Compiling towers of categories

to unlock the full potential of constructive category theory

Fabian Zickgraf

University of Siegen

March 09, 2022

Building towers of categories (see preceding talk)

- Benefits:
  - reach advanced and complex applications
  - while the mathematics and the software are still (relatively) simple
  - high-level algorithms which are
    - modular
    - reusable
    - easy to comprehend and check
- CAP: **C**ategories, **A**lgorithms, and **P**rogramming
- One drawback: computational overhead
- Solution: the package `CompilerForCAP`

wrapping and unwrapping is the most common source of overhead
Example:

- $C := \texttt{CategoryOfRows}(\mathbb{Z}) = \texttt{MatrixCategory}(\mathbb{Z})$:
  - objects: $0, 1, 2, 3, \ldots$
  - morphisms:
    $$m \xrightarrow{A} n \quad \text{with} \quad A \in \mathbb{Z}^{m \times n}$$
  - composition:
    $$m \xrightarrow{A} n \xrightarrow{B} k \quad = \quad m \xrightarrow{A \cdot B} k$$

- $D := \texttt{CategoryOfColumns}(\mathbb{Z})$:
  - objects: $0, 1, 2, 3, \ldots$
  - morphisms:
    $$m \xleftarrow{A} n \quad \text{with} \quad A \in \mathbb{Z}^{m \times n}$$
  - composition:
    $$m \xleftarrow{A} n \xleftarrow{B} k \quad = \quad m \xleftarrow{A \cdot B} k$$

- $D \cong C^{\text{op}} = \texttt{OppositeCategory}(C)$

- `OppositeCategory(CategoryOfRows(`$\mathbb{Z}$`))` is a (simple) tower of categories
- Data structures:
  - $0, 1, 2, 3, \ldots$ are wrapped as objects $\boxed{0}, \boxed{1}, \boxed{2}, \boxed{3}, \ldots$ in `CategoryOfRows(`$\mathbb{Z}$`)`
  - triples $(\boxed{m}, A, \boxed{n})$ are wrapped as morphisms $\boxed{(\boxed{m}, A, \boxed{n})}$ in `CategoryOfRows(`$\mathbb{Z}$`)`
  - $\boxed{0}, \boxed{1}, \boxed{2}, \boxed{3}, \ldots$ are wrapped as objects $\boxed{\boxed{0}}, \boxed{\boxed{1}}, \boxed{\boxed{2}}, \boxed{\boxed{3}}, \ldots$ in `OppositeCategory(CategoryOfRows(`$\mathbb{Z}$`))`
  - triples are wrapped as morphisms $\boxed{(\boxed{n}, \boxed{(\boxed{m}, A, \boxed{n})}, \boxed{m})}$ in `OppositeCategory(CategoryOfRows(`$\mathbb{Z}$`))`

Implementation of **AdditionForMorphisms** in CategoryOfRows:

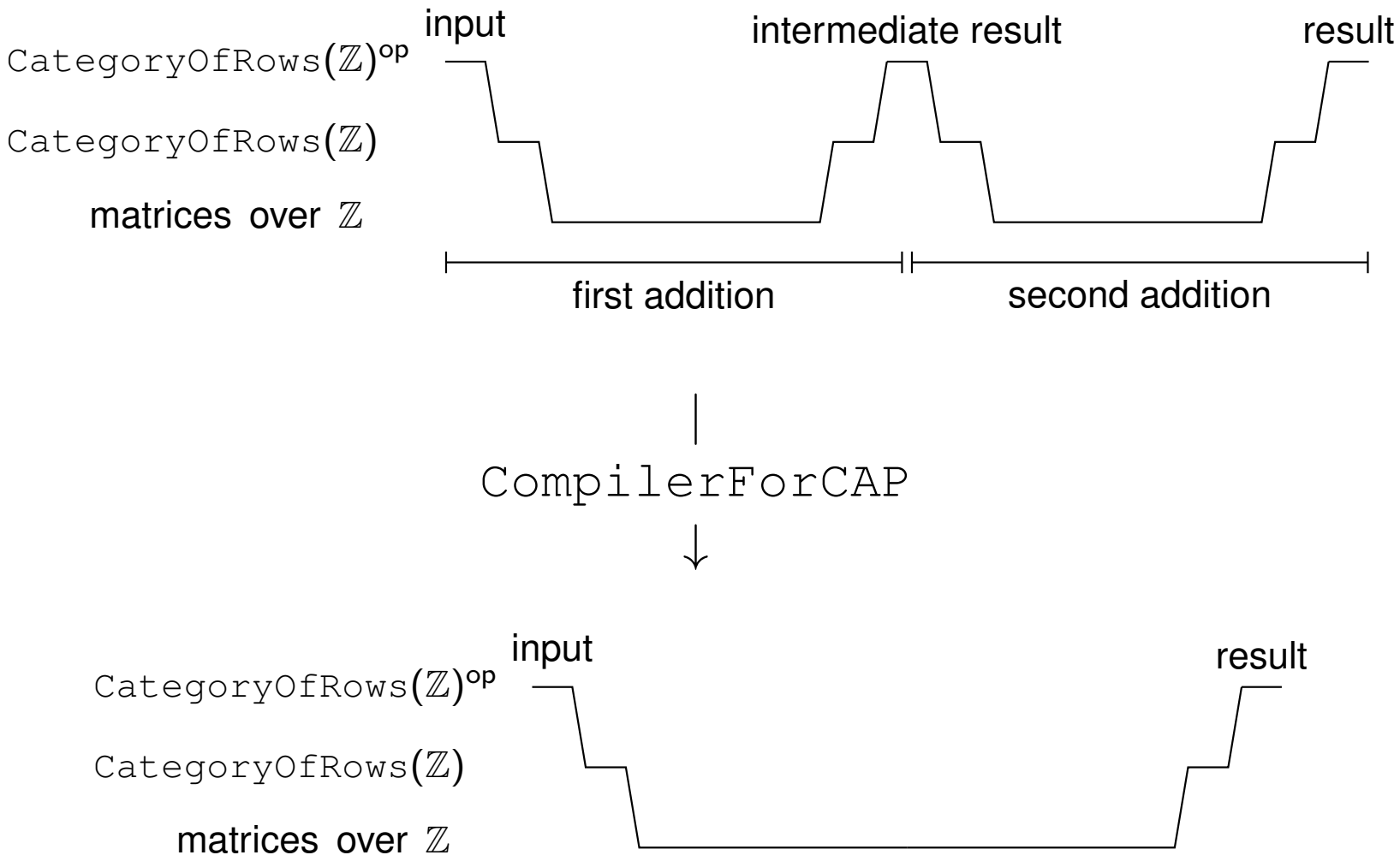
```
function ( cat, alpha, beta )
  local ...;
    matrix_alpha := UnderlyingMatrix( alpha );
    matrix_beta := UnderlyingMatrix( beta );
    matrix_result := matrix_alpha + matrix_beta;
    return CategoryOfRowsMorphism(
        Source( alpha ),
        matrix_result,
        Range( alpha )
    );
end
```

Implementation of **AdditionForMorphisms** in OppositeCategory:

```
function ( cat, alpha, beta )
  local ...;
    underlying_alpha := UnderlyingMorphism( alpha );
    underlying_beta := UnderlyingMorphism( beta );
    underlying_result := AdditionForMorphisms(
        UnderlyingCategory( cat ),
        underlying_alpha,
        underlying_beta
    );
    return MorphismInOppositeCategory(
        Source( alpha ),
        underlying_result,
        Range( alpha )
    );
end
```

Now consider the following function adding three morphisms:

```
function ( alpha, beta, gamma )
  local ...;
    alpha_plus_beta :=
        AdditionForMorphisms(
            alpha,
            beta
        );
    return
        AdditionForMorphisms(
            alpha_plus_beta,
            gamma
        );
end
```


CategoryOfRows(ℤ)^op
CategoryOfRows(ℤ)
matrices over ℤ
input
intermediate result
result
first addition
second addition
CompilerForCAP
CategoryOfRows(ℤ)^op
CategoryOfRows(ℤ)
matrices over ℤ
input
result

Inlined version of the function adding three morphisms:

```
function ( alpha, beta, gamma )
  return MorphismInOppositeCategory(
    Source( alpha ),
    CategoryOfRowsMorphism(
      Source( UnderlyingMorphism( alpha ) ),
      UnderlyingMatrix(
        UnderlyingMorphism(
          MorphismInOppositeCategory(
            Source( alpha ),
            CategoryOfRowsMorphism(
              Source( UnderlyingMorphism( alpha ) ),
              UnderlyingMatrix( UnderlyingMorphism( alpha ) )
                + UnderlyingMatrix( UnderlyingMorphism( beta ) ),
              Range( UnderlyingMorphism( beta ) )
            ),
            Range( alpha )
          )
        )
      ) + UnderlyingMatrix( UnderlyingMorphism( gamma ) ),
      Range( UnderlyingMorphism( gamma ) )
    ),
    Range( alpha ) ); end
```

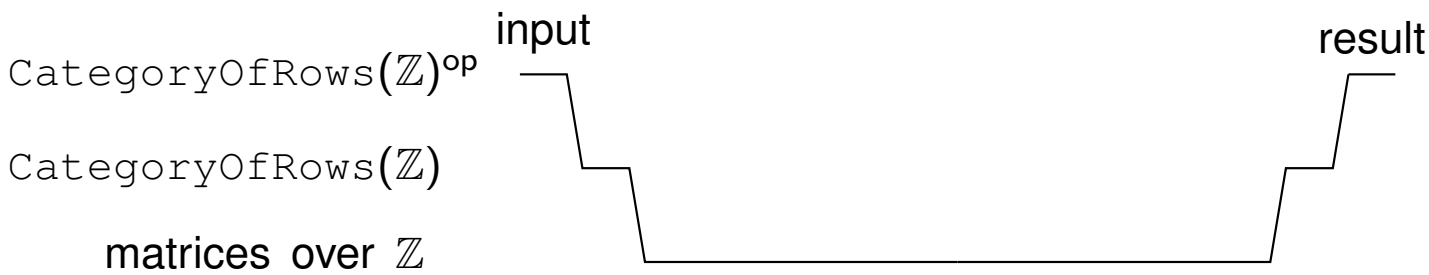
Can automatically be achieved by adding custom rewrite rules:

```
rec(
  variable_names := [ "source", "matrix", "range" ],
  src_template := """
    UnderlyingMatrix( CategoryOfRowsMorphism(
      source, matrix, range
    ) )
  """,
  dst_template := "matrix",
)

rec(
  variable_names := [ "source", "u_mor", "range" ],
  src_template := """
    UnderlyingMorphism( MorphismInOppositeCategory(
      source, u_mor, range
    ) )
  """,
  dst_template := "u_mor",
)
```

Inlined version of the function adding three morphisms:

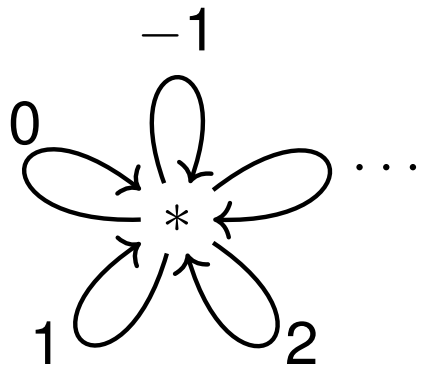
```
function ( alpha, beta, gamma )
  return MorphismInOppositeCategory(
    Source( alpha ),
    CategoryOfRowsMorphism(
      Source( UnderlyingMorphism( alpha ) ),
        UnderlyingMatrix( UnderlyingMorphism( alpha ) )
          + UnderlyingMatrix( UnderlyingMorphism( beta ) )
          + UnderlyingMatrix( UnderlyingMorphism( gamma ) ),
      Range( UnderlyingMorphism( gamma ) )
    ),
    Range( alpha ) ); end
```



Remarks

- this already happens for towers of height 1
- gets worse for higher towers
- gets much worse if a category wraps multiple objects or morphisms
- Example: the `AdditiveClosure` of a (pre-additive) category C
  - formal direct sums
  - objects: $[A_1, A_2, \dots]$ for objects A_i in C
  - morphisms: matrices (lists of lists) of morphisms in C

- Example: `AdditiveClosure`(`RingAsCategory`($\mathbb{Z}$))
- `RingAsCategory`($\mathbb{Z}$):



- example of a morphism in `AdditiveClosure`(`RingAsCategory`($\mathbb{Z}$)):

$$\boxed{(\ldots, \begin{pmatrix} \boxed{(\boxed{*}, 1, \boxed{*})} & \boxed{(\boxed{*}, 2, \boxed{*})} \\ \boxed{(\boxed{*}, 3, \boxed{*})} & \boxed{(\boxed{*}, 4, \boxed{*})} \end{pmatrix}, \ldots)} \Rightarrow \boxed{(\ldots, \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \ldots)}$$

- data structures can be changed using a concept called `WrapperCategory` and compilation

Implementation of **PreCompose** in AdditiveClosure (generalized matrix multiplication):

```
function ( cat, mor_1, mor_2 )
    ⋮
    List( [ 1 .. NrRows( matrix_1 ) ], i ->
      List( [ 1 .. NrCols( matrix_2 ) ], j ->
        Sum(
          List( [ 1 .. NrCols( matrix_1 ) ], k ->
            PreCompose( matrix_1[i, k], matrix_2[k, j] )
          )
        )
      )
    )
    ⋮
end
```

Compiled implementation of **PreCompose** in AdditiveClosure(RingAsCategory($\mathbb{Z}$)) with improved data structure:

```
function ( cat, mor_1, mor_2 )
  ⋮
  List( [ 1 .. NrRows( matrix_1 ) ], i ->
    List( [ 1 .. NrCols( matrix_2 ) ], j ->
      Sum(
        List( [ 1 .. NrCols( matrix_1 ) ], k ->
          matrix_1[i, k] * matrix_2[k, j]
        )
      )
    )
  )
  ⋮
end
```

With a custom rewrite rule for matrix multiplication we get:

```
function ( cat, mor_1, mor_2 )
  local ...;
    matrix_1 := UnderlyingMatrix( mor_1 );
    matrix_2 := UnderlyingMatrix( mor_2 );
    matrix_result := matrix_1 * matrix_2;
    return MorphismInAdditiveClosure(
        Source( mor_1 ),
        matrix_result,
        Range( mor_2 )
    );
end
```

We see

$\texttt{CategoryOfRows}(\mathbb{Z}) \cong \texttt{AdditiveClosure}(\texttt{RingAsCategory}(\mathbb{Z}))$

- “Wow, so you have found a very complicated way of obtaining matrix multiplication.”
- Yes, but we have “separation of concerns”:
  - higher-level matrix calculus
  - lower-level ring multiplication and addition
- We can apply the high-level algorithms to different pre-additive categories where there are no classical implementations.
- In this example, we knew the classical result.
- But what if we don’t?

Implementation of **HomomorphismStructureOnMorphisms** in `AdditiveClosure`:

```
MorphismBetweenDirectSums(
  List( [ 1 .. size_j ], j ->
    List( [ 1 .. size_i ], i ->
      MorphismBetweenDirectSums(
        List( [ 1 .. size_s ], s ->
          List( [ 1 .. size_t ], t ->
            HomomorphismStructureOnMorphisms(
              matrix_1[i, j], matrix_2[s, t]
) ) ) ) ) ) )
```

For `AdditiveClosure(RingAsCategory(`R`))`:

- If R is commutative: `TransposedMatrix`, `KroneckerMatrix`
- If R is an exterior algebra over a field: `TransposedMatrix`, `KroneckerMatrix`, `DualKroneckerMatrix`, `CoefficientsWithGivenMonomials`

```
function ( cat, source, alpha, beta, range )
  return CategoryOfRowsMorphism(
    source,
    CoercedMatrix( ring, field,
      CoefficientsWithGivenMonomials(
        KroneckerMat(
          TransposedMatrix( UnderlyingMatrix( alpha ) ),
          DualKroneckerMat(
            generating_system_as_column,
            UnderlyingMatrix( beta )
          )
        ),
        DiagMat(
          ListWithIdenticalEntries(
            NumberRows( UnderlyingMatrix( alpha ) ) *
              NumberColumns( UnderlyingMatrix( beta ) ),
            generating_system_as_column
          )
        )
      )
    ),
    range
  );
end
```

Further problems and solutions

- Computations inside loops which are actually independent of the loop: ~~caching~~ hoisting
- Duplicate computations: ~~caching~~ deduplication
- Computations of intermediate results which do not affect the final result: ~~laziness~~ drop unused code
- Computations in special cases which allow simplifications: ~~laziness and logic~~ custom rewrite rules

Limitations

The usual limitations of any compiler apply:

- When to inline functions?
  - code size
  - potential for optimizations
- compilation time matters
- *Confluence*: Is the result independent of the order of application of rewrite rules?
- *Termination*: Does application of the rewrite rules stop after finitely many steps?

Only works for a functional subset of GAP:

- no `while`, `for`, or `repeat ... until` loops
- no side effects

Correctness

- easy to mess up custom rewrite rules (like conventional programming)
- no formal verification (feasible in GAP?)

But:

- high code coverage ($> 90\%$)
- generic code
- applied to a large existing code base
- strict type checking
- comparison with existing, manually compiled categories

$\rightarrow$ very few correctness bugs

Future: Meta-CAP might allow formal verification

Conclusion

- building towers of categories has many advantages
- only one drawback: computational overhead
- this can be avoided using `CompilerForCAP`, while still keeping high-level algorithms and low-level optimizations separate
- this unlocks the full potential of the categorical approach

References


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.


Sebastian Gutsche, Øystein Skartsæterhagen, and Sebastian Posur, *The* `CAP` *project – Categories, Algorithms, Programming*, (`https://homalg-project.github.io/prj/CAP_project`), 2013–2022.


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