The Structure Identity Principle in Cubical Agda

Anders Mörtberg

Memorial Conference for Erik Palmgren, November 20, 2020

Introduction

Erik was not only doing great mathematics, he also formalized it in Agda and Coq

His last paper, *From type theory to setoids and back*, is about a formalization of a setoid model of Martin-Löf extensional type theory with universes in Agda

Introduction

Erik was not only doing great mathematics, he also formalized it in Agda and Coq

His last paper, *From type theory to setoids and back*, is about a formalization of a setoid model of Martin-Löf extensional type theory with universes in Agda

```
Den tis 27 aug. 2019 19:43 Erik Palmgren <palmgren@math.su.se> skrev:

Hello
could you please do me favor and check whether

http://staff.math.su.se/palmgren/MLTT-and-setoids-2019-08-26.zip

loads in some recent standard version of Agda? (this is 2.5.2)

Loading V-model-all-rules.agda verifies all relevant files. Should take about 10 minutes.

If it works I will list the version, otherwise I stick to 2.5.2.

Best regards
Erik
```

The V-model-all-rules.agda file and dependencies is 16000 lines of code!

Homotopy Type Theory and Univalent Foundations

A new foundation of mathematics, initially developed by Vladimir Voevodsky, building on ideas from type theory, (higher) category theory and homotopy theory

Offers an alternative to setoids, both for formalization and constructive maths

Homotopy Type Theory and Univalent Foundations

A new foundation of mathematics, initially developed by Vladimir Voevodsky, building on ideas from type theory, (higher) category theory and homotopy theory

Offers an alternative to setoids, both for formalization and constructive maths

The two main innovations are

- Univalence:

$$\mathsf{ua} : \forall\, A\, B \rightarrow A \simeq B \rightarrow A \equiv B$$

- Higher inductive types (HITs): types with path-constructors; quotient types

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

Cubical type theory gives these computational content

Cubical Type Theory & Cubical Agda

Extension of the dependently typed programming language Agda with *cubical* features:

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

Cubical Type Theory & Cubical Agda

Extension of the dependently typed programming language Agda with *cubical* features:

- Interval and Path types (I, PathP, _≡_, ...)
- 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!

agda/cubical library

We are developing an open-source library:

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

Currently: $\sim$ 39000 LOC, $\sim$ 400 files, 47 contributors

It contains a variety of results about synthetic homotopy theory, $\mathbb{Z}$-cohomology, constructive algebra, datastructures...

Many proofs are simpler compared to their HoTT counterparts, and everything is fully constructive so that we can run the programs/proofs

This work

With Carlo Angiuli, Evan Cavallo and Max Zeuner we have studied how to formalize (set level) constructive mathematics and computer science in `Cubical Agda`, without using setoids

This work

With Carlo Angiuli, Evan Cavallo and Max Zeuner we have studied how to formalize (set level) constructive mathematics and computer science in `Cubical Agda`, without using setoids

In particular, we have focused on the following problems:

1. Program/proof transfer: transport programs and proofs between 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)

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]*

Example: Monoid of unary and binary numbers

Consider the following goal[1] expressed using the additive monoid on $\mathbb{N}$:

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

[1] 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.

Example: Monoid of unary and binary numbers

Consider the following goal[1] expressed using the additive monoid on $\mathbb{N}$:

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

Trying to replace ? with refl will crash your computer!

[1]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.

Example: Monoid of unary and binary numbers

Consider the following goal[1] expressed using the additive monoid on $\mathbb{N}$:

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

Trying to replace ? with refl will crash your computer!

If we could transport goal to the equivalent monoid on $\text{Bin}\mathbb{N}$ we would get:

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

Replacing ? with refl will now terminate in no time!

[1] 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.

Example: Monoid of unary and binary numbers

So what we want to do is to construct a *path* from the monoid on $\mathbb{N}$ to the one on $\mathrm{Bin}\mathbb{N}$ and transport along it

Example: Monoid of unary and binary numbers

So what we want to do is to construct a *path* from the monoid on $\mathbb{N}$ to the one on $\text{Bin}\mathbb{N}$ and transport along it

Let

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

The lemma we need is then:

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

where $\text{MonoidEquiv}\ M\ N$ is the type of MonoidStr preserving equivalences

Example: Monoid of unary and binary numbers

So what we want to do is to construct a *path* from the monoid on $\mathbb{N}$ to the one on $\mathrm{Bin}\mathbb{N}$ and transport along it

Let

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

The lemma we need is then:

$$\mathrm{MonoidPath} : \forall\ (M\ N : \mathrm{Monoid}) \to \mathrm{MonoidEquiv}\ M\ N \to M \equiv N$$

where $\mathrm{MonoidEquiv}\ M\ N$ is the type of $\mathrm{MonoidStr}$ preserving equivalences

With this we can easily do the above example and solve our goal, but we can also transport any *proof* of a property for M to a proof of that property for N

Example: Monoid of unary and binary numbers

So what we want to do is to construct a *path* from the monoid on $\mathbb{N}$ to the one on $\text{Bin}\mathbb{N}$ and transport along it

Let

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

The lemma we need is then:

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

where MonoidEquiv M N is the type of MonoidStr preserving equivalences

With this we can easily do the above example and solve our goal, but we can also transport any *proof* of a property for M to a proof of that property for N

MonoidPath can be proved by hand using ua, but it gets hard for more complex structures... So we need some form of SIP that helps us prove this kind of results!

The Structure Identity Principle

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

The Structure Identity Principle

The SIP is an informal principle that reasoning about mathematical structures should be invariant under 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 formalization 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 the structure)

The Cubical Structure Identity Principle

For which S and ι can we prove:

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

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 $\mathsf{ua}\ e$ 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 is 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)

The Cubical Structure Identity Principle

With this notion of UnivalentStr we can now prove our SIP:[2]

```
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

[2]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 ⇒ automation!

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]*

In CS it is very common to consider structured relations between types which are weaker than isomorphisms; e.g. one-to-many or many-to-many correspondences

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]*

In CS it is very common to consider structured relations between types which are weaker than isomorphisms; e.g. one-to-many or many-to-many correspondences

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

The SIP lets us identify isomorphic structured types, but can it be used also to prove relational results?

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 $(A \times \mathbb{N})$

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 multiset $\{1, -2, 1, 3, 1, 3, -5\}$ 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 is more complicated

Finite multisets

Clearly these two types are in a many-to-many relationship, so the SIP doesn't directly apply...

Finite multisets

Clearly these two types are in a many-to-many relationship, so the SIP doesn't directly apply... But we can fix this using HITs:

```
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?

Observation: there are many examples where we start with two related types and by adding path constructors they become isomorphic

Generalization?

Observation: there are many examples where we start with two related types and by adding path constructors they become isomorphic

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

Answering this let us develop a *relational* SIP which in turn makes it applicable to many-to-many correspondences from traditional CS examples

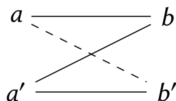
Zigzag-complete Relations

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:



Intuition: symmetric and transitive heterogeneous relations

Quasi-equivalence relations

Definition (Quasi-equivalence relations)

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

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

Quasi-equivalence relations

Definition (Quasi-equivalence relations)

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 $\text{fwd} : A \to B$ such that $\forall\ a \to R\ a\ (\text{fwd}\ a)$
3. We have $\text{bwd} : B \to A$ such that $\forall\ b \to R\ (\text{bwd}\ b)\ b$

We furthermore define

$$\mathrm{R}^A : A \to A \to \text{Type} \qquad \mathrm{R}^A\ a_1\ a_2 = R\ a_1\ (\text{fwd}\ a_2)$$

$$\mathrm{R}^B : B \to B \to \text{Type} \qquad \mathrm{R}^B\ b_1\ b_2 = R\ (\text{bwd}\ b_1)\ b_2$$

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

The Relational Structure Identity Principle

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

$$\text{StrRel}\ S = \{A\ B : \text{Type}\} \to (A \to B \to \text{Type}) \to (S\ A \to S\ B \to \text{Type})$$

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:

$$\mathsf{StrRel}\ S = \{A\ B : \mathsf{Type}\} \to (A \to B \to \mathsf{Type}) \to (S\ A \to S\ B \to \mathsf{Type})$$

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

Theorem (Angiuli-Cavallo-M.-Zeuner)

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

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 queues and finite multisets as special cases

For technical details see:

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

Summing up

The SIP helps with characterizing equality in Σ-types (usually a hard problem) for a large class of algebraic and categorical structures

The proofs that structures are univalent are very modular and can to be automated

This lets us transport programs and proofs between equivalent *structured* types

Summing up

The SIP helps with characterizing equality in Σ-types (usually a hard problem) for a large class of algebraic and categorical structures

The proofs that structures are univalent are very modular and can to be automated

This lets us transport programs and proofs between equivalent *structured* types

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)

Thank you for your attention!

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