# Pragmatic Isomorphism Proofs Between Coq Representations: Application to Lambda-Term Families

Catherine Dubois
Samovar, ENSIIE, 1 square de la résistance, 91025 Évry-Courcouronnes, France

Nicolas Magaud
Lab. ICube UMR 7357 CNRS Université de Strasbourg, 67412 Illkirch, France

Alain Giorgetti
Université de Franche-Comté, CNRS, Institut FEMTO-ST, F-25030 Besançon, France

## Abstract

There are several ways to formally represent families of data, such as lambda terms, in a type theory such as the dependent type theory of Coq. Mathematical representations are very compact ones and usually rely on the use of dependent types, but they tend to be difficult to handle in practice. On the contrary, implementations based on a larger (and simpler) data structure combined with a restriction property are much easier to deal with.

In this work, we study several families related to lambda terms, among which Motzkin trees, seen as lambda term skeletons, closable Motzkin trees, corresponding to closed lambda terms, and a parameterized family of open lambda terms. For each of these families, we define two different representations, show that they are isomorphic and provide tools to switch from one representation to another. All these datatypes and their associated transformations are implemented in the Coq proof assistant. Furthermore we implement random generators for each representation, using the QuickChick plugin.

**2012 ACM Subject Classification** Theory of computation → Lambda calculus; Theory of computation → Logic and verification; Theory of computation → Type theory

**Keywords and phrases** Data Representations, Isomorphisms, dependent Types, formal Proofs, random Generation, lambda Terms, Coq

**Digital Object Identifier** 10.4230/LIPIcs.TYPES.2022.11

**Supplementary Material** *Software (Source Code)*: `https://github.com/magaud/postTYPES2022`
archived at `swh:1:dir:f01d28048087438c173ad32a5536f79755285465`

**Funding** *Alain Giorgetti*: Supported by the EIPHI Graduate School (contract ANR-17-EURE-0002).

## 1 Introduction

Choosing the most appropriate implementation of mathematical objects to perform computations and proofs is challenging. Indeed, efficient (well-suited for computations) representations are often difficult to handle when it comes to proving properties of these objects. Conversely, well-suited representations for proofs often have fairly poor performances when it comes to computing. The simplest example is the implementation of natural numbers. Using a unary representation, proofs (especially inductive reasoning) are easy to carry out but computing is highly inefficient. Using a binary representation makes computations faster, however it is more difficult to use reasoning principles such as the induction principle on natural numbers.

In the field of $\lambda$-calculus, representations that are closest to mathematics are usually implemented using dependent types. This makes them easily readable and understandable by mathematicians. However it is rather challenging and requires a strong background in functional programming and theorem proving to handle them smoothly. Representations based on a larger type (a non-dependent one) and a restriction property are easier to handle in practice, but less intuitive. In addition, one needs to take extra care to make sure that the combination of the larger type and the restriction property exactly represents the expected objects.

Overall there is no perfect representation for a given mathematical object. To overcome this challenge, we propose to deal with several different isomorphic representations of families of mathematical objects simultaneously. To do that, we present a rigorous methodology to partially automate the construction of the transformation functions between two isomorphic representations and prove these transformations correct. We apply these techniques to some families of objects related to $\lambda$-calculus, namely closable Motzkin trees, uniquely closable Motzkin trees and $m$-open $\lambda$-terms.

Our first examples revisit an article of Bodini and Tarau [3] in which they define Prolog generators for closed lambda terms and their skeletons seen as Motzkin trees, efficient generators for closable and uniquely closable skeletons and study their statistical properties. Our contributions are to formalize in Coq these different notions, prove the equivalence of several definitions that underlie the generators designed by Bodini and Tarau, and write random generators to be used with QuickChick [12]. We then extend the discourse to a parameterized family of open $\lambda$-terms, named $m$-open $\lambda$-terms. All the considered representations, transformations between isomorphic representations and isomorphism proofs are formalized[1] in the Coq proof assistant [2, 7]. We propose some generic tools to help setting up the correspondence between two isomorphic types more easily. We hope such a methodology could be reused to deal with other families of objects, having different and isomorphic representations.

**Related Work.** Dealing with various isomorphic representations of the same mathematical objects is a common issue in computer science. Research results span from theoretical high-level approaches such as homotopy type theory [19] or cubical type theory [5] to more pragmatic proposals such as ours. In the context of formal specifications and proofs about mathematical concepts, several frameworks have been proposed to deal with several types and their transformation functions. A seminal work on changing (isomorphic) data representation was implemented by Magaud [14] as a plugin for Coq in the early 2000s. In this approach, the transformation functions were provided by the user and only the proofs were ported. Here, we aim at helping the programmer to write the transformation functions as well as their proofs of correctness. In [6], Cohen et al. focus on refining from abstract representations, well-suited for reasoning, to computationally well-behaved representations. In our work, both representations are considered of equal importance, and none of them is preferred. Finally, our work is closely related to the concept of views, introduced by Wadler in [20] and heavily used in the dependently-typed programming language Epigram [16]. In this approach, operations are made independent of the actual implementation of the types they work on. Pattern-matching on an element of type $A$ can be carried out following the structure of the type $B$ provided $A$ and $B$ are isomorphic types. The correspondence functions we shall implement in this article provide an example of a concrete implementation of views.

Regarding random generators and enumerators, Paraskevopoulou et al. [17] recently proposed a new framework, on top of the QuickChick testing tool for Coq. It allows to automatically derive such generators by extracting the computational contents from inductive relations.

[1] The Coq code is available at `https://archive.softwareheritage.org/browse/origin/https://github.com/alaingiorgetti/postTYPES2022`.

**Paper Outline.** In Sect. 2, we present a general methodology and interfaces to capture all the features of two representations of a given family of objects, and to switch easily from one representation to the other. In Sect. 3, we show how our approach applies to representations of closable Motzkin trees – that are the skeletons of closed $\lambda$-terms – and to representations of uniquely closable Motzkin trees. In Sect. 4, we adapt our approach to the parameterized family of $m$-open $\lambda$-terms. In Sect. 5, several applications of the presented isomorphic types are exposed. In Sect. 6, we draw some conclusions and present some promising perspectives.

## 2 Specifying Families Using Two Different Representations

As we shall see with examples related to pure $\lambda$-terms, a family of mathematical objects can usually be defined formally in two different but equivalent ways: either using an inductive datatype, possibly dependent, or using a larger non dependent datatype, together with a restriction property. In this section, we summarize which elements are required to specify the two datatypes and their basic properties. We then show how to derive the isomorphism properties automatically. One of these isomorphism properties can always be derived automatically, using a generic approach based on a functor, whereas the other one, which relies on a proof by induction on the data, is carried out using Ltac. In the examples presented in this article, the types have at most one level of dependency. Even if the Ltac code is as generic as possible, it may not generalize well when the level and complexity of the dependencies increase.

### 2.1 Types

A *restricted type* (T,is_P) is a dependent pair defined by a type T : Type, called its *base type*, and a predicate is_P : T → Prop, called its *restriction* or *filter*. The restricted type (T,is_P) is intended to represent the inhabitants of T satisfying the restriction is_P. For practical reasons, these two objects are encapsulated together as a record type rec_P isomorphic to the Σ-type $\{x : \texttt{T} \mid \texttt{is\_P}\ x\}$.

```
Record rec_P := Build_rec_P {
  P_struct :> T;
  P_prop : is_P P_struct
}.
```

In addition to this practical type, we assume that we also have another possibly dependent type P for the same family of objects. This type is usually closer to the way mathematicians would define such objects. However, it may be less convenient to handle in practice (e.g. when proving in a proof assistant such as Coq) and thus we shall prefer using the larger type T and the restriction is_P rather than the type P when programming operations and proving lemmas on such a family.

### 2.2 Transformations and Their Properties

Once the datatypes T and P and the filter is_P are defined, we build the expected isomorphisms as two transformation functions rec_P2P (from rec_P $\equiv \{x : \texttt{T} \mid \texttt{is\_P}\ x\}$ to P) and P2rec_P (from P to rec_P $\equiv \{x : \texttt{T} \mid \texttt{is\_P}\ x\}$). The first function rec_P2P can be defined as follows, with an auxiliary function T2P : ∀ (x:T), is_P x → P transforming any element $x$ : T that satisfies is_P into an element of P.

```
Definition rec_P2P m := T2P (P_struct m) (P_prop m).
```

To define the reverse transformation P2rec_P, we first implement a function P2T from P to T and then prove that the image of any $x$ by P2T satisfies the predicate is_P, i.e. we prove the following lemma:

```
Lemma is_P_lemma: ∀ v, is_P (P2T v).
```

Then the transformation P2rec_P can be defined as follows:

```
Definition P2rec_P (x:P) : rec_P := Build_rec_P (P2T x) (is_P_lemma x).
```

## 2.3 Partial Automation of Specification and Proofs

In order to automate some parts of the process, we provide an abstract definition of the minimum requirements for the two involved types, as shown in the module type declaration (a.k.a. *interface* or *signature*) family reproduced in the following code snippet.

```
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.
```

We assume that we have the type T and a restricting predicate is_P as well as the type P. We also provide two conversion functions T2P and P2T, together with two proofs: a proof is_P_lemma that is_P holds for all images (P2T v) of the inhabitants v of P, and a proof P2T_is_P that for all inhabitants t : T satisfying the predicate is_P, P2T is a left inverse of T2P.

Then, the roundtrip lemma P2rec_PK stating that P2rec_P is a left inverse for rec_P2P can be proved automatically using the functor equiv_family, reproduced in the following code snippet.

```
Module Type equiv_sig (f:family).
Import f.
Parameter rec_P : Type.
Parameter rec_P2P : rec_P → P.
Parameter P2rec_P : P → rec_P.
Parameter P2rec_PK : ∀ x: rec_P, P2rec_P (rec_P2P x) = x.
End equiv_sig.

Module equiv_family (Import f:family) <: equiv_sig(f).
  Record rrec_P := Build_rrec_P {
    P_struct :> T;
    P_prop : is_P P_struct
  }.

  Definition rec_P := rrec_P.

  Definition rec_P2P m := T2P (P_struct m) (P_prop m).
  Definition P2rec_P (x:P) : rec_P := Build_rrec_P (P2T x) (is_P_lemma x).
  Lemma P2rec_PK : ∀ x: rec_P, P2rec_P (rec_P2P x) = x.
  Proof.
    unfold rec_P2P, P2rec_P; intros; simpl.
    generalize (is_P_lemma (T2P (P_struct x) (P_prop x))).
    rewrite P2T_is_P.
    intros; destruct x; simpl in *.
    rewrite (proof_irr _ P_prop0 i).
    reflexivity.
  Qed.
End equiv_family.
```

The proof of `P2rec_PK` is generic and only relies on the components of the module `f` which has type `family`.

The proof of the other roundtrip lemma `rec_P2PK` cannot be derived abstractly using a functor. Indeed, the argument of this lemma is an element `m` of the inductively-defined type `P`. Therefore no proof can be carried out before we have an explicit definition of `P`. Once this definition is provided, the proof of the second lemma is rather straightforward and can be automated using some Ltac constructs. Although the Ltac proof is not generic, it works easily for all examples provided in this paper. We believe that this could be generalized to arbitrary datatypes by using some meta-programming tools such as Coq-elpi [10] or MetaCoq [18].

In the next subsection, we shall extend our interface and build a new functor to automatically generate some random generators for the two representations `P` and $\{x : \texttt{T} \mid \texttt{is\_P}\ x\}$ at stake.

## 2.4 Random Generators

Property based testing (PBT) has become famous in the community of functional languages. Mainly popularized by QuickCheck [4] in Haskell, PBT is also available in proof assistants. In Coq, the random testing plugin QuickChick [12] allows us to check the validity of executable conjectures with random inputs, before trying to write formal proofs of these conjectures. QuickChick is mainly a generic framework providing combinators to write testing code, in particular random generators, and also to prove their correctness.

Our general framework also provides guidelines to develop random generators for all the datatypes under study. Generators, either user-defined or automatically derived by QuickChick, have a type `G Ty` where `Ty` is the type of the generated values and `G` is an instance of the Coq `Monad` typeclass. They are usually parameterized by a natural number `n` that controls their termination (called *fuel* in the Coq community). It may also serve as a bound on the depth of the generated values, even if it is not always guaranteed.

Let us assume that a random generator of values of type `T`, named `gen_T : nat → G T`, is available. We are interested in providing a generator for each datatype: (i) a generator of values of type `T` satisfying the property `is_P`, (ii) a generator of values of type `rec_P` embedding a value of type `T` and a proof that the latter satisfies the property `is_P`, (iii) a generator of values of type `P`. Thanks to QuickChick and the bijections we have previously defined, they can be obtained quite easily, using three new functors explained below. All these generators come in a sized version, i.e. they are parameterized with a natural number which is randomly chosen, when used with a QuickChick test command.

The first functor we propose, `generators_family1`, allows the definition of the random generator `gen_filter_P` which implements the strategy *generate and test*. It can be obtained when are available an executable version of the predicate `is_P`, named `is_Pb`, and a proof of decidability of `is_P`, named `is_P_dec`. A value `default_P` of the considered family - which is guaranteed by a proof `default_is_P` - is also required.

```
Module Type family_for_generators1 (Import f : family).
  Import f.
  Module facts := equiv_family (f).
  Parameter is_Pb : T → bool.
  Parameter is_P_dec : ∀ x:T, is_P x ↔ is_Pb x = true.
  Parameter gen_T : nat → G T.
  Parameter default_P : T.
  Parameter default_is_P : is_Pb default_P = true.
End family_for_generators1.

Module generators_family1 (f : family) (g : family_for_generators1 f).
  Import f.
  Import g.
  Import g.facts.

  Definition filter_max := 100.
  Fixpoint gen_filter_P_aux nb n :=
  match nb with
  | 0 ⇒ returnGen default_P
  | S p ⇒ do! val ← gen_T n;
          if is_Pb val then returnGen val
          else gen_filter_P_aux p n
  end.
  Definition gen_filter_P : nat → G T := gen_filter_P_aux filter_max.
End generators_family1.
```

The random generator `gen_filter_P` randomly produces a value `val` of type `T` thanks to `gen_T` and checks whether `is_Pb val` is true, in which case it outputs `val`. Otherwise, it discards the value and tries again. If the maximum number of tries `filter_max` is reached, it yields the provided default value `default_P`.

The next two functors can be used to derive a random generator for one family representation from that of the alternative representation. When the random generator `gen_P` of values of type `P` is available, using the functor `generators_family3` shown below, we can obtain a random generator of values of type `rec_P`, i.e. a value of type `T` and a proof that it satisfies the property `is_P` (thanks to the functions and lemmas derived using `equiv_family`). The functor `generators_family2` (omitted here) does the opposite job.

```
Module Type family_for_generators3 (Import f : family).
  Parameter gen_P : nat → G P.
End family_for_generators3.

Module generators_family3 (Import f : family)
 (Import g : family_for_generators3 f)
 (Import facts : equiv_sig f).
  Definition gen_rec_P n : G rec_P :=
  do! p ← gen_P n;
  returnGen (P2rec_P p).
End generators_family3.
```

In the next section, we shall see how to instantiate our framework with two different representations of closable Motzkin trees and uniquely closable Motzkin trees, to automatically prove the equivalence between the representations and to automatically derive random generators.

## 3 Two Instances: Closable Motzkin Trees and Uniquely Closable Motzkin Trees

This section presents two simple examples of infinite families of objects with two representations in Coq. These examples are presented as applications of our formal framework, including formal proofs of isomorphisms between representations and the design of their corresponding random generators. Whereas our methodology applies to any pair of isomorphic datatypes, we have chosen to focus our applications primarily on data families related to the $\lambda$-terms from the pure (i.e., untyped) $\lambda$-calculus.

Let us briefly recall that the $\lambda$-calculus is a universal formalism to represent computations with functions. A *(pure) $\lambda$-term* is either a variable ($x$, $y$, ...), an *abstraction* $\lambda x.t$, that *binds* the variable $x$ in the $\lambda$-term $t$, or a term of the form $t\,u$ for two $\lambda$-terms $t$ and $u$. The term $\lambda x.t$ represents a function of the variable $x$. The term $t\,u$ represents an *application* of the function (represented by) $t$ to the function (represented by) $u$. A variable $x$ in *free* in the term $t$ if it is not bound in $t$ (by some $\lambda x$). A *closed* term is a term without free variables. Terms are considered up to renaming of their bound variables.

The two examples come from a study for the efficient enumeration of closed $\lambda$-terms, by Bodini and Tarau [3], that starts from binary-unary trees, a.k.a. Motzkin trees, that can be seen as skeletons of $\lambda$-terms. For self-containment, all the definitions and properties of this study that are formalized here are kindly reminded to the reader.

A *Motzkin tree* is a rooted ordered tree built from binary, unary and leaf nodes. Thus the set of Motzkin trees can be seen as the free algebra generated by the constructors `v`, `l` and `a` of respective arity 0, 1 and 2. Their type in Coq, named `motzkin`, is the following inductive type.

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

### 3.1 Closable Motzkin Trees

The *skeleton* of the $\lambda$-term $t$ is the Motzkin tree obtained by erasing all the occurrences of the variables in $t$. A Motzkin tree is *closable* if it is the skeleton of at least one closed $\lambda$-term. As in [3], we define a predicate for characterizing closable Motzkin trees:

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

This predicate only requires the presence of at least one occurrence of the unary node on each rooted path of the Motzkin tree. For instance, the tree `l (a v (l v))` is closable (it is the skeleton of the closed $\lambda$-term $\lambda x.x(\lambda y.y)$), whereas the tree `a (l v) v` is not closable.

Bodini and Tarau proposed a grammar generating closable Motzkin trees [3, Section 3], that we adapt in Coq as an inductive type, named `closable`.

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

**Table 1** Two instances of the `Module Type family` and the functor `equiv_family` representing closable Motzkin trees and uniquely closable Motzkin trees. Statements required in the functor `Module Type family` (upper part of the array) are proven automatically. The roundtrip statement `rec_P2PK` (last line of the array), which corresponds to `rec_closable2closableK` and `rec_ucs2ucsK` does not belong to the functor but can be proven automatically in both settings.

| Abstraction | Closable Skeletons | Uniquely Closable Skeletons |
|---|---|---|
| `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` | `proof_irr_is_closable` | `proof_irr_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 | |

For example, `La (a v (l v))` is the `closable` term corresponding to the Motzkin tree `l (a v (l v))`.

To prove that there is a bijection between closable Motzkin trees specified using the type `rec_closable` and inductive objects whose type is `closable`, using our approach, we simply need to provide two functions `motzkin2closable` and `closable2motzkin`.

```
Fixpoint motzkin2closable (m : motzkin) : is_closable m → closable :=
  match m as m0 return (is_closable m0 → closable) with
  | v ⇒ fun H : is_closable v ⇒ let H0 := match H return closable with end in H0
  | l m0 ⇒ fun _ : is_closable (l m0) ⇒ La m0
  | a m1 m2 ⇒ fun H : is_closable (a m1 m2) ⇒
      match H with
      | conj Hm1 Hm2 ⇒ Ap (motzkin2closable m1 Hm1) (motzkin2closable m2 Hm2)
      end
  end.
```

```
Fixpoint closable2motzkin c :=
  match c with
  | La m ⇒ l m
  | Ap c1 c2 ⇒ a (closable2motzkin c1) (closable2motzkin c2)
  end.
```

Because it involves dependent pattern matching, defining directly `motzkin2closable` as a function is not immediate. However it is easily carried out interactively as a lemma, in a proof-like manner, using the tactic `fix`.

The transformation functions and the isomorphism properties between the two types `closable` and `rec_closable` can then be automatically generated, as summarized in the second column of Table 1.

### 3.1.1 Random Generators

Random generators for closable and rec_closable have been used to test the different lemmas before proving them, for example the roundtrip lemma rec_closable2closableK, which is an instance of the pattern rec_P2PK. Corresponding QuickChick commands can be found in our formal development.

The generator for Motzkin trees, gen_motzkin, required by any of the other generators, is obtained automatically, thanks to QuickChick:

```
Derive (Arbitrary, Show) for motzkin.
```

In the context of closable Motzkin trees, the gen_closable generator associated to the tailored simple inductive type closable can be easily obtained using QuickChick. Thanks to the functor generators_family3, we can derive the random generator of values of the corresponding restricted type, as it is illustrated by the following snippet of code, where closable is an instance of the family, and fact_cl is defined as the module equiv_family (closable).

```
Module gen_closable3 : family_for_generators3 (closable).
Definition gen_P := gen_closable.
End gen_closable3.

Module V3 := generators_family3 closable gen_closable3 facts_cl.
```

To test the motzkin2closable function (T2P in the family interface), we need a generator that produces closable Motzkin trees. It is not relevant to use the previously defined generator which we have derived from that of closable values and thus obtained using, as a main ingredient, the function under test itself. For that purpose, the generator gen_filter_P obtained by applying the functor generators_family1 can be useful, however such a generator usually discards many values to produce the required ones. A handmade generator, as gen_closable_struct defined below, is usually preferred.

As a representative of this kind of custom generators, we expose its code in the following code snippet and explain it.

```
Fixpoint gen_closable_struct_aux (k : nat) (n : nat) : G motzkin :=
match n with
| 0 ⇒ match k with
      0 ⇒ returnGen default_closable
      | _ ⇒ returnGen v
      end
| S p ⇒
  match k with
  0 ⇒ oneOf [
    (returnGen default_closable);
    (do! mt ← gen_closable_struct_aux (S k) p; returnGen (l mt));
    (do! mt0 ← gen_closable_struct_aux k p; do! mt1 ← gen_closable_struct_aux k p;
      returnGen(a mt0 mt1)) ]
  | _ ⇒ oneOf [
    (returnGen v);
    (do! mt ← gen_closable_struct_aux (S k) p; returnGen (l mt));
    (do! mt0 ← gen_closable_struct_aux k p; do! mt1 ← gen_closable_struct_aux k p;
      returnGen(a mt0 mt1)) ]
  end
end.

Definition gen_closable_struct : nat → G motzkin := gen_closable_struct_aux 0.
```

We first define an intermediate function that uses the additional parameter `k` denoting the number of `l` constructors at hand. So, if both `k` and `n` are equal to 0, the generator emits the default value (here `l v`, stored in `default_closable`). If `n` is 0 but at least one `l` is available, then the generator produces the leaf `v`. When `n` is not 0, again we have two treatments depending on whether we have already introduced the constructor `l` or not. In both cases, the generator picks one of the several ways to produce a value – thanks to `oneOf`, and thus either stops with a value (resp. `l v` or `v`), recursively produces a closable Motzkin tree which is used to build a resulting unary Motzkin tree, or recursively generates two closable Motzkin trees used to produce a binary Motzkin tree. The final custom generator is obtained using the previous intermediate function with `k` equal to 0.

We recommend testing that this generator does produce Motzkin trees which are closable, as follows:

```
QuickCheck (sized (fun n ⇒ forAll (gen_closable_struct n) is_closableb)).
(*  +++ Passed 10000 tests (0 discards) *)
```

To define the proof-carrying version of the custom generator, we follow a similar scheme but also produce a proof that the produced value `mt` is closable, i.e. a term of type `is_closable mt`. We use the `Program` facility which allows us to produce certified programs and generates proof obligations. Here these proof obligations are automatically solved.

## 3.2 Uniquely Closable Motzkin Trees

A Motzkin tree is *uniquely closable* if there exists exactly one closed $\lambda$-term having it as its skeleton.

We first define a predicate `is_ucs` for characterizing uniquely closable skeletons. This predicate specifies that a Motzkin tree is uniquely closable if and only if there is exactly one unary node on each rooted path.

```
Fixpoint is_ucs_aux m b :=
  match m with
  | v ⇒ b = true
  | l m ⇒ if b then False
         else is_ucs_aux m true
  | a m1 m2 ⇒ is_ucs_aux m1 b ∧ is_ucs_aux m2 b
  end.

Definition is_ucs m := is_ucs_aux m false.
```

This Coq predicate corresponds to the second Prolog predicate `uniquelyClosable2` introduced by Bodini and Tarau [3, Section 4], after a first Prolog predicate `uniquelyClosable1` using a natural number to count the number of $\lambda$ binders above each leaf, instead of a Boolean flag as here. A Coq formalization of this other characterization of uniquely closable Motzkin trees, and a formal proof of their equivalence, are presented in Section 5.4.

We then define an inductive type `ucs` that also represents uniquely closable Motzkin trees.

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

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

Even though we use the abbreviations `ca` for `ClosedAbove` and `ucs` for `UniquelyClosable`, these types exactly correspond to Haskell datatypes given in [3]. For instance, the Motzkin tree `l (a v v)` and the corresponding `ucs` term `L (B V V)` represent uniquely closable skeletons. The closable tree `l (a (l v) v)` is not uniquely closable, because it is the skeleton of two closed $\lambda$-terms, namely $\lambda x.(\lambda y.y)x$ and $\lambda x.(\lambda y.x)x$.

Using the same infrastructure as for closable Motzkin trees, the transformation functions and the isomorphism properties between the two types `ucs` and `rec_ucs` can be automatically generated, as summarized in the last column of Table 1.

We proceed in the same way for random generators. Using QuickChick, the generator `gen_ucs` is automatically derived from the definition of the inductive types `ca` and `ucs`. The user-defined generator `gen_ucs_struct` is very close to `gen_closable_struct`. Similarly we use `Program` to define the one producing values and proofs.

## 4 Pure Open $\lambda$-Terms in De Bruijn Form

Let us now address the questions of formal representations and random generation of pure open $\lambda$-terms modulo variable renaming. The definitions in this section are not present in Bodini and Tarau's work [3].

To get rid of variable names, we adopt de Bruijn's proposal to replace each variable in a $\lambda$-term by a natural number, called its *de Bruijn index* [8]. When a de Bruijn index is not too high, it encodes a variable bound by the number of $\lambda$'s between its location and the $\lambda$ that binds it. Otherwise, it encodes a free variable. We consider de Bruijn indices from 0, to ease their formalization with the Coq type `nat` for natural numbers. For instance, the term $\lambda.(1\ (\lambda.1))$ in de Bruijn form represents the term $\lambda x.(y\ (\lambda z.x))$ with the free variable $y$.

### 4.1 Types

The tree structure of open $\lambda$-terms in de Bruijn form can be represented by unary-binary trees whose leaves are labeled by a natural number. They are the inhabitants of the following inductive Coq type `lmt` (acronym for `l`abeled `M`otzkin `t`ree).

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

However the property of being closed cannot be defined by induction on this definition of $\lambda$-terms. Indeed, if the term $\lambda\, t$ is closed, then the term $t$ is not necessarily closed, it can also have a free variable. The more general property of $m$-openness overcomes this limitation: for any natural number $m$, the $\lambda$-term $t$ is said to be *$m$-open* if the term $\lambda \dots \lambda\, t$ with $m$ abstractions before $t$ is closed. Whereas the "$m$-open" terminology is recent [1], the notion has been studied since 2013, by Grygiel and Lescanne [11, 13].

With the following definition, `(is_open m t)` holds iff the labeled Motzkin tree `t` encodes an `m`-open $\lambda$-term. This function call indeed visits the tree `t` and counts (from `m`) the number of $\lambda$s (constructor `lam`) traversed so far. At each leaf (constructor `var`) it checks that its de Bruijn indice `i` is lower than this number `m` of traversed abstractions.

```
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.
```

For instance, the tree `lam (app (var 0) (lam (var 1)))` is 0-open (its skeleton is the closable term `l (a v (l v))`), whereas the tree `lam (app (var 1) (lam (var 1)))` is 1-open, but not 0-open.

Because of the extra parameter $m$, the formal framework presented in Sect. 2 must be adapted and we propose a new module type `param_family` together with a functor `equiv_param_family` to automatically prove one of the roundtrip lemmas. The other one can be easily proved correct using the same sequences of Ltac constructs as for the non dependent case.

The following record type parameterized by `m` is such that `(rec_open m)` describes `m`-open terms. As previously, the first field stores the datum, here a labeled Motzkin tree (i.e., `T` is `lmt`), and the second field stores a proof that it is `m`-open.

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

It is however more natural to describe `m`-open terms with a dependent type `(open m)` enclosing the condition `i < m` at leaves, as follows.

```
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.
```

## 4.2 Transformations and Their Properties

In order to switch from one representation to the other whenever needed, we provide two functions `rec_open2open m` and `open2rec_open m`, and Coq proofs for two roundtrip lemmas justifying that they are mutual inverses.

**From the Record Type to the Dependent Type.** The function `rec_open2open m` from the record type `(rec_open m)` to the dependent type `(open m)` is defined by

```
Definition rec_open2open (m : nat) (r : rec_open m) :=
  lmt2open (open_struct m r) m (open_prop m r).
```

where `lmt2open` is the following dependent recursive function.

```
Fixpoint lmt2open (t:lmt) : ∀ m:nat, is_open m t → open m :=
  match t as u return (∀  m0 : nat, is_open m0 u → open m0) with
  | var n ⇒ fun (m0 : nat) (H : is_open m0 (var n)) ⇒ open_var m0 n H
  | lam u ⇒ fun (m0 : nat) (H : is_open m0 (lam u)) ⇒
      open_lam m0 (lmt2open u (S m0) H)
  | app u w ⇒
      fun (m0 : nat) (H : is_open m0 (app u w)) ⇒
        match H with
        | conj H0 H1 ⇒ open_app m0 (lmt2open u m0 H0) (lmt2open w m0 H1)
        end
end.
```

It is rather difficult to define this function directly. We choose to develop it as a proof, as advocated by McBride [15], in an interactive manner, letting Coq handle the type dependencies. Once the term is built, we simply revert the proof and declare it directly as a fixpoint construction to make it look like a function, more readable and understandable for humans than a proof script.

**From the Dependent Type to the Record Type.** The process to define the inverse function `open2rec_open m` from the dependent type (`open m`) to the record type (`rec_open m`) is rather different, and can be decomposed as follows. First of all, a function (`open2lmt m`) turns each dependent term `t` of type `open m` into a labeled Motzkin tree.

```
Fixpoint open2lmt (m:nat) (t : open m) : lmt :=
  match t with
  | open_var m i _ ⇒ var i
  | open_lam m u ⇒ lam (open2lmt (S m) u)
  | open_app m t1 t2 ⇒ app (open2lmt m t1) (open2lmt m t2)
  end.
```

Then we prove automatically, using the same Ltac constructs as for the previous examples, the following lemma that states that the function `open2lmt m` always outputs an `m`-open term.

```
Lemma is_open_lemma : ∀ m t, is_open m (open2lmt m t).
```

Once this lemma is proved, we can derive automatically the transformation `open2rec_open`, by using the functor `equiv_param_family`.

```
Definition open2rec_open m t := Build_rec_open m (open2lmt m t) (is_open_lemma m t).
```

As we did in the previous sections, we then need to prove a lemma `open2lmt_is_open` which relates the functions `open2lmt` and `lmt2open`, without taking into account the restriction property.

```
Lemma open2lmt_is_open : ∀ m t H, open2lmt m (lmt2open t m H) = t.
```

Both lemmas are part of the interface `param_family` for a parametric family, extending the interface `family`. Thus, applying the appropriate functor, we automatically derive a proof of the first roundtrip lemma:

```
Lemma open2rec_openK : ∀ m r, open2rec_open m (rec_open2open m r) = r.
```

The proof of the second roundtrip lemma proceeds by induction on `x` of type `open m`. It is immediately proven using the Ltac constructs proposed in the previous sections.

```
Lemma rec_open2openK : ∀ m x, rec_open2open m (open2rec_open m x) = x.
```

## 4.3 Random Generators

The required generator `gen_lmt` is automatically derived by QuickChick from the definition of the inductive type `lmt`. The custom generators for $\lambda$-terms satisfying the `open m` property, with or without proofs, are written following the same canvas as before. The generator corresponding to the inductive type `open` is no longer derived automatically by QuickChick, in particular because proofs have to be inserted when using the `open_var` constructor. However it is easy to define it manually.

## 4.4 Characterization of Open $\lambda$-Terms From Their Skeleton

This subsection presents definitions and formal proofs relating Bodini and Tarau's skeletons for $\lambda$-terms (Section 3) with $m$-open $\lambda$-terms introduced in this section, not present in Bodini and Tarau's work.

The skeleton of a $\lambda$-term is the Motzkin tree obtained by erasing the labels at its leaves.

```
Fixpoint skeleton (t: lmt) : motzkin :=
  match t with
  | var _ ⇒ v
  | lam t1 ⇒ l (skeleton t1)
  | app t1 t2 ⇒ a (skeleton t1) (skeleton t2)
  end.
```

This function (specified by `toMotSkel` in [3]) connects Motzkin trees without labels (Sect. 3) and Motzkin trees with labels defined in this section.

As the `skeleton` function cannot be inverted functionality, we define a pseudo-reverse, from Motzkin trees without labels to labeled Motzkin trees, as the following family of inductive relations (`label m`), for all natural numbers `m`.

```
Inductive label : nat → motzkin → lmt → Prop :=
| Lvar : ∀ m i, i < m → label m v (var i)
| Llam : ∀ m mt t, label (S m) mt t → label m (l mt) (lam t)
| Lapp : ∀ m mt1 mt2 t1 t2, label m mt1 t1 → label m mt2 t2
    → label m (a mt1 mt2) (app t1 t2).
```

The label-removing function `skeleton` and the label-adding relation `label` can be used together as follows, to define a second characterization of `m`-open $\lambda$ terms among labeled Motzkin trees `t`.

```
Definition skeleton_open (m:nat) (t:lmt) : Prop := label m (skeleton t) t.
```

The proof of the following equivalence with the first characterization (`is_open`, introduced in Section 4) is straightforward.

```
Lemma skeleton_is_open_eq : ∀ m t, skeleton_open m t ↔ is_open m t.
```

An $m_1$-open $\lambda$-term is also an $m_2$-open $\lambda$-term for all $m_2 \geq m_1$.

```
Lemma label_mon : ∀ m1 mt t, label m1 mt t → ∀ m2, m1 ≤ m2 → label m2 mt t.
```

Consequently, for any labeled Motzkin tree $t$, there is a minimal natural number $m$ such that $t$ is an $m$-open $\lambda$-term. It can be computed for instance by the following function.

```
Fixpoint minimal_openness (t : lmt) : nat :=
  match t with
  | var i ⇒ i+1
  | lam t ⇒ match minimal_openness t with S m ⇒ m | _ ⇒ 0 end
  | app t1 t2 ⇒ max (minimal_openness t1) (minimal_openness t2)
  end.
```

The function `skeleton` and the relation `label` are pseudo-inverses in the sense of the following two lemmas.

```
Lemma label_skeletonK : ∀ t : lmt, label (minimal_openness t) (skeleton t) t.
```

```
Lemma skeleton_labelK : ∀ m : nat, ∀ mt : motzkin, ∀ t : lmt,
  label m mt t → skeleton t = mt.
```

The lemmas `label_skeletonK` and `skeleton_is_open_eq` jointly establish that the labeled Motzkin tree `t` is a (`minimal_openness t`)-open $\lambda$-term.

```
Lemma lmt_minimal_openness : ∀ t : lmt, is_open (minimal_openness t) t.
```

Finally, it is easy to prove by induction that `minimal_openness t` indeed computes the smallest openness $m$ such that `t` is an $m$-open $\lambda$-term.

```
Lemma minimality : ∀ t : lmt, ∀ m : nat, is_open m t → m ≥ minimal_openness t.
```

## 5 Use Cases

In this section we use the previous examples of types to formalize all the propositions in Bodini and Tarau's work [3] that are related to Motzkin trees and pure $\lambda$-terms.

### 5.1 Another Definition for Closable Skeletons

Bodini and Tarau [3, section 3] first defined closable skeletons with a Prolog predicate – named `isClosable` – whose adaptation in Coq is

```
Fixpoint isClosable2 (mt: motzkin) (V: nat) :=
  match mt with
  | v ⇒ V > 0
  | l m ⇒ isClosable2 m (S V)
  | a m1 m2 ⇒ isClosable2 m1 V ∧ isClosable2 m2 V
  end.

Definition isClosable (mt: motzkin) := isClosable2 mt 0.
```

For each $\lambda$ binder this function increments a counter `V` (starting at `0`). Then it checks at each leaf that its label is strictly positive. This definition is slightly more complicated than that of the Coq predicate `is_closable` presented in Sect. 3. We have proved formally that both definitions are equivalent:

```
Lemma is_closable_isClosable_eq : ∀ (mt: motzkin), is_closable mt ↔ isClosable mt.
```

The two implications of this equivalence are proved by structural induction and thanks to the following two lemmas, themselves proved by structural induction.

```
Lemma isClosable2_S : ∀ m n, isClosable2 m n → isClosable2 m (S n).
Lemma isClosable_l : ∀ m, isClosable (l m).
```

We can notice that this proof is simpler than expected: Although the generalization `isClosable2` is required to define the predicate `isClosable`, the proof avoids the effort to invent generalizations to `isClosable2` of the predicate `is_closable` and the equivalence lemma. Similarly, after "packing" the predicate `isClosable` in the following record type, it was possible to define and prove isomorphism with the algebraic datatype `closable` without having to generalize the record and the datatype to `isClosable2`.

```
Record recClosable : Type := Build_recClosable {
  Closable_struct : motzkin;
  Closable_prop : isClosable Closable_struct
}.
```

### 5.2 Two Definitions for the Size of Terms

Bodini and Tarau [3, Proposition 1] state the following proposition to justify that two different size definitions lead to the same sequence of numbers of closed $\lambda$-terms modulo variable renaming, counted by increasing size.

▶ **Proposition 1.** *The set of terms of size $n$ for size defined by the respective weights 0, 1 and 2 for variables, abstractions and applications is equal to the set of terms of size $n+1$ for size defined by weight 1 for variables, abstractions and applications.*

This proposition holds not only for all Motzkin trees (without labels), but also for closable ones, labeled ones, and for $m$-open $\lambda$-terms. Since we proposed two Coq types for closable Motzkin trees and for $m$-open $\lambda$-terms, we formalize Proposition 1 by six propositions in Coq, all of the form

```
Proposition proposition1X : ∀ t : X, size111X t = size012X t + 1.
```

with `X` in {`motzkin`, `rec_closable`, `closable`, `lmt`, `rec_open`, `open`}, and with adequate functions `size111X` and `size012X`, not detailed here, defining both sizes for each type. More precisely, thanks to the coercion (`P_struct :> T`) in the record types, the functions `size*rec_P` are not defined, but advantageously replaced by the functions `size*T`. Here, `*` is either `111` or `012` and (`T`,`P`) is either (`motzkin`,`closable`) or (`lmt`,`open`). For record types, the proposition then takes the following form:

```
Proposition proposition1rec_P : ∀ t : rec_P, size111T t = size012T t + 1.
```

This proposition is a straightforward consequence of the corresponding proposition on the type `T` (named `proposition1T`, according to our naming conventions). This mechanism being similar for all record types, it can easily be mechanized.

The situation is very different with – potentially – dependent types (named `P` in our general framework), if we forbid ourselves to use their isomorphism with a record type to prove their proposition (named `proposition1P`, according to our naming conventions). Here, the propositions for `P` in {`closable`,`open`} are proved by structural induction and linear arithmetic, because the latter suffices to inductively define the size functions. However, the general situation may be arbitrarily more complex, so no general mechanization can be considered.

## 5.3 Characterization of Closable Motzkin Trees

This section and the next one present two propositions from Bodini and Tarau's work [3] that cannot be formalized with the single unlabeled notion of skeleton introduced in that work, but also require a formalization of $\lambda$-terms with labels for their variables, such that the one introduced in Section 4 of the present paper.

The first of these two propositions is the following characteristic property for closable Motzkin trees [3, Proposition 2].

▶ **Proposition 2.** *A Motzkin tree is the skeleton of a closed $\lambda$-term if and only if it exists at least one $\lambda$-binder on each path from the leaf to the root.*

After defining a closed $\lambda$-term as a 0-open $\lambda$-term, we can state Proposition 2 in Coq, as follows.

```
Definition is_closed t := is_open 0 t.
Proposition proposition2 : ∀ mt : motzkin,
  (∃ t : lmt, skeleton t = mt ∧ is_closed t) ↔ is_closable mt.
```

This formalization is close to the text of Proposition 2. It relies on the base type `motzkin` and the restriction `is_closable`. A formulation of this proposition with the type `rec_closable` or `closable` would be useless, because these more precise types already include the *closability* property characterized by the proposition.

The proof of this proposition is straightforward.

## 5.4 Characterization of Uniquely Closable Motzkin Trees

Bodini and Tarau propose the following characteristic property for uniquely closable Motzkin trees [3, Proposition 4].

▶ **Proposition 3.** *A skeleton is uniquely closable if and only if exactly one lambda binder is available above each of its leaf nodes.*

The predicates `is_ucs` and `is_ucs_aux` presented in Section 3.2 correspond to the Prolog predicate `uniquelyClosable2` of [3, Section 4] and to the characteristic property "exactly one lambda binder is available above each of its leaf nodes" of Proposition 4 of [3]. Therefore, proving Proposition 3 consists in showing that this property is equivalent to the definition "We call a skeleton *uniquely closable* if it exists exactly one closed lambda term having it as its skeleton." [3, page 6], which gives the following Coq code.

```
Proposition proposition4: ∀ mt : motzkin,
  (∃! t, skeleton t = mt ∧ is_closed t) ↔ is_ucs mt.
```

However, this proposition cannot be proved directly, because `(is_closed t)` is a special case of `(is_open m t)`, which is parametrized by a natural number `m`, while `(is_ucs mt)` is a special case of of `(is_ucs_aux mt b)`, which is only parameterized by a Boolean `b`. The rest of this section addresses this issue by generalizing the proposition to any natural number `m`, using a characterization `(ucs1_aux mt m)` parametrized by this integer and put in correspondence with `(is_ucs_aux mt b)`.

The following predicates `ucs1_aux` and `ucs1` adapt in Coq the Prolog predicate named `uniquelyClosable1` in [3].

```
Fixpoint ucs1_aux (t:motzkin) (n:nat) : Prop :=
  match t with
  | v ⇒ (1 = n)
  | l m ⇒ ucs1_aux m (S n)
  | a m1 m2 ⇒ ucs1_aux m1 n ∧ ucs1_aux m2 n
  end.
Definition ucs1 (t:motzkin) := ucs1_aux t 0.
```

We then use the predicate `ucs1_aux` to state a generalization of Proposition 3 to any openness $m$, then the predicate `ucs1` to state its specialization when $m = 0$, which is a variant of Proposition 3.

```
Lemma proposition4ucs1_aux : ∀ (mt : motzkin) (m : nat),
  (∃! t, skeleton t = mt ∧ is_open m t) ↔ ucs1_aux mt m.
```

```
Corollary proposition4ucs1: ∀ mt : motzkin,
  (∃! t, skeleton t = mt ∧ is_closed t) ↔ ucs1 mt.
```

Independently, we can prove that the two charaterizations of uniquely closable Motzkin trees are equivalent.

```
Lemma ucs1_is_ucs_eq : ∀ mt : motzkin, ucs1 mt ↔ is_ucs mt.
```

As usually when formalizing pen-and-paper proofs, we get more precise statements and more detailed proofs. For example, we formally proved Proposition 1 in [3] as four propositions, corresponding to four distinct data families.

## 6 Conclusions and Perspectives

We have presented a framework to define and formally prove isomorphisms between Coq datatypes, and to produce random generators for them. After applying it to several examples related to lambda term families, we have formalized in Coq a large subset of the computational and logical content of Bodini and Tarau's paper [3] about pure $\lambda$-terms. Although our work is clearly dealing with Coq representations, our technique could be useful to other proof assistant tools and could be developed for example in Isabelle/HOL or Agda.

Technically, our present approach using interfaces allows us to automatically derive only one of two round-trip properties, that state that the considered transformations are inverse bijections. The other one, which proceeds by induction on the type `P`, cannot be generated automatically by a functor, however, we can prove it automatically using some advanced tactic combinations using Ltac. In the near future, we plan to investigate in more details whether using external tools like MetaCoq [18] or elpi [10] and Coq-elpi [10] would increase the genericity of our approach compared to simply relying on Ltac.

Our framework obviously applies to other formalization topics. It was inspired by previous work, including one on Coq representations of permutations and combinatorial maps [9]. We plan to complete this work and revisit it using this structuring framework. The proofs of isomorphisms presented in this paper were elementary because the two types in bijection were very close to one another. In the more general case of two different points of view on the same family (e.g., permutations seen as injective endofunctions or products of disjoint cycles), isomorphisms can be arbitrarily more difficult to prove.

## References

1 Maciej Bendkowski, Olivier Bodini, and Sergey Dovgal. Statistical Properties of Lambda Terms. *The Electronic Journal of Combinatorics*, 26(4):P4.1, October 2019. `doi:10.37236/8491`.

2 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions.* Springer-Verlag, Berlin/Heidelberg, May 2004. 469 pages.

3 Olivier Bodini and Paul Tarau. On uniquely closable and uniquely typable skeletons of lambda terms. In Fabio Fioravanti and John P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation. LOPSTR 2017*, volume 10855 of *Lecture Notes in Computer Science*, pages 252–268. Springer, Cham, 2018. `doi:10.1007/978-3-319-94460-9_15`.

4 Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, volume 35 of *SIGPLAN Not.*, pages 268–279. ACM, New York, 2000. `doi:10.1145/351240.351266`.

5 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2015.5`.

6 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs. CPP 2013*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, Cham, 2013. `doi:10.1007/978-3-319-03545-1_10`.

7 Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.13.2.* INRIA, 2021. URL: `http://coq.inria.fr`.

8 N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972. doi:10.1016/1385-7258(72)90034-0.

9 Catherine Dubois and Alain Giorgetti. Tests and proofs for custom data generators. *Formal Aspects of Computing*, 30(6):659–684, July 2018. doi:10.1007/s00165-018-0459-1.

10 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: Fast, Embeddable, $\lambda$Prolog Interpreter. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2015*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, Berlin, Heidelberg, 2015. doi:10.1007/978-3-662-48899-7_32.

11 Katarzyna Grygiel and Pierre Lescanne. Counting and generating lambda terms. *Journal of Functional Programming*, 23(5):594–628, September 2013. doi:10.1017/S0956796813000178.

12 Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, August 2022. Version 1.3.1 https://softwarefoundations.cis.upenn.edu/qc-1.3.1.

13 Pierre Lescanne. On counting untyped lambda terms. *Theoretical Computer Science*, 474:80–97, February 2013. doi:10.1016/j.tcs.2012.11.019.

14 Nicolas Magaud. Changing data representation within the Coq system. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics. TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 87–102. Springer, Berlin, Heidelberg, 2003. doi:10.1007/10930755_6.

15 Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs. TYPES 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, Berlin, Heidelberg, 2000. doi:10.1007/3-540-45842-5_13.

16 Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. doi:10.1017/S0956796803004829.

17 Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 966–980. ACM, New York, 2022. doi:10.1145/3519939.3523707.

18 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq project. *Journal of Automated Reasoning*, 64(5):947–999, 2020. doi:10.1007/s10817-019-09540-0.

19 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

20 Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313. ACM Press, 1987. doi:10.1145/41625.41653.