A Category of Finite Ordinals and Functions with Computable Pullbacks and Pushouts in Rocq

Samuel Arsac[a]

[a]ENS de Lyon, CNRS, UCBL1, LIP, Plume team, UMR 5668, Lyon, France

Introduction

As part of a formalization of category theory for graph rewriting [1] and in the context of the ANR CoREACT project, which aims to formalize diagrammatic reasoning in Rocq, we are interested in being able to efficiently compute some basic operations within Rocq, namely pullbacks and pushouts [2].

Our library already implements the needed definitions and some instances of categories. These instances are not satisfactory when it comes to computation, especially when it comes to pushouts. Some of them require some axioms to compute pushouts, such as the category of types and functions or the category of simple directed graphs and graph homomorphisms, which require predicate extensionality and constructive definite description to build quotients. For other instances that are not reliant on axioms, the computation is not possible for other reasons, for instance the category of finite types and finite functions, defined with `fintype` and `finfun` from MathComp, doesn't have computable pushouts and pullbacks since computation is locked in MathComp.

We thus implement an instance with finite ordinals and functions. The simplicity of the objects makes operations easier to compute, and since finite ordinals are a fortiori finite types, we can aim at using them to compute pullbacks and pushouts in finite types.

We know that a pullback can be built from a product and an equalizer, and dually a pushout from a coproduct and a coequalizer [2]. Using this result, it is enough to prove that our category has all products and equalizers to get all pullbacks, and similarly for pushouts. This is both easier to prove, as we nicely separate the tasks, and stronger, since having all pullbacks does not imply having all equalizers. This strategy is inspired from the AlgebraicJulia project [3].

To ensure the computation goes well, we have to make sure to get rid of proof contents as much as we can, and to avoid having terms uncontrollably growing in size. In addition, as we use theoretical results from the library, such as the construction of pullbacks (pushouts) from (co)equalizers and (co)products, we need to make sure that their proofs preserve computation and are efficient enough.

The code is available on a GitLab repository.[1]

Formalization

Setup

This formalization relies on an existing library for category theory, which notably contains definitions for pullbacks and pushouts. For ordinals we use the definition from the MathComp library. In Mathcomp, given some `n: nat`, elements of the finite ordinal `n` are simply natural numbers `m: nat` with a proof of `m < n`.

We can then define our category of ordinals by taking `nat` as the type of objects and functions between ordinals as morphisms.

However MathComp does not fully cover our use case, since it locks computation. For instance, the function to enumerate the elements of an ordinal (to get a list `[0, 1, ..., n-1]`) does not reduce to a list. We have to define our own enumeration function and then show that it produces the expected output.

A major concern when doing computation within Rocq is to keep the size of proof terms limited. In our case the proof of inequality carried by the elements of an ordinals grows quickly, even when we are careful to make all proofs opaque. We use a common trick to regularly replace these proofs with `erefl`.

[1] https://gitlab.com/SamuelArsac/graph-rewriting

Similarly, terms for morphisms tend to grow. To counter that, we evaluate them to get the list of their images, and the new definition then accesses that list to obtain the return value. While this can be detrimental to the performance in some specific cases, this ensures a consistent performance once the computation is done. Additionally, most morphisms are fully evaluated anyway when they are used to compute other objects or morphisms, so this is rarely detrimental in practice.

Pullbacks & pushouts

As mentioned in the first part, we start by proving that Ord has all products and coproducts. This is not difficult, since the product in the category is the product in `nat`, and the coproduct is the sum. The definition of the projections and coprojections is also easy, especially since MathComp provides functions to treat sums of natural numbers as direct sums on sets.

Equalizers in the category of sets and functions are $\{x \in A \mid f(x) = g(x)\}$, where $f, g : A \to B$. In the category of ordinals, this corresponds to filtering the enumeration of the elements of A. This is rather straightforward to define and prove correct. Combined with the result mentioned in the introduction, this gives a proven construction for pullbacks.

Coequalizers on the other hand are more complicated; in the category of sets, we can define them as $B/\sim$ with $\sim$ being the smallest equivalence relation containing $\{(x, y) \in B^2 \mid \exists a \in A, f(a) = x \wedge g(a) = y\}$ (the relation "have a common preimage"), which means we need a way to compute quotients of types. To achieve this we have implemented a basic union-find data structure which allows us to get the desired equivalence classes by successive unions. We then have the quotient object by counting the number of equivalence classes. The correctness of this implementation of union-find is admitted for the moment. Once this is defined, we can have pushouts in the same way as pullbacks. Another approach we attempted was to only have pushouts along monomorphisms, as this is enough for our intended use case. In this case a quotient is no longer needed, but several other operations have to be combined, which is cumbersome to implement and ultimately no more efficient than the previous method when using lists.

An extra functionality we have implemented is a function to check that a given commutative square is a pushout. It works by building the canonical pushout, and then checking that it is isomorphic to the given one by building the canonical morphism from the canonical pushout to the given square and testing if this morphism is an iso. It is almost proven correct, only some small details of the proof are still admitted.

Conclusion & future work

We have implemented a category of finite ordinals and functions with computable and mostly proven pullbacks and pushouts.

The last step to complete this work would be to prove the union-find structure correct, which would complete the proof for pushouts. Another option could be to replace the implementation by an existing, proven one [4].

Using lists and unary numbers greatly limits the performance of our implementation. We could improve it by using better data structures, such as binary integers with the `positive` type and positive maps or primitive arrays and primitive integers. For the latter, the standard library is currently very limited in definitions and lemmas, so we would need to use an extended library or write our own. We could also make some use of parametricity [5] to implement this.

The category theory library contains a category of finite types and functions, using `fintype` from MathComp. We could use the category of ordinals to compute pullbacks and pushouts in this category, using the function from ordinals to finite types and vice-versa. Such functions are provided by MathComp, but do not compute, so we would have to write our own.

Acknowledgements

This work was partly funded by the ANR CoREACT project. We especially thank Kazuhiko Sakaguchi for helping to refactor the code, as well as Nicolas Behr, Manuel Catz, Russ Harmer and Damien Pous for their valuable input.

Bibliography

[1] S. Arsac, R. Harmer, and D. Pous, "Adhesive Category Theory for Graph Rewriting in Rocq," in *Proceedings of the 15th ACM SIGPLAN International Conference on Certified Programs and Proofs*, in CPP '26. New York, NY, USA: Association for Computing Machinery, Jan. 2026, pp. 59–74. doi: 10.1145/3779031.3779105.

[2] J. Adamek, H. Herrlich, and G. Strecker, "Abstract and Concrete Categories - The Joy of Cats," *Reprints in Theory and Applications of Categories*, no. 17, pp. 1–507, 2006.

[3] The AlgebraicJulia Team, "AlgebraicJulia." [Online]. Available: https://www.algebraicjulia.org/

[4] T. Braibant and D. Pous, "Deciding Kleene Algebras in Coq," *Logical Methods in Computer Science*, vol. Volume 8, Issue 1, Mar. 2012, doi: 10.2168/LMCS-8(1:16)2012.

[5] C. Cohen, E. Crance, and A. Mahboubi, "Trocq: Proof Transfer for Free, With or Without Univalence," in *Programming Languages and Systems (ESOP 2024)*, S. Weirich, Ed., in Lecture Notes in Computer Science, vol. 14576. Luxembourg, Luxembourg: Springer Nature Switzerland, Apr. 2024, pp. 239–268. doi: 10.1007/978-3-031-57262-3_10.