: Abstraction Theorem can be stated and proved *within* type theory
- *Externalized parametricity*: Abstraction Theorem is stated through a meta-theoretic translation.

Equality in Dependent Type Theory

Type-checking requires checking term equality

Judgmental Equality $\Gamma \vdash A = A' \; \mathit{type}$

Set of equality rules that are (inductively) defined

Definitional Equality type-checker silently coerces between definitionally equal types

Propositional Equality $\mathsf{Eq}_A(x, y)$

Proof of equality between two elements

Equality in Type Theory

Extensional type theory : equality reflection

$$\frac{p : \mathsf{Eq}_A(x, y)}{x = y}$$

Uniqueness of Identity Proofs (UIP) : Any two elements of $\mathsf{Eq}_A(x, y)$ are equal.

Streicher's *Axiom K*

For the context of parametricity: Allow coercions between parametrically related terms!

Realizability Semantics

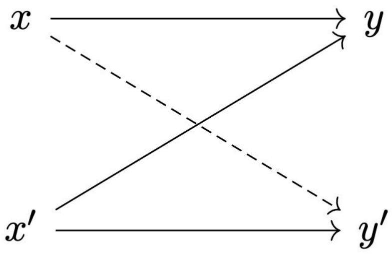
- Taking the Brouwer–Heyting–Kolmogorov (BHK) Interpretation to heart
  - The interpretation of a logical formula is the **proof (realizer)**
  - e.g. `P /\ Q` interprets to `<a, b>` where `a` is a proof of `P` and `b` is a proof of `Q`
- What if you have a formula which you have a proof of…
  - But your typing rules do not "type-check" the formula?
- **It must be true! Add the formula to the theory!**

Syntactic formalisms cannot show all truths!

Gödel's Incompleteness Theorem (1931)

Realizability-style Model

- Interprets types as relations (*logical relations*)
- Quasi-PERs (QPERs) to show heterogeneous equivalences
  - Typically, the interpretation is a partial equivalence relation (aka PER, a symmetric and transitive relation)
  - Symmetry requires homogeneity (relation must relate two terms of equal types)



$$\textit{if } (x, y) \in R,\ (x', y') \in R,\ \textit{and } (x', y) \in R,\ \textit{then } (x, y') \in R.$$

- Can use a single relational model for relating terms at different types
  - (Instead of requiring a PER model of types and a relational model between PERs)

Internalizing relational parametricity

- Relationally parametric model of an <u>extensional</u> Calculus of Constructions
- <u>Realizability-style</u> interpretation of types
  - Types interpreted as relations
  - Realizer: Exhibit a term that is related to itself at the type (semantically well-typed term)
- Can add "validated axioms" to the theory which have realizers

$$(e \ , e \ ) \in [\![X]\!]$$

relational interpretation $[\![\quad]\!]$
realizer e
axiom X

Adding axioms with computational content to theory

- Dependent pairs (Σ-types)
- Induction principle for natural numbers
- Quotient types

The axiom may not be syntactically well-typed, but the realizer of the axiom (i.e. the proof of the axiom) is semantically well-typed!

Bird's Eye View

	Main Technique	Type theory	Result
Krishnaswami & Dreyer 2013	Internalized parametricity with *realizability semantics*	Extensional Calculus of Constructions	1. Relationally parametric model 2. Adding **semantically well-typed terms** as axioms with computational content
Tabareau et al. 2019	Marriage of Univalence and Parametricity		
Angiuli et al. 2021	Internalizing Representation Independence with Univalence		

Marriage of Univalence and Parametricity

Goal: Automated Proof Transport

Given two implementation of natural numbers, we should be able to *reuse* proofs between them

```
Inductive nat : Set :=
| O : nat
| S : nat → nat
```

Easy to reason about

```
Inductive Bin : Set :=
| O_Bin : Bin
| pos_Bin : positive → nat

Inductive positive : Set :=
| xI : positive → positive
| xO : positive → positive
| xH : positive
```

Efficient

Goal: Automated Proof Transport

Given two implementation of natural numbers, we should be able to *reuse* proofs between them

```
Lemma plus_comm : ∀ n m : nat, n + m = m + n.
Proof.
    ...
Qed.
```

```
Lemma plus_Bin_comm : ∀ n m : Bin, n + m = m + n.
Proof.
    transport plus_comm. (* automatically inferred *)
Qed.
```

Using parametricity for refinement

Given two implementation of natural numbers, we should be able to *reuse* proofs between them

```
Lemma plus_comm : ∀ n m : nat, n + m = m + n.
Proof.
    ...
Qed.
```

```
Lemma plus_Bin_comm : ∀ n m : Bin, n + m = m + n.
Proof.
    transport plus_comm. (* automatically inferred *)
Qed.
```

1. Specifying a common abstract interface a priori can be difficult

"Anticipation Problem"

Usually, parametricity states a relation between two expressions on the *same type*

(i.e. homogeneous parametricity)

$$\textit{If } \Gamma \vdash t : A \;\; \textit{then } [\![\Gamma]\!] \;\vDash [\![t]\!] \;\; : [\![A]\!]\; t\; t$$

Heterogeneous parametricity can relate two expressions to each other *directly*

$$\text{If } \Gamma \vdash t : A \text{ and } [\Gamma] \vdash [t] : [A] \text{ then } [\![\Gamma]\!] \;\vDash [\![t]\!] \;\; : [\![A]\!]\; t\; [t]$$

2. Limits of parametricity in an Intensional Type Theory

"Computation Problem"

Parametrically-related functions behave the same *propositionally* but not *definitionally*

(i.e. parametrically related definitions are not equal by conversion)

Univalence to the rescue!

Univalence

Isomorphic types are treated the "same" (isomorphic objects enjoy same structural properties)

Isomorphic types are *equal*

Voevodsky (2009)

Every **equivalence** (isomorphism) between types A and B leads to an identity proof **Id (A, B)**

Type Equivalence (Isomorphism)

$f : A \to B$ is an *equivalence* iff there exists a function $g : B \to A$ paired with proofs that f and g are inverses of each other.

$$\forall a : A,\ \mathsf{Eq}(g(f(a)), a)$$

$$\forall b : B,\ \mathsf{Eq}(f(g(b)), b)$$

Type equivalence ($A \simeq B$)

Two types A and B are equivalent to each other iff there exists a function $f : A \to B$ that is an equivalence.

Univalence

All about coercions!

For any two types A and B, the canonical map **Id**(A, B) → (A ≃ B) is an equivalence.

<u>Indiscernibility of Equivalents</u>

For any P: Type → Type, and any two types A and B such that A ≃ B, we have P A ≃ P B

<u>Immediate ***transport*** using univalence</u>

For any P: Type → Type, and any two types A and B such that A ≃ B,

there exists a function ***transport*** $\uparrow_{\blacksquare}$: P A → P B

N.B. : Realizing Univalence

- Homotopy Type Theory: *axiomatized* univalence
- Use of axioms breaks **computational adequacy** (“stuck terms”, “canonicity”)

“All closed terms of a natural number type compute numerals”

- Alternative : Cubical Type Theory
  - De Morgan Cubical Type Theory
  - Cartesian Cubical Type Theory

(we’ll brush on it a little later)

- Tabareau et al.’s approach painstakingly maneuvers coercions between typeclasses that simulate *computational rules* that are at the foot of cubical type theory

Univalent Parametricity

- Restriction of parametricity to relations that correspond to **equivalences**

$[\![\mathsf{Type}_i]\!]\ A\ B$

relation **R : A → B → Type$_i$**

equivalence **e : A ≃ B**

coherence condition **Π a b . (R a b) ≃ (a = ↑$_e$ b)**

$$[\![\mathsf{Type}_i]\!]\ A\ B \triangleq \Sigma(R : A \to B \to \mathsf{Type}_i)(e : A \simeq B).\, \Pi a\ b.\, (R\ a\ b) \simeq (a = \uparrow_e b)$$

Univalent Parametricity in Action

```
Definition square (x : nat) : nat := x * x.

Definition square_Bin■: Bin → Bin := ↑■ square. (* Transport using univalence *)


Check eq_refl : square_Bin■ = (fun x:Bin ⇒ ↑■ (square (↑■ x))). (* Inefficient *)


Definition univrel_mult : mult ≈ mult_Bin.  (* Additional proof *)


Definition square_Bin□: Bin → Bin := ↑□ square. (* Transport using parametricity *)

Check eq_refl : square_Bin□ = (fun x ⇒ (x * x)%Bin). (* Infers new univalent relations
*)
```

Bird's Eye View

	Main Technique	Type theory	Result
Krishnaswami & Dreyer 2013	Internalized parametricity with *realizability semantics*	Extensional Calculus of Constructions	1. Relationally parametric model 2. Adding **semantically well-typed terms** as axioms with computational content
Tabareau et al. 2019	Univalent parametricity (Externalized parametricity with univalence)	Calculus of Inductive Constructions (CIC) with axiomatized univalence	**Automated** proof transport between isomorphic representations
Angiuli et al. 2021	**Internalizing Representation Independence with Univalence**		

Internalizing Representation Independence with Univalence

Cubical Type Theory

Axiomatized univalence bites the programmer's neck

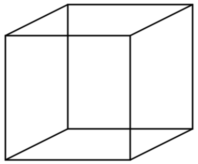
Stuck terms that are unable to reduce (i.e. lacks **computational adequacy**)

Cubical type theory: *constructive interpretation* of univalence

Path types: information about how two types are equal

Angiuli et al. 2021

Cubical Type Theory



<u>Path Types</u>

Maps out of an interval type **I** which has two elements **i0: I** and **i1: I** that are *behaviorally equal* but *not definitionally equal*

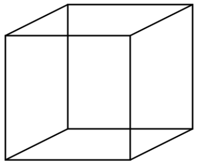
- <u>Behavioral equality</u>: no function $\mathbf{f : I \to A}$ can distinguish the elements

$\mathbf{PathP : (A : I \to Type\ l) \to A\ i0 \to A\ i1 \to Type\ l}$ specifies the behavior of their elements at i0 and i1

$\mathbf{\_{\equiv}\_\ \{A = A\}\ x\ y = PathP\ (\lambda\ \_ \to A)\ x\ y}$ homogeneous equality using path type

Angiuli et al. 2021

Higher Inductive Types



Each constructor carries *paths between elements*

Set quotients quotient a type with an arbitrary relation (resulting in a set).

$$
\begin{aligned}
&\mathsf{data}\ \_/\_\{A : \mathsf{Type}\} \to \{R : A \to A \to \mathsf{Type}\} \to \mathsf{Type}\ \mathsf{where} \\
&\quad [\_] : \{a : A\} \to A/R \\
&\quad \mathsf{eq}/ : \{a\, b : A\} \to \{r : R\, a\, b\} \to [a] \equiv [b] \\
&\quad \mathsf{squash}/ : \mathsf{isSet}(A/R).
\end{aligned}
$$

Queue up!

Let's say we want a **Queue** implementation with a standard **dequeue** and **enqueue** operation.

Basic implementation: **ListQueue**

```
ListQueue (A : Type) → Queue A
ListQueue A = queue (List A) [] _::_ last
```

Faster, faster..

Okasaki's **BatchedQueue** representation:

The queue is a tuple **Q = List A × List A** (first queue for *enqueue*, second queue for *dequeue)*

(amortized constant-time!)

```
BatchedQueue : (A : Type) → Queue A
BatchedQueue A = queue (List A x List A) ([], [])
    (fun x (xs, ys) → fastcheck (x :: xs, ys))
    (fun {(_, []) → nothing ; (xs, x :: ys) → just (fastcheck (xs, ys), x)})
where
    fastcheck : {A : Type} → List A * List A → List A * List A
    fastcheck (xs, ys) = if isEmpty ys then ([], reverse xs ) else (xs, ys)
```

Structure-preserving Correspondence

```
appendReverse : {A : Type} → BatchedQueue A Q → ListQueue A Q
appendReverse (xs, ys) = xs ++ reverse ys
```

Structure-preserving – preserves **empty**, and commutes with **enqueue** and **dequeue**

Thus, **ListQueue** and **BatchedQueue** are contextually equivalent!

What's the problem?

([], [1,0]) and ([0], [1]) maps to [0, 1]   **Not** an isomorphism!

Structure-preserving Equivalence

A **structure** is a function S : Type → Type, and an *S-structure* is a dependent pair of a type and its application to the structure.

$$\mathsf{TypeWithStr}\ \mathsf{S} = \Sigma[X \in \mathsf{Type}](S\ X)$$

An **S-structure-preserving** equivalence `StrEquiv` is a term with two S-structures and an equivalence between their underlying types.

$$\mathsf{StrEquiv}\ S = (A\ B : \mathsf{TypeWithStr}\ S) \to \mathsf{fst}\ A \simeq \mathsf{fst}\ B \to \mathsf{Type}$$

$$A \simeq [\ \iota\ ]B = \Sigma[e \in \mathsf{fst}\ A \simeq \mathsf{fst}\ B](\iota\ A\ B\ e) \qquad \iota : \mathsf{StrEquiv}\ S$$

Structure Identity Principle

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

Univalent Structure (S, ι)

$$\begin{aligned}\mathsf{UnivalentStr}\ S\ \iota = \{A\ B : \mathsf{TypeWithStr}\ S\}(e : \mathsf{fst}\ A \simeq \mathsf{fst}\ B) \\ \to (\iota\ A\ B\ e) \simeq \mathsf{PathP}(\lambda i \to S(\mathsf{ua}\ e\ i))(\mathsf{snd}\ A)(\mathsf{snd}\ B)\end{aligned}$$

Structure Identity Principle (SIP)

For $\mathrm{S} : \mathsf{Type} \to \mathsf{Type}$ and $\iota : \mathsf{StrEquiv}\ \mathrm{S}$, we have a term

$$\mathsf{SIP} : \mathsf{UnivalentStr}\ \mathrm{S}\ \iota \to (\mathrm{A}\ \mathrm{B} : \mathsf{TypeWithStr}\ \mathrm{S}) \to (\mathrm{A} \simeq[\ \iota\ ]\ \mathrm{B}) \simeq (\mathrm{A} \equiv \mathrm{B})$$

Using the SIP

Given a set **A** fixed, the raw queue structure contains the empty queue, and the enqueue/dequeue functions.

$$\mathsf{RawQueueStructure}\ X = X * (A \to X \to X) * (X \to \mathsf{Maybe}(X * A))$$

Set quotients can identify any two **`BatchedQueue`**s sent to the same list by `appendReverse`.

$$\begin{aligned}
&\mathsf{data\ BatchedQueueHIT} : \mathsf{Type\ where} \\
&\quad Q\langle _, _ \rangle : \mathsf{List}\ A \to \mathsf{List}\ A \to \mathsf{BatchedQueueHIT} \\
&\quad \mathsf{tilt} : \forall\ xs\ ys\ a \to Q\langle xs \mathbin{++} [a], ys \rangle \equiv Q\langle xs, ys \mathbin{++} [a] \rangle \\
&\quad \mathsf{squash} : \mathsf{isSet\ BatchedQueueHIT}
\end{aligned}$$

Using the SIP

The structure-map between the structures, **appendReverse**, can be extended to an equivalence **BatchedQueueHIT ≃ List A** which induces a raw queue structure on **BatchedQueueHIT**

Finally, an appeal to the SIP will transfer any **ListQueue** axioms to the quotiented **BatchedQueue** operations

Recall Structure Identity Principle (SIP)

For $\mathrm{S} : \mathsf{Type} \to \mathsf{Type}$ and $\iota : \mathsf{StrEquiv}\ \mathrm{S}$, we have a term

$$\mathsf{SIP} : \mathsf{UnivalentStr}\ \mathrm{S}\ \iota \to (\mathrm{A}\ \mathrm{B} : \mathsf{TypeWithStr}\ \mathrm{S}) \to (\mathrm{A} \simeq[\ \iota\ ]\ \mathrm{B}) \simeq (\mathrm{A} \equiv \mathrm{B})$$

Bird’s Eye View

	Main Technique	Type theory	Result
Krishnaswami & Dreyer 2013	Internalized parametricity with *realizability semantics*	Extensional Calculus of Constructions	Adding **semantically well-typed terms** and free theorems
Tabareau et al. 2019	Univalent parametricity (Externalized parametricity with univalence)	Calculus of Inductive Constructions (CIC) with axiomatized univalence	**Automated** proof transport between isomorphic representations
Angiuli et al. 2021	Univalence (Structure Identity Principle)	De Morgan Cubical Type Theory	Proof transport between **non-isomorphic** representations

Induced Equivalence from QPERs

Canonically Induced PER

Every QPER $Q \subseteq R \times S$ induces an equivalence relation $\sim_Q \subseteq Q \times Q$ (and hence a PER on $R \times S$), defined as $(a_1, a_2) \sim_Q (b_1, b_2)$ iff the zigzag $\{(a_1, a_2), (b_1, b_2), (a_1, b_2), (b_1, a_2) \subseteq Q\}$.

Global Context for Univalent Parametricity

$$\Xi_0 = \cdot$$

Two constants

Witness that two constants are parametrically related

$$\Xi_1 = (\underline{c_1^\circ : A_1^\circ;\ c_1^\bullet : A_1^\bullet};\ \underline{c_1^\otimes : [A_1]_u^{\Xi_0}\ c_1^\circ\ c_1{}^\bullet})$$

$$\ldots$$

$$\Xi_n = \Xi_{n-1}, (c_n^\circ : A_n^\circ;\ c_n^\bullet : A_n^\bullet;\ c_n^\otimes : [A_n]_u^{\Xi_{n-1}}\ c_n^\circ\ c_n^\bullet)$$

Universes

$$[\![\mathsf{Type}_i]\!]\ A\ B \triangleq \Sigma(R : A \to B \to \mathsf{Type}_i)(e : A \simeq B).\,\Pi a\ b.\,(R\ a\ b) \simeq (a = \uparrow_e b)$$

$$[\mathsf{Type}_i]_u : [\![\mathsf{Type}_{i+1}]\!]_u\ \mathsf{Type}_i\ \mathsf{Type}_i \equiv$$
$$\Sigma(R : \mathsf{Type}_i \longrightarrow \mathsf{Type}_i \longrightarrow \mathsf{Type}_{i+1})(e : \mathsf{Type}_i \simeq \mathsf{Type}_i).\ \Pi a\ b.(R\ a\ b) \simeq (a = \uparrow_e b).$$

$$[\mathsf{Type}_i]_u \triangleq (\lambda\ (A\ B : \mathsf{Type}_i),\ \Sigma(R : A \longrightarrow B \longrightarrow \mathsf{Type}_i)(e : A \simeq B).$$
$$\Pi ab.(R\ a\ b) \simeq (a = \uparrow_e b); \mathrm{id}_{\mathsf{Type}_i}; \mathsf{univ}_{\mathsf{Type}_i})$$