Isomorphisms and Interoperability

Catherine Dubois

ENSIIE, Samovar, INRIA, France

Contributors : Raphaël Cauderlier (Nomadic labs, France), Alain Giorgetti (Univ. Franche-Comté, France) and Nicolas Magaud (Univ. Starsbourg, France)

Isomorphisms

In Maths, common practice to have several constructions for the same objects, identified later modulo isomorphisms

In Computer Science, in proof assistants, also common practice to have several representations for the same objects

- formal verification of Maths constructions
- alternative representations to dependent types
- reuse of formal developments in the same or another proof assistant (to bridge two different representations)
- data refinement (one rep. well suited for proving while another one allows more efficient representation, e.g. Peano vs binary natural numbers)
- random generation (testing before proving)

Related issues

1. Prove that the several representations are isomorphic (transformation functions and roundtrip lemmas)
2. Transfer theorems from one representation to another one
   Definition : let A, B, two isomorphic structures, ϕ_A a formula on A, ϕ_B the corresponding formula on B, $\phi_A \Rightarrow \phi_B$: a transfer theorem,
   e.g. `transfer` tactic in Isabelle (Huffman, Kuncar), `transfer` tactic in Coq (Zimmermann, Herbelin), `transfer` tactic in Dedukti (Cauderlier)

Proof Interoperability

Motivations

- Proof development is *expensive*
  - 4-color theorem, Kepler conjecture, Feit-Thomson theorem
- Proof assistants are *specializing*
  - Counterexamples, proof by reflection, decision procedures, ...

Obstacles

- Logical problem :
  We need to combine the logics of PA A and PA B in a consistent way.
- Mathematical problem : L and L are not identical
  Theories such as arithmetic are independently defined in System A and System B.
  We need to identify similar concepts (through isomorphisms)

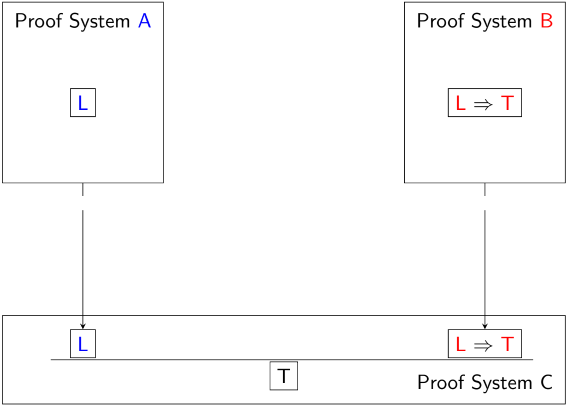
In the rest of the talk

We need to identify similar concepts (through isomorphisms)

- Illustration 1 : Composite checked proofs in DEDUKTI (presented in detail inTetrapod 2018)
  Use case : a composite (HOL/Coq) checked proof of correctness of Eratosthenes Sieve

- Illustration 2 : Practical isomorphisms for families of objects in Coq
  Use case : transfer a theorem about a family f from System A (here Coq) to System B
  - f is defined using a dependent pair (rec_P) in A while B lacks dependent types .
  - fortunately there exists a simpler isomorphic type P that can implemented in B.
  - so we have to provide the transformation functions and roundtrip lemmas in A and then go ahead with our favorite framework for interoperability.

Illustration 1 : Composite checked proofs with DEDUKTI [CD17]



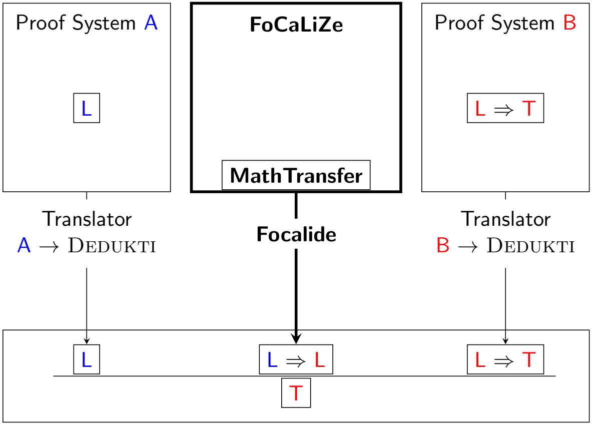
[CD17]Raphaël Cauderlier, Catherine Dubois : FoCaLiZe and Dedukti to the Rescue for Proof Interoperability. ITP 2017

Composite checked proofs with DEDUKTI [CD17]

DEDUKTI (`http://dedukti.gforge.inria.fr/`) : a universal proof checker / logical framework developed by Dowek and his group based on $\lambda\Pi$-calculus modulo theory = dependent types *à la* LF + (user-defined) rewriting rules

DEDUKTI can check proofs from iProverModulo (resolution proofs) : Zenon modulo (f.o. tableaux proofs), HOL provers (open theory format), Matita and Coq (CIC proofs), FoCaLiZe, thanks to translators.

Composite checked proofs with DEDUKTI [CD17]



MathTransfer = a FoCaLiZe library of transfer theorems about natural number arithmetic

Composite checked proofs with DEDUKTI [CD17]

- A = HOL (OpenTheory)
- B = Coq
- T = correctness of the Sieve of Eratosthenes
- L = prime divisor lemma

$$\mathrm{L} := \forall n \neq 1.\ \exists p.\ \mathrm{prime}(p) \wedge p \mid n$$

proved in HOL/OpenTheory `natural-prime` library

Illustration 2 : Isomorphic representations for families of objects in Coq [DMG22]

In this work :

- study of several families related to lambda terms
- family = subset of a larger type = aka PVS *predicate subtypes*
- for each family 2 different *isomorphic* representations, roundtrip theorems
- implemented in Coq
- random generators (using QuickChick, *testing before proving*)
- functors to deal with isomorphic representations for some families of objects

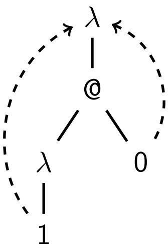
[DMG22] Catherine Dubois, Nicolas Magaud, Alain Giorgetti. Pragmatic Isomorphism Proofs Between Coq Representations : Application to Lambda-Term Families. In *TYPES 2022 : 11 :1-11 :19*

Pure lambda terms in de Bruijn form

$$T ::= \mathbb{N} \mid \lambda\, T \mid T\ T$$

Implemented in Coq with an inductive type :
unary-binary trees, aka labeled Motzkin trees

```
Inductive lmt : Set :=
| var : nat → lmt
| lam : lmt → lmt
| app : lmt → lmt → lmt.
```



for
$\lambda x.(\lambda y.x)\ x$

```
Definition ex1 := lam (app (lam (var 1)) (var 0)).
```

Motzkin trees

A Motzkin tree is a rooted ordered tree built from binary nodes, unary nodes and leaf nodes.

```
Inductive motzkin : Set :=
| v : motzkin
| l : motzkin → motzkin
| a : motzkin → motzkin → motzkin.
```

Closable Motzkin trees

A closable Motzkin tree is the skeleton of a closed lambda-term.

A Motzkin tree is a skeleton of a closed lambda term if and only if it exists at least one λ binder on each path from the leaf to the root. [BT17]

```
Fixpoint is_closable (mt: motzkin) :=
 match mt with
 | v ⇒ False
 | l m ⇒ True
 | a m1 m2 ⇒ is_closable m1 ∧ is_closable m2
 end.
```

```
    a
   / \
  l   l
  |   |
  l   a
  |  / \
  v v   v
```

a closable Motzkin tree

```
    a
   / \
  l   a
  |  / \
  l v   v
  |
  v
```

a non closable Motzkin tree

[BT17] Olivier Bodini and Paul Tarau. On uniquely closable and uniquely typable skeletons of lambda terms. In *Logic-Based Program Synthesis and Transformation, LOPSTR 2017*

From a representation to another

```
Record rec_closable : Type := Build_rec_closable {
  closable_struct :> motzkin;
  closable_prop : is_closable closable_struct
}.
```

$$\texttt{rec\_closable2closable} \downarrow \quad \uparrow \texttt{closable2rec\_closable}$$

(two roundtrip lemmas proved in Coq)

```
Inductive closable :=
| La : motzkin → closable
| Ap : closable → closable → closable.
```

From a representation to another, an example

```
   a
  / \
 l   l
 |   |
 l   a
 |  / \
 v v   v
```

+ a proof that it's a closable Motzkin tree

`rec_closable2closable` $\downarrow$ $\uparrow$ `closable2rec_closable`

```
    Ap
   /  \
 La    La
 |     |
 l     a
 |    / \
 v   v   v
```

Random generators

We use QuickChick (Coq port of Haskell QuickCheck) to test a Coq conjecture before proving it

- Executable properties and random generators needed !
- With the help of QuickChick combinators, write (verified) random generators (sometimes derived)

We provide random generators for

- Motzkin trees (derived from `motzkin` def.)
- closable Motzkin trees (obtained by filtering, not efficient)
- closable Motzkin trees (handwritten)
- closable Motzkin trees of type `rec_closable`
- objects of type `closable` (derived from `closable` def.)

```
(** ** Tests for [closable2rec_closableK] *)
QuickCheck (sized (fun n⇒
  forAll (gen_closable n) (fun c⇒
  (rec_closable2closable (closable2rec_closable c)) =? c)))
(*  +++ Passed 10000 tests  *)
```

Uniquely Closable Motzkin trees

A Motzkin tree is uniquely closable if it is the skeleton of a unique closed lambda-term.

A Motzkin tree is uniquely closable if and only if exactly one lambda binder/unary binder is available above each of its leaf nodes. [BT17, Prop. 4]

```
Definition is_ucs: motzkin → Prop := ....

Record rec_ucs : Type := Build_rec_ucs {
  ucs_struct : motzkin;
  ucs_prop : is_ucs ucs_struct
}.
```

$$\texttt{rec\_ucs2ucs} \downarrow \quad \uparrow \texttt{ucs2rec\_ucs}$$

(two roundtrip lemmas proved in Coq)

```
Inductive ca :=
| V : ca
| B : ca → ca → ca.
```

```
Inductive ucs :=
| L : ca → ucs
| A : ucs → ucs → ucs.
```

And random generators for `rec_ucs`, `ca` and `ucs`

Characterization of closable Motzkin trees

A Motzkin tree is the skeleton of a closed λ-term
if and only if
it exists at least one λ-binder on each path from the leaf to the root [BT17, Proposition 2]

How to formalize closed λ-terms ?

"to be closed" cannot be defined recursively on the structure of labeled Motzkin trees : $(\lambda\, t)$ can be closed for terms t that are not closed themselves
$\leadsto$ extension to *m-open terms*

The lambda term t is *m-open* if the term $(\lambda \ldots \lambda\, t)$ with m abstractions before t is closed, aka. a lambda term containing at most m distinct free variables.

A closed lambda term is defined as a 0-open term.

m-open λ-terms, formally

```
Fixpoint is_open (m: nat) (t: lmt) : Prop :=
  match t with
  | var i ⇒ i < m
  | lam t1 ⇒ is_open (S m) t1
  | app t1 t2 ⇒ is_open m t1 ∧ is_open m t2
  end.

Record rec_open (m:nat) : Set := Build_rec_open {
  open_struct :> lmt;
  open_prop : is_open m open_struct
}.
```

`rec_open2open` ↓ ↑ `open2rec_open`

(two roundtrip lemmas proved in Coq)

```
Inductive open : nat → Set :=
| open_var : ∀ (m i:nat), i < m → open m
| open_lam : ∀ (m:nat), open (S m) → open m
| open_app : ∀ (m:nat), open m → open m → open m.
```

A general framework to prove isomorphisms

- Generic interface : an abstract representation of two datatypes

```
Module Type family.
  Parameter T : Set.
  Parameter is_P : T → Prop.
  Parameter P : Set.
  Parameter T2P : ∀ (x:T), is_P x → P.
  Parameter P2T : P → T.
  Parameter is_P_lemma : ∀ v, is_P (P2T v).
  Parameter P2T_is_P :
    ∀ (t : T) (H : is_P t), P2T (T2P t H) = t.
  Parameter proof_irr :
    ∀ x (p1 p2:is_P x), p1 = p2.
End family.
```

- Functor parametrized by a module of type `family`

Closable and uniquely closable Motzkin trees as two instances

Abstraction	Closable MT	Uniquely Closable MT
`T`	`motzkin`	`motzkin`
`is_P`	`is_closable`	`is_ucs`
`P`	`closable`	`ucs`
`T2P`	`motzkin2closable`	`motzkin2ucs`
`P2T`	`closable2motzkin`	`ucs2motzkin`
`is_P_lemma`	automatically proved using Ltac	
`P2T_is_P`	automatically proved using Ltac	
`proof_irr`	`PI_is_closable`	`PI_is_ucs`
`rec_P`	automatically derived in the functor	
`rec_P2P`	automatically derived in the functor	
`P2rec_P`	automatically derived in the functor	
`P2rec_PK`	automatically derived in the functor	
`rec_P2PK`	automatically proved using Ltac	

Random generators

3 interfaces for random generators, parametrized by a `family` module and 3 functors

- one pair (interface,functor) to derive `gen_P_filter` from `gen_T` and an executable version of `is_P`
- one pair to derive `gen_P_rec` from `gen_P` using the transformation `P_rec2P`
- one pair to derive `gen_P` from `gen_P_rec` using the transformation `T_rec2P`

```
Module Type family_gen3 (Import f : family).
  Parameter gen_P : nat→ G P.
End family_gen3.

Module genfamily3(Import f : family)(Import g : family_gen3 f)
 (Import facts : equiv_sig f).
  Definition gen_rec_P n : G rec_P :=
  do! p← gen_P n;
  returnGen (P2rec_P p).
End genfamily3.
```

Discussion

In use case 1, MathTransfer is an external tool providing the bridge while in use case 2, it is the responsability of System A (or System B) to provide the bridge.

Could these approches be generalized and pushed further to make a bridge through systems ?