Tactic and (co)algebraic reasoning

Rodrigo Raya

Advisor: Prof. Viktor Kunčăk

Laboratory for Automated Reasoning and Analysis (LARA)

January 21, 2020

Abstract

The purpose of this report is to introduce tactical reasoning and the theory of (co)algebras in the design of the Isabelle proof assistant. In a previous presentation, I studied how tactical reasoning was introduced by Milner in the LCF system. In this work, I study how tactical reasoning is achieved in one of LCF's descendants. Two particular tactics were developed in the context of mathematical theorem proving. Furthermore, I describe the general theory of (co)algebras and report on its use in Isabelle to provide a solid foundation for its (co)datatype package. Further research is needed to compare this approach to other systems that reduce the study of coalgebras to algebras and discuss the advantages and disadvantages of both approaches.

1 Tactical reasoning

In [13], Milner discussed the introduction of tactics and tacticals. He implemented these features in the Edinburgh LCF proof assistant. LCF, the mechanization of Scott's Logic of Computable Functions, was probably the first theoretically based yet practical tool for machine assisted proof construction and was quoted as one of the reasons why Milner received the Turing Award in 1991. A summary of this work can be seen in [16].

Modern proof assistants such as Isabelle provide users with a large collection of specialized tactics to facilitate the theorem proving process. These tactics come in three different levels of abstraction. Users are normally presented the Isar language [25]. This language permits modular reasoning similar to that found in normal high-level proofs. The underlying implementation uses a metalanguage written in ML that encodes the logic and the transformations allowed on it. This is the level in which Milner developed his theory of tactics and our study goal in this semester project. Finally, the Eisbach language was developed to provide users with a high-level language to design new proof methods by reusing the predefined ones.

Figure 1 gives the high-level representation of Isar proof processing in contraposition with the usual state of things in tactical theorem proving. The richness of Isar language greatly facilitates the proving process. However, I illustrate in this section some examples in which the user might need to resort to specialized tactics, forcing us to descend to the metalanguage of the system.

I give first some background references that may help during the process of learning to program in Isabelle/ML. The Isabelle/Isar Implementation manual comes with the distribution and provides a detailed overview of the Isabelle/ML level. I would recommend [25] for the foundations of the system. For a practical introduction to program Isabelle/ML is the Isabelle Cookbook [24]. I focused on the study of on tactical reasoning. However, the sections teaching how to create new proof commands, parsing specifications and writing new definitional packages helped me in understanding what is the connection between Isabelle/Isar and Isabelle/ML. A fresh perspective of the system can be found on [26].

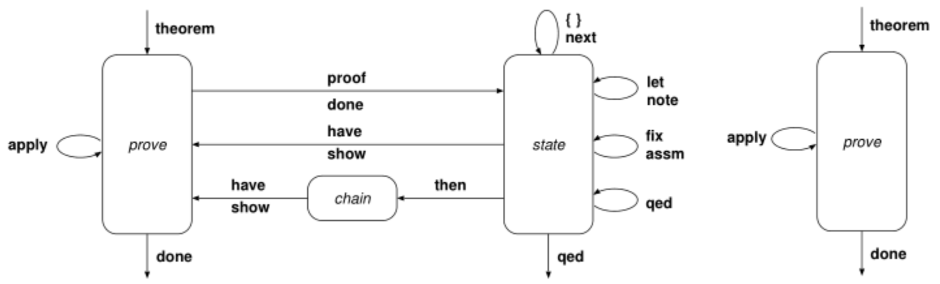

Figure 1: Transitions of Isar proof processing and tactical theorem proving (taken from [25])

1.1 An associativity proof

I recently finished a proof of the group law for elliptic curves on Edwards form [19]. The hardest part on this classical exercise in theorem proving is showing that the group law is associative. In my setting, this amounts in showing that the following equation holds:

$$(p_1 \oplus_i p_2) \oplus_j p_3 = p_1 \oplus_k (p_2 \oplus_l p_3)$$

Here $i, j, k, l \in \{0, 1\}$ and $\oplus_0, \oplus_1$ are well-defined operations on elliptic curves points. After completing the proof I found that the different proofs were actually very similar. The proof can be found in [17]. In essence, we abstract the different 16 proofs into a single function yielding for each parameter combination the corresponding theorem. The theorem to be proven as well as the method of proving it is a function of the input parameters. This process can be observed in function *concrete_assoc*.

To the side, I also investigated the design of the rewrite tactic. This method supersedes the substitution tactic which needs to specify the position of the subterm to which we apply certain rule. In [14], the authors explain the design of a language to specify subterms to be rewritten with a pattern-matching approach. Internally, the implementation added commands *in* (selecting of subterms of a term) and *concl* (selecting the conclusion of the goal as the target term) to our apply-style commands. However, I only found this out when debugging the ML implementation for the rewriting library. This leads me to the last remark of this section.

Some tips on how to debug ML source code may be appreciated for future users. In [8], it is explained how to edit theories that are read-only by default. In [9], it is explained how to browse ML files. For limited debugging support of ML code, see chapter 5 of the JEdit manual.

This section provides a good example for the section on structured proofs in the Isabelle Cookbook.

1.2 Rewriting sines

I present a small tactic illustrating some common idioms for Isabelle/ML. The idea is to rewrite sine expressions whose arguments contain sums of multiples of π. I chose to encode this as a *simproc*. These simprocs are special rewriting procedures called by the simplifier to solve theory-specific goals. Other examples, like a tactic to facilitate the application of Taylor theorem contribute to bridge the gap between mathematicians expectations and theorem proving tools.

The high-level idea of the tactic [7] is as follows:

1. Define a function SIN_SIMPROC_ATOM x n = x + of_int n * pi.

2. Write a conversion sin_atom_conv that rewrites of_int n * pi to SIN_SIMPROC_ATOM 0 n and everything else into SIN_SIMPROC_ATOM x 0.

3. Write a conversion that descends through +, applies sin_atom_conv to every atom, and then applies some kind of combination rule like:

   SIN_SIMPROC_ATOM x1 n1 + SIN_SIMPROC_ATOM x2 n2 =
   SIN_SIMPROC_ATOM (x1 + x2) (n1 + n2).

4. In the end, I have rewritten the original term to the form sin (SIN_SIMPROC_ATOM x n), and then I apply some suitable rule to that.

A conversion is essentially a function of type cterm → thm which takes a cterm ct and returns a theorem of the form ct ≡ . . . , i.e. it's a form of deterministic rewriting (a conversion can also fail by throwing a CTERM exception, by convention). There are a lot of combinators for building and using these in Pure/conv.ML

The resulting theorem is then provided to the user transparently, since the simproc is activated by the simplifier. The details of the implementation can be seen in [18].

2 (Co)algebras

I also tried to answer the question: what is an algebraic datatype? I studied the answer to this question from the point of view of category theory [1]. This theory models (co)datatypes and (co)induction through the concept of coalgebras [21] [5] [20]. It is somewhat surprising that these concepts have found their way into the implementation of proof assistants [23] [2]. There seem to be problems with other implementations reducing coinduction to induction [3] [22] [10] [6].

2.1 Theoretical background

Let $F : Set \to Set$ be a functor. An F-algebra is given by a set A and a mapping $\alpha : F(A) \to A$. α is called the structure mapping of the F-algebra. A F-homomorphism f between two F-algebras $(A, \alpha), (B, \beta)$ is a mapping that makes the following diagram commutative:

$$\begin{tikzcd}
F(A) \arrow[r, "F(f)"] \arrow[d, "\alpha"] & F(B) \arrow[d, "\beta"] \\
A \arrow[r, "f"] & B
\end{tikzcd}$$

One can form a category Set^F whose objects are all F-algebras and its morphisms are the families of F-homomorphisms. Given an F-algebra $\mathcal{A} = (A, s)$, a subalgebra of $\mathcal{A}$ is given by a $S \subseteq A$ together with a morphism $\beta_S : F(S) \to S$ such that the inclusion $i : S \to A$ is an homomorphism. The notions of coalgebra, coalgebra homomorphism, category Set_F of all coalgebras and subcoalgebra are obtained reversing the arrows in the above definitions.

An initial F-algebra is an initial object in Set^F. A terminal F-coalgebra is a terminal object in Set_F. As a consequence, if any of these exist, it is unique up to isomorphism. A classical result by Lambeck states that the structure mapping of an initial algebra (dually, a final coalgebra) is an isomorphism. As a consequence, the powerset functor $\mathcal{P} : Set \to Set$ cannot have initial or final coalgebras since their existence would contradict Cantor's theorem.

A relation $R \subseteq S \times T$ is an F-congruence if there exists an F-algebra structure (R, γ) such that the projections π_i are F-homomorphisms:

$$\begin{tikzcd}
F(S) \arrow[d, "\alpha"] & F(R) \arrow[l, "F(\pi_1)"'] \arrow[r, "F(\pi_2)"] \arrow[d, dashed, "\gamma"] & F(T) \arrow[d, "\beta"] \\
S & R \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & T
\end{tikzcd}$$

dually, one gets the notion of F-bisimulation. If one defines the diagonal of a set A, as the set:

$$\Delta(A) = \{(a, a).a \in A\}$$

one can state the (co)induction principles as follows:

- Induction: every congruence relation on an initial algebra contains the diagonal.
- Coinduction: every bisimulation relation on a final coalgebra contains the diagonal.

it is straight-forward to reduce the traditional induction principle on natural numbers, coinduction principle on streams or the view on (co)induction as least (greatest) fixed points of F to the above principles.

2.2 Functional programming and (co)algebras

Let $A_1, \ldots, A_k$ be sets and $n_1, \ldots n_k \in \mathbb{N}$. F is polynomial if it is of the form:

$$F(X) = A_1 \times X^{n_1} + \ldots + A_k \times X^{n_k}$$

where $\times$ is the cartesian product and $+$ is the disjoint union of sets. In this categorical view, a datatype is the initial algebra and a codatatype is the final coalgebra of a polynomial functor. If (T, c) is an initial algebra (final coalgebra) for F, one can write:

$$c : A_1 \times T^{n_1} + \ldots + A_k \times T^{n_k} \to T$$

and decompose $c = [c_1, \ldots, c_k]$ as:

$$c_i : A_i \times T^{n_i} \to T$$

Since $+$ stands for the disjoint union of sets. By Lambek's lemma, T is completely determined by the c_i.

In a programming language, the corresponding constructs are written as follows:

$$\begin{aligned} \text{datatype } T &= c_1 \ of \ A_1 \times T^{n_1} \\ &\vdots \\ &| = c_k \ of \ A_k \times T^{n_k} \end{aligned} \qquad\qquad \begin{aligned} \text{codatatype } T &= c_1 \ of \ A_1 \times T^{n_1} \\ &\vdots \\ &| = c_k \ of \ A_k \times T^{n_k} \end{aligned}$$

Geuvers [11] gives a clear account on how iteration, primitive recursion and case analysis can be understood categorically. For instance, checking initiality of datatypes leads to definitions by iteration. To see an example, consider the natural numbers $\mathbb{N}$ as an algebra over the functor $F(X) = 1 + X, F(f) = id + f$ where $1 = \{*\}$ and $*$ is a non-natural number. The structure map we consider is $[zero, succ]$ where $zero$ assigns $*$ to 1 and $succ$ assigns each natural number to its successor. We show that there exists a unique f such that the following diagram commutes:

$$\begin{tikzcd} 1 + \mathbb{N} \arrow[r, "F(f)"] \arrow[d, "{[zero,succ]}"'] & 1 + X \arrow[d, "\alpha"] \\ \mathbb{N} \arrow[r, dashed, "\exists! f"'] & X \end{tikzcd}$$

That is, $\alpha \circ N(f) = f \circ [zero, succ]$ or equivalently $f(0) = \alpha(*)$ and $\forall x \in \mathbb{N}.f(x+1) = \alpha(f(x))$. By natural induction, there is a unique f satisfying this definition. Since f is defined iterating over α, it is sometimes denoted $iter(\alpha)$. More generally, we can see our initial algebra endowed with an iterator mapping $iter$ sending each α to the function f it defines.

Primitive recursion on natural numbers can be encoded with the following scheme:

- $h(0) = g_1(*)$
- $h(succ(n)) = g_2(h(n), n)$

If $g : 1 + B \times \mathbb{N} \to B$ is given, the above definition corresponds to the existence of a unique mapping h making the following diagram commutative:

$$\begin{tikzcd} 1 + \mathbb{N} \arrow[r, dashed, "F\langle h{,}id\rangle"] \arrow[d, "{[zero,succ]}"'] & 1 + B \times \mathbb{N} \arrow[d, "g"] \\ \mathbb{N} \arrow[r, dashed, "\exists! h"'] & B \end{tikzcd}$$

In general, $h = \pi_1 \circ iter\langle g, in \circ F(\pi_2)\rangle$. Since h is defined recurring to g, we write it $h = rec(g)$ and for any initial algebra we can establish a mapping rec assigning to each extra parameter supplier g, the function h it defines.

Finally case analysis on natural numbers corresponds to the following scheme:

- $h(0) = g_1(*)$

- $h(succ(n)) = g_2(n)$

If $g : 1+\mathbb{N} \to B$ is given, the above definition corresponds to the existence of a unique mapping h making the following diagram commutative:

$$\begin{tikzcd}
1+\mathbb{N} \arrow[d, "{[zero,succ]}"'] \arrow[dr, "g"] & \\
\mathbb{N} \arrow[r, dashed, "\exists! h"'] & B
\end{tikzcd}$$

Coiteration, corecursion and cocase analysis are encoded as the dual of the above notions.

2.3 Building initial and final (co)algebras

In this section, I state some theorems that guarantee the existence of usual (co)datatypes. The first result is due to Adámek and Barr and can be seen as a generalization of Kleene's fixed-point theorem:

- Let $\mathcal{C}$ be a category with initial object 0 and colimits for any ω-chain. If $F : \mathcal{C} \to \mathcal{C}$ preserves the colimit of the initial ω-chain, then the initial F-algebra is $\mu(F) = \mathrm{colim}_{n<\omega} F^n 0$.
- Let $\mathcal{C}$ be a category with terminal object 1 and limits for any ω^{op}-chain. If $F : \mathcal{C} \to \mathcal{C}$ preserves the limit of the terminal ω^{op}-chain, then the terminal F-coalgebra is $\nu(F) = \lim_{n<\omega^{op}} F^n 1$.

As a consequence, we have:

- Any polynomial functor on Set admits an initial algebra and a final coalgebra.

For the construction in Isabelle, there are two points to be made. First, the above theorems do not apply to all the functors of interest for Isabelle theories. Some examples that are not dealt with include: the finite powerset ('a fset), countable powerset ('a cset), finite multisets ('a multiset) and the discrete probability distributions ('a pmf). As an example, finite depth finitely branching trees can be defined with this functor:

```
datatype 'a tree = Node 'a ('a tree fset)
```

Instead, the notion of bounded endofunctor is introduced, see section 4.6 of [12]. As remarked there both approaches seem to be related by transfinite induction. A second point to be made is that the above construction is not enough even for the simple polynomial functors that constituted our (co)datatypes. This is because arbitrary limits require reasoning about infinite type families, which goes beyond HOL capabilities.

2.4 The Isabelle approach

In a real proof assistant, the user specifies some type constructors, such as:

```
'a List = Nil | Cons 'a ('a List)
```

which can be understood as a fixed point equation $X = 1 + A \times X$ on types. The Isabelle's (co)induction package has to derive appropriate theorems for such an equation. Given such a user definition, how do we derive the corresponding functorial structure? Given the functorial structure, how do we derive in the logic the (co)induction principles, introduction rules and (co)case analysis? This study can be found on [23] and the practical implementation in [15] which shows in apply-style the rewriting steps performed by the definitional package.

2.4.1 Functorial structure

HOL can be modeled as a category using the universe of types U as objects and functions between types as morphisms. A type constructor $(\alpha_1, \ldots, \alpha_n)F$ together with a mapping:

$$\mathrm{Fmap} : \overline{\alpha} \to \overline{\beta} \to \overline{\alpha}F \to \overline{\beta}F$$

such that Fmap id = id and $\mathrm{Fmap}(\overline{g} \circ \overline{f}) = \mathrm{Fmap}\,\overline{g} \circ \mathrm{Fmap}\,\overline{f}$ gives the notion of functor $(\mathrm{F}, \mathrm{Fmap})$ whose action on objects is F and on morphisms is Fmap. In order to formalize initial and final (co)algebras, the notion of functor is enriched into the notion of bounded natural functor. An n-ary bounded natural functor is a tuple (F, Fmap, Fset, Fbd) where:

- F is an n-ary type constructor.
- Fmap : $\overline{\alpha} \to \overline{\beta} \to \overline{\alpha} F \to \overline{\beta} F$
- $\forall i \in \{1, \ldots, n\}$. $\mathrm{Fset}_i : \overline{\alpha} F \to \alpha_i$ set
- Fbd is an infinite cardinal number.

satisfying the following:

- (F,Fmap) is a binary functor.
- $\mathrm{Fset}_i : \overline{\alpha} F \to \alpha_i$ set is a natural transformation from: $((\alpha_1, \ldots, \alpha_{i-1}, \_, \alpha_{i+1}, \ldots, \alpha_n)F, \mathrm{Fmap})$ to (set, image).
- (F,Fmap) preserves weak pullbacks.
- $\forall a \in \mathrm{Fset}_i\, x, i \in \{1, \ldots, n\}. f_i\, a = g_i\, a \implies \mathrm{Fmap}\, \overline{f}\, x = \mathrm{Fmap}\, \overline{g}\, x$
- The following cardinal bound conditions hold: $\forall x : \overline{\alpha} F, i \in 1, \ldots, n. |\mathrm{Fset}_i\, x| \leq \mathrm{Fbd}$

There is a key observation to this definition. The authors call it the *shape and content* intuition. Indeed, if one writes the definition of natural transformation for the second axiom for the inclusion mapping $f = i$, one obtains that $\mathrm{Fmap}\, i = i$. So inclusion lifts to the inclusion. Thus, the notion of subalgebra is greatly simplified in this context. If (A, t) is a F-subalgebra of the algebra (B, s), then the inclusion $i : A \to B$ is an F-algebra homomorphism and $\mathrm{Fmap}\, i = i$ so the subalgebra equation simplifies to $s \circ i = i \circ t$ which implies that $t = s|_{F(A)}$. Thus, for a BNF, a subalgebra can be given by a subset together with a particular restriction of the structure mapping.

2.4.2 Induction principle

In this section, I describe the construction of the initial algebra in Isabelle/HOL. The derived induction principles can be seen in [23]. The construction happens in two stages: first we build a weakly initial algebra and then the initial algebra is built from the former. The authors present the second step with a construction *from above* of the minimal algebra generated by $\emptyset$, but in practice, they implement a construction *from below* that is less intuitive since it uses transfinite induction. Here, I describe the abstract construction which benefits from the remarks made in the previous section on the characterization of subalgebras for bounded natural functors.

Let $\mathcal{A} = (A, s)$ be an F-algebra. Set $M_s = \bigcap_{B.(B,s) \text{is a subalgebra of (A,s)}} B$ then:

$$\mathcal{M}(\mathcal{A}) = \left(M_s, s\Big|_{M_s} \right)$$

is the F-subalgebra generated by $\emptyset$. We call it the minimal algebra generated by $\mathcal{A}$. $\mathcal{M}(\mathcal{A})$ is said to be the subalgebra generated by $\emptyset$ in the sense that it is the intersection of all subalgebras containing $\emptyset$.

There exists at most one morphism from $\mathcal{M}(\mathcal{A})$ to any other F-algebra (Y, t) (*). Since, if f, g are two such morphisms, we can show that:

$$B = \mathcal{M}(\mathcal{A}) \cap \{x \in \mathcal{A}. f(x)_= g(x)\}$$

is a F-subalgebra of $\mathcal{A} = (A, s)$. Indeed, by our remarks, it suffices to note that $M_s \cap \{x \in \mathcal{A}. f(x)_= g(x)\} \subseteq M_s$ and consider the structure map $s|_B$. This leads to a subalgebra of $\mathcal{M}(\mathcal{A})$ which can be naturally seen as a subalgebra of $\mathcal{A}$. By definition of $\mathcal{M}(\mathcal{A})$, $M_s \supseteq B$ and thus $\forall x \in M_s. f(x) = g(x)$. Thus, the morphisms are equal. We will use this observation in what follows.

Here is the naive approach to the construction of an initial F-algebra.

1. Set $\mathcal{R} = \prod \{\mathcal{A}. \mathcal{A} \text{ is an algebra}\}$.

2. Given an algebra $\mathcal{A}$, note h the projection morphism from $\mathcal{R}$ to $\mathcal{A}$.
3. Then $h|_{\mathcal{M}(\mathcal{R})}$ is the unique morphism between $\mathcal{M}(\mathcal{R})$ and $\mathcal{A}$.
4. Since the construction does not depend on the chosen algebra $\mathcal{A}$, $\mathcal{M}(\mathcal{R})$ is the desired initial algebra.

The naive approach cannot be encoded in HOL. First, one cannot quantify over infinite type collections. Second, the product of the carrier sets of all algebras, fails itself to be a set. Instead, we split the construction in two phases:

1. Given an F-algebra $\mathcal{A}$ we know that there exists at most one morphism $\mathcal{M}(\mathcal{A}) \to \mathcal{A}$. But from our remarks above, for bounded natural functors, the inclusion is one such morphism. So there is exactly one morphism $g : \mathcal{M}(\mathcal{A}) \to \mathcal{A}$.

2. On the other hand, we would like some set of algebras $\mathcal{R}$ such that from $\mathcal{R}$ there is a unique morphism to any $\mathcal{M}(\mathcal{A})$. The strategy is two find a sufficiently large type T_0 such that its cardinality is an upperbound for any $\mathcal{A}$. The reason is a theorem stating that if we can bound the cardinality of a set by some ordinal then the set has a bijective representation on the carrier of the wellorder inducing the ordinal. The crucial lemma is ex_bij_betw:

   $$|A| \leq_o (r ::' b\,\mathrm{set}) \implies \exists f\ B ::' b\,\mathrm{set}.\mathrm{bij\_betw} f\ B\ A$$

   More precisely, the package shows that for all algebras $\mathcal{A}$, if M denotes the carrier of $\mathcal{M}(\mathcal{A})$ then $|M| \leq_o 2 \wedge_c k$. Then, the package witnesses a type T_0 with this cardinality and defines:

   $$\mathcal{R} = \prod \{\mathcal{A}.\mathcal{A} = (A, s) \text{ is an algebra with structure map } s : T_0 F \to T_0\}$$

   By means of *ex_bij_betw* the minimal algebras $\mathcal{M}(\mathcal{A})$ have isomorphic representants on a component of $\mathcal{R}$. Thus, the corresponding projection from the product to $\mathcal{M}(\mathcal{A})$ restricted to $\mathcal{M}(\mathcal{R})$ is the unique morphism f between the two.

   Then, $f \circ g : \mathcal{M}(\mathcal{R}) \to \mathcal{A}$ is a suitable morphism. One shows it is the unique morphism between the two with a similar argument as in our observation (*).

3 Conclusion

For a follower of computational trinitarianism [4], the design and implementation of proof assistants is the perfect playground mixing logic, language design and, as we have seen here, some notions of category theory. A proof assistant is not a programming language and thus, the focus is on the foundations and not on efficiency. I believe that category theory contributes to clarify these foundations and that some of the technology developed for proof assistants may influence future programming languages or software verifiers.

In this work, I have tried to move my understanding from the user level of the proof assistant Isabelle into the design of some of its features, namely tactical reasoning and its formalization of (co)datatypes. A natural continuation of this work would be the design of a system from scratch, focusing on the reasoning principles behind the system and providing a modern implementation in Scala.

References

[1] Michael Barr and Charles Wells. *Category theory for computing science*. Vol. 49. Prentice Hall New York, 1990.

[2] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. "Cardinals in Isabelle/HOL". In: *International Conference on Interactive Theorem Proving*. Springer. 2014, pp. 111–127.

[3] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. "Foundational extensible corecursion: a proof assistant perspective". In: *ACM SIGPLAN Notices*. Vol. 50. 9. ACM. 2015, pp. 192–204.

[4] *Computational trinitarianism*. `https://ncatlab.org/nlab/show/computational+trinitarianism`. 2017.

[5] Klaus Denecke and Shelly L Wismath. *Universal algebra and coalgebra*. World Scientific, 2009.

[6] *Deprecate positive coinductive types*. `https://github.com/coq/coq/pull/7536`. 2018.

[7] Manuel Eberl. *Rewriting sine using simprocs in Isabelle*. `https://stackoverflow.com/questions/58859417/rewriting-sine-using-simprocs-in-isabelle`. [Online; accessed 01-01-2017]. 2019.

[8] *Explore finished theories in Isabelle*. `https://stackoverflow.com/questions/55028386/explore-finished-theories-in-isabelle`. [Online; accessed 01-01-2017]. 2019.

[9] *Exploring ML files in Isabelle*. `https://stackoverflow.com/questions/59091016/exploring-ml-files-in-isabelle`. [Online; accessed 01-01-2017]. 2019.

[10] *Fix loss of positive coinductive SR by restricting return clauses*. `https://github.com/coq/coq/pull/10764`. 2019.

[11] Herman Geuvers and Erik Poll. "Iteration and primitive recursion in categorical terms". In: (2007).

[12] Bart Jacobs. "Introduction to coalgebra". In: *Towards mathematics of states and observations* (2005).

[13] Robin Milner. "The use of machines to assist in rigorous proof". In: *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences* 312.1522 (1984), pp. 411–422.

[14] Lars Noschinski and Christoph Traut. "Pattern-based Subterm Selection in Isabelle". In: ().

[15] *Operations on Bounded Natural Functors*. `https://www.isa-afp.org/entries/BNF_Operations.html`. 2017.

[16] Rodrigo Raya. *Presentation of the paper "The use of machines to assist in rigorous proof"*. `https://github.com/rjraya/SAV/blob/master/presentation/presentation.pdf`. [Online; accessed 01-01-2017]. 2017.

[17] Rodrigo Raya. *Proof of associativity for elliptic Edwards curves*. `https://github.com/rjraya/Isabelle/blob/master/curves/Hales.thy`. [Online; accessed 01-01-2017]. 2019.

[18] Rodrigo Raya. *Sine tactic implementation*. `https://github.com/rjraya/Tactics/blob/master/Experiment3.thy`. [Online; accessed 01-01-2017]. 2019.

[19] Rodrigo Raya and Manuel Eberl. "Gauss Sums and the Pólya–Vinogradov Inequality". In: *Archive of Formal Proofs* (Dec. 2019). `http://isa-afp.org/entries/Gauss_Sums.html`, Formal proof development. ISSN: 2150-914x.

[20] Jan Rutten. "The Method of Coalgebra: exercises in coinduction". In: (2019).

[21] Michael B Smyth and Gordon D Plotkin. "The category-theoretic solution of recursive domain equations". In: *SIAM Journal on Computing* 11.4 (1982), pp. 761–783.

[22] *Towards a fix to the loss of subject reduction with corecursion?* `https://github.com/coq/coq/issues/6768`. 2018.

[23] Dmytro Traytel. "A category theory based (co)datatpye package for Isabelle/HOL". Master Thesis. TUM, 2012.

[24] Christian Urban. *The Isabelle Cookbook*. `https://nms.kcl.ac.uk/christian.urban/Cookbook/`. [Online; accessed 01-01-2017]. 2019.

[25] Markus M Wenzel. "Isabelle/Isar—a versatile environment for human-readable formal proof documents". PhD thesis. Technische Universität München, 2002.

[26] Burkhart Wolff. "My Personal, Ecclectic Isabelle Programming Manual". In: (2019).