Programming and Proving with the Structure Identity Principle in Cubical Agda

Carlo Angiuli, Evan Cavallo, **Anders Mörtberg** and Max Zeuner

Carnegie Mellon University and Stockholm University

CMU HoTT seminar, November 13, 2020

This talk

1. Introduction: Computer Science & HoTT
2. Cubical Type Theory & Cubical Agda
3. A Cubical Structure Identity Principle
4. A Relational Structure Identity Principle
5. Conclusion

Overview

1. Introduction: Computer Science & HoTT
2. Cubical Type Theory & Cubical Agda
3. A Cubical Structure Identity Principle
4. A Relational Structure Identity Principle
5. Conclusion

Introduction: Computer Science & HoTT

Observation: not many CS applications of HoTT in the literature, despite a lot of initial interest

[1]Datastructures, algorithms, PL theory, automata, complexity, ...

Introduction: Computer Science & HoTT

Observation: not many CS applications of HoTT in the literature, despite a lot of initial interest

Problem: univalence and HITs are axiomatic in HoTT $\Rightarrow$ cannot run programs written using them... ☹

[1] Datastructures, algorithms, PL theory, automata, complexity, ...

Introduction: Computer Science & HoTT

Observation: not many CS applications of HoTT in the literature, despite a lot of initial interest

Problem: univalence and HITs are axiomatic in HoTT $\Rightarrow$ cannot run programs written using them... ☹

Solution: cubical type theory gives univalence and HITs computational content, so HoTT programs can be run! ☺

[1] Datastructures, algorithms, PL theory, automata, complexity, ...

Introduction: Computer Science & HoTT

Observation: not many CS applications of HoTT in the literature, despite a lot of initial interest

Problem: univalence and HITs are axiomatic in HoTT ⇒ cannot run programs written using them... ☹

Solution: cubical type theory gives univalence and HITs computational content, so HoTT programs can be run! ☺

Question: many traditional CS applications[1] are about sets, can we still benefit from univalence & HITs?

[1] Datastructures, algorithms, PL theory, automata, complexity, ...

Yes!

Univalence and HITs are in fact very useful for tackling CS problems related to programming and reasoning about programs

Yes!

Univalence and HITs are in fact very useful for tackling CS problems related to programming and reasoning about programs

We tackle the following problems using `Cubical Agda`:

1. Program/proof transfer: transport programs and proofs between equivalent/isomorphic *structured* types
2. Representation independence: relate implementations of abstract interfaces using *structured* relations

The solutions to both of these problems require a cubical version of the structure identity principle (SIP)

Overview

1. Introduction: Computer Science & HoTT
2. Cubical Type Theory & Cubical Agda
3. A Cubical Structure Identity Principle
4. A Relational Structure Identity Principle
5. Conclusion

Cubical Agda

Extension of the dependently typed programming language `Agda` with:

- Interval and Path types (I, PathP, _≡_, ...)
- Transport (via transp and hcomp)
- Univalence (via Glue types)
- Higher inductive types

Cubical Agda

Extension of the dependently typed programming language `Agda` with:

- Interval and Path types (I, PathP, $_\equiv_$, ...)
- Transport (via transp and hcomp)
- Univalence (via Glue types)
- Higher inductive types

Get a proof assistant for cubical type theory with all of `Agda`'s features:

- Advanced dependent pattern matching (also for HITs!)
- Inductive and coinductive types
- `Agda` emacs mode
- Inductive families
- ...

Big plus: easy for `Agda` users to try out cubical type theory!

Demo!

`https://github.com/agda/cubical/`

($\sim$ 39000 LOC, $\sim$ 400 files, 47 contributors)

Overview

1. Introduction: Computer Science & HoTT
2. Cubical Type Theory & Cubical Agda
3. A Cubical Structure Identity Principle
4. A Relational Structure Identity Principle
5. Conclusion

Proof-oriented vs. computation-oriented types

Proof-oriented: unary numbers $\mathbb{N}$ (generated by 0 and succ)

Computation-oriented: binary numbers Bin$\mathbb{N}$ (lists of $0/1$ with no trailing 0)

Proof-oriented vs. computation-oriented types

Proof-oriented: unary numbers $\mathbb{N}$ (generated by 0 and succ)

Computation-oriented: binary numbers Bin$\mathbb{N}$ (lists of 0/1 with no trailing 0)

Tension: want to develop theory using $\mathbb{N}$, but run computations using Bin$\mathbb{N}$

> *"We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day, so to speak. [...] But nothing is gained – on the contrary! – by tackling these various aspects simultaneously. It is what I sometimes have called the* ***separation of concerns****." [Dijkstra, 1974]*

Proof vs. computation oriented types

Does univalence help with this problem?

[2] The substUA function is in fact an equivalence itself, but that is not important here.

Proof vs. computation oriented types

Does univalence help with this problem?

Yes, kind of... It's easy to prove the following result:

$$\text{substUA} : (S : \text{Type} \to \text{Type})\ \{A\ B : \text{Type}\}\ (e : A \simeq B) \to S\ A \to S\ B$$

So we can transport any structure S along an equivalence of A and B.[2]

Let's see how this can be useful to transfer programs and proofs!

[2] The substUA function is in fact an equivalence itself, but that is not important here.

Example: Monoid of unary and binary numbers

Consider your favorite implementation of monoid structures on a type A:

$$\text{MonoidStr} \; : \; (A : \text{Type}) \; \rightarrow \; \text{Type}$$

Example: Monoid of unary and binary numbers

Consider your favorite implementation of monoid structures on a type A:

$$\text{MonoidStr} \ : \ (A : \text{Type}) \ \to \ \text{Type}$$

This could be a record or nested Σ-types with

$$\varepsilon : A \qquad\qquad _\cdot_ : A \to A \to A$$

Satisfying:

$$(x : A) \to (x \cdot \varepsilon \equiv x) \times (\varepsilon \cdot x \equiv x)$$

$$(x\ y\ z : A) \to x \cdot (y \cdot z) \equiv (x \cdot y) \cdot z$$

$$\text{isSet}\ A$$

Monoid of unary and binary numbers

It is easy to use $0 : \mathbb{N}$ and $_+_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ to construct an instance

$$\mathsf{Monoid}\mathbb{N} : \mathsf{MonoidStr}\ \mathbb{N}$$

We can also prove

$$\mathsf{natEquiv} : \mathbb{N} \simeq \mathsf{Bin}\mathbb{N}$$

Monoid of unary and binary numbers

It is easy to use $0 : \mathbb{N}$ and $_+_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ to construct an instance

$$\mathsf{Monoid}\mathbb{N} : \mathsf{MonoidStr}\ \mathbb{N}$$

We can also prove

$$\mathsf{natEquiv} : \mathbb{N} \simeq \mathsf{Bin}\mathbb{N}$$

So we get:

$$\mathsf{MonoidBin}\mathbb{N} : \mathsf{MonoidStr}\ \mathsf{Bin}\mathbb{N}$$
$$\mathsf{MonoidBin}\mathbb{N} = \mathsf{substUA}\quad \mathsf{MonoidStr}\quad \mathsf{natEquiv}\quad \mathsf{Monoid}\mathbb{N}$$

We have hence successfully transported a structure on $\mathbb{N}$ to $\mathsf{Bin}\mathbb{N}$! This works for any structure we can think of, so we can transport proofs from the proof-oriented type to the computation-oriented one...

Program/proof transfer

But what about computations? Consider the following goal:[3]

$$\text{goal} : 2^{30} + 2^{30} \equiv 2^{31}$$
$$\text{goal} = ?$$

[3] A similar goal actually occurred in *A Computer-Algebra-Based Formal Proof of the Irrationality of* $\zeta(3)$ by Chyzak, et. al. in ITP 2014.

Program/proof transfer

But what about computations? Consider the following goal:[3]

$$\text{goal} : 2^{30} + 2^{30} \equiv 2^{31}$$
$$\text{goal} = ?$$

Trying to replace ? with refl will crash your computer! The problem is that it's expressed using unary numbers, with binary it's trivial to check...

If we replace the goal with the transported monoid structure on binary numbers we get:

$$\text{goal} : 100..00 +_{\text{Bin}\mathbb{N}} 100..00 \equiv 100..00$$
$$\text{goal} = ?$$

Will refl directly prove this goal?

[3] A similar goal actually occurred in *A Computer-Algebra-Based Formal Proof of the Irrationality of* $\zeta(3)$ by Chyzak, et. al. in ITP 2014.

No!

No!

The problem is that $+_{\mathsf{Bin}\mathbb{N}}$ is not the function we want to add binary numbers with... As it has been transported over from $\mathbb{N}$, the expression

$$x +_{\mathsf{Bin}\mathbb{N}} y$$

will reduce to

$$\mathsf{transport}\ \mathbb{N}{\equiv}\mathsf{Bin}\mathbb{N}\ (\mathsf{transport}\ (\mathsf{sym}\ \mathbb{N}{\equiv}\mathsf{Bin}\mathbb{N})\ x + \mathsf{transport}\ (\mathsf{sym}\ \mathbb{N}{\equiv}\mathsf{Bin}\mathbb{N})\ y)$$

So Agda will convert the numbers to unary, add them using $+$ for unary numbers and then transport back...

No!

The problem is that $+_{\mathsf{Bin}\mathbb{N}}$ is not the function we want to add binary numbers with... As it has been transported over from $\mathbb{N}$, the expression

$$x +_{\mathsf{Bin}\mathbb{N}} y$$

will reduce to

$$\mathsf{transport}\ \mathbb{N}{\equiv}\mathsf{Bin}\mathbb{N}\ (\mathsf{transport}\ (\mathsf{sym}\ \mathbb{N}{\equiv}\mathsf{Bin}\mathbb{N})\ x + \mathsf{transport}\ (\mathsf{sym}\ \mathbb{N}{\equiv}\mathsf{Bin}\mathbb{N})\ y)$$

So Agda will convert the numbers to unary, add them using $+$ for unary numbers and then transport back...

Conclusion: plain univalence doesn't give us what we want!

Program/proof transfer

What we really want to do is to transport the goal expressed using the *monoid* of unary numbers so that it uses the *monoid* of binary numbers instead...

Program/proof transfer

What we really want to do is to transport the goal expressed using the *monoid* of unary numbers so that it uses the *monoid* of binary numbers instead...

Let

$$\text{Monoid} = \Sigma[\ X \in \text{Type}\ ]\ (\text{MonoidStr}\ X)$$

we then want

$$(M\ N : \text{Monoid}) \to \text{MonoidEquiv}\ M\ N \to M \equiv N$$

where MonoidEquiv M N is the type of monoid-structure preserving equivalences

This can be proved by hand, but it gets hard for more complex structures...

Program/proof transfer

What we really want to do is to transport the goal expressed using the *monoid* of unary numbers so that it uses the *monoid* of binary numbers instead...

Let

$$\text{Monoid} = \Sigma[\ X \in \text{Type}\ ]\ (\text{MonoidStr}\ X)$$

we then want

$$(M\ N : \text{Monoid}) \to \text{MonoidEquiv}\ M\ N \to M \equiv N$$

where MonoidEquiv M N is the type of monoid-structure preserving equivalences

This can be proved by hand, but it gets hard for more complex structures...
So we need some form of SIP that proves this kind of theorem once and for all!

The Structure Identity Principle

The SIP is an informal principle that reasoning about mathematical structures should be invariant under equivalence/isomorphism of such structures

The Structure Identity Principle

The SIP is an informal principle that reasoning about mathematical structures should be invariant under equivalence/isomorphism of such structures

There are many different ways to formalize this principle in HoTT/UF:

- Coquand-Danielsson: *Isomorphism is equality*
- Aczel: *Homotopy Type Theory and the Structure Identity Principle*
- HoTT book chapter 9.8
- Awodey: *Structuralism, Invariance, and Univalence*
- Ahrens-Lumsdaine: Displayed categories
- Escardó: *Intro. to Univalent Foundations of Mathematics with Agda*
- Ahrens-North-Shulman-Tsementzis: *A Higher Structure Identity Principle*
- Buchholtz-Schipp von Branitz: *Displayed Univalent Reflexive Graphs*
- ...

Some of these are more categorical while others are more type theoretic

The Cubical Structure Identity Principle

We found Escardó's variation closest to our needs as it's:

1. Type theoretic
2. Natural to formalize in `Cubical Agda`
3. Modular (complex structures can be decomposed into simpler ones)

The last point is surprisingly important to make things practical—it allows us to automate almost all of the proofs whenever we consider a new structure!

The version we have formalized is based on Escardó's, but some key definitions are different to work better with the cubical machinery $\Rightarrow$ shorter proofs

The Cubical Structure Identity Principle

With this SIP a structure is just a function

$$S : \mathsf{Type} \to \mathsf{Type}$$

and the type of *S-structures* is defined as:

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

The Cubical Structure Identity Principle

With this SIP a structure is just a function

$$S : \mathsf{Type} \to \mathsf{Type}$$

and the type of *S-structures* is defined as:

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

An *S-structure-preserving equivalences* is a term ι : StrEquiv S:

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

The type of structure-preserving equivalences is then given by:

$$A \cong B = \Sigma[\ e \in \mathsf{fst}\ A \simeq \mathsf{fst}\ B\ ]\ (\iota\ A\ B\ e)$$

(For monoids we take $S =$ MonoidStr and ι states that e preserves ε and _·_)

The Cubical Structure Identity Principle

For which S and ι can we prove:

$$(A\ B : \mathsf{TypeWithStr}\ S) \to A \cong B \to A \equiv B$$

Using ua e we can easily get the fst part of the path, but for the snd part we need a suitable path-over

The Cubical Structure Identity Principle

For which S and ι can we prove:

$$(A\ B : \mathsf{TypeWithStr}\ S) \to A \cong B \to A \equiv B$$

Using ua e we can easily get the fst part of the path, but for the snd part we need a suitable path-over

With this in mind we define *univalent* structures as:

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

(This is where we deviate from Escardó: as he's working in HoTT/UF he cannot use PathP, so he defines an alternative *standard notion of structure* which is in fact equivalent to our UnivalentStr's)

The Cubical Structure Identity Principle

With this notion of UnivalentStr we can now prove our SIP:[4]

```
sip : UnivalentStr S ι → (A B : TypeWithStr S) → A ≃ B → A ≡ B
sip θ A B (e , φ) i = (p i , q i)
  where
  p : fst A ≡ fst B
  p = ua e

  q : PathP (λ i → S (p i)) (snd A) (snd B)
  q = θ e .fst φ
```

Note that we defined univalent structures so that the proof of sip would be easy. So in order to use it we now need to prove that various structures are univalent

[4] The function sip is in fact an equivalence, but for the applications that is not important.

The Cubical Structure Identity Principle

To prove that monoids are univalent observe that any MonoidStr on A is equivalent to an element of

$$\text{RawMonoidStr}\ A = A \times (A \to A \to A)$$

combined with an element of

$$\text{isMonoid} : \text{RawMonoidStr} \to \text{hProp}$$

The Cubical Structure Identity Principle

To prove that monoids are univalent observe that any MonoidStr on A is equivalent to an element of

$$\text{RawMonoidStr } A = A \times (A \to A \to A)$$

combined with an element of

$$\text{isMonoid} : \text{RawMonoidStr} \to \text{hProp}$$

Theorem (Escardó, Angiuli-Cavallo-M.-Zeuner)

Constants and function types are univalent structures.
Univalent structures are closed under _×_ and adding prop-valued axioms.

Corollary (Escardó, Angiuli-Cavallo-M.-Zeuner)

MonoidStr is univalent and monoid-isomorphisms are equivalent to paths.

Other algebraic structures follow exactly the same pattern!

Algebra demo!

(j.w.w. Felix Wellen)

Another example: matrices

Matrices form another example where we want to use program/proof transfer between proof-oriented and computation-oriented representations

Proof-oriented:

```
data Fin : ℕ → Type where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} (i : Fin n) → Fin (suc n)

FinMatrix : (A : Type) (m n : ℕ) → Type
FinMatrix A m n = Fin m → Fin n → A
```

Another example: matrices

Matrices form another example where we want to use program/proof transfer between proof-oriented and computation-oriented representations

Proof-oriented:

```
data Fin : ℕ → Type where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} (i : Fin n) → Fin (suc n)

FinMatrix : (A : Type) (m n : ℕ) → Type
FinMatrix A m n = Fin m → Fin n → A
```

Computation-oriented:

```
data Vec (A : Type) : ℕ → Type where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

VecMatrix : (A : Type) (m n : ℕ) → Type
VecMatrix A m n = Vec (Vec A n) m
```

Matrix demo!

Summary Cubical Structure Identity Principle

- Helps us characterize equality in Σ-types (usually a hard problem)
- Applies to a large class of algebraic and categorical structures, e.g. Noetherian local rings[5]
- The proofs that structures are univalent are modular and can to a large extent be automated
- Lets us transport programs and proofs between equivalent *structured* types
- This means that we can use proof-oriented types for developing theory and computation-oriented types for computing, leading to a *separation of concerns*

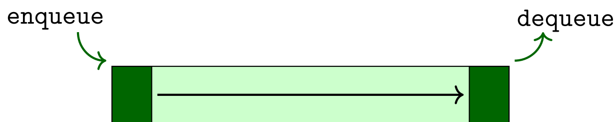
[5] See: `https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/HoTT-UF-Agda.html#noetherian-ring-sip`

Overview

1. Introduction: Computer Science & HoTT
2. Cubical Type Theory & Cubical Agda
3. A Cubical Structure Identity Principle
4. A Relational Structure Identity Principle
5. Conclusion

Queues

A *queue* is a classical datastructure where you can *enqueue* elements on the "back" and *dequeue* from the "front"



This makes them into a "first in, first out" (FIFO) datastructure: the first element added to the queue will be the first to be removed

Queues in Cubical Agda

Given A : Type, a simple queue interface is:

```
record QueueStr (A : Type) : Type₁ where
  constructor queue
  field
    Q : Type
    ∅ : Q
    enqueue   : A → Q → Q
    dequeue   : Q → Maybe (Q × A)
    setQ      : isSet Q
    deqEmpty : dequeue ∅ ≡ nothing
    dequeueEnqueue : (a : A) (q : Q) →
      dequeue (enqueue a q) ≡
      just (Maybe.rec (∅ , a) (λ {(q , b) → enqueue a q , b}) (dequeue q))
```

This interface is similar to the algebraic structures and it comes with a natural notion of queue-structure preserving equivalence. This is even a univalent structure and the SIP applies

Queues in Cubical Agda

Let's say that someone writes a

$$\text{computation} : (Q : \text{QueueStr}\ \mathbb{N}) \to \mathbb{N}$$

As Q is a variable, computation must typecheck for any implementation of Q. But to run computation, we must apply it to a concrete implementation

Queues in Cubical Agda

Let's say that someone writes a

$$\text{computation} : (Q : \text{QueueStr}\ \mathbb{N}) \to \mathbb{N}$$

As Q is a variable, computation must typecheck for any implementation of Q. But to run computation, we must apply it to a concrete implementation

Using the SIP we can easily prove that

$$\text{QueueEquiv}\ Q_1\ Q_2 \to \text{computation}\ Q_1 \equiv \text{computation}\ Q_2$$

This is great, but is it really enough?

Not really...

The simplest implementation of QueueStr is to use lists (omitting proofs):

$$
\begin{array}{l}
\mathsf{ListQueue} : (A : \mathsf{Type}) \to \mathsf{QueueStr}\ A \\
\mathsf{ListQueue}\ A = \mathsf{queue}\ (\mathsf{List}\ A)\ []\ _{::}_\ \mathsf{last}
\end{array}
$$

It is straightforward to prove that this is an implementation of the queue interface, but it's quite inefficient as dequeueing will traverse the whole queue to extract the last element

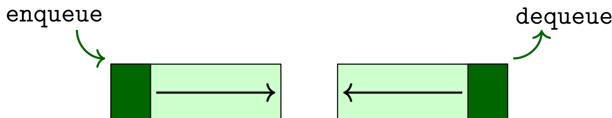
Not really...

The simplest implementation of QueueStr is to use lists (omitting proofs):

ListQueue : (*A* : Type) → QueueStr *A*
ListQueue *A* = queue (List *A*) [] _::_ last

It is straightforward to prove that this is an implementation of the queue interface, but it's quite inefficient as dequeueing will traverse the whole queue to extract the last element

A more efficient representation using *batched queues* can be found in the 1996 CMU PhD thesis of Chris Okasaki: *Purely Functional Data Structures*



Batched Queues in Cubical Agda

This can be implemented as:

```
BatchedQueue : (A : Type) → QueueStr A
BatchedQueue A =
  queue (List A × List A)
        ([] , [])
        (λ x (xs , ys) → fastcheck (x :: xs , ys))
        (λ { (_ , []) → 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)
```

Batched queues are a much faster implementation of queues, but it's more complex to reason about them

Relating List and Batched Queues

Consider the map:

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

This is queue-structure preserving, so we should now be able to reap the benefits of the SIP...

Relating List and Batched Queues

Consider the map:

$$\text{appendReverse} : \{A : \text{Type}\} \to \text{Q}\ (\text{BatchedQueue}\ A) \to \text{Q}\ (\text{ListQueue}\ A)$$
$$\text{appendReverse}\ (xs\ ,\ ys) = xs \mathbin{++} \text{reverse}\ ys$$

This is queue-structure preserving, so we should now be able to reap the benefits of the SIP...

But this is **not** an equivalence!

$$\text{appendReverse}([0],[1]) = [0,1] \qquad \text{appendReverse}([],[1,0]) = [0,1]$$

These two types are hence related by a many-to-one relationship and the SIP doesn't apply... ☹

The Problem of Representation Independence

> *"Representation independence: programs should not depend on the way stacks are represented, only on the behavior of stacks with respect to push and pop operations." [Mitchell, POPL'86]*

We now have two implementations of the queue interface which have the same behavior with respect to enqueue/dequeue according to appendReverse, but we cannot equate them using the SIP as it is only about equivalent/isomorphic types

The Problem of Representation Independence

> *"Representation independence: programs should not depend on the way stacks are represented, only on the behavior of stacks with respect to push and pop operations." [Mitchell, POPL'86]*

We now have two implementations of the queue interface which have the same behavior with respect to enqueue/dequeue according to appendReverse, but we cannot equate them using the SIP as it is only about equivalent/isomorphic types

In fact, in CS there is a wide range of relations that are respected by quantification over (structured) types — this is the realm of *parametricity theorems*

So can HoTT/UF really say anything about all of these applications?

Yes!

It turns out that the solution is very simple using HITs:

```
data BatchedQueueHIT (A : Type) : Type where
  Q⟨_,_⟩ : List A → List A → BatchedQueueHIT A
  tilt : ∀ xs ys a → Q⟨ xs ++ [ a ] , ys ⟩ ≡ Q⟨ xs , ys ++ [ a ] ⟩
  squash : isSet (BatchedQueueHIT A)
```

The tilt constructor lets us prove that

$$Q\langle[0],[1]\rangle \equiv Q\langle[],[1,0]\rangle$$

So we can define a new appendReverse function from BathedQueueHIT to ListQueue which is an equivalence! ☺

Another example: Finite multisets

A typical interface for finite multisets FMSetStr$A$ would contain: $\emptyset$, insert, $\in$, count, $\cup$, $\cap$, ..., and suitable axioms

Possible implementations are List A or the more efficient association lists: List $(A \times \mathbb{N})$

Another example: Finite multisets

A typical interface for finite multisets FMSetStr$A$ would contain: $\emptyset$, insert, $\in$, count, $\cup$, $\cap$, ..., and suitable axioms

Possible implementations are List A or the more efficient association lists: List $(A \times \mathbb{N})$

For example the integer set $\{1, -2, 1, 3, 1, 3, -5\}$ could either be represented (up to permutations) by $[1, -2, 1, 3, 1, 3, -5]$ or $[(1, 3), (-2, 1), (3, 2), (-5, 1)]$

If a concrete set has very many repeated elements the association list representation is much more efficient, but instantiating the interface with it is once again more complicated

Finite multisets

Clearly these two types are in a many-to-many relationship, so the SIP doesn't directly apply... But we can once again fix this by adding some path constructors:

```
data FMSet (A : Type) : Type where
  []    : FMSet A
  _::_  : A → FMSet A → FMSet A
  comm  : ∀ x y xs → x :: y :: xs ≡ y :: x :: xs
  trunc : isSet (FMSet A)
```

Finite multisets

Clearly these two types are in a many-to-many relationship, so the SIP doesn't directly apply... But we can once again fix this by adding some path constructors:

```
data FMSet (A : Type) : Type where
  []    : FMSet A
  _::_  : A → FMSet A → FMSet A
  comm  : ∀ x y xs → x :: y :: xs ≡ y :: x :: xs
  trunc : isSet (FMSet A)
```

```
data AssocList (A : Type) : Type where
  ⟨⟩ : AssocList A
  ⟨_,_⟩::_ : A → ℕ → AssocList A → AssocList A
  per : ∀ a b m n xs → ⟨ a , m ⟩:: ⟨ b , n ⟩:: xs ≡ ⟨ b , n ⟩:: ⟨ a , m ⟩:: xs
  agg : ∀ a m n xs → ⟨ a , m ⟩:: ⟨ a , n ⟩:: xs ≡ ⟨ a , m + n ⟩:: xs
  del : ∀ a xs → ⟨ a , 0 ⟩:: xs ≡ xs
  trunc : isSet (AssocList A)
```

Generalization?

These two examples suggest that there might be something more general going on

Question: under what circumstances can we make a relation between two structured types into an equality?

Answering this let us develop a *relational* structure identity principle which in turn let us apply it to many-to-many correspondences

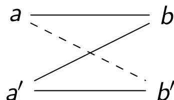
Zigzag-complete Relations

Definition (Zigzag-completeness)

Let $A\ B$: Type. A relation $R : A \to B \to$ hProp is *zigzag-complete* if

$$\forall\ \{a\ a' :\ A\}\ \{b\ b' :\ B\} \to R\ a\ b \to R\ a'\ b \to R\ a'\ b' \to R\ a\ b'$$

Pictorially we get:



Quasi-equivalence relations

Definition (QER)

We say that $R : A \to B \to \text{hProp}$ is a *quasi-equivalence relation* (QER) if

1. R is zigzag-complete
2. We have $\text{fwd} : A \to B$ s.t. $\forall\ a \to R\ a\ (\text{fwd}\ a)$
3. We have $\text{bwd} : B \to A$ s.t. $\forall\ b \to R\ (\text{bwd}\ b)\ b$

Quasi-equivalence relations

Definition (QER)

We say that $R : A \to B \to \mathsf{hProp}$ is a *quasi-equivalence relation* (QER) if

1. R is zigzag-complete
2. We have $\mathsf{fwd} : A \to B$ s.t. $\forall\, a \to R\, a\, (\mathsf{fwd}\ a)$
3. We have $\mathsf{bwd} : B \to A$ s.t. $\forall\, b \to R\, (\mathsf{bwd}\ b)\, b$

We furthermore define

$$\mathsf{R}^{\to} : A \to A \to \mathsf{Type} \qquad\qquad \mathsf{R}^{\leftarrow} : B \to B \to \mathsf{Type}$$

$$\mathsf{R}^{\to}\ a_1\ a_2 = R\ a_1\ (\mathsf{fwd}\ a_2) \qquad\qquad \mathsf{R}^{\leftarrow}\ b_1\ b_2 = R\ (\mathsf{bwd}\ b_1)\ b_2$$

Theorem (Angiuli-Cavallo-M.-Zeuner)

Let $A\ B : \mathit{Type}$ and $R : A \to B \to \mathit{hProp}$ a QER, then we have that

1. *$R^{\leftarrow}$ and $R^{\to}$ are proposition-valued equivalence relations on A and B.*
2. *fwd and bwd induce an equivalence $e^{\leftrightarrow} : A \,/\, R^{\leftarrow} \simeq B \,/\, R^{\to}$*

The Relational Structure Identity Principle

What remains now is to take into account a structure $S : \mathsf{Type} \to \mathsf{Type}$. For this we define a notion of structured relation:

$$\mathsf{StrRel} : (S : \mathsf{Type} \to \mathsf{Type}) \to \mathsf{Type}_1$$
$$\mathsf{StrRel}\ S = \{X\ Y : \mathsf{Type}\} \to (X \to Y \to \mathsf{Type}) \to (S\ X \to S\ Y \to \mathsf{Type})$$

The Relational Structure Identity Principle

What remains now is to take into account a structure $S : \mathsf{Type} \to \mathsf{Type}$. For this we define a notion of structured relation:

$$\begin{aligned}&\mathsf{StrRel} : (S : \mathsf{Type} \to \mathsf{Type}) \to \mathsf{Type}_1\\&\mathsf{StrRel}\ S = \{X\ Y : \mathsf{Type}\} \to (X \to Y \to \mathsf{Type}) \to (S\ X \to S\ Y \to \mathsf{Type})\end{aligned}$$

This lets us sketch the statement of the relational version of the SIP:

Theorem (Angiuli-Cavallo-M.-Zeuner)

Let $\rho : \mathit{StrRel}\ S$ be a suitable structured relation, and let $(A\ ,\ s) : \mathit{TypeWithStr}\ S$ and $(B\ ,\ t) : \mathit{TypeWithStr}\ S$.
For any QER $R : A \to B \to \mathit{Type}$ structured by some $r : (\rho\ R)\ s\ t$, there exists structures $\bar{s} : S\,(A\,/\,R^{\leftarrow})$ and $\bar{t} : S\,(B\,/\,R^{\rightarrow})$ which are related by ρ according to a path induced $e^{\leftrightarrow}$.

The Relational Structure Identity Principle

We have similar lemmas as for the cubical SIP for proving that structured relations are closed under various type formers, making many proofs automatic

We have applied this general framework to obtain both the queues and finite multisets as special cases

For technical details see the paper!

Internalizing Representation Independence with Univalence
Carlo Angiuli, Evan Cavallo, Anders Mörtberg, Max Zeuner
`https://arxiv.org/abs/2009.05547`
(to appear in POPL'21)

Overview

1. Introduction: Computer Science & HoTT
2. Cubical Type Theory & Cubical Agda
3. A Cubical Structure Identity Principle
4. A Relational Structure Identity Principle
5. Conclusion

Summing up

The SIP and HITs are useful for traditional CS application like program/proof transfer and representation independence

Everything in the talk can be done in HoTT, but by working in Cubical Agda we don't loose computational content

Summing up

The SIP and HITs are useful for traditional CS application like program/proof transfer and representation independence

Everything in the talk can be done in HoTT, but by working in Cubical Agda we don't loose computational content

Current/future work:

- Displayed SIP (c.f. DURGs)
- More substantial examples: FP style monads, dictionaries, trees, automata, ...
- Applications to computing $\mathbb{Z}$-cohomology?
- Applications in algebra with a view towards algebraic geometry (wip: ring localization)

Conclusion: Computer Science & HoTT

Question: how to get CS people working on formalization interested in HoTT?

Conclusion: Computer Science & HoTT

Question: how to get CS people working on formalization interested in HoTT?

There's still quite a lot of skepticism and confusion... But one of the reviewers of our paper said the following:

> *I've watched the HoTT movement as an outsider. Supporters have argued all along that the most compelling application to computer science is convenient reasoning about substituting different data-structure implementations. However, I've never felt like the details were presented clearly. [...]*
>
> *This paper is the closest I've seen to finally revealing the details. Thus, from my baseline position of skepticism for this topic, I was surprised to find myself reading enthusiastically.*

Thank you for your attention!

`https://github.com/agda/cubical/`