An Attempt at Necromancy

Experience Report on Modernising a Domain Theory Library

Meven Lennon-Bertrand
Université Paris Cité, INRIA, CNRS, IRIF
France
meven.bertrand@inria.fr

Abstract

For an ongoing mechanisation attempt, I needed some domain theory. A number of libraries on this topic have been developed since the early days of Rocq, but all of them are currently outdated – the most recent has been in maintenance mode since 2014. I propose an experience report on my attempt at porting this library to modern Rocq, both from a technical point of view (use of quotients, automation, hierarchy of structures, etc.) and on "softer" aspects.

1 Introduction

This work started with the goal to better understand a paper of Coquand and Huber [6]. Of course, the best way to understand a paper is to mechanise it, so I set myself to the task. However, the paper essentially starts as follows:

> *We shall use the following Scott domain, least solution of a recursive domain equation [...]:*

$$D = [D \to D] + \Pi\, D\, [D \to D] + \mathsf{N} + 0 + \mathsf{S}\, D + \mathsf{U}_i$$

This domain is the central object of the paper, which builds a semantics for a dependent type system in it and derives interesting syntactic properties from this model. Thus, to formalise it, I needed a domain theory library.[1]

Mechanised domain theory is as old as proof assistants: LCF itself [10] has an intended model in domains! In Rocq, domains were already formalised in version 5 [9], but the most interesting and up-to-date libraries date back to Rocq 8.2 (2009) [4] and 8.4 (2014) [7]. Because it was more recent, seemed more complete, and appeared to suit my needs, I elected the latter, by Rob Dockins, as my starting point.

2 What porting, exactly?

Porting Rocq 8.4 to Rocq 9.1, and again. I obviously did not want to work in Rocq 8.4, so the first step was to port to recent versions. This had already been done by Rob Dockins and Arthur Azevedo de Amorim until Rocq 8.16, and I completed it to 9.1. I cannot comment for their part, but on my end, thanks to Rocq's excellent care for backwards compatibility, this was a breeze, taking only a day for a ~50kLoC library. Most changes were well-documented change in default behaviour at various places, such as locality information.

However, this was only the start of the process. Indeed, although the library was compiling with Rocq 9.1, it was still clearly written for Rocq 8.4: it used few tactics, made only minimal use of canonical structures [8], in the fashion of old school MathComp,[2] etc. The real porting was not to get a *working* development but an *idiomatic* one, on which I could happily base my own work. These changes were much heavier, and have been ongoing on-and-off for months.

My main efforts were along two lines. I first replaced the pervasive use of setoids with a lightweight form of observational equality [1, 12], critically featuring quotients. Second, I introduced Hierarchy Builder [5], complemented with type classes [14], to keep invisible mathematics [3] invisible.

Porting as program revival. In the end, virtually all the library needed rewriting, mostly for the best: order theoretic proofs are typically shorter by a factor of 2 or 3 in my version, exposing the core arguments while removing a lot of bookkeeping silently handled by the infrastructure.

Yet, this was challenging, as Dockins' paper focuses on the library's mathematics, with only a few passing comments on mechanisation. There is no trace of the reasoning behind most mechanisation choices, which complicated my work. These issues are in my view well captured by Naur's idea of *Programming as Theory Building* [11], and its explanation of the difficulties of *program revival*: while the text of the mechanisation remains, the *theory*[3] behind it is gone.

I essentially reconstructed a new theory supporting my own choices. I often depart from the original definitions, without clear understanding of whether this is a sensible consequence of my basic design, or whether I was falling again in a trap Dockins knew and avoided. As for proofs, I was more radical, and did not try to reproduce Dockins': the level of granularity of lemmas is small enough that the best strategy often was to rewrite an entirely new (and much shorter) proof. At times, though, being able to go back to the original proof and explore it interactively was very useful to figure out a tricky step that I could not easily reinvent on my end. Luckily, I could follow Dockins' mathematics even though the mechanisations diverged deeply, and my

[1] An alternative strategy is to bypass domain theory by working concretely with a type representing the domain's compact elements. This is doable, but not the approach I explore here.

[2] Featuring handcrafted mixins, manually declared coercions, and the like.

[3] Roughly, the knowledge and understanding which supports and leads the choices made in the activity at hand, here mechanisation.

development, even though it is a deep reworking, was still significantly easier than having to start a formalisation from scratch.

3 A bright future of synthetic setoids?

All hail OTT! On the technical front, my main decision was to avoid setoids. Dockins parametrises every structure by an equivalence relation corresponding to its "natural" notion of identification, which might not be equality. This was key for a sub-library of finite sets, represented as lists which are extensionally the same, *i.e.* related by the following relation

```
Definition list_ext A : relation (list A) :=
  fun l l' ⇒ forall x, In x l ↔ In x l'.
```

Setoids incur a proliferation of axioms and lemmas about relation preservation. While, in cases as the above, these bear mathematical content, they are often purely bureaucratic. Moreover, setoids come with generalised rewriting, which, while powerful, is not as smooth as the `rewrite` tactic.

An alternative is to use an extensional flavour of type theory, such as Observational Type Theory (OTT) [1, 12], where equality accurately reflects type's "natural" identification. Although an implementation of OTT in Rocq is underway, for now I went quick and simple, postulating function and proposition extensionality, and, most importantly, quotients (see Appendix A), for cases like finite sets where one genuinely wants to change a type's equality.

To circumvent the fact that, contrarily to "true" OTT, my equality does not compute on types, I designed `ext`, a small tactic which applies lemmas characterising each type's equality, collected in a database. I enrich `ext` with a characterisation of complex types' equality, *e.g.* equality of monotone maps being equality of the underlying functions. Although this particular tactic is very specific to my setting, I feel like similar small simplification tactics are quite useful, particularly if they can be extended throughout the development – for which I use the Smpl plugin [13], since this use is not really well handled by hint databases.

OTT has a model in setoids over intentional equality, so I am not really changing the mathematical content, merely delegating bureaucracy to the model: setoids are transparent everywhere, except where they matter, *i.e.* around quotients. Quotients also enforce honesty: the type theory ensures the designated equivalence must be respected.

Or? While good hygiene, the restrictions enforced by quotients in OTT can be too strong. I faced an issue with the union operation of enumerable sets `eset`, which are represented as a quotient of `nat → option A`. This union operation, of type `eset (eset A) → eset A`, essentially amounts to pre-composing an `f : nat → option (nat → option A)` with a pairing function `nat*nat → nat`. With quotients, however, one is stuck with some `g : nat → (eset A)`: we would like to un-quotient all elements in g's range at once, getting some `g' : nat → nat → option A` – and the obligation of showing that the final result is independent of the particular `g'`. Yet this is countable choice, unprovable in basic OTT! In setoids, this poses no issue, as one can locally disregard the relation, and show at the end it is in fact respected. Luckily, countable choice is true in the intended model – its justification is essentially the manipulation used to define the union in setoids – so I simply added it to the postulates.

Another annoyance with quotients is their most general elimination principle, which involves some transport yoga. There are two ways to neutralise the issue, using two specialised induction principles (see Appendix B). The first is non-dependent recursion, where the obligation stays but the transport disappears. The second is dependent induction in an h-proposition, where the obligation disappears altogether. The latter is more general than induction with a `Prop`-valued predicate, and is handy to show instances of the `Decidable` class, which, although not in `Prop`, is an h-proposition.

A last issue is the absence of unique choice,[4] which makes the interaction between `Prop` and h-proposition somewhat unnatural. In particular there are two, non-isomorphic forms of propositional truncation – and thus of existential. This caused some headaches when working with semi-decidable predicates, although in the end I managed to avoid the h-propositions' existential. Still, altogether OTT remains a vast improvement over vanilla Rocq.

4 Inferring data and structure

I use Hierarchy Builder (HB) [5] heavily to handle order-theoretic structures, also reusing Arsac et al.'s [2] HB-based category theory library. HB in its current incarnation was much more attractive than handcrafted canonical structures – which Dockins foregoes in later parts of the library, probably due to the bureaucratic weight that HB alleviates. Although I believe HB's approach has a lot of potential, I sadly found it difficult to use in its current incarnation, and hope my issues can help in making HB better and/or serve designers of tools with similar aims.

A first issue is that HB does a lot, which is good, but also makes it easy to get confused or use it not quite the way one should. Unfortunately, when one strays off the happy path where everything just works, HB issues become very hard to understand or debug. This is compounded by the fact that, by design, HB is mostly a black box, making it hard to figure out what the root cause of an issue is.

On a more technical side, in a number of situations I decided to resort to type classes instead of canonical structures. In some cases, this was forced by other parts of Rocq: the `transitivity` tactic or the infrastructure around relation preservation (`Proper`) are handled using type classes. However, I believe that type classes like `Proper` or `Decidable` are

[4] Which unfortunately does *not* hold in the vanilla setoid model.

in any case more naturally expressed in the unbundled fashion which goes with type classes, than in a bundled one, which canonical structures use. Especially so as they do not belong in a deep hierarchy, and are naturally used inside other structures, for instance for decidable orders. Luckily, with a bit of hacking they can be made to collaborate well with HB – and I hope that the ongoing work by Quentin Vermande will make even this hacking unnecessary.

More worrying, it seems like HB and/or canonical structures are not quite flexible enough to capture certain inference patterns. One example is set-like structures, whose carrier is a function `set : Type → Poset`: canonical structures have difficulties to infer the poset structure on `set A` due to the head constant being a variable. Another example is a category being a groupoid, where the structure on the category implies structure on its morphisms, a form of cross-hierarchy inheritance pattern. Maybe here too some creative use of a type-class-like mechanism might help.

Code

The original library is available on Dockins' GitHub: https://github.com/robdockins/domains. The port that this abstract describes is hosted on my own fork of the repository, currently on the port_categories branch.

Acknowledgments

This work benefitted from discussions with Adrien Mathieu (on the domain theory), Loïc Pujet (regarding setoids, OTT and choice), and Cyril Cohen and Cécile Marcon (for HB). Many thanks for their help! I also thank the anonymous reviewers for their nice feedback.

References

[1] Thorsten Altenkirch, Conor McBride and Wouter Swierstra. 2007. Observational Equality, Now! In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification* (PLPV '07). Association for Computing Machinery, Freiburg, Germany, 57–68. doi:10.1145/1292597.1292608.

[2] Samuel Arsac, Russ Harmer and Damien Pous. 2026. Adhesive category theory for graph rewriting in Rocq. *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs*, (Jan. 2026).

[3] Andrej Bauer. 2023. Formalizing invisible mathematics. In *Machine assisted proofs*.

[4] Nick Benton, Andrew Kennedy and Carsten Varming. 2009. Some Domain Theory and Denotational Semantics in Coq. In *Theorem Proving in Higher Order Logics*. Springer Berlin Heidelberg, 115–130. doi:10.1007/978-3-642-03359-9_10.

[5] Cyril Cohen, Kazuhiko Sakaguchi and Enrico Tassi. 2020. Hierarchy Builder: algebraic hierarchies made easy in Coq with Elpi. In *FSCD 2020 - 5th International Conference on Formal Structures for Computation and Deduction* number 167. Paris, France, (June 2020). doi:10.4230/LIPIcs.FSCD.2020.34.

[6] Thierry Coquand and Simon Huber. 2018. An Adequacy Theorem for Dependent Type Theory. *Theory of Computing Systems*, 63, 4, (July 2018), 647–665. doi:10.1007/s00224-018-9879-9.

[7] Robert Dockins. 2014. Formalized, Effective Domain Theory in Coq. In *Interactive Theorem Proving*. Springer International Publishing, 209–225. doi:10.1007/978-3-319-08970-6_14.

[8] François Garillot, Georges Gonthier, Assia Mahboubi and Laurence Rideau. 2009. Packaging Mathematical Structures. In *Theorem Proving in Higher Order Logics*. Stefan Berghofer, Tobias Nipkow, Christian Urban and Makarius Wenzel, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 327–342.

[9] Gilles Kahn. Elements of Domain Theory. Coq users contributions library, (1993). https://github.com/rocq-archive/domain-theory.

[10] Robin Milner. 1973. Models of LCF. Tech. rep. Stanford University.

[11] Peter Naur. 1985. Programming as Theory Building. *Microprocessing and Microprogramming*, (May 1985). doi:10.1016/0165-6074(85)90032-8.

[12] Loïc Pujet and Nicolas Tabareau. 2022. Observational Equality: Now for Good. *Proc. ACM Program. Lang.*, 6, POPL, Article 32, 27 pages. doi:10.1145/3498693.

[13] [SW] Sigurd Schneider, Rocq Plugin smpl. URL: https://github.com/uds-psl/smpl/.

[14] Matthieu Sozeau and Nicolas Oury. 2008. First-Class Type Classes, 278–293. doi:10.1007/978-3-540-71067-7_23.

A Quotient axiomatisation

```
(** The quotient type *)
Axiom quot : forall
  {T : Type} (R : relation T) `{! Equivalence R},
  Type.

(** The quotient element constructor *)
Axiom to_quot : forall
  {T : Type} {R : relation T} `{! Equivalence R},
  T -> quot R.

(** The quotient path constructor *)
Axiom quot_ext : forall
  {T : Type} {R : relation T} `{! Equivalence R}
  {t t' : T}, R t t' -> to_quot t = to_quot t'.

(** Quotient effectivity axiom *)
Axiom quot_eq : forall
  {T : Type} {R : relation T} `{! Equivalence R}
  (t t' : T),
  to_quot t = to_quot t' -> R t t'.

(** Induction for quotients *)
Axiom quot_rect : forall
  {T : Type} {R : relation T} `{! Equivalence R}
  (P : quot R -> Type)
  (f : forall (t : T), P (to_quot t)),
  (forall (x y : T) (r : R x y),
    (quot_ext r) # (f x) = f y :> P (to_quot y)) ->
  forall u : quot R, P u.

(** Propositional computation rule *)
Axiom quot_rect_eq : forall
  {T : Type} {R : relation T} `{! Equivalence R}
  (P : quot R -> Type)
```

```
  (f : forall (t : T), P (to_quot t))
  (r : forall (x y :T) (e : R x y),
    (quot_ext e) # (f x) = f y :> P (to_quot y))
  (x : T),
  quot_rect P f r (to_quot x) = f x.
```

In the type of `quot_rect`, `e # t` denotes the transport of `t` along the proof of equality `e`, necessary to bridge the gap between `f x : P (to_quot x)` and `f y : P (to_quot y)`.

B Specialised quotient induction

The specialised variants of `quot_rect` are the following:

```
Definition quot_rec
  {T : Type} {R : relation T} `{! Equivalence R}
  {P : Type} (f : T -> P) `{p : Proper _ (R ⟹ eq) f} :
  quot R -> P := …

Definition quot_rect_irr
  {T : Type} {R : relation T} `{! Equivalence R}
  (P : quot R -> Type)
  (f : forall (t : T), P (to_quot t))
  `{p : forall (t : T), ProofIrrel (P (to_quot t))} :
  forall u : quot R, P u := …
```

where `ProofIrrel` is the following type class for h-propositions:

```
Class ProofIrrel (T : Type) :=
  proof_irrel : forall x y : T, x = y.
```