idris-ct: A Library to do Category Theory in Idris

Fabrizio Genovese
0000-0001-7792-1375

Alex Gryzlov
0000-0001-6188-0417

Jelle Herold
0000-0002-1966-2536

Andre Knispel

Marco Perone
0000-0002-1004-0431

Erik Post
0000-0002-8111-9593

André Videla
0000-0002-7298-6230

Statebox Team
research@statebox.io

We introduce idris-ct, a Idris library providing verified type definitions of categorical concepts. idris-ct strives to be a bridge between academy and industry, catering both to category theorists who want to implement and try their ideas in a practical environment and to businesses and engineers who care about formalization with category theory: It is inspired by similar libraries developed for theorem proving but remains very practical, being aimed at software production in business. Nevertheless, the use of dependent types allows for a formally correct implementation of categorical concepts, so that guarantees can be made on software properties.

1 Introduction and motivation

The Statebox project [24] is centered around the idea of building a graphical programming language using the well-known correspondence between Petri nets and free symmetric monoidal categories [25]. Trying to turn this idea into practice, our team was confronted with the problem that many research results in the field were strictly theoretic – e.g. [22, 2] – with implementation never to be considered a priority. The effort to overcome these issues led to some novel research – e.g. [9, 10] – but more importantly to the development of some basic computational tools needed to carry out the job, as the one we are going to introduce here.

This paper has the purpose of announcing and describing idris-ct, which is open to external contributions, to the ACT community: We are already receiving pull requests, and would very much appreciate an involvement of the ACT community in this open project. Here our scope will remain broad, and we will explain some design choices at the expense of in-depth technical specification: For instance, the examples we provide may look trivial to expert programmers and researchers. We redirect them to the idris-ct Github repository [23], where everything is worked out and documented in a greater level of detail, and more involved concepts are covered.

2 What is idris-ct?

idris-ct is an Idris library focused on providing verified type definitions of important concepts in category theory, such as “category”, “functor”, “monoidal category”, etc. The project is open source,

John Baez and Bob Coecke (Eds.): Applied Category Theory 2019
EPTCS 323, 2020, pp. 246–254, doi:10.4204/EPTCS.323.16

licensed under AGPL 3.0 [8] (license exemptions are available), and its source code can be found on Github [23]. Installation instructions are available on the repository.

The original requirements laid down for `idris-ct` focused on obtaining a library practical enough to be production-ready in software development for business and enterprise. In this respect, pre-existing solutions – see e.g. [18] – did not fit our requirements. Indeed, we wanted a codebase that allowed us to answer questions such as "What kind of runtime costs are we looking at" and "What would be a more ergonomic API for the users of the library". Similar goals have been set by Rydeheard and Burstall in [21] but, as they themselves declare, "we hoped that [this work] would provide a tool for advanced programming, harnessing the abstraction of category theory for use in program design. These hopes have not really been realized by our work so far." As a result, we felt compelled to develop an alternative solution.

2.1 Why Idris

One might ask why we chose to use Idris for our coding endeavours. The main reason is that one of our priorities is *correctness*: We want to implement the categorical connection between nets and symmetric monoidal categories *formally*.

In order to achieve this goal, we want to employ a programming language that is powerful enough to let us *prove* properties about our code. For example, we want to implement structures so that defining a category in our model is equivalent to *mathematically proving* that it is indeed a category in the "traditional" [14] sense.

At the same time we need a language backed by a strong community, well versed in the problems faced with software *engineering*. Its ecosystem should provide HTTP and parsing libraries, C-FFI, filesystem access and web capabilities. In short, we require a very specific blend of formal methods and pragmatic solutions.

There exist multiple popular functional programming languages allowing different degrees of formalism and pragmatism, such as Coq [27], Haskell [12], Idris [4, 5] and Agda [1, 17]. We decided to go for Idris because we strongly believe in its goal to be a "Pacman-complete" [1] programming language first, and a theorem prover second. This design choice makes an important difference, as it is highlighted by practical projects as [7] and [15]. Additionally, Idris' successor Blodwen [3] already shows promise by featuring linear types and a more efficient compiler.

It is worth noting that our design choices are, at the moment, quite unique in the applied category theory landscape: Other projects, like [20], [21] and [6], are implemented using languages that are either imperative or have a weaker type system, hence coming with less formal guarantees (e.g. composing morphisms using the algorithms in [21] can throw exceptions). Conversely, many projects striving to implement categorical gadgets in a formally verified way are confined to the realm of theorem proving (e.g. [18, 11]), with no plans – to our knowledge – to provide the needed integrations to make them usable in industry.

[1] *Pacman-completeness* is a folklore concept in programming communities representing the idea that having a powerful language is not enough for it to be useful. For instance, PowerPoint presentations are Turing-complete [28], and yet we are not building operating systems with them. Pacman-completeness serves the idea that a programming language is "useful enough" so that you can use it to implement the game "Pacman" without having a seizure.

2.2 Our Idris is literate

Another noteworthy design decision is to write `idris-ct` in *literate* Idris. Literate programming is a paradigm where, instead of prepending a symbol to instruct the compiler that a given line is a comment (such as % in LaTeX), the user does the opposite, prepending a symbol to the lines that *do not* have to be ignored by the compiler.

This programming mode allows for having extensive blocks of text documenting code, and to present the code itself as a story (hence the "literate"). In our case, `idris-ct` can be compiled both to Idris, obtaining a compiled source in the usual sense, or to LaTeX, obtaining a pdf explaining the code in detail.

The choice of documenting our project with literate Idris stems from the fact that `idris-ct` is not aimed exclusively at academia. In fact, we believe that many developers in industry use concepts from category theory without even knowing it – this is surely the case for many people working with Haskell, Purescript and other functional languages. Literate programming constitutes a great solution for these developers to learn, by providing enough space to introduce categorical concepts by example as the code progresses, without forcing them – often used to just skim through READMEs – to resort to external sources to educate themselves.

Note that at the moment not all of our source files are adequately presented as LaTeX documents. This is intended, since it is desirable to have the source in a stable form before putting massive effort towards documenting it.

3 Some excerpts of the code

Let us now dive into the code, showing its design principles. We will present how we can implement the definition of *category*. We deem this useful, both to demonstrate code sintax and because such definition in Idris may feel very different than the "traditional" one, at least to someone not used to dependently typed coding. In `idris-ct`, we define a category as follows:

```
1  record Category where
2    constructor MkCategory
3    obj            : Type
4    mor            : obj → obj → Type
5    identity       : (a : obj) → mor a a
6    compose        : (a, b, c : obj)
7                   → (f : mor a b)
8                   → (g : mor b c)
9                   → mor a c
10   leftIdentity   : (a, b : obj)
11                  → (f : mor a b)
12                  → compose a a b (identity a) f = f
13   rightIdentity  : (a, b : obj)
14                  → (f : mor a b)
15                  → compose a b b f (identity b) = f
16   associativity  : (a, b, c, d : obj)
17                  → (f : mor a b)
18                  → (g : mor b c)
19                  → (h : mor c d)
20                  → compose a b d f (compose b c d g h)
21                   = compose a c d (compose a b c f g) h
```

On line 1, we say that `Category` is a *record*, that is, a collection of Idris values. To build our record we provide a `constructor`, which is a function which receives the single values, as specified in lines

3 – 21, and returns the whole data type. Our record is composed of the fields `obj`, `mor`, `identity`, `compose`, `leftIdentity`, `rightIdentity` and `associativity`. Let us review them in detail.

- `obj : Type` just says that any type can be taken to be the set of objects of our category. The objects of the category will be the terms of that type.
- `mor : obj →obj →Type`, instead, is a *function type*. It says that once we specify two objects `a` and `b` of type `obj` (standing for the homset source and target), then `mor a b` is the type of the morphisms going from `a` to `b`. This type is not definable in languages that are not dependently typed: The type `mor a a` depends on the value `a`!

  Note how we opted to define a category by specifying homsets, and not by considering the set of *all* morphsims as a whole and then defining source and target as functions `mor → obj`, as it is done for instance in [14].
- `identity` asks for an object `a` and produces a term having type `mor a a`. This can be thought of as a function that points out which term of type `mor a a` has to be considered the identity morphism on `a`.
- `compose` expects objects `a, b, c` and two morphsims `f: a →b` and `g: b →c`, and produces a morphism of type `mor a c`, so going from `a` to `c`.

  We decided to make types `a, b, c` explicit for now because it is easier to track the values involved, which is useful for experimenting and debugging. These types will be made implicit (to reduce bookkeeping) as soon as the code is stable enough.
- `leftIdentity` is where Idris's dependent type system really shines: It expects two objects `a` and `b` and a morphism `f: a →b`. From this, it produces a term of type `compose a a b (identity a) f = f`. A term of this type is a *proof* that the right-hand side and the left-hand sides of that equation are equal, so nothing more than a proof that, for a given morphism `f`, composing on the left with the identity morphism of its source amounts to doing nothing. Having this kind of terms that represent proofs would be impossible without dependent types.
- `rightIdentity` is analogous to `leftIdentity`.
- Finally `associativity`, albeit perhaps a bit difficult to parse, implements exactly what its name suggests: It expects three morphisms which can be sequentially composed, and produces a proof that the order of composition does not matter. Hence, a term of type `associativity` is just a proof that three fixed morphisms have associative composition.

This explanation should help to clarify why we claim that our definitions are *verifiably correct*: In `idris-ct` a term of type `Category` is not just a collection of objects, identities, morphisms and compositions, but it also comes with *proofs* that the categorical axioms hold!

To further illustrate this point, *let us prove that Idris types and functions form a category in our framework*. This is useful, since it provides a bridge between Idris' inner workings and the formal environment we are defining. To do this, we will require some helper functions:

```
1 TypeMorphism : Type → Type → Type
2 TypeMorphism a b = a → b
3
4 identity : (a : Type) → TypeMorphism a a
5 identity a = id
6
7 compose :
8     (a, b, c : Type)
```

```
9   →  (f : TypeMorphism a b)
10  →  (g : TypeMorphism b c)
11  →  TypeMorphism a c
12  compose a b c f g = g . f
```

These functions describe the main structure of our category: morphisms, identities and compositions. In detail:

- `TypeMorphism` expects two Idris types `a` and `b`, and returns the type of all Idris functions from `a` to `b`.
- To define identities, we just resort to Idris' identity functions, defined for each type.
- We proceed by defining morphism composition. Unsurprisingly, we set this to be Idris function composition.

It is crucial to note that merely *defining* what the objects, morphisms, identities and compositions are is not sufficient in our library. We also need to provide *proofs* that the categorical laws hold. To accomplish this, we write such proofs as functions:

```
14  leftIdentity :
15       (a, b : Type)
16  →  (f : TypeMorphism a b)
17  →  f . (identity a) = f
18  leftIdentity a b f = Refl
19
20  rightIdentity :
21       (a, b : Type)
22  →  (f : TypeMorphism a b)
23  →  (identity b) . f = f
24  rightIdentity a b f = Refl
```

As before, `discreteLeftIdentity` and `discreteRightIdentity` are conceptually equal, so we focus on the former: We are just using the fact that – via η-reduction – Idris considers composing a function with an identity equal to the function itself. This is denoted in the language using the equality type inhabitant `Refl`. We implement the associativity law in a very similar way:

```
26  associativity :
27       (a, b, c, d : Type)
28  →  (f : TypeMorphism a b)
29  →  (g : TypeMorphism b c)
30  →  (h : TypeMorphism c d)
31  →  (h . g) . f = h . (g . f)
32  associativity a b c d f g h = Refl
```

Here, we rely on the fact that Idris automatically reduces both sides to the same normal form, so we can again use `Refl`. Finally, we can put everything together, and obtain:

```
34  typesAsCategory : Category
35  typesAsCategory = MkCategory
36    Type
37    TypeMorphism
38    identity
39    compose
40    leftIdentity
41    rightIdentity
42    associativity
```

Here we are saying that our set of objects is given by the type of all types. The morphisms are specified by the type `TypeMorphism`, while `identity` and `compose` take care of specifying, respectively, what the identity morphisms and morphism compositions are. Finally, we plug in the laws we implemented above. The category `typesAsCategory` is important, because it allows us to functorially map other categories into actual Idris computations: With it, we can formally translate a morphism into a list of functions our machine has to compute using functors.

3.1 What else can you do, and does it scale?

While the example above may seem a bit limited, `idris-ct` is far richer than this may suggest: We have implemented categories, functors, natural transformations, products and coproducts, initial and terminal objects, monoidal categories, strict monoidal categories and strict symmetric monoidal categories. Moreover, there is work in progress, which could be found in the Github repository, on monads and Kleisli categories, F-algebras and Eilenberg-Moore categories, wiring diagrams of discrete dynamical systems. Curiously, in implementing non-strict, symmetric monoidal categories we seem to have pushed the Idris compiler to its limits [19], and we still need to figure out a way to help it recognize that our code should indeed typecheck.

We also have provided some useful instances of categories, e.g. we proved that categories and functors themselves – that is, terms of type `Category` and `Functor` in our implementation – form a category. This means that we have implemented **Cat** in `idris-ct`. We also proved that monoids are categories, and that they can be used to *freely generate* the objects of a monoidal category. Having done so, we are now focused on implementing free categories, such as the free strict symmetric monoidal category generated by a set of objects and a set of morphisms.

As one can see, at the moment we are focusing on a narrow subfield of category theory: Basic definitions, monoidal categories and limits. This is dictated by internal business reasons, but we hope that the library will be expanded in different directions with the help of the community (in fact, we already received external pull requests defining opcategories, cocones, and dualizing our definitions of product and terminal object).

As for scalability, up to now we mainly found two types of blockers: In defining free structures, we often had to rely on quotients, which are not easily implemented in Idris. This is a well known problem which depends on how dependently typed languages deal with equality, which could be circumvented as explained in [13]. In some circumstances – namely when a quotient-free way of defining the structure we want to implement exists – this issue can be circumvented by just implementing the quotient-free construction as it is. This is the case, for instance, of *free categories*, which can be implemented without relying on quotiens by using the notion of *path category*. [16]

Another source of problems arises when trying to tie our categorical definitions with Idris' internal structure. For example, we want to prove that Idris types and functions do not just form a category, but a monoidal one where the usual product defines the tensor. Since our definition of monoidal category is quite verbose, doing this directly is difficult. Instead, we opted for proving that products and terminal objects define a monoidal structure in the general case, and then use this proof to implement our claim. In practice, this required developing tools to comfortably deal with commutative diagrams and diagram chasing, since dealing with commutative diagrams naïvely makes everything untractable fairly quickly.

Moreover, when dealing directly with Idris functions, sometimes we were forced to assume function extensionality. Natural transformations, for example, are considered equal if their families of morphisms are the same. These families of morphisms are modelled as functions, so extensionality is necessary to arrive at the correct notion of equality. This is not a problem per sé since Idris cannot internally

distinguish between intensional and extensional functions, but we deemed it worth noticing.

4 Open problems and future work

Future work includes extending `idris-ct` via external contributions. Indeed, raising awareness about this project within the research community is the main reason for this submission. Figuring out which companies focused on using applied category theory in developing products, and thinking about ways that could speed adoption of category theory into industry were a big topic of discussion at ACT 2018. We decided to make the `idris-ct` library open source to help this process, and also as an act of gratitude towards the whole ACT community, which helped us figuring out things many times. We would be very happy, then, if other researchers from the ACT community would converge on `idris-ct`, by opening pull requests or by forking the repository, so that the library could be extended beyond the use that Statebox wants to make of it.

On a different note, many type definitions contain arguments that will have to be made implicit. This will require code refactoring, which will surely happen in the future. At the moment we are postponing this task since the conversion of arguments to implicits will result in less bookkeeping (which is good), but it will also make compiler debugging trickier and compilation longer (which is bad). Hence, the team is waiting for the code to become stable and crystallized enough before undertaking such task.

Another direction of future work includes interfacing the `idris-ct` library with `typedefs` [26], the other big Idris project the team is pursuing at the moment. *`typedefs` is a language-agnostic type construction language based on polynomials*, that makes it possible to pass types and terms around between different programming languages with ease. Incredibly, as of now (April 2019), `typedefs` is the most trending Idris repository on Github, a clear signal that the functional programming community considers it to be a useful project. In our long-term vision, developers will be able to use `typedefs` to interface the non-core parts of their projects, such as the user interface, with the core parts – written in Idris using `idris-ct`– where the verified implementation of some categorical gadget will reside. This will allow for an easy passing of data between formally consistent environments, where core evaluations are carried out, and more flexible environments, where the frontend material is provided. In practice, this requires to prove that `typedefs` is a monoidal category, in a similar fashion to what we did with Idris types and functions.

Finally, it has to be considered that the library is still a prototype: It may require tweaking and optimization in order to become fully performant. However, this will also have to wait until the codebase is in a more settled state. Another important consideration in this regard is the status of the successor to Idris, which has been tentatively named Blodwen. It may make sense to postpone optimization efforts only after it has been released and become viable.

Acknowledgements

We want to thank: Our fellow team members at Statebox and the friendly, stimulating environment that Statebox itself provides, which made this contribution possible; our reviewers, which provided valuable advice and observations which we did our best to incorporate in the paper; the whole ACT community, which is always welcoming and supporting.

Finally we want to thank the unsung hero of our library, Github user `mstn` (unfortunately we do not know his/her real identity), that keeps enriching `idris-ct` day after day with valuable commits and pull requests.

References

[1] agda. *Agda Homepage*. URL: https://github.com/agda/agda (cit. on p. 247).

[2] Paolo Baldan, Roberto Bruni, and Ugo Montanari. "Pre-Nets, Read Arcs and Unfolding: A Functorial Presentation". In: *Recent Trends in Algebraic Development Techniques*. Ed. by Martin Wirsing, Dirk Pattinson, and Rolf Hennicker. Vol. 2755. Springer Berlin Heidelberg, 2003, pp. 145–164. DOI: 10.1007/978-3-540-40020-2_8. (Visited on Apr. 20, 2019) (cit. on p. 246).

[3] Edwin Brady. *Edwinb/Blodwen*. URL: https://github.com/edwinb/Blodwen (cit. on p. 247).

[4] Edwin Brady. "Idris, a General-Purpose Dependently Typed Programming Language: Design and Implementation". In: *Journal of Functional Programming* 23.05 (Sept. 15, 2013), pp. 552–593. DOI: 10.1017/S095679681300018X (cit. on p. 247).

[5] Edwin Brady. *Type-Driven Development With Idris*. Manning Publications, 2017. 480 pp. (cit. on p. 247).

[6] Conexus Team. *CQL Homepage*. URL: http://cql.conexus.ai/index.php (cit. on p. 247).

[7] Simon Fowler and Edwin Brady. "Dependent Types for Safe and Secure Web Programming". In: *Proceedings of the 25th Symposium on Implementation and Application of Functional Languages - IFL '13*. The 25th Symposium. ACM Press, 2014, pp. 49–60. DOI: 10.1145/2620678.2620683 (cit. on p. 247).

[8] Free Software Foundation. *GNU Affero General Public License Version 3 (AGPL-3.0)*. 2007. URL: https://www.gnu.org/licenses/agpl-3.0.en.html (cit. on p. 247).

[9] Fabrizio Genovese and Jelle Herold. "Executions in (Semi-)Integer Petri Nets Are Compact Closed Categories". In: *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 31, 2019), pp. 127–144. DOI: 10.4204/EPTCS.287.7 (cit. on p. 246).

[10] Fabrizio Genovese et al. "Computational Petri Nets: Adjunctions Considered Harmful". In: *arXiv* (Apr. 29, 2019). arXiv: 1904.12974 [cs, math]. (Visited on May 2, 2019) (cit. on p. 246).

[11] Jason Gross, Adam Chlipala, and David I. Spivak. "Experience Implementing a Performant Category-Theory Library in Coq". In: *Interactive Theorem Proving*. Ed. by Gerwin Klein and Ruben Gamboa. Red. by David Hutchison et al. Vol. 8558. Springer International Publishing, 2014, pp. 275–291. DOI: 10.1007/978-3-319-08970-6_18. (Visited on Apr. 23, 2019) (cit. on p. 247).

[12] Haskell.org. *Haskell Homepage*. URL: https://www.haskell.org/ (cit. on p. 247).

[13] Dan Licata. *Running Circles Around (In) Your Proof Assistant; or, Quotients That Compute*. 2011. URL: https://homotopytypetheory.org/2011/04/23/running-circles-around-in-your-proof-assistant/ (cit. on p. 251).

[14] Saunders MacLane. *Categories for the Working Mathematician*. Vol. 5. Springer New York, 1978. DOI: 10.1007/978-1-4757-4721-8. (Visited on May 5, 2018) (cit. on pp. 247, 249).

[15] Brian McKenna and Nathan Dots. *Puffnfresh/Iridium*. URL: https://github.com/puffnfresh/iridium (cit. on p. 247).

[16] nCatLab. *Path Category*. URL: https://ncatlab.org/nlab/show/path+category (cit. on p. 251).

[17] Ulf Norell. "Towards a Practical Programming Language Based on Dependent Type Theory". PhD Thesis. Chalmers University, 2007 (cit. on p. 247).

[18] Daniel Peebles et al. *Copumpkin/Categories*. URL: https://github.com/copumpkin/categories (cit. on p. 247).

[19] Marco Perone. *Issue: Type Checker Hangs and Errors*. Issue 4690. Apr. 18, 2019. URL: https://github.com/idris-lang/Idris-dev/issues/4690 (cit. on p. 251).

[20] David Reutter and Jamie Vicary. "High-Level Methods for Homotopy Construction in Associative n-Categories". In: *arXiv* (Feb. 11, 2019). arXiv: 1902.03831 [math]. (Visited on Apr. 23, 2019) (cit. on p. 247).

[21] David E. Rydeheard and Rod M. Burstall. *Computational Category Theory*. Prentice Hall International, 1988. URL: www.cs.man.ac.uk/~david/categories/book/book.ps (cit. on p. 247).

[22] Vladimiro Sassone. "On the Category of Petri Net Computations". In: *TAPSOFT '95: Theory and Practice of Software Development*. Ed. by Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach. Red. by Gerhard Goos, Juris Hartmanis, and Jan Leeuwen. Vol. 915. Springer Berlin Heidelberg, 1995, pp. 334–348. DOI: 10.1007/3-540-59293-8_205. (Visited on May 5, 2018) (cit. on p. 246).

[23] Statebox Team. *Idris-Ct Github Page*. 2019. URL: https://github.com/statebox/idris-ct (cit. on pp. 246, 247).

[24] Statebox Team. *Statebox, Compositional Diagrammatic Programming Language*. 2017. URL: https://statebox.org (cit. on p. 246).

[25] Statebox Team. "The Mathematical Specification of the Statebox Language". In: *arXiv* (June 18, 2019). arXiv: 1906.07629 [cs, math]. (Visited on June 28, 2019) (cit. on p. 246).

[26] Statebox Team. *Typedefs Github Page*. 2018. URL: http://typedefs.com (cit. on p. 252).

[27] The Coq Consortium. *The Coq Proof Assistant*. URL: https://coq.inria.fr/ (cit. on p. 247).

[28] Tom Wildenhain. "On the Turing Completeness of MS PowerPoint". In: *The Official Proceedings of the Eleventh Annual Intercalary Workshop about Symposium on Robot Dance Party in Celebration of Harry Q Bovik's 26th Birthday*. SIGBOVIK 2017. Lulu, 2017, pp. 102–106. URL: http://www.sigbovik.org/2017/proceedings.pdf (cit. on p. 247).