CompilerForCAP – A CATEGORY THEORY AWARE COMPILER
BUILDING AND COMPILING TOWERS OF CATEGORIES

MOHAMED BARAKAT, TOM KUHMICHEL, AND FABIAN ZICKGRAF

1. Towers of categories and compositionality

In our effort to organize computer algebra using category theory we naturally encounter the phenomenon of compositionality. In this section we demonstrate this on one specific example: A canonical decomposition for constructible sets in algebraic varieties was introduced in [BLH22, Def. 2.4] and was applied to compute constructible images of rational morphisms [BLH22, Sections 6,7] and to verify Terao's freeness conjecture of rank 3 simple arrangements with up to 14 hyperplanes in any characteristic [BBJ+21, BK21].

The idea for this canonical decomposition arose as a result of compositional thinking, which in our computer algebra contexts manifests itself in the concept of **towers of category constructors**. We now sketch such a tower consisting of 8 levels that describes the Boolean algebra of constructible sets in an affine scheme $\operatorname{Spec} R$:

($\ell = 0$) Let R be *computable* commutative unital ring. The notion of computability was introduced in [BLH11] and will be defined using categorical vocabulary in Step ($\ell = 2$).

($\ell = 1$) `RingAsCategory`: View R as a pre-additive category on one object.

($\ell = 2$) `AdditiveClosure`: A skeletal model of the (universal) additive closure $R^\oplus$ of R is given by the category of matrices over R, with $\mathbb{N}$ as the set of objects. The category $R^\oplus$ is a self-dual compact closed category with the Kronecker product as tensor product and $1 \in \mathbb{N}$ as tensor unit (corresponding to the free R-module of rank 1). The commutative ring R is *computable* iff $R^\oplus$ algorithmically admits weak kernels and decidable lifts [Pos21a, Section 3].

($\ell = 3$) `SliceCategoryOverTensorUnit`: The slice category $R^\oplus/1$ models the category of explicitly finitely generated ideals. An object, which is a morphism in $R^\oplus$, is given by a column matrix, its entries being the generators of the ideal. This category is not additive anymore but is still finite complete and cocomplete, and carries the induced closed monoidal structure. The coproduct, product, tensor product, and internal hom of two objects model the ideal sum $I + J$, intersection $I \cap J$, product $I \cdot J$, and quotient $I : J$, respectively.

($\ell = 4$) `PosetCategory`: The poset category $\mathcal{I} := \mathcal{P}(R^\oplus/1)$ identifies all parallel morphisms in $R^\oplus/1$ and hence models the poset of finitely generated ideals "forgetting" the generators, i.e., identifying all possible finite generating sets of the same ideal. The poset is a lattice with an induced closed monoidal structure modeling the ideal product and the ideal quotient.[1]

($\ell = 5$) `StablePosetCategory`: The stabilization $\mathcal{R} := \underline{\mathcal{P}(R^\oplus/1)}$ is defined by $J \leq_{\mathcal{R}} I :\iff \top \leq_{\mathcal{I}} I : J^\infty$, where $I : J^\infty$ is the iterated internal hom (= ideal quotient) $(I : J) : \cdots : J$.[2] In this particular context this is nothing but the ideal saturation. This construction ends up identifying the categorical product and the tensor product, i.e., identifying the ideal intersection and the ideal product. In particular, the internal hom is now an exponential. The resulting category is the Heyting algebra of *radical* ideals of R (or, equivalently, of Zariski open sets in $\operatorname{Spec} R$).

($\ell = 6$) `OppositeCategory`: The opposite category $\mathcal{C} := \underline{\mathcal{P}(R^\oplus/1)}^{\mathrm{op}}$ then models the co-Heyting algebra of closed sets in $\operatorname{Spec} R$.

($\ell = 7$) `Differences`: The poset $\mathcal{D} := (\underline{\mathcal{P}(R^\oplus/1)}^{\mathrm{op}})^-$ of formal differences defined by $A - A' \leq_{\mathcal{D}} B - B'$ iff $A \leq_{\mathcal{C}} (A' \vee B)$ and $(A \wedge B') \leq_{\mathcal{C}} A'$ models the meet-semilattice of locally closed subsets of $\operatorname{Spec} R$.

($\ell = 8$) `Unions`: The skeletal thin category $\mathcal{B} := ((\underline{\mathcal{P}(R^\oplus/1)}^{\mathrm{op}})^-)^\cup$ of formal unions models the Boolean algebra of constructible sets of $\operatorname{Spec} R$. In total we get the following tower of category constructors:

```
Unions(Differences(OppositeCategory(StablePosetCategory(PosetCategory(
  SliceCategoryOverTensorUnit(AdditiveClosure(RingAsCategory(R)))))))).
```

The first few steps of the above tower can be altered when considering projective schemes or smooth toric varieties. The above tower is implemented in the package `ZariskiFrames` [BKLH22] building on several other GAP-packages from the `homalg`-project [BLH12] and the CAP-project [GPS18, GP19, Pos21b].

Starting from Step ($\ell = 2$), all the above category constructors are *doctrine-based*: Such a constructor takes a category as an instance of a specific categorical doctrine as input and produces another category as an instance of the same doctrine or another doctrine as output. In particular, the implementation of a doctrine-based category constructor is written in terms of the categorical operations guaranteed by the input doctrine, and does not depend on the internals of the implementation of the input category. Designing and writing algorithms in terms of towers of category constructors is a defining characteristic of our approach to computer algebra.

2010 *Mathematics Subject Classification.* 13P10, 13P15, 68W30, 14Q20, 14R20.

Key words and phrases. CompilerForCAP, category constructor, towers of categories, Zariski frames/coframes, constructible sets .

This work is a contribution to the Qompiler Project – Supported by the German Federal Ministry for Economic Affairs and Climate Action.

[1] Up to this point we were able to find pointers in the literature [Ros90].

[2] We were unable to find this construction in the literature. Our construction avoids the explicit computation of radical ideals, the latter being a serious bottleneck in many computations.

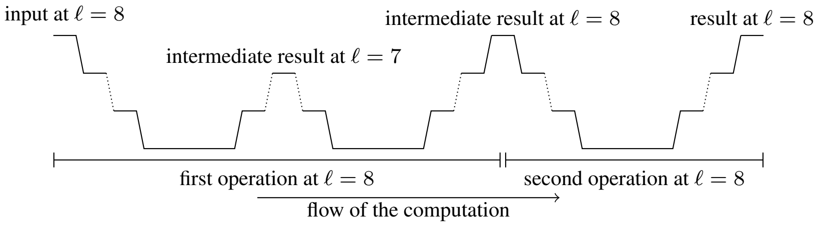
While CAP currently only offers a proprietary mechanism for defining categorical doctrines, CatLab.jl [HPBF20] follows a more systematic approach by using generalized algebraic theories (GATs). However, CAP's central design feature is its support for building arbitrarily high towers of category constructors.

Finally, avoiding side-effects in the implementation of categorical algorithms allows for parallel code execution. Many category-theoretic constructions are inherently parallelizable. For example, many operations of the above `AdditiveClosure` constructor are trivially parallelizable.

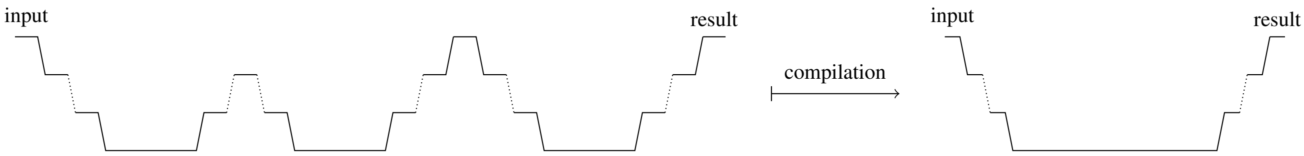
2. `CompilerForCAP`: Compiling the tower and computational performance

Structuring the software as a tower of category constructors naturally comes with a measurable performance overhead. We start by describing three sources of overhead which to a great extent could be solved by a generic compiler, for example JULIA's JIT compiler. The first source of overhead is the excessive amount of superfluous wrapping and unwrapping of data structures occurring during the computations, especially in an interpreted language like GAP. For example, a simple computation consisting of two operations at the topmost level of the tower in Section 1, where the first operation itself consists of two operations at level $\ell = 7$, can be visualized as follows:

$\ell = 8$:	`Unions(Differences(`$\ldots(R)\ldots$`))`
$\ell = 7$:	`Differences(`$\ldots(R)\ldots$`)`
$\vdots$	$\vdots$
$\ell = 0$:	R



Passing one level down or up during the computation means unwrapping or wrapping a corresponding data structure. The picture illustrates that data which is wrapped to form an intermediate result will again be immediately unwrapped by the next operation at the same level. A generic compiler can rewrite code so that the given input objects and morphisms only have to be unwrapped once at the beginning of a computation and wrapped again once the computation is finished:



The second source of overhead is that a high-level computation will often trigger repeated evaluation of low-level algorithms with the same input, that is, will compute the same thing multiple times. To avoid this, one typically uses caching, which is, however, detrimental to parallelization. A generic compiler can use common subexpression elimination and hoisting to solve this problem without caching.

The third source of overhead comes from computing intermediate lower-level results that end up not affecting the final high-level result. This can be addressed using lazy data structures, whereas a generic compiler can more elegantly just eliminate dead code.

However, to simplify the categorical code and improve the performance even further we also need support for

(1) changing the data structures of objects and morphisms at the topmost level to completely avoid the repeated (un)wrapping, even at the beginning and the end, so that the compiled code contains no references to the tower anymore;

(2) full control over the compilation process, for example to provide doctrine-specific logical rewriting rules at any chosen level of the categorical tower;
(3) the flexibility to compile against any chosen level ℓ of the categorical tower generating readable categorical code, for example to detect missing advantageous logical rewriting rules;
(4) transpiling the compiled code generated in (3) for level $\ell = 1$ into native code (at level $\ell = 0$) in various programming languages or computer algebra systems (some of which support parallel computing), which can be further compiled to (parallelized) machine code, e.g., by JULIA's JIT compiler.

This can only be achieved by a *category-theory-aware compiler* taking advantage of the strict hierarchy of data types provided by category theory. Such a compiler is implemented in our package `CompilerForCAP` [Zic22]. Additionally, we are using `CompilerForCAP` equipped with its logical rewriting rules for code verification or, "dually", as a yet simple-minded proof assistant. On the one hand, we can regard a low-level translation of a high-level algorithm[3] as adequately verified once we succeed in producing it using `CompilerForCAP`. On the other hand, `CompilerForCAP` helps us find such translations if not yet known. In any case, the compiled low-level code typically reaches a level of complexity that would make a direct implementation, let alone a manual verification, error-prone and practically impossible.

[3] e.g., how ideal intersection in the above tower translates to the computation of syzygies, which in categorical terms are weak-kernels.

References

[BBJ+21] Mohamed Barakat, Reimer Behrends, Christopher Jefferson, Lukas Kühne, and Martin Leuner, *On the generation of rank 3 simple matroids with an application to Terao's freeness conjecture*, SIAM J. Discrete Math. **35** (2021), no. 2, 1201–1223, (arXiv:1907.01073). MR 4269491 1

[BK21] Mohamed Barakat and Lukas Kühne, *Computing the nonfree locus of the moduli space of arrangements and Terao's freeness conjecture*, (arXiv:2112.13065), 2021. 1

[BKLH22] Mohamed Barakat, Tom Kuhmichel, and Markus Lange-Hegermann, `ZariskiFrames` – *(Co)frames/Locales of Zariski closed/open subsets of affine, projective, or toric varieties*, 2018–2022, (https://homalg-project.github.io/pkg/ZariskiFrames). 1

[BLH11] Mohamed Barakat and Markus Lange-Hegermann, *An axiomatic setup for algorithmic homological algebra and an alternative approach to localization*, J. Algebra Appl. **10** (2011), no. 2, 269–293. MR 2795737 (2012f:18022) 1

[BLH12] Mohamed Barakat and Markus Lange-Hegermann, *The* `homalg` *project*, *Computeralgebra-Rundbrief*, **51**, 6–9, October, 2012, (http://www.fachgruppe-computeralgebra.de/data/CA-Rundbrief/car51.pdf), pp. 6–9. 1

[BLH22] Mohamed Barakat and Markus Lange-Hegermann, *An algorithmic approach to Chevalley's theorem on images of rational morphisms between affine varieties*, Math. Comput. **91** (2022), no. 333, 451–490, (arXiv:1911.10411). 1

[GP19] Sebastian Gutsche and Sebastian Posur, *Cap: categories, algorithms, programming*, *Computeralgebra-Rundbrief*, **64**, 14–17, March, 2019, (https://fachgruppe-computeralgebra.de/data/CA-Rundbrief/car64.pdf), pp. 14–17. 1

[GPS18] Sebastian Gutsche, Sebastian Posur, and Øystein Skartsæterhagen, *On the syntax and semantics of* `CAP`, In: O. Hasan, M. Pfeiffer, G. D. Reis (eds.): Proceedings of the Workshop Computer Algebra in the Age of Types, Hagenberg, Austria, 17-Aug-2018, published at http://ceur-ws.org/Vol-2307/, 2018. 1

[HPBF20] Micah Halter, Evan Patterson, Andrew Baas, and James Fairbanks, *Compositional scientific computing with catlab and semanticmodels*, 2020. 2

[Pos21a] Sebastian Posur, *A constructive approach to Freyd categories*, Appl. Categ. Structures **29** (2021), no. 1, 171–211. MR 4204560 1

[Pos21b] Sebastian Posur, *Methods of constructive category theory*, Representations of Algebras, Geometry and Physics, Contemp. Math., vol. 769, Amer. Math. Soc., [Providence], RI, 2021, pp. 157–208. MR 4254099 1

[Ros90] Kimmo I. Rosenthal, *Quantales and their applications*, Pitman Research Notes in Mathematics Series, vol. 234, Longman Scientific & Technical, Harlow; copublished in the United States with John Wiley & Sons, Inc., New York, 1990. MR 1088258 1

[Zic22] Fabian Zickgraf, `CompilerForCAP` – *Speed up computations in CAP categories*, 2020–2022, (https://homalg-project.github.io/pkg/CompilerForCAP). 2

Department of mathematics, University of Siegen, 57068 Siegen, Germany
Email address: mohamed.barakat@uni-siegen.de

Department of mathematics, University of Siegen, 57068 Siegen, Germany
Email address: tom.kuhmichel@student.uni-siegen.de

Department of mathematics, University of Siegen, 57068 Siegen, Germany
Email address: fabian.zickgraf@uni-siegen.de