CAP – a categorical (re)organization of computer algebra

Mohamed Barakat

Synthetic mathematics, logic-affine computation
and efficient proof systems
CIRM, Luminy
8 – 12 September, 2025

Joint work with Sebastian Posur, Kamal Saleh, Fabian Zickgraf, Tom Kuhmichel, Juan Cuadra Díaz

Motivation: A typical scenario in computer algebra

Kaplansky's fifth conjecture

A finite dimensional semisimple Hopf-algebra A over a field F is cocommutative.

Juan asked me the following in June 2024:

- Consider F-algebra $A = F \times F \times F^{2\times 2}$ of dimension 6
- Classify bi- and Hopf algebra structures over A when $\mathrm{char}(F) \mid 6$
- Check Kaplansky's fifth conjecture

Software demo

The motivation for the CAP project

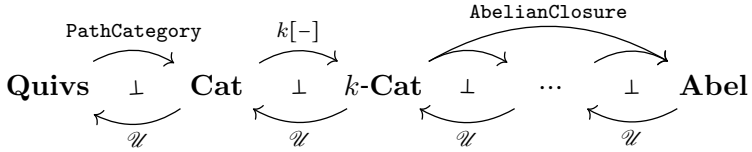
- homalg was well-desigend for the intended application
- however, not modular enough to cover more applications
- implementing more complicated categories became increasingly difficult, e.g.,
- generalizing from f.p. modules to coh. sheaves was a pain

Rectify: Take category theory more seriously

- category theory should guide all design decisions
- categories, functors, ... should become first class citizens
- turn category theory into a programming language:
- write all algorithms using categorical vocabulary

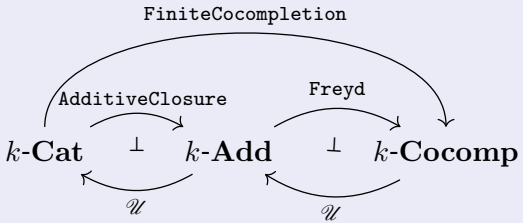
A categorical tower for `AbelianClosure`

A categorical tower of biadjunction yields `AbelianClosure` as a **categorical tower** of 2-*adjunctions*:



Zooming in

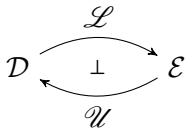
`FiniteCocompletion` as a **categorical tower** of biadjunctions



- `AdditiveClosure` formally adds direct sums
- `AdditiveClosure` invents matrices
- `Freyd` formally adds cokernels
- `Freyd` is a quotient of the arrow category

Free-forgetful 2-adjunctions

The above tower of categorical constructors is typically composed of several free-forgetful 2-adjunctions



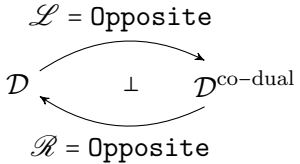
between a 2-category $\mathcal{D}$ of categories (called **doctrine**) and another doctrine $\mathcal{E}$ of categories with extra structure.

Software demo

https://homalg-project.github.io/nb/DigraphOfKnownDoctrines

FiniteCompletions

The dual category construction is also a 2-adjunction on each doctrine



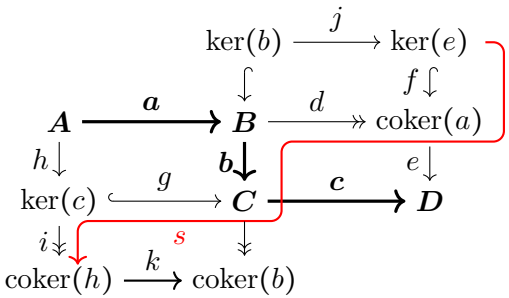
Implementing `Opposite` requires a lot of meta programming.

More categorical towers of biadjunctions

- `CoFreyd` $\coloneqq$ `Opposite` $\circ$ `Freyd` $\circ$ `Opposite`
- `FiniteCompletion` $\coloneqq$ `Opposite` $\circ$ `FiniteCocompletion` $\circ$ `Opposite`
- `FpCoPreSheaves` $\coloneqq$ `Opposite` $\circ$ `FpPreSheaves` $\circ$ `Opposite`
- `CategoryOfComonoids` $\coloneqq$ `Opposite` $\circ$ `CategoryOfMonoids` $\circ$ `Opposite`

Simplest diagram chasing: The connecting morphism

Snake Lemma: Given three composable morphisms $A \xrightarrow{a} B \xrightarrow{b} C \xrightarrow{c} D$ in an Abelian category with $abc = 0$.



$\rightsquigarrow \exists$ an *ess. unique natural* morphism $\ker(e) \xrightarrow{s} \operatorname{coker}(h)$ with $\ker(b) \xrightarrow{j} \ker(e) \xrightarrow{s} \operatorname{coker}(h) \xrightarrow{k} \operatorname{coker}(b)$ an exact sequence.

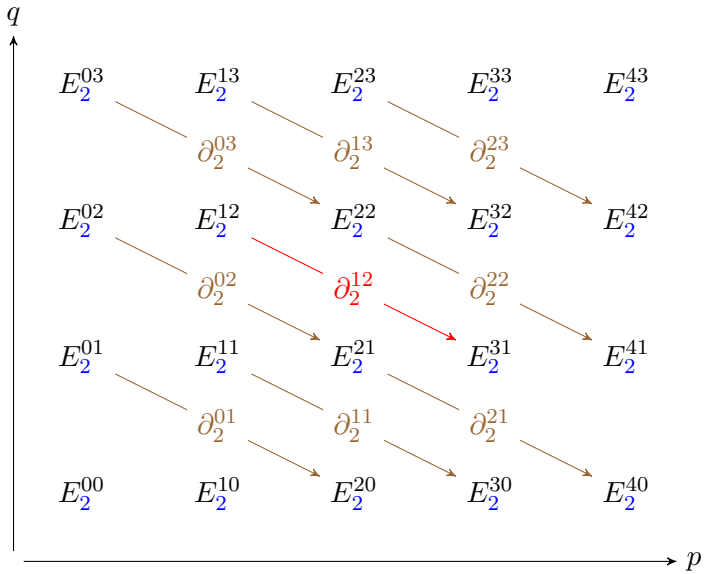
A computational proof of the snake lemma

Software demo

`https://homalg-project.github.io/nb/SnakeInFreeAbelian`

Exercise: Along the same lines treat spectral sequences of bicomplexes.

Spectral sequences of bicomplexes



Extracting the snake lemma program

Having constructed the connecting morphism s in the *syntactically* free model

$$\mathscr{L}(\mathbf{D}) = \texttt{AbelianClosure}(\underbrace{\mathbb{Q}[A \xrightarrow{a} B \xrightarrow{b} C \xrightarrow{c} D]/abc}_{\mathbf{D}})$$

we can apply the counit of the adjunction (the syntax-semantics-evaluation)

$$\varepsilon_{\mathscr{L}(\mathbf{D})} : \mathscr{L}(\mathscr{U}(\mathscr{L}(\mathbf{D}))) \to \mathscr{L}(\mathbf{D})$$

to the syntactic s an extract the program

$$\begin{aligned}
&\texttt{ConnectingMorphism} \coloneqq \texttt{function}(a, b, c)\\
&\quad k \coloneqq \texttt{KernelEmbedding}(b \cdot c); \quad \ell \coloneqq \texttt{KernelLift}(b \cdot c, a);\\
&\quad \texttt{InverseForMorphisms}(\\
&\qquad \texttt{CokernelColift}(\ell, \texttt{KernelLift}(\texttt{CokernelColift}(a, b \cdot c), k \cdot \texttt{CokernelProjection}(a)))) \cdot\\
&\quad \texttt{CokernelColift}(\ell, \texttt{KernelLift}(c, k \cdot b) \cdot \texttt{CokernelProjection}(\texttt{KernelLift}(c, a \cdot b)));
\end{aligned}$$

(up to some rewriting rules in $\texttt{AbelianClosure}(\mathbf{D})$).

Examples of categorical towers

We can model

- free left R-modules of finite rank via $\mathcal{C}(R)^{\oplus}$
- free right R-modules of finite rank via $(\mathcal{C}(R)^{\oplus})^{\mathrm{op}}$
- finitely presented left R-modules via $\mathbf{Freyd}(\mathcal{C}(R)^{\oplus})$
- finitely presented right R-modules via $\mathbf{Freyd}((\mathcal{C}(R)^{\oplus})^{\mathrm{op}})$
- quivers via $\mathbf{Func}(\mathcal{C}(\mathfrak{A} \rightrightarrows \mathfrak{V}), \mathbf{Sets})$
- ZX-diagrams via $\mathbf{Sub}(\mathbf{Csp}(\mathbf{Slice}(\mathbf{Func}(\mathcal{C}(\mathfrak{A} \rightrightarrows \mathfrak{V}), \mathbf{Sets}))))$
- free Abelian categories for theorem proving via $\mathbf{Freyd}(\mathbf{Freyd}(((-)^{\oplus})^{\mathrm{op}})^{\mathrm{op}})$
- linear representations of a group G over a field k via $\mathbf{Func}(\mathcal{C}(G), k^{\oplus})$
- radical ideals of a ring R via $\mathbf{StablePoset}(\mathbf{Poset}(\mathbf{Slice}(\mathcal{C}(R)^{\oplus})))$

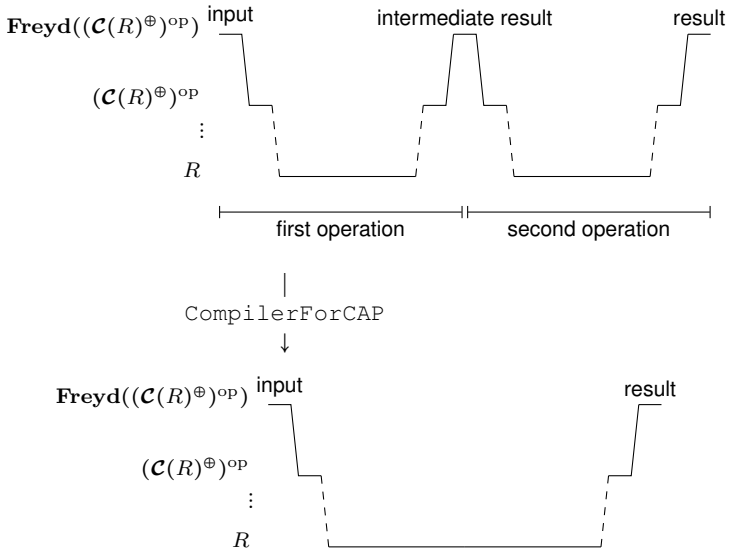
Advantages:

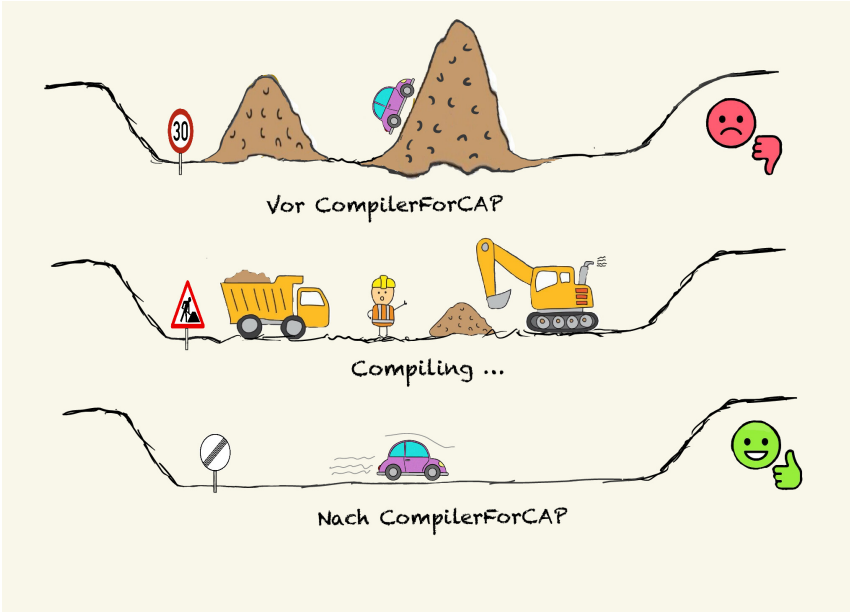
- **Reusability**: Building blocks can appear in multiple different contexts.
- **Separation of concerns**: Focus on a single concept at a time.
- **Verifiability**: Every constructor has a limited scope.
- **Emergence**: The whole is greater than the sum of its parts.

Effects on computer implementations

- **Efficient development** thanks to
  - reusability
  - separation of concerns
  - verifiability
  - emergence
- **Inefficient execution** due to computational overhead :-(
- Solution: compilation

Overhead of boxing and unboxing





Benchmarks

Consider a computation in the categorical tower

$$\mathbf{Freyd}((\mathcal{C}(R)^{\oplus})^{\mathrm{op}}) \simeq \textbf{fpmod-}R$$

problem size	**original code** (s)	**compiled code** (s)	**factor**
1	0.2	0.05	≈ 5
2	2.4	0.06	≈ 50
3	19.1	0.07	≈ 250
4	118.9	0.09	≈ 1250
5	584.5	0.12	≈ 5000
10	N/A	0.35	N/A
20	N/A	1.34	N/A
30	N/A	3.53	N/A

We see a difference between “finishes in seconds” and “will never finish”.

Further applications

`CompilerForCAP` can also be used

- for removing additional sources of overhead,
- apply categorical and mathematical rewriting rules while compiling
- as a simple-minded **proof assistant** for verifying categorical implementations

Conclusion

- **Algorithmic category theory** is a high-level programming language
- Using this language for building **categorical towers** allows
  - to reach a wide range of advanced and complex applications
  - allowing reusability, separation of concerns, verifiability, and emergence
- This approach naturally comes with a computational overhead.
- `CompilerForCAP` can avoid this overhead, allowing us to make full use of the advantages of building categorical towers

Thank you