`CompilerForCAP` – Building and compiling categorical towers in algorithmic category theory

DISSERTATION

zur Erlangung des Grades eines
Doktors der Naturwissenschaften

vorgelegt von

M.Sc. Fabian Zickgraf

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2024

Betreuer und erster Gutachter

Prof. Dr. Mohamed Barakat

Universität Siegen

Zweiter Gutachter

Prof. Dr. Benedikt Ahrens

Delft University of Technology

Tag der mündlichen Prüfung

17. Mai 2024

Acknowledgments

I would like to thank everyone who has helped and supported me during the years I have been working on this thesis.

First of all, I wish to express my sincere gratitude to Mohamed Barakat for introducing me to algorithmic category theory and giving me the unique opportunity of combining category theory and computer sciences in a single project.

Furthermore, I thank Benedikt Ahrens for his valuable comments and many fruitful insights into type theory.

I would also like to thank my current and former colleagues at the university for the open and pleasant working atmosphere. Special thanks go to Kamal Saleh for many conversations and for company at lunch, even in times of Corona when the university was often quite deserted.

Last, I would like to thank my parents, my sister, and Lina for their unconditional and limitless support.

Abstract

In this thesis we develop `CompilerForCAP`, a compiler for optimizing and verifying **categorical towers** in **algorithmic category theory**. To see the need for a compiler, we start with some setup: First, we show how algorithmic category theory can be interpreted as a high-level programming language, and introduce the software framework `CAP`: Categories, Algorithms, and Programming. Moreover, we introduce programming conventions which allow us to prove that categorical algorithms written in `CAP` are **faithful** to the mathematics done on paper. Next, we introduce the concept of categorical towers, which has been used before to make constructions in categories algorithmic. We will see that computations in categorical towers naturally come with a sizeable performance overhead. This shows the need for a compiler like `CompilerForCAP`.

Afterwards, we present various applications of categorical towers exhibiting the advantages of the approach. The two main applications are the computation of lifts in categories of finitely presented modules over certain rings and algorithms for the closed monoidal structure of the category of **ZX-diagrams**, a category appearing in quantum computing. As an application, we use the category of ZX-diagrams to model a foundational functional programming language for quantum computers.

Afterwards, we see in detail how `CompilerForCAP` can optimize the categorical towers in the presented applications and provide benchmarks showing the performance gains in concrete computations. As we will see, `CompilerForCAP` can make the difference between “finishes in seconds” and “will never finish”. The central mechanism that makes these optimizations possible are **reinterpretations** of categorical towers, which allow to simplify the data structures of categorical towers. Finally, we show that `CompilerForCAP` can also be used as a **proof assistant** for verifying categorical implementations.

In summary, `CompilerForCAP` can generate efficient and verified implementations, allowing us to make full use of the advantages of building categorical towers on a computer.

Zusammenfassung

In dieser Arbeit entwickeln wir CompilerForCAP, einen Compiler zur Optimierung und Verifikation von **kategoriellen Türmen** in der **algorithmischen Kategorientheorie**. Um die Notwendigkeit eines Compilers zu erkennen, benötigen wir einige Vorbereitungen: Zuerst zeigen wir, wie algorithmische Kategorientheorie als höhere Programmiersprache aufgefasst werden kann, und stellen das Software-Framework CAP vor: *Categories, Algorithms and Programming*. Außerdem führen wir Programmierkonventionen ein, mit deren Hilfe wir beweisen können, dass kategorielle Algorithmen in CAP der Mathematik auf dem Papier **treu** sind. Als Nächstes führen wir das Konzept der kategoriellen Türme ein, das bereits früher verwendet wurde, um Konstruktionen in Kategorien zugänglich für Algorithmen zu machen. Wir werden sehen, dass Berechnungen in kategoriellen Türmen natürlicherweise mit einem beträchtlichen Performance-Overhead verbunden sind. Dies zeigt die Notwendigkeit eines Compilers wie CompilerForCAP.

Anschließend stellen wir verschiedene Anwendungen von kategoriellen Türmen vor, die die Vorteile des Ansatzes zeigen. Die zwei Hauptanwendungen sind zum einen die Berechnung von Lifts in Kategorien endlich präsentierter Moduln über bestimmten Ringen, und zum anderen Algorithmen für die geschlossene monoidale Struktur der Kategorie der **ZX-Diagramme**, einer Kategorie aus dem Bereich des Quantencomputing. Als Anwendung verwenden wir die Kategorie der ZX-Diagramme, um eine elementare funktionale Programmiersprache für Quantencomputer zu modellieren.

Anschließend sehen wir im Detail, wie CompilerForCAP die kategoriellen Türme in den vorgestellten Anwendungen optimieren kann, und führen Benchmarks durch, die die Performancezuwächse bei konkreten Berechnungen zeigen. Wie wir sehen werden, kann CompilerForCAP den Unterschied zwischen “terminiert in Sekunden” und “wird nie terminieren” ausmachen. Der zentrale Mechanismus, der diese Optimierungen ermöglicht, sind **Reinterpretationen** von kategoriellen Türmen, die es erlauben, die Datenstrukturen von kategoriellen Türmen zu vereinfachen. Zum Schluss zeigen wir, dass CompilerForCAP auch als **Beweisassistent** zur Verifikation kategorieller Implementierungen verwendet werden kann.

Zusammenfassend sehen wir, dass CompilerForCAP effiziente und verifizierte Implementierungen erzeugen kann, was es uns ermöglicht, die Vorteile kategorieller Türme auf dem Computer voll auszuschöpfen.

Contents

Introduction

Category theory has many rich applications in numerous different areas not only in mathematics but also in science and technology. Thanks to **algorithmic** category theory, these applications are not limited to theoretical aspects but can involve practical computations. We will see various examples of such applications throughout this thesis. For doing actual computations in algorithmic category theory on a computer, we need a computer framework for category theory. Such a framework is given by `CAP`: Categories, Algorithms, and Programming [GPSZ24].

Making constructions in categories algorithmic can be a challenging task. Here, **categorical towers** can be of great help. An example of a categorical tower of height 2 appears in [Pos21a], where categories of finitely presented modules over a ring are modeled as **Freyd categories** of **categories of matrices**. Modeling categories of finitely presented modules in two steps simplifies the search for an algorithm for computing lifts in these categories. Another example of a categorical tower appears in [Cic18], where the category of **ZX-diagrams**, a category appearing in quantum computing, is modeled as a **category of cospans** of a certain **slice category** of the **category of quivers**. Again, modeling the category of **ZX-diagrams** in three steps simplifies the search for algorithms for the closed monoidal structure of this category. We will see those and other examples of categorical towers in great detail in this thesis.

As we have seen, categorical towers are built by composing **category constructors**, like constructors for categories of matrices, for Freyd categories, for the category of quivers, for slice categories, and for categories of cospans. Modeling categories as categorical towers has many advantages:

- **Reusability**: The category constructors used to build one categorical tower can be reused for building categorical towers in other contexts.
- **Separation of concerns**: Every category constructor in a categorical tower can focus on a single concept, simplifying the definitions.
- **Verifiability**: The category constructors used to build a categorical tower can be verified independently of each other. Each category constructor has a limited scope and is hence relatively simple to comprehend and verify.
- **Emergence**: Simple and natural constructions at each level of the tower can lead to convoluted structures of the tower as a whole.

We will see several instances of these advantages through this thesis.

However, we will also see that computations in categorical towers on a computer naturally come with a sizeable performance overhead. For example, the organization as a tower requires additional communication between the different layers of the tower, and categorical towers often have more generic and hence complex data structures than actually required in a concrete application. Due to the resulting performance overhead, formerly in many cases large computations in categorical towers were not feasible on a computer. To make computations in categorical towers feasible, we need a **compiler**. Such a compiler, called `CompilerForCAP` [Zic24a], is developed in this thesis. `CompilerForCAP` can merge the algorithms in the different layers of the categorical towers into a single algorithm which avoids the additional communication between the layers.

Furthermore, to introduce more efficient data structures, we have to "restructure" the tower. We model this by an isomorphism of categories in a process which we call a **reinterpretation** of the categorical tower. The concept of reinterpretations of categorical towers was developed in the context of `CompilerForCAP` as the central mechanism for generating efficient primitive implementations from categorical towers in `CAP`. Using a reinterpretation, we can convert the data structure of the tower to a desired, more efficient data structure, which allows `CompilerForCAP` to fully optimize the algorithms. We will see examples of implementations where `CompilerForCAP` makes the difference between "finishes in seconds" and "will never finish". With this, large computations in categorical towers are finally feasible, often even beating the performance of implementations which have formerly been optimized by hand. Summing up, by using `CompilerForCAP` we can avoid the performance overhead naturally appearing in computations in categorical towers completely, which allows us to make full use of the advantages of building categorical towers on a computer.

Moreover, `CompilerForCAP` can also be used for code verification. In our context, this automatically means that `CompilerForCAP` also acts as a **proof assistant**, that is, as a tool for formalizing and verifying proofs on a computer. In fact, we will see that the steps used for verifying an implementation correspond to the steps in a proof of a corresponding mathematical statement on paper. Summing up, `CompilerForCAP` can not only generate efficient implementations, but can do so starting from a verified input. Assuming the compilation steps are correct, this guarantees that the compiled implementation is correct.

Outline This thesis is structured as follows: In Chapter 1, we first lay the mathematical foundations for this thesis. Afterwards, we introduce **algorithmic category theory** interpreted as a **high-level programming language**, together with `CAP` as a software implementation of algorithmic category theory. In the last section of the chapter, we discuss the **faithfulness** of computer implementations, that is, the question whether the computations on the computer actually model the mathematics done on paper. Answering this question is crucial if we want to interpret the results of computations on a computer mathematically.

In Chapter 2, we introduce various categorical constructions which we need in later chapters. In Chapter 3, we introduce the central concepts of **categorical towers** and **reinterpretations**, and also see the overhead of a computation in a categorical tower. In Chapter 4, we present various mathematical applications of categorical towers. Moreover, in Chapter 5, we present an application of categorical towers in quantum computing: We use a categorical tower to model a foundational functional programming language for quantum computers. This can be achieved via a connection between functional programming languages, so-called **lambda calculi**, and closed monoidal categories.

In Chapter 6, we finally introduce `CompilerForCAP`. We show how the compilation process is carried out by `CompilerForCAP` and look at some benchmarks showing the performance gains for the two main towers in Chapter 4 and Chapter 5. Moreover, we describe various situations in which `CompilerForCAP` can optimize code during compilation, and also explain the corresponding compiler techniques. In Chapter 7, we show how `CompilerForCAP` can be used for code verification and as a proof assistant. Finally, we summarize the results and conclude in Chapter 8.

Funding The author was supported by "Deutsche Forschungsgemeinschaft (DFG) grant SFB-TRR 195: Symbolic Tools in Mathematics and their Application" for 9 months in 2020. Since January 2022, the author is supported by the Federal Ministry for Economic Affairs and Climate Action on the basis of a decision by the German Bundestag.

Conventions We make the following global conventions:

- All rings and algebras are unital and associative.
- The natural numbers start at 0.

Chapter 1

Algorithmic category theory

The main goal of this chapter is to introduce **algorithmic category theory** seen as a **high-level programming language**, together with CAP, a software implementation of algorithmic category theory [GPSZ24]. In Section 1.1, we lay the mathematical foundations for this: We define the notion **category** and discuss various subtleties in how one can state the definition. Moreover, in Section 1.1.4, we introduce an important notation used throughout this thesis: To easily distinguish objects and morphisms of different categories in contexts with multiple categories, we write objects and morphisms inside a box to which we attach the category as an index. This notation will also form the basis for the software implementation of categories in CAP.

In Section 1.2, we explain in which sense algorithmic category theory can be seen as a high-level programming language. Subsequently, in Section 1.3 we introduce CAP as a software implementation of algorithmic category theory. In particular, we see how one can implement categories and categorical algorithms in CAP.

In Section 1.4, we discuss the **faithfulness** of computer implementations, that is, the question whether the computations on the computer actually model the mathematics done on paper. Answering this question is crucial if we want to interpret the results of computations on a computer mathematically. To this end, we introduce the concept of **pure functions** in computer programs in Section 1.4.2: This notion is, for example, used in the context of functional programming languages and makes it possible to identify functions in a computer program which essentially behave like mathematical functions. Consequently, we require all implementations in CAP to be given by pure functions, which allows us to conclude that algorithms in CAP are faithful to the mathematics.

1.1 Mathematical foundations

In this section, we introduce the mathematical foundations for this thesis.

1.1.1 The definition of *category*

We first define the term **category** and give examples of categories. We deliberately try to keep the definition flexible with regard to the chosen foundation of mathematics to emphasize that `CAP` is not biased towards a certain foundation as long as the ideas can be implemented on a computer.

Remark 1.1.1 (Collections of objects and sets of morphisms)**.** We will define categories with a **collection of objects**. The term **collection** means that we do not require any structure on objects at all in the definition.

Moreover, we will define categories with **sets of morphisms**. In this context, a **set** is a collection endowed with an equality such that two things cannot be equal in more than one way.

In Section 1.1.2, we discuss reasons for these choices and various consequences.

Definition 1.1.2 (Categories)**.** A **category $\mathbf{C}$** consists of the following data:

- a collection[1] $\mathrm{Obj}_{\mathbf{C}}$ of **objects**,
- for each two objects A and B a set[2] $\mathrm{Hom}_{\mathbf{C}}(A, B)$ of **morphisms** with **source** A and **target** B, which are denoted by $f : A \to B$,
- for each three objects A, B and C and each two morphisms $f : A \to B$ and $g : B \to C$ a morphism
$$f \cdot g : A \to C$$
called the **composite of f and g**,
- for each object A a morphism
$$\mathrm{id}_A : A \to A$$
called the **identity morphism of A**,

fulfilling the following conditions:

- composition is compatible with the equality on the sets of morphisms, that is, for $f_1, f_2 : A \to B$ with $f_1 = f_2$ and $g_1, g_2 : B \to C$ with $g_1 = g_2$ we have
$$f_1 \cdot g_1 = f_2 \cdot g_2,$$
- composition is associative, that is, for $f : A \to B$, $g : B \to C$, and $h : C \to D$ we have
$$(f \cdot g) \cdot h = f \cdot (g \cdot h),$$
- identity morphisms are neutral elements of the composition, that is, for each object A, each morphism f with source A and each morphism g with target A we have
$$\mathrm{id}_A \cdot f = f \qquad \text{and} \qquad g \cdot \mathrm{id}_A = g.$$

We call two morphisms $f : A \to B$ and $g : B \to C$ with a common object B as target and source **composable**. Note that we use the diagrammatic notation $f \cdot g$ instead of the set-theoretic notation $g \circ f$ for the composite.

Moreover, we call two morphisms $f, g : A \to B$ having a common source and a common target **parallel**.

[1] Here, the term "collection" should be read in the sense of Remark 1.1.1.

[2] Here, the term "set" should be read in the sense of Remark 1.1.1.

For an alternative definition of the notion of **category**, which only uses a single set of morphisms, see Section A.1. For the applications in this thesis, the definition given above is more useful.

Example 1.1.3 (The category of matrices over $\mathbb{Z}$)**.** We define the **category $\mathbf{Mat}_{\mathbb{Z}}$ of matrices over $\mathbb{Z}$** as follows:

- $\mathrm{Obj}_{\mathbf{Mat}_{\mathbb{Z}}} := \mathbb{N}$,
- $\mathrm{Hom}_{\mathbf{Mat}_{\mathbb{Z}}}(m, n) := \mathbb{Z}^{m \times n}$ with entrywise equality of integers,
- composition of two morphisms is given by matrix multiplication,
- identity morphisms are given by identity matrices.

One can easily check that this indeed defines a category. We can generalize this to an arbitrary (unital) ring R and obtain a category $\mathbf{Mat}_R$ of matrices over R in exactly the same fashion.

Example 1.1.4 (The category of sets)**.** We define the **category of sets Sets** as follows:

- its objects are sets,
- the morphisms with source M and target N are functions from M to N with the usual pointwise equality of functions,
- composition of morphisms is given by composing functions,
- the identity morphism of M is given by the identity function on M.

One can easily check that this indeed defines a category.

If we restrict the construction to **finite** sets, that is, sets which admit a bijection with $\{1, \ldots, n\}$ for some natural number n, we obtain the **category of finite sets FinSets**.

Example 1.1.5 (The opposite category of a category)**.** Given a category $\mathbf{C}$ we define its **opposite category $\mathbf{C}^{\mathrm{op}}$** as follows:

- $\mathrm{Obj}_{\mathbf{C}^{\mathrm{op}}} := \mathrm{Obj}_{\mathbf{C}}$,
- $\mathrm{Hom}_{\mathbf{C}^{\mathrm{op}}}(A, B) := \mathrm{Hom}_{\mathbf{C}}(B, A)$, keeping the equality,
- composition of two morphisms is given by composing in $\mathbf{C}$ with the order swapped,
- identity morphisms are given by the identity morphisms of $\mathbf{C}$.

One can easily check that this indeed defines a category.

1.1.2 Elaboration of the definition

The definition of the notion of **category** has many subtleties which we now elaborate on.

Remark 1.1.6 (**Collections** of objects)**.** We talk about a **collection** of objects instead of a **class**, **set**, or **type** to not restrict our setup to a certain mathematical foundation like a kind of set theory or type theory. For the same reason, we avoid using the membership symbol "$\in$".

Actually, in our computational context many peculiarities of the different foundations are irrelevant. For example, it is not relevant if we talk about about sets or proper classes because a computer can only represent finitely many things at a given time anyway due to its memory being finite, and even the collection of things theoretically representable on the infinite tape of a Turing machine is by definition a countably infinite set.

We will nevertheless specialize the definition to various contexts in Section 1.1.3.

Remark 1.1.7 (Equalities on objects)**.** We were able to state the definition without explicitly referencing a notion of equality on objects. This is in line with the **principle of equivalence**: Categorical reasoning should be invariant under equivalence of categories (Definition 2.1.4), which in general does not preserve notions of equality which we might impose on objects. Equivalences of categories do preserve isomorphisms though, so a natural mathematical notion of equality on objects would be "being isomorphic". However, deciding if two objects are isomorphic can be computationally expensive or even undecidable, so in general this is not useful for algorithmic applications.

Nevertheless, there is a notion of equality on objects hidden in the definition: If we want to check soundness of constructions, for example whether two morphisms are actually composable, we need a **meta-theoretical equality** on objects, which we denote by the symbol "$\equiv$". On paper, there usually is an obvious ad-hoc notion of equality, for example symbolic equality like $A \equiv A$ or definitional equality like $\mathrm{Id}(A) \equiv A$ where Id is the identity functor. On the computer, we will derive a suitable technical equality from the data structure of objects and require this technical equality to coincide with the meta-theoretical equality in Convention 1.4.24.

Note that if two objects A and B are equal with regard to the meta-theoretical equality, they should be basically indistinguishable for the mathematics, that is, replacing all occurrences of B by A should not change anything mathematically.

Remark 1.1.8 (**Sets** of morphisms)**.** We now explain why we specifically talk about **sets** of morphisms. Recall that we defined a **set** as a collection with an equality such that two things **cannot be equal in more than one way**. Without the latter restriction, in an intensional type theory morphisms could be equal in more than one way. In this case we would have to consider the pentagon in Figure 1.1 when composing four morphisms f, g, h, and i. The requirement that the composition is compatible with the equality would make all morphisms in the pentagon equal, but the equality

$$((f \cdot g) \cdot h) \cdot i = f \cdot (g \cdot (h \cdot i))$$

obtained via the left path in the pentagon would not necessarily coincide with the equality obtained via the right path in the pentagon. A category with this kind of ambiguity has been called a **wild** category in homotopy type theory [KvR21, Definition 5]. To avoid this situation, we restrict the definition to **sets** of morphisms.

Remark 1.1.9 (Equalities on morphisms)**.** The sets of morphisms could, for example, consist of equivalence classes, that is, the equality could be given by a non-trivial equivalence relation. This is why the equality on the sets of

$$\begin{tikzcd}
& ((f \cdot g) \cdot h) \cdot i \arrow[dl, "="'] \arrow[dr, "="] & \\
(f \cdot (g \cdot h)) \cdot i \arrow[d, "="'] & & (f \cdot g) \cdot (h \cdot i) \arrow[ddl, "="] \\
f \cdot ((g \cdot h) \cdot i) \arrow[dr, "="'] & & \\
& f \cdot (g \cdot (h \cdot i)) &
\end{tikzcd}$$

Figure 1.1: A pentagon including all possibilities of putting parentheses in the composition of four morphisms

morphisms is an important and explicit datum and why the requirement that composition is compatible with the equality is in general a non-trivial property.

We will also introduce a technical equality on morphisms in Remark 1.4.21. If we want to distinguish this technical equality from the equality coming with the notion “set of morphisms”, we call the latter equality the **mathematical equality**.

1.1.3 Specializing the definition to various contexts

We now specialize Definition 1.1.2 to various contexts.

Remark 1.1.10 (Specializing the definition of **category** to set theory)**.** If we replace the word **collection** (of objects) by **class** and interpret the terms **class** (of objects) and **set** (of morphisms) in Zermelo–Fraenkel set theory with the axiom of choice (ZFC), we obtain the classical notion of a (locally small) category. In this context, we automatically have an equality on objects, which we can use as the meta-theoretical equality introduced in Remark 1.1.7.

Remark 1.1.11 (Specializing the definition of **category** to (homotopy) type theory)**.** If we replace the word **collection** (of objects) by **type** and interpret the terms **type** (of objects) and **set** (of morphisms) in homotopy type theory (HoTT), we obtain the notion of a **precategory** as in [Uni13, Definition 9.1.1]. When adding the **univalence axiom** for categories, that is, if we require the identity type of objects to be equivalent to the type of isomorphisms of objects, we obtain the notion of a category as in [Uni13, Definition 9.1.6]. In this context, we have a judgemental and a propositional equality on objects, and the judgemental equality plays the role of the meta-theoretical equality introduced in Remark 1.1.7.

If the propositional equality on the sets of morphisms does not coincide with the desired mathematical equality on morphisms, one can also use the notion of **E-categories** [Pal19, Section 3], where the mathematical equality on morphisms is explicitly given as an equivalence relation.

Remark 1.1.12 (Specializing the definition of **category** to programming languages)**.** If we replace the word **collection** (of objects) by **data type** and the word **set** (of morphisms) by **data type together with a function for deciding equality**, and interpret all the terms in some programming language, we obtain a notion of a category which can be implemented on a computer. Note that a data type in a programming language might come with multiple notions of equality (for instance, see Example 1.1.13). Thus, we must be careful when deciding which equality we use for the meta-theoretical equality on objects introduced in Remark 1.1.7 and the mathematical equality on morphisms: We have to make sure that we stay faithful to the mathematics, that is, that we actually model the mathematics done on paper. We will discuss this in much more detail in Section 1.4.

Example 1.1.13 (Multiple notions of equality on the same data type)**.** Data types in programming languages often come with multiple notions of equality. For example, integers in `GAP` [GAP22] come with two different notions of equality: The equality denoted by the symbol "`=`" matches the mathematical equality of integers. Additionally, one can compare any two values in `GAP` by the operation `IsIdenticalObj`, which compares the representations of the values in memory. On 64-bit systems, integers from -2^{60} to $2^{60}-1$ are represented directly as binary numbers in memory, so for those integers the operation `IsIdenticalObj` coincides with "`=`". Integers outside of this range, however, are stored in more complex data structures involving pointers to memory locations. When computing an integer outside of this range twice, the pointers in general do not match. This is the reason why

```
IsIdenticalObj( 2^60, 2^60 )
```

returns `false` in `GAP`.

Remark 1.1.14 (Differences between the contexts)**.** Although one can often translate concepts between the different contexts in obvious ways, one must be aware of subtle differences. For one, we have to be careful with the different notions of equalities, as seen in the preceding remarks. Additionally, the translation might not preserve the complete information. As an example, consider a morphism $f : \{0,1\} \to \{0,1,2\}$ in the category **FinSets** defined pointwise via

$$f(0) \coloneqq 0,$$
$$f(1) \coloneqq 0.$$

We compare representations of f in different contexts:

In a set-theoretic foundation of mathematics, one might define functions from M to N as left-total and univalent relations[3] between M and N. This means that f would be identified with its graph

$$\{(0,0),(1,0)\}.$$

Note that we can read off the source $\{0,1\}$ of f from the graph but not the target $\{0,1,2\}$.

[3] A binary relation $R \subseteq M \times N$ is called **left-total** (resp. **univalent**) if each $m \in M$ appears at least (resp. at most) once as a first entry of elements of R.

In a type theory with function types, f would usually be an element of the type of functions from $\{0, 1\}$ to $\{0, 1, 2\}$. In particular, both the source and the target of f would typically be encoded in the type of f.

In GAP, we could represent f as

```
function ( x )
    return 0;
end
```

From this representation, neither the source nor the target of f can be read off.

Hence, when translating between different contexts, we have to be careful to not lose any information. We will therefore introduce Notation 1.1.16 in the next section to solve this problem for categorical information in a generic way, both on paper and when used as a model for a computer implementation.

1.1.4 Boxing of objects and morphisms

In this section, we introduce a notation which helps us to keep track of sources and targets of morphisms and to distinguish objects and morphisms of different categories in contexts with multiple categories. We first give an illustrative example and afterwards introduce the notation in a formal way. The notation will also be the basis for how we model objects and morphisms in CAP, for example in Section 1.3.2.

Example 1.1.15 (Boxing)**.** Consider the category $\mathbf{Mat}_{\mathbb{Z}}$ of matrices over the integers. Its objects are given by natural numbers and its morphisms are given by matrices over the integers of suitable dimensions. To visually distinguish between a "plain" natural number n and the natural number n interpreted as an object in $\mathbf{Mat}_{\mathbb{Z}}$, we denote the latter by

$$\boxed{n}_{\mathbf{Mat}_{\mathbb{Z}}}.$$

Similarly, to visually distinguish between a "plain" matrix over the integers $M \in \mathbb{Z}^{m \times n}$ and the matrix M interpreted as a morphism from $\boxed{m}_{\mathbf{Mat}_{\mathbb{Z}}}$ to $\boxed{n}_{\mathbf{Mat}_{\mathbb{Z}}}$ in $\mathbf{Mat}_{\mathbb{Z}}$, we denote the latter by

$$\boxed{M}_{\mathbf{Mat}_{\mathbb{Z}}} : \boxed{m}_{\mathbf{Mat}_{\mathbb{Z}}} \to \boxed{n}_{\mathbf{Mat}_{\mathbb{Z}}}.$$

Now consider the category ${\mathbf{Mat}_{\mathbb{Z}}}^{\mathrm{op}}$. Its objects and morphisms are given by objects and morphisms of $\mathbf{Mat}_{\mathbb{Z}}$. To view $\boxed{M}_{\mathbf{Mat}_{\mathbb{Z}}}$ as a morphism in ${\mathbf{Mat}_{\mathbb{Z}}}^{\mathrm{op}}$, we would add additional boxes as follows:

$$\boxed{\boxed{M}_{\mathbf{Mat}_{\mathbb{Z}}}}_{{\mathbf{Mat}_{\mathbb{Z}}}^{\mathrm{op}}} : \boxed{\boxed{n}_{\mathbf{Mat}_{\mathbb{Z}}}}_{{\mathbf{Mat}_{\mathbb{Z}}}^{\mathrm{op}}} \to \boxed{\boxed{m}_{\mathbf{Mat}_{\mathbb{Z}}}}_{{\mathbf{Mat}_{\mathbb{Z}}}^{\mathrm{op}}}.$$

This allows us to visually distinguish between objects and morphisms of $\mathbf{Mat}_{\mathbb{Z}}$ and objects and morphisms of ${\mathbf{Mat}_{\mathbb{Z}}}^{\mathrm{op}}$, despite there being no mathematical difference.

We formalize the previous example:

Notation 1.1.16 (Boxing)**.** To visually distinguish between "plain" values x and the value x interpreted as an object in a category $\mathbf{C}$, we denote the latter by

$$\boxed{x}_{\mathbf{C}}.$$

Similarly, to visually distinguish between a “plain” value x and a value x interpreted as a morphism from A to B in a category $\mathbf{C}$, we denote the latter by

$$\boxed{x}_{\mathbf{C}} : A \to B.$$

If the category is clear from the context but we still want to distinguish plain values from their interpretations as objects or morphism, we will usually omit the category in the notation. Additionally, if source and target of a morphism are given in the context or are uniquely determined, we sometimes also omit those.

Given a morphism $\boxed{x}_{\mathbf{C}} : A \to B$, it now makes sense to talk about the source $s(\boxed{x}_{\mathbf{C}}) \coloneqq A$ and the target $t(\boxed{x}_{\mathbf{C}}) \coloneqq B$ of $\boxed{x}$, even if x itself does not uniquely determine A and B.[4] Here, s and t should not be seen as mathematical functions but as a meta-theoretical notation which retrieves the correct objects from the context.[5]

Note that the boxes are only meant for visual distinction. Mathematically we do not want to distinguish between x and $\boxed{x}_{\mathbf{C}}$, that is, by specifying more information in the context, we should always be able to drop the boxes.

While the concept of boxing values has its origins in computer science, the notation will also prove itself extremely useful in the mathematical context of **categorical towers** in Chapter 3. We can already give some examples showing the advantages of the notation here.

Example 1.1.17 (Advantages of the notation)**.** Using the notation, we can make the following statement: If $\boxed{M} : \boxed{m} \to \boxed{n}$ and $\boxed{N} : \boxed{n} \to \boxed{\ell}$ are two morphisms in $\mathbf{Mat}_{\mathbb{Z}}$, then by definition we have

$$\boxed{M} \cdot \boxed{N} = \boxed{M \cdot N}.$$

Note that “$\cdot$” on the left-hand side denotes the composition of morphisms while “$\cdot$” on the right-hand side denotes matrix multiplication. We could of course also write this statement without boxes but then we would have to specify more information in the context, for example by renaming the left “$\cdot$” to “$\cdot_{\mathbf{Mat}_{\mathbb{Z}}}$”.

More examples arise in contexts with multiple categories. For example, consider a category $\mathbf{C}$ and its opposite category $\mathbf{C}^{\mathrm{op}}$. For two objects

$$A, B \in \mathrm{Obj}_{\mathbf{C}} = \mathrm{Obj}_{\mathbf{C}^{\mathrm{op}}},$$

the expression

$$f : A \to B \tag{1.1}$$

is ambiguous because it can either be interpreted as

$$f \in \mathrm{Hom}_{\mathbf{C}}(A, B)$$

[4] The situation that x might not uniquely determine A and B is discussed in more detail in Section A.1.

[5] We can link this to the definition of **categories with a single set of morphisms** in Section A.1: In many cases, the meta-theoretical notation s and t behaves like the mathematical functions s and t in the definition of categories with a single set of morphisms. Hence, this notation blurs the lines between the definition of **categories with a family of sets of morphisms** and the definition of **categories with a single set of morphisms** in Section A.1.

or as

$$f \in \mathrm{Hom}_{\mathbf{C}^{\mathrm{op}}}(A, B) = \mathrm{Hom}_{\mathbf{C}}(B, A).$$

Similarly, for two morphisms $f, g \in \mathrm{Hom}_{\mathbf{C}^{\mathrm{op}}}(A, A) = \mathrm{Hom}_{\mathbf{C}}(A, A)$, the expression

$$f \cdot g \tag{1.2}$$

is ambiguous because it could refer to the composite of f and g in $\mathbf{C}$ or the composite of f and g in $\mathbf{C}^{\mathrm{op}}$.

Using the notation with boxes, a possible unambiguous version of the first ambiguous expression (1.1) would be

$$\boxed{f}_{\mathbf{C}^{\mathrm{op}}} : \boxed{A}_{\mathbf{C}^{\mathrm{op}}} \to \boxed{B}_{\mathbf{C}^{\mathrm{op}}}.$$

Let us decode this expression:

- The expressions $\boxed{A}_{\mathbf{C}^{\mathrm{op}}}$ and $\boxed{B}_{\mathbf{C}^{\mathrm{op}}}$ are objects in $\mathbf{C}^{\mathrm{op}}$.
- Objects in $\mathbf{C}^{\mathrm{op}}$ are given by objects in $\mathbf{C}$, so A and B should be objects in $\mathbf{C}$.
- The expression $\boxed{f}_{\mathbf{C}^{\mathrm{op}}}$ is a morphism in $\mathrm{Hom}_{\mathbf{C}^{\mathrm{op}}}(\boxed{A}_{\mathbf{C}^{\mathrm{op}}}, \boxed{B}_{\mathbf{C}^{\mathrm{op}}})$.
- Morphisms in $\mathbf{C}^{\mathrm{op}}$ are given by morphisms in $\mathbf{C}$, so f should be a morphism in $\mathbf{C}$.
- Finally, f must have source B and target A if $\boxed{f}_{\mathbf{C}^{\mathrm{op}}}$ has source $\boxed{A}_{\mathbf{C}^{\mathrm{op}}}$ and target $\boxed{B}_{\mathbf{C}^{\mathrm{op}}}$.

Thus, the intended meaning $f \in \mathrm{Hom}_{\mathbf{C}}(B, A)$ is clear. Similarly, we can rephrase the second ambiguous expression (1.2) as

$$\boxed{f}_{\mathbf{C}^{\mathrm{op}}} \cdot \boxed{g}_{\mathbf{C}^{\mathrm{op}}}$$

if the intended interpretation is that "$\cdot$" refers to the composition in $\mathbf{C}^{\mathrm{op}}$.

1.2 Algorithmic category theory as a high-level programming language

In this section, we explain in which sense algorithmic category theory can be seen as a high-level programming language. For this we first need the insight that a categorical operation is not defined by a concrete construction but by how its result behaves with respect to other objects and morphisms. We explain this concept by comparing the definition of disjoint unions of two finite sets and the definition of binary coproducts in a category.

Definition 1.2.1 (Disjoint union of two finite sets)**.** Given two sets N and M, we define their disjoint union $N \sqcup M$ to be the following set:

$$\big\{(1, n) \mid n \in N\big\} \cup \big\{(2, m) \mid m \in M\big\} \subseteq \{1, 2\} \times (N \cup M)$$

Definition 1.2.2 (Binary coproducts)**.** Let N and M be two objects in a category $\mathbf{C}$. Their (binary) coproduct is an object $N \sqcup M$ in $\mathbf{C}$ together with two morphisms

$$\iota_N^{\sqcup} : N \to N \sqcup M \qquad \text{and} \qquad \iota_M^{\sqcup} : M \to N \sqcup M$$

with the following universal property: For every two morphisms $\tau_N : N \to T$ and $\tau_M : M \to T$, there exists a unique morphism $u : N \sqcup M \to T$ such that the triangles in the following diagram commute:

$$\begin{tikzcd}
N \arrow[dr, "\iota_N^{\sqcup}"] \arrow[ddr, bend right, "\tau_N"'] & & M \arrow[dl, "\iota_M^{\sqcup}"'] \arrow[ddl, bend left, "\tau_M"] \\
& N \sqcup M \arrow[d, dashed, "\exists_1 u"] & \\
& T &
\end{tikzcd}$$

Remark 1.2.3 (Category theory as a high-level language for mathematics)**.** One can easily show that the disjoint union of two sets N and M together with the obvious embeddings forms a coproduct in the category of sets (which justifies that we use the same symbol for both concepts). However, the two definitions have different semantics: The definition of the disjoint union of two sets N and M specifies how the set $N \sqcup M$ is constructed using symbols and constructions from set theory. On the contrary, the definition of binary coproducts does not specify how the object $N \sqcup M$ is constructed but only how it behaves with respect to other objects and morphisms of the category.

In the context of programming languages, the definition of the disjoint union of two sets would be called a definition for a **data structure**, that is, a definition for a concrete representation of data. On the contrary, the definition of binary coproducts would be called an **abstract data type** which is purely defined by its behavior. While low-level programming languages commonly allow and sometimes require to manipulate data structures directly, high-level languages usually provide interfaces which hide many implementation details. For example, the low-level programming language[6] C represents integers as bit strings and allows the programmer to access and manipulate single bits of such bit strings. On the contrary, the high-level programming language GAP has an implementation for (big) integers whose internal data structure is hidden from the user but which mirrors the mathematical behavior which one would expect, for example having an associative addition.

This mirrors the situation of our two definitions above. In this sense we view category theory as a **high-level language for mathematics**.

Remark 1.2.4 (**Algorithmic** category theory as a high-level **programming** language)**.** It remains to explain in which sense **algorithmic** category theory can be seen as a high-level **programming** language. The definition of binary coproducts above can be read as a specification for the inputs and outputs of an algorithm: Given two objects N and M in a category $\mathbf{C}$, compute an object $N \sqcup M$ in $\mathbf{C}$ together with two morphisms

$$\iota_N^{\sqcup} : N \to N \sqcup M \qquad \text{and} \qquad \iota_M^{\sqcup} : M \to N \sqcup M$$

with the following universal property: For every two morphisms $\tau_N : N \to T$ and $\tau_M : M \to T$ we can compute a unique morphism $u : N \sqcup M \to T$ such that the triangles in the following diagram commute:

[6] Of course, what constitutes a low-level or high-level programming language depends on the context.

$$\begin{tikzcd}
N \arrow[dr, "\iota_N^{\sqcup}"] \arrow[ddr, bend right, "\tau_N"'] & & M \arrow[dl, "\iota_M^{\sqcup}"'] \arrow[ddl, bend left, "\tau_M"] \\
& N \sqcup M \arrow[d, dashed, "\exists_1 u"] & \\
& T &
\end{tikzcd}$$

In a similar way, we can also interpret other categorical constructions like the composite of morphisms as specifications of algorithms. In this sense, **algorithmic category theory** is a **high-level programming language** which we call ALCT:

- the primitive data types of ALCT are categories, objects, and morphisms,
- the operations of ALCT are exactly the algorithms appearing in categorical definitions, for example computing the composite of morphisms or computing coproducts.

In ALCT, we can express **categorical algorithms**. We do not fully formalize in a strict mathematical sense what a categorical algorithm is, since this would require a much more elaborate formal setup of the theory of programming languages going far beyond the scope of this thesis. A suitable formal setup could, for example, be provided by [Hag87], which introduces a programming language based on category theory. Instead, we now state an informal definition:

Definition 1.2.5 (Categorical algorithms)**.** A **categorical algorithm** is an algorithm on objects and morphisms which only interacts with objects and morphisms by applying categorical operations, for example by computing a composition or a coproduct, but not by applying operations specific to concrete data types like multiplication of matrices or addition of integers.

Example 1.2.6 (Computing binary pushouts from coproducts and coequalizers)**.** As an example of a categorical algorithm, we show how to compute binary pushouts (Definition 2.7.8) from coproducts (Definition 2.7.1) and equalizers (Definition 2.7.3).

Let $\mathbf{C}$ be a category and let $f : C \to A$ and $g : C \to B$ be two morphisms in $\mathbf{C}$ with a common source. We can compute a pushout P of f and g from coproducts and coequalizers as follows:

- compute $A \sqcup B$ together with $\iota_A^{\sqcup} : A \to A \sqcup B$ and $\iota_B^{\sqcup} : B \to A \sqcup B$,
- define two auxiliary morphisms $d \coloneqq f \cdot \iota_A^{\sqcup}$ and $e \coloneqq g \cdot \iota_B^{\sqcup}$,
- define $P \coloneqq \mathrm{coeq}(d, e)$ with morphisms $\iota_A \coloneqq \iota_A^{\sqcup} \cdot \pi^{\mathrm{coeq}}$ and $\iota_B \coloneqq \iota_B^{\sqcup} \cdot \pi^{\mathrm{coeq}}$.

We can visualize this construction as follows:

$$\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"'] \arrow[dr, shift left, "d"] \arrow[dr, shift right, "e"'] & A \arrow[d, "\iota_A^{\sqcup}"] \arrow[ddr, bend left, "\iota_A"] & \\
B \arrow[r, "\iota_B^{\sqcup}"'] \arrow[drr, bend right, "\iota_B"'] & A \sqcup B \arrow[dr, "\pi^{\mathrm{coeq}}"] & \\
& & \mathrm{coeq}(d, e)
\end{tikzcd}$$

Using the commutativity of the triangles in the diagram and the definition of the coequalizer, one can check that

$$f \cdot \iota_A = g \cdot \iota_B.$$

Given two morphisms $\tau_A : A \to T$ and $\tau_B : B \to T$ such that $f \cdot \tau_A = g \cdot \tau_B$, we can compute $u : P \to T$ with the required properties by first applying the universal property of the coproduct and then the universal property of the coequalizer. Hence, the above construction actually constructs a pushout of f and g.

The computations of P, ι_A, ι_B, and u only interact with objects and morphisms by categorical operations, for example by taking coproducts and coequalizers. Hence, they form categorical algorithms.

Note that the above construction can be generalized to pushouts of arbitrary finite families of morphisms in $\mathbf{C}$ with a common source.

1.3 Introduction to `CAP`

In this section, we introduce `CAP`, a software implementation of ALCT.

1.3.1 `CAP` as a dialect of ALCT

`CAP` [GPSZ24] is a dialect of ALCT in that it

- defines a concrete vocabulary for constructing categories, objects, and morphisms, which we will see in Section 1.3.2, and
- defines concrete interfaces for various categorical operations.

For example, `CAP` defines two interfaces for the categorical operation "composition of morphisms": Given two morphisms $f : A \to B$ and $g : B \to C$, their composite $f \cdot g$ can be computed by

`PreCompose( f, g )` or `PostCompose( g, f )`.

`CAP`'s interpreter[7] for the categorical operations includes many automatic consistency checks. For example, before executing `PreCompose( f, g )`, the interpreter checks if `f` and `g` are actually composable.

Moreover, `CAP` includes a standard library providing many derivations of categorical operations from other categorical operations. For example, if the user provides an implementation of `PreCompose`, `CAP` will automatically derive an implementation for `PostCompose` (and vice versa) by simply swapping the arguments. A more sophisticated example would be deriving an algorithm for computing pushouts from algorithms for coproducts and coequalizers (see Example 1.3.2).

Finally, `CAP` also includes some fundamental constructions like the terminal category and a constructor for opposite categories.

Most aspects of `CAP` can easily be extended by other packages: Packages can introduce interfaces for further categorical operations, add new derivations, and can provide implementations for more categories. For example, the package

[7] This interpreter is built on `GAP`'s method dispatch mechanism.

MonoidalCategories [BGK+24] introduces support for the categorical operations of monoidal categories (see Section 2.8) together with many derivations. The package FinSetsForCAP [BMZ24] provides an implementation of the category of finite sets. More packages building on and extending CAP can be found in [Pac24].

CAP is currently implemented in GAP [GAP22] but does not rely on any special features of GAP and hence could be implemented in many other programming languages. A project which demonstrates this by automatically translating the implementation of CAP to Julia[8] is in development and currently in a proof-of-concept state. Hence, while most of the code in this thesis is GAP code, the presented concepts are actually programming language agnostic.

1.3.2 Implementation of categories in CAP

As a guiding example for implementations of categories in CAP, we show in a very detailed way how one can implement the category $\mathbf{Mat}_{\mathbb{Z}}$ from Example 1.1.3 in CAP. The code in this section can be executed in GAP 4.12.2 with CAP 2024.01-03.

First, we have to load CAP:

```
gap> LoadPackage( "CAP", false );
true
```

Then, we start the implementation of $\mathbf{Mat}_{\mathbb{Z}}$ by creating a new category with a suitable name:

```
gap> Mat_ZZ := CreateCapCategory( "Mat_ZZ" );
Mat_ZZ
```

This category does neither have definitions for the data types of objects and morphisms nor any operations yet. Nevertheless, we can already technically consider objects in this category. For example, to view the integer 1 as an object $\boxed{1}$ in $\mathbf{Mat}_{\mathbb{Z}} =$ Mat_ZZ, we use the function AsCapCategoryObject:

```
gap> 1_boxed := AsCapCategoryObject( Mat_ZZ, 1 );
<An object in Mat_ZZ>
```

To view the object 1_boxed $= \boxed{1}$ as the integer 1 again, we can use the function AsPrimitiveValue:

```
gap> AsPrimitiveValue( 1_boxed );
1
```

The morphisms are given by matrices over the integers. Note that we need support for $m \times 0$ and $0 \times n$ matrices, which is not provided natively by GAP. However, support for such matrices is conveniently provided by the package MatricesForHomalg:

```
gap> LoadPackage( "MatricesForHomalg", false );
true
```

The package MatricesForHomalg also also provides a constructor for the ring of integers:

```
gap> ZZ := HomalgRingOfIntegers( );
Z
```

[8] Julia combines the advantages of high-level programming languages with a compiler typical for low-level programming languages [BEKS17]. In Section 6.4, we will see that Julia's compiler would nevertheless not be able to replace CompilerForCAP.

Now, as an example, we can create the 1×1 identity matrix I_1 over $\mathbb{Z} =$ `ZZ` and view it as a morphism $\boxed{I_1} : \boxed{1} \to \boxed{1}$ in `Mat_ZZ` by using the function `AsCapCategoryMorphism`:

```
gap> I_1 := HomalgIdentityMatrix( 1, ZZ );;
gap> I_1_boxed := AsCapCategoryMorphism( Mat_ZZ, 1_boxed, I_1, 1_boxed );
<A morphism in Mat_ZZ>
```

Just as for objects, we can use the function `AsPrimitiveValue` to view the morphism `I_1_boxed` $= \boxed{I_1}$ as a matrix again:

```
gap> AsPrimitiveValue( I_1_boxed );
<An unevaluated 1 x 1 identity matrix over an internal ring>
```

Note that technically we can view any value as an object (or morphism) in `Mat_ZZ`, for example also rational numbers:

```
gap> one_half_boxed := AsCapCategoryObject( Mat_ZZ, 1/2 );
<An object in Mat_ZZ>
```

So while `one_half_boxed` claims to technically be an object in `Mat_ZZ`, mathematically it is not. Thus, we have to distinguish between technical objects which are also mathematical objects and technical objects which are not. We call the former objects **well-defined** technical objects. Of course we use an analogous terminology for morphisms.

To make `CAP` aware of this distinction, we should add information about the data types of objects and morphisms to `Mat_ZZ`. There are two ways of doing this: If the data types of objects and morphisms correspond to `GAP` data types[9], we can provide those to `CAP` in the call of `CreateCapCategory`. Here, this is not the case because `GAP`'s type system has no data types for natural numbers and "matrices over `ZZ`". Thus, we use the alternative method: We provide implementations of two `CAP` operations `IsWellDefinedForObjects` and `IsWellDefinedForMorphisms` for `Mat_ZZ`, which check if technical objects and morphisms are well-defined. We model natural numbers as non-negative integers in `GAP`, so $\boxed{n}$ is a well-defined technical object of `Mat_ZZ` if and only if n is a non-negative integer:

```
gap> AddIsWellDefinedForObjects( Mat_ZZ, function ( Mat_ZZ, n_boxed )
>     local n;
>       n := AsPrimitiveValue( n_boxed );
>       return IsInt( n ) and n >= 0;
>   end );
```

The operation `IsWellDefinedForObjects` is our first example of a `CAP` operation. One can provide an implementation of a `CAP` operation for a specific category by calling the corresponding `Add` function, for example the function `AddIsWellDefinedForObjects`, with the category and an implementation as above.[10]

Now that `Mat_ZZ` has an implementation of `IsWellDefinedForObjects`, we can check if the technical objects in `Mat_ZZ` created above are well-defined:

[9] In `GAP`, data types are called **filters**.

[10] Note that the implementation always gets the category explicitly as its first argument. This has the advantage that implementations do not have to rely on information from the function's context, which is particularly useful for operations like `ZeroObject` which get no argument except the category. Note that this does not mean that all `CAP` operations have to be called with the category as the first argument: As long as the category can be determined from the remaining arguments, `CAP` provides convenience methods which automatically do so.

```
gap> IsWellDefinedForObjects( 1_boxed );
true
gap> IsWellDefinedForObjects( one_half_boxed );
false
```

As expected, `1_boxed` is well-defined while `one_half_boxed` is not.

We continue with the well-definedness of technical morphisms. Given two well-defined technical objects $\boxed{m}$ and $\boxed{n}$ in `Mat_ZZ`, a technical morphism $\boxed{M} : \boxed{m} \to \boxed{n}$ is well-defined if and only if M is an $m \times n$ (homalg) matrix over `ZZ`:

```
gap> AddIsWellDefinedForMorphisms( Mat_ZZ, function ( Mat_ZZ, M_boxed )
>     local M, m, n;
>       M := AsPrimitiveValue( M_boxed );
>       m := AsPrimitiveValue( Source( M_boxed ) );
>       n := AsPrimitiveValue( Target( M_boxed ) );
>       return IsHomalgMatrix( M ) and HomalgRing( M ) = ZZ and
>                NrRows( M ) = m and NrCols( M ) = n;
>   end );
```

Of course, this check relies on the assumption that the matrix `M` is internally consistent. For example, `M` could claim to be a matrix over `ZZ` but accessing an entry explicitly could still return an element of a different ring. Hence, as always in computer algebra, we have to decide case by case which things we explicitly check and which things we take as given.[11] Since we will not explicitly access entries of the matrices in the following algorithms, it makes sense to simply assume internal consistency of the given matrices.

We quickly check that the technical morphism constructed above is well-defined:

```
gap> IsWellDefinedForMorphisms( I_1_boxed );
true
```

Next, we should provide the set structure for the sets of morphisms, that is, provide the mathematical equality on morphisms. This mathematical equality is called **congruence** in CAP to indicate that this in general is an important and explicit datum and not simply some trivial equality given by the system.[12] In $\mathbf{Mat}_{\mathbb{Z}}$, the mathematical equality is defined as the entrywise equality of integers, which is just the usual equality for matrices over the integers in `MatricesForHomalg`:

```
gap> AddIsCongruentForMorphisms( Mat_ZZ, function ( Mat_ZZ, M_boxed, N_boxed )
>     local M, N;
>       M := AsPrimitiveValue( M_boxed );
>       N := AsPrimitiveValue( N_boxed );
>       return M = N;
>   end );
```

`IsCongruentForMorphisms` assumes that the two morphisms in the input are parallel. As explained in Remark 1.1.7, checking such soundness conditions requires a meta-theoretical equality on objects, which we should provide next.

[11] If one takes memory corruption into account, even input checks and verified stacks of software and firmware are not sufficient for guaranteeing correct outputs.

[12] There also exists a CAP operation for a meta-theoretical equality on morphisms, called `IsEqualForMorphisms`. Since this operation is rarely used, we will avoid it in this thesis as not to cause any confusion with the mathematical equality. Note that we use "=" for the mathematical equality as usual, while the documentation of CAP uses "=" for the meta-theoretical equality and "∼" for the mathematical equality.

Here, the obvious choice is to view two objects as equal if and only if they are equal as natural numbers:

```
gap> AddIsEqualForObjects( Mat_ZZ, function ( Mat_ZZ, m_boxed, n_boxed )
>      local m, n;
>        m := AsPrimitiveValue( m_boxed );
>        n := AsPrimitiveValue( n_boxed );
>        return m = n;
>    end );
```

It remains to add implementations for composing morphisms via the CAP operation PreCompose and for constructing identity morphisms via the CAP operation IdentityMorphism. Recall that composition of two morphisms $\boxed{M} : A \to B$ and $\boxed{N} : B \to C$ in $\mathbf{Mat}_{\mathbb{Z}}$ is just given by $\boxed{M \cdot N} : A \to C$, where "$\cdot$" denotes matrix multiplication:

```
gap> AddPreCompose( Mat_ZZ, function ( Mat_ZZ, M_boxed, N_boxed )
>      local A, C, M, N, M_times_N;
>        A := Source( M_boxed );
>        C := Target( N_boxed );
>        M := AsPrimitiveValue( M_boxed );
>        N := AsPrimitiveValue( N_boxed );
>        M_times_N := M * N; # matrix multiplication
>        return AsCapCategoryMorphism( Mat_ZZ, A, M_times_N, C );
>    end );
```

PreCompose assumes that the two morphisms are composable. Since we have provided IsEqualForObjects, CAP will automatically check this condition every time PreCompose is called with a concrete input.

Finally, we provide an implementation for the construction of identity morphisms, which are given by identity matrices:

```
gap> AddIdentityMorphism( Mat_ZZ, function ( Mat_ZZ, m_boxed )
>      local m, I_m;
>        m := AsPrimitiveValue( m_boxed );
>        I_m := HomalgIdentityMatrix( m, ZZ );
>        return AsCapCategoryMorphism( Mat_ZZ, m_boxed, I_m, m_boxed );
>    end );
```

The final step of the implementation should always be to finalize the category:

```
gap> Finalize( Mat_ZZ );;
```

This tells CAP that we will not add more operations to Mat_ZZ and triggers the installation of derivations.

For a very quick check of the implementations above, we test if the identity morphism on $\boxed{1}$ is idempotent with regard to composition:

```
gap> IsCongruentForMorphisms(
>      PreCompose( IdentityMorphism( 1_boxed ), IdentityMorphism( 1_boxed ) ),
>      IdentityMorphism( 1_boxed )
>    );
true
```

Finally, we look at an example of a derivation of a categorical operation which CAP has automatically derived for us: IsIdempotent. Instead of the above, we can simply write

```
gap> IsIdempotent( IdentityMorphism( 1_boxed ) );
true
```

and again obtain true as expected.

Remark 1.3.1 (Boxed and unboxed values)**.** On a technical level, the integer `1` and the object `1_boxed` are not the same thing. For example, trying to compute `IdentityMorphism( 1 )` will signal an error. This is why we have to make the boxing and unboxing explicit. Yet, if we consistently use boxes in our mathematical notation, everything translates exactly into code.

Of course, one could extend the categorical operations to also accept unboxed values. However, just as in our mathematical notation this would mean that more information has to be specified in the context, namely the category and sources and targets of morphisms. Explicitly specifying the category is possible since every `CAP` operation can be called with the category as the first argument. However, sources and targets of morphisms are usually inferred from the boxed morphisms where possible. For example, `PreCompose` accepts two morphisms but would in addition have to accept three objects if the two morphisms should be given by unboxed values. This would lead to a very verbose and thus cumbersome interface.

This argument does not apply to objects, but there is an additional drawback of using unboxed values in our context: Debugging would be much more difficult. Once we start building so-called **categorical towers** in Chapter 3, our contexts will always contain multiple categories and often some of those categories will mathematically have the same objects (or morphisms), for example natural numbers. In such a context, if we would not distinguish between interpretations of the same natural number as objects (or morphisms) in different categories on a technical level, programming errors could occur much more easily and finding such errors would be difficult.

Finally, we look at an example of a categorical algorithm in `CAP`.

Example 1.3.2 (Computing binary pushouts from coproducts and coequalizers in `CAP`)**.** A categorical algorithm as defined in Definition 1.2.5 only interacts with objects and morphisms by applying categorical operations, for example by computing a composition or a coproduct, but not by applying operations specific to concrete data types like multiplication of matrices or addition of integers. In `CAP`, this means that the algorithm only interacts with objects and morphisms by applying `CAP` operations. Since `CAP` operations cannot be applied to unboxed values, this also means that it never makes sense to unbox objects or morphisms in such an algorithm.

As an example of a categorical algorithm in `CAP`, we consider the algorithm computing binary pushouts from coproducts and coequalizers in Example 1.2.6. The algorithm for computing the pushout object P could be written in `CAP` as follows:

```
function ( cat, f, g )
  local C, A, B, iA, iB, d, e, P;

    if not IsEqualForObjects( cat, Source( f ), Source( g ) ) then
        Error( "f and g must have a common source" );
    fi;

    C := Source( f );
    A := Target( f );
    B := Target( g );

    A_u_B := Coproduct( cat, [ A, B ] );
    iA := InjectionOfCofactorOfCoproduct( cat, [ A, B ], 1 );
    iB := InjectionOfCofactorOfCoproduct( cat, [ A, B ], 2 );

    d := PreCompose( cat, f, iA );
    e := PreCompose( cat, g, iB );

    P := Coequalizer( cat, A_u_B, [ d, e ] );

    return P;
end
```

Note that no boxes appeared in Example 1.2.6, and consequently no boxing or unboxing appears in the CAP algorithm as expected. In fact, since the category is not fixed, we cannot know what data type the unboxed values of f and g would have, so we could not sensibly interact with the unboxed values anyway.

1.4 Faithfulness of computer implementations

In this section, we discuss various foundational issues arising when connecting mathematics to computer implementations. To this end, we introduce the concept of **pure functions** in computer programs, consider equalities of data structures in computer implementations, and discuss subtleties arising from the explicit boxing and unboxing in CAP. Readers who are mainly interested in applications might prefer to skip this section.

1.4.1 Introduction

Many programming languages, including GAP and therefore also CAP, provide more flexibility than we typically have in mathematics. For example, in many programming languages we can compare values by their representations in memory (for instance, see Example 1.1.13), or modify values in-place. This cannot be done in mathematics because there is no notion of memory.

This observation raises the question in which sense an implementation is **faithful** to mathematics, that is, actually models the mathematics done on paper. Specifically, we would like to prove the following meta-theorem:

Meta-Theorem 1.4.1. *Algorithms in CAP which box values, apply a categorical algorithm, and unbox the result are faithful to the mathematics.*

To prove this meta-theorem, we have to introduce some terminology and make a few programming conventions: In Section 1.4.2, we introduce the notion of **pure functions** and show how pure functions can rule out unfaithfulness in Remark 1.4.7. The definition of purity of functions requires a technical notion of equality, which we introduce for non-categorical data types in Section 1.4.3. In Section 1.4.4, we look at the technical equalities of objects and morphisms in CAP categories and make some programming conventions linking the purity of CAP operations and the technical equalities of objects and morphisms to the mathematical definitions. Assuming all implementations follow these conventions, this allows us to finally prove the meta-theorem near the end of Section 1.4.4.

Before we continue, we first look at two instructive examples showing how unfaithfulness in implementation could arise in the first place.

Example 1.4.2 (Unfaithfulness of implementations)**.** Consider the following part of a naive algorithm computing a saturation of some set S:

```
repeat
    let old_S be equal to S;
    compute some new elements from the elements of S;
    add those newly computed elements to S;
until S = old_S;
```

If one naively translates this algorithm to GAP, one obtains the following implementation:

```
repeat
    old_S := S;
    X := ComputeSomeNewElements( S );
    for x in X do
        AddSet( S, x );
    od;
until S = old_S;
```

This implementation would always terminate after the first execution because the assignment `old_S := S;` makes `old_S` point to the same memory location as `S`, so adding elements to `S` would also affect the value of `old_S`. Hence, this implementation has not captured the intention of the algorithm and is not faithful to mathematics.

Example 1.4.3 (Faithfulness of boxing and unboxing)**.** Recall that in Notation 1.1.16 the boxes were only meant for visual distinction, that is, we should always be able to drop the boxes by specifying more information in the context. In Remark 1.3.1, however, we specifically distinguish between boxed and unboxed values on the computer. This also raises some questions regarding faithfulness: If a value and its boxed version are not the same thing on a technical level, how can we ensure that boxing does not introduce any additional information? For example, how can we ensure that two values are equal if and only if their boxed versions are equal? And what actually is the proper notion of equality here in the first place? We will answer these questions in Corollary 1.4.22, but first have to introduce the notion of **pure** functions for this.

1.4.2 Purity of functions

One possible bridge between mathematics and computer implementations is given in [Gut17, Section II.1], which introduces a translation between mathematical objects and integers on a computer. Here, we want to take a higher-level approach by using the notion of a **pure** function in a computer program, a notion for example used in the context of functional programming languages. We do not formalize this notion in a strict mathematical sense because this would require a much more elaborate formal setup of the theory of programming languages going far beyond the scope of this thesis. Nevertheless, in practice there will be little doubt as to whether we want to regard a function as pure or impure.

There are many different definitions of **pure functions** (see, for example [Pł22, Chapter 2] and [Lon15, Chapter 03]), but they all share the same spirit. We use the following definition:

Definition 1.4.4 (Pure functions)**.** A function in a computer program is called **pure** if

- its output is fully determined by its inputs, and

- the function has no side effects, that is, evaluating the function has no observable effects on its environment.

Example 1.4.5 ((Im-)pure functions)**.** We consider some examples of pure and impure functions. For this, we assume a programming language with arbitrary-precision integers to avoid having to consider possible issues arising from integer overflow.

- We consider a function which takes two integers and simply returns their sum. This function is pure because the sum of two integers if fully determined by the summands and there are no side effects.
- We consider a function which
  1. takes an integer n,
  2. obtains a pseudorandom integer r between 0 and 100 from a global pseudorandom number generator (PRNG), and
  3. returns the sum of n and r.

  This function is not pure: The output is not fully determined by the input n because it also depends on the state of the global PRNG. Additionally, obtaining a pseudorandom integer from a PRNG changes the state of the PRNG. If the state of the global PRNG can be queried from the environment, changing this state is a side effect.
- We consider a function which
  1. takes an integer n,
  2. creates a PRNG seeded with the current Unix time at the time of execution,
  3. obtains a pseudorandom integer r between 0 and 100 from this PRNG, and
  4. returns the sum of n, r, and $-r$.

  This function is pure: As in the previous example, r is not fully determined by the input. Nevertheless, the final output $n+r-r = n$ is fully determined by the input because the nondeterministic effect cancels out.

  Also as in the previous example, obtaining a pseudorandom integer from a pseudorandom number generator changes the state of the pseudorandom number generator. However, in this example this effect cannot be observed from outside the function and hence does not qualify as a side effect.

Remark 1.4.6 (Subtleties in the definition of pure functions)**.** The definition of purity has some subtleties. First, it hides a notion of equality which we will discuss in great detail in Section 1.4.3. Moreover, the term "observable" leaves room for interpretation what qualifies as a side effect. For example, `GAP` has a concept called **attributes**. An example of an attribute is the `CAP` operation `IdentityMorphism`: If we compute

```
IdentityMorphism( some_object ),
```

`GAP` attaches the result of the computation to `some_object`. If we compute the same expression a second time, the stored result is returned instead of running the computation again, which in more complex examples can potentially give a significant speedup. By computing

```
HasIdentityMorphism( some_object )
```

we can check if `IdentityMorphism` has already been computed for `some_object`. Hence, the first execution of `IdentityMorphism( some_object )` has an observable effect on `some_object`. Additionally, we can also observe this effect by timing the execution of `IdentityMorphism( some_object )`. However, if we make sure to not use `HasIdentityMorphism` in our code, we can safely ignore all of this as an implementation detail which is only relevant for the performance, and hence we can still consider `IdentityMorphism` as possibly pure.

Remark 1.4.7 (Ruling out unfaithfulness)**.** Requiring functions to be pure is useful because it can rule out unfaithfulness.

For example, the unfaithfulness in Example 1.4.2 could only appear due to the side effect of `AddSet`, which modifies its first argument. We could rewrite the implementation as follows:

```
repeat
    old_S := S;
    X := ComputeSomeNewElements( S );
    for x in X do
        S := Concatenation( S, [ x ] );
    od;
until S = old_S;
```

This replaces `AddSet` by `Concatenation`, which has no side effect and simply returns the new result instead of modifying its arguments.

Additionally, we can now properly capture the problems which arise when comparing values by their representations in memory. As an example, consider the exponentiation of integers in Example 1.1.13: Strictly speaking, the inputs `2` and `60` do not fully determine the output `2^60` because the output is stored in memory using pointers which can point to different locations. This problem is part of the data structure of integers and affects all computations with integers outside of the range from -2^{60} to $2^{60}-1$. Of course we do not want to rule out all computations with integers outside of this range, so instead we specify that we want the output to be fully determined **up to the equality given by** "`=`". Since "`=`" does not take the different representations in memory into account, the impurity disappears. We will discuss such equalities in much more detail in Section 1.4.3.

We end this section with a proof showing that composing pure functions again gives a pure function, a property which will later be essential when proving the purity of functions.

Proposition 1.4.8. *A function f which is constructed by the nested application of pure functions is itself pure.*

Proof. We proceed by induction on the nesting depth.

If the nesting depth is zero, no function is applied at all in the construction of f, so f simply returns one of the inputs. In this case, the output is certainly fully determined by the input and there are no side effects.

Assume that the claim holds for any nesting depth smaller or equal to n and consider a function f of nesting depth $n+1$. The return value of f must arise from a function application of a pure function g to some input terms $t_1, \dots, t_m$. These terms can be interpreted as functions with the same inputs as f (possibly ignoring some inputs). By assumption, these functions have a nesting depth at most n and we can apply the induction assumption: The value of each of the terms t_i is fully determined by the inputs of f and evaluation of the terms does not cause side effects. The absence of side effects guarantees that the evaluation of a term does not mutate the inputs, so the tuple of values of the terms t_i is fully determined by the inputs of f independent of the order in which we evaluate the terms.[13] Since g is a pure function, the output of g is fully determined by the values of the input terms t_i and thus transitively by the inputs of f. Additionally, the evaluation of g does not cause side effects.

Summing up, the output of f is fully determined by its inputs and evaluating f does not cause any side effects, so f is pure. ■

1.4.3 Technical equalities in CAP

The statement

> "The output of a function is fully determined by its inputs."

in Definition 1.4.4 hides a notion of equality because it basically translates to

> "Applying the function twice to equal inputs gives equal outputs."

Thus, we have to define technical notions of equality for all things appearing in our algorithms. We will do this in the following definition and afterwards discuss various aspects in more detail. The most complex equality is the one for functions. Since this equality does not appear in the applications in this thesis, we omit discussing it at this point and refer to Section A.2 for the full picture.

Definition 1.4.9 (Technical equalities in CAP)**.** We use the following **technical equalities** in CAP:

- For integers, booleans, and characters we use the default equality given by the comparison operator "=" in GAP.
- When creating a new composite data type, one must specify a technical equality subject to the following conditions: The constructor must be pure, which imposes a limit on how fine the equality can be, and all interfaces interacting with elements of the data type must be pure, which imposes a limit on how coarse the equality can be. Typically, the coarsest equality fulfilling these conditions is used.

Any two things which are not equal by the above are considered as unequal. In particular, no integer is equal to a boolean, as is sometimes the case in other programming languages.

Remark 1.4.10 (Technical equality of integers, booleans, and characters)**.** For integers, booleans and characters, the equality in Definition 1.4.9 which is given by the comparison operator "=" in GAP is just the expected mathematical equality. Note that in Example 1.1.13 we saw that GAP actually also has another equality for integers, so choosing a technical equality for integers is indeed a proper choice.

[13] Of course we could also simply fix an order in which we evaluate function arguments, for example "left to right". However, such an order would not have a mathematical meaning, so we avoid it.

Now, we explain the ideas behind the technical equalities of composite data types using two examples.

Example 1.4.11 (Technical equality of finite lists)**.** For composite data types, we have imposed the following requirement in Definition 1.4.9: The constructors for elements of the data type and all interfaces interacting with elements of the data type must be pure.

For the constructors this means that if two elements of a composite data type are constructed from equal things, they must also be equal. For example, two finite lists of the same length which are constructed from entrywise equal elements must be equal. This imposes a limit on how fine the equality of finite lists can be.

Conversely, the purity of the interfaces interacting with elements of the data type imposes a limit on how coarse the equality can be. For example, an interface for finite lists should certainly include a function for obtaining the length of a list and a function for obtaining the entry of a list at a given position. Now if we would consider two finite lists as equal even if they are of different length or are not entrywise equal, then this interface would not be pure.

Summing up, the equality of finite lists is uniquely determined: Two finite lists are equal if and only if they are of the same length and are entrywise equal.

We also look at an example similar to the case of finite lists but with a different equality:

Example 1.4.12 (Technical equality of finite sets)**.** Let us consider a possible composite data type modeling finite sets and let us compare the situation to the situation for finite lists. Possibly, we would like to be able to construct a finite set by specifying its elements, similar to how we have constructed finite lists in the previous example. Moreover, an interface for finite sets should certainly include a function for obtaining the cardinality. However, the interface cannot include a function for accessing an "entry at a given position" because this notion is meaningless for sets. Instead, the interface would contain common operations on sets like taking subsets, unions, intersections, et cetera. This allows for a coarser equality than in the case of lists, that is, we are free to consider two finite sets as equal if and only if they are constructed from equal elements **up to permutation and removal of duplicates**.

Note that we could in principle use the same equality as for finite lists, but would then simply get finite lists with an overly constrained interface which does not even provide access to the typical semantics of lists. This shows why typically the coarsest possible equality is chosen.

The technical equality of finite lists in Example 1.4.11 was uniquely determined because we could recover all the entries of a list in the correct order from the list itself. This will often be the case, so we want to give this situation a special name:

Definition 1.4.13 (Fully deconstructible composite data types)**.** Let us consider a composite data type together with a constructor C and interfaces for interacting with elements of the data type. We call this data type **fully deconstructible** if the inputs $x_1, \ldots, x_n$ for the constructor can be recovered (up to technical equality) in the original order from the constructed object $C(x_1, \dots, x_n)$ via the interfaces.[14]

Example 1.4.14. We consider finite lists and sets as in Example 1.4.11 and Example 1.4.12.

Finite lists are fully deconstructible: If a list is constructed from elements $x_1, \dots, x_n$, then we can access the length n and each entry x_i for $1 \leq i \leq n$ via the interface.

Finite sets are not a fully deconstructible: When constructing a set from the elements 2, 1, and 1, then the interfaces for sets allow neither to recover that 1 appeared two times in the input nor to recover the order in which elements appeared in the input, because this would contradict the purity of the interfaces with regard to the technical equality for sets defined in Example 1.4.12.

Remark 1.4.15 (Technical equality on fully deconstructible data types)**.** The technical equality on fully deconstructible data types is uniquely determined:

As always, the purity of the constructor gives that two elements of the data type constructed from equal things are equal. If the data type is fully deconstructible, the purity of the interfaces also gives the converse: Equal elements of the data type are constructed from equal things. So in this case two elements of the data type are equal if **and only if** they are constructed from equal things.

Remark 1.4.16 (Relation between mathematical and technical equalities of non–categorical data types)**.** We look at the relation between equalities constructed in mathematical contexts and the technical equalities in Definition 1.4.9.

- For primitive mathematical objects like integers or booleans, the chosen technical equality coincides with the usual mathematical equality.
- Mathematical constructions usually respect mathematical equalities on the input. For example, when constructing tuples from pointwise mathematically equal elements, we get mathematically equal tuples. This matches the rule that constructors of composite data types must respect the technical equality. By careful design of the interfaces for non-categorical composite data types[15], we can choose the technical equality to match the mathematical equality, for instance as in Example 1.4.12.

Consequently, we make the following programming convention:

Convention 1.4.17 (Equalities of non-categorical data types)**.** The mathematical and technical equalities of non-categorical data types coincide.

We will look at the situation for categorical composite data types, that is, objects and morphisms of CAP categories, in the next section. For this, we will need the following lemma:

Lemma 1.4.18 (Purity of equalities)**.** *Let "=" be some equality in CAP. Then "=" is pure if and only if "=" has no side effects and is coarser than the technical equality.*

[14] This is similar to how **inductive types** in type theory [Uni13, Chapter 5] are **freely** generated by their constructors. However, composite data types in GAP are not formalized in a way which allows us to make this similarity fully precise.

[15] This technique does not work for categorical composite data types, that is, CAP objects and morphisms, because the interfaces for those are fixed by CAP.

Proof. Let us denote the technical equality by "$\equiv$" in this proof.

First, assume that "$=$" is pure. By definition, "$=$" has no side effects and the output is fully determined by its inputs, that is:

$$\text{If } x \equiv x' \text{ and } y \equiv y', \text{ then } (x = y) \equiv (x' = y').$$

Since the technical equality on booleans coincides with the mathematical equality, we can rephrase this statement as:

$$\text{If } x \equiv x' \text{ and } y \equiv y', \text{ then } x = y \text{ if and only if } x' = y'.$$

In the special case that x and x' both coincide with y, we obtain the following statement:

$$\text{If } y \equiv y \text{ and } y \equiv y', \text{ then } y = y \text{ if and only if } y = y'.$$

Due to reflexivity of "$\equiv$" and "$=$", this can be simplified to

$$\text{If } y \equiv y', \text{ then } y = y'.$$

This just means that "$=$" is coarser than "$\equiv$".

Conversely, assume that "$=$" has no side effects and is coarser than "$\equiv$". Moreover, assume that we have values x, x', y, and y' with $x \equiv x'$ and $y \equiv y'$. Since "$=$" is coarser than "$\equiv$", we have $x = x'$ and $y = y'$. Using the symmetry and the transitivity of "$=$", we can conclude:

$$x = y \text{ if and only if } x' = y'.$$

Hence, we have

$$\text{If } x \equiv x' \text{ and } y \equiv y', \text{ then } (x = y) \equiv (x' = y').$$

This means that the output of "$=$" is fully determined by its inputs, and since it has no side effects, "$=$" is pure. ■

1.4.4 Faithfulness of categorical algorithms in `CAP`

In this section, we want to finally prove Meta-Theorem 1.4.1. In Remark 1.4.7, we have seen that requiring functions to be pure can rule out unfaithfulness. Hence, we make the following programming convention:

Convention 1.4.19 (Purity of boxing, unboxing, and `CAP` operations)**.** Boxing, unboxing, and all implementations of `CAP` operations are pure functions.

To understand the implications of this convention, we first have to look at the technical equalities of categories, objects and morphisms.

Remark 1.4.20 (Categories as fixed parameters instead of variable inputs)**.** As we have seen in Section 1.3.2, `CAP` categories are constructed by a sequence of functions: the function `CreateCapCategory`, followed by the `Add` functions, and ultimately a call of `Finalize`. In particular, there is not a unique function which we could call the "constructor" for `CAP` categories, so the requirement that for composite data types the constructor must be pure does not apply. In particular, we have no proper notion of a technical equality of categories.

However, this is not a problem in most of our applications: Usually all categories appearing in the implementation are fixed after the creation of the category. For example, the local variable `Mat_ZZ` appearing in the implementations of `PreCompose` and `IdentityMorphism` in Section 1.3.2 will always reference the same category as the global variable `Mat_ZZ`. Hence, we do not have to view `Mat_ZZ` as a variable input for the function `AsCapCategoryMorphism`. Instead we can view `AsCapCategoryMorphism` as a family of functions and `Mat_ZZ` as a fixed parameter selecting the function

```
AsCapCategoryMorphism( Mat_ZZ, ... )
```

from the family. With this point of view, we can mostly avoid the need for a proper notion of a technical equality of categories.[16]

Remark 1.4.21 (Technical equalities of objects and morphisms)**.** The function used for boxing, that is, `AsCapCategoryObject` applied to a category and a value, is actually a constructor of a composite data type with the interface given by the function `AsPrimitiveValue` used for unboxing. As explained in Remark 1.4.20, we consider the category as a fixed parameter. Unboxing allows us to recover the value which was boxed, so we deal with a fully deconstructible data type. Hence, as explained in Remark 1.4.15 the technical equality on objects is uniquely determined by the technical equality of the unboxed values.

Similarly, the constructor `AsCapCategoryMorphism` together with the interface given by the functions `Source` and `Target` as well as `AsPrimitiveValue` used for unboxing fully determines a technical equality on morphisms: Two morphisms can only be equal if they are parallel with regard to the technical equality on objects. If they are parallel in this sense, the technical equality is given by the technical equality of the unboxed values.

With this, we can finally give an answer to the questions raised in Example 1.4.3:

Corollary 1.4.22 (Boxes on the computer do not introduce additional information)**.** *Two boxed values are equal with regard to the technical equality if and only if their unboxed versions are equal with regard to the technical equality. In particular, the result of any pure function acting on boxed objects and morphisms is fully determined by the unboxed values and the additional context introduced by boxing, like source and targets of morphism. This formalizes that we really could drop the boxes as long as we would provide the needed context like source and target of morphisms in a more verbose way.*

Convention 1.4.19 also requires that implementations of `IsEqualForObjects` are pure functions. This has some implications:

Remark 1.4.23 (Purity and the meta-theoretical equality on objects)**.** We look at the implications of Convention 1.4.19 for implementations of the `CAP` operation `IsEqualForObjects`, the meta-theoretical equality on objects. Lemma 1.4.18 states that `IsEqualForObjects` is pure if and only if the meta-theoretical equality on objects is coarser than the technical equality on objects, which in turn is given by the technical equality on unboxed objects, see Remark 1.4.21. Actually, for the proof of Meta-Theorem 1.4.1, we want the technical equality

[16] This does not work when categories are variable, for example when they form objects in the **category of categories**. In the future, the existing function `CategoryConstructor` could be enhanced to allow creating arbitrary categories in a pure way. This would define a proper technical equality on categories, although this equality would in general not be decidable due to involving the equality of the functions implementing the various `CAP` operations.

on objects to coincide with the meta-theoretical equality on objects. If this is not the case a priori, there are two possible solutions:

- We can create a new composite data type as suggested in Remark 1.4.16: By careful design of a new composite data type for unboxed objects with a suitable interface, we can choose the technical equality of unboxed objects in such a way that it gives rise to the desired technical equality on objects coinciding with the meta-theoretical equality.
- We can simply use the technical equality as the meta-theoretical equality. This does not affect the category theory because the meta-theoretical equality on objects was not a part of the definition of **category** anyway.

We emphasize the central idea of the previous remark:

Convention 1.4.24. The technical equality and the meta-theoretical equality on objects of a `CAP` category coincide.

Now, we can finally prove the meta-theorem.

Meta-Theorem (Recapitulation of Meta-Theorem 1.4.1)**.** *Algorithms in `CAP` which box values, apply a categorical algorithm, and unbox the result are faithful to the mathematics.*

Proof. We assume that all programming conventions we have made above are fulfilled:

- Convention 1.4.17: The mathematical and technical equalities of non-categorical data types coincide.
- Convention 1.4.19: Boxing, unboxing, and all implementations of `CAP` operations are pure functions.
- Convention 1.4.24: The technical equality and the meta-theoretical equality on objects of a `CAP` category coincide.

We consider an algorithm as in the statement of the meta-theorem, that is, an algorithm in `CAP` which boxes values, applies a categorical algorithm, and unboxes the result. Due to Convention 1.4.19 and Proposition 1.4.8, the algorithm is pure. In particular, the unboxed result is fully determined by the unboxed values of the input with regard to the technical equality, which matches the mathematical equality due to Convention 1.4.17. The specifications of `CAP` operations are just given by the categorical definitions. Hence, the algorithm is faithful to category theory.

The only remaining aspect external to category theory is the meta-theoretical equality on objects. By Convention 1.4.24, the meta-theoretical equality on objects coincides with the technical equality on objects, which in turn is respected due to the purity of the algorithm.

Summing up, the algorithm is faithful to the mathematics. ■

Example 1.4.25 (Purity of the implementation of $\mathbf{Mat}_{\mathbb{Z}}$)**.** We revisit our implementation of $\mathbf{Mat}_{\mathbb{Z}}$ in `CAP` in Section 1.3.2 and check if all implementations of `CAP` operations are pure. First, we check that the technical equalities of unboxed objects and morphisms coincide with the mathematical equalities of unboxed objects and morphisms as expected by Convention 1.4.17. Objects are given by integers, for which the technical equality and the mathematical equality coincide. Morphisms are given by matrices over ZZ in the package MatricesForHomalg. The constructor for matrices in MatricesForHomalg is called HomalgMatrix and accepts a ring, the number of rows and columns, and the entries of the matrix. Since the only ring we construct is ZZ, for simplicity we view the ring as a fixed parameter, just as we view categories as fixed parameters since Remark 1.4.20. The interface for a matrix M in the package MatricesForHomalg consists of the following functions:

- NrRows(M) gives the number of rows of M,
- NrCols(M) gives the number of columns of M,
- M[i,j] gets the entry of M at position (i, j).

Hence, we deal with a fully deconstructible data type. Therefore, the technical equality of matrices over ZZ is uniquely determined by the number of rows and columns and entrywise equality of integers. This technical equality of matrices matches the usual mathematical equality of matrices over the integers.

Now, we go through the implementations of all CAP operations. The implementation of IsEqualForObjects simply unboxes objects as integers and checks if those are equal. Similarly, the implementation of IsCongruentForMorphisms simply unboxes morphisms as matrices and checks if those are equal. Unboxing is pure by Convention 1.4.19. Checking the technical equality of two things is always compatible with the technical equality (by transitivity) and has no side effects. Hence, by Proposition 1.4.8 we conclude that the implementations of IsEqualForObjects and IsCongruentForMorphisms are pure.

Next, matrix multiplication certainly respects the technical equality of matrices and has no side effects. Since boxing and unboxing is pure, we can conclude that the implementation of PreCompose is pure.

Furthermore, an identity matrix over a given ring is fully determined by the integer defining its dimension, and creating an identity matrix has no side effects[17], so the implementation of IdentityMorphism is pure.

For the implementation of IsWellDefinedForObjects, things get more intricate: The specifications of IsEqualForObjects, IsCongruentForMorphisms, PreCompose and IdentityMorphism only consider well-defined technical objects and morphisms. However, when checking well-definedness, of course we have to consider any technical object or morphism. Hence, the variable n in the implementation in IsWellDefinedForObjects does not necessarily have to be an integer. However, our definition of the technical equalities strictly separates between integers and non-integers, that is, if two things are equal than either both are integers or neither of the two is an integer. Additionally, checking if something is an integer has no side effect. Thus, IsInt is pure. Checking if an integer is non-negative also certainly is pure. Hence, we conclude that IsWellDefinedForObjects is pure.

Similar to IsInt, the function IsHomalgMatrix is pure. Since the only ring we construct is ZZ, for simplicity we exclude HomalgRing from consideration in this example. Next, NrRows and NrCols are pure functions by construction of the technical equality on matrices. Finally, checking the technical equality of integers is pure. Hence, we conclude that IsWellDefinedForMorphisms is pure.

[17] This holds up to implementation details, see Remark 1.4.6.

To sum up, all implementations of `CAP` operations in Section 1.3.2 are pure.

Chapter 2

Categorical constructions

In this chapter, we introduce various categorical constructions which we will need in the upcoming chapters. The constructions are not novel, but by using CompilerForCAP [Zic24a] as a proof assistant, we can now formalize many of the proofs, which we will demonstrate in Chapter 7. We mostly follow the usual definitions but explicitly require some constructions to be compatible with the (mathematical) equality on morphisms, owing to the fact that equality on morphisms is an explicit part of our definition of **category** in Definition 1.1.2. The definitions can be read both in a non-constructive and in a constructive sense. In the context of implementations in CAP, we always read definitions in a constructive sense, that is, we require algorithms for computing the objects and morphisms specified in the definitions. Readers who are mainly interested in the applications might prefer to skip this chapter and use the back references in the upcoming chapters to fill in the details later.

The chapter is structured as follows: In Section 2.1, we introduce functors and natural transformations as well as equivalences and isomorphisms of categories, which will appear in all applications in the upcoming chapters.

We continue by introducing the constructions which we need for our applications in Section 4.1 and Section 4.2, where we show how one can compute lifts in categories of finitely presented modules. We introduce lifts and colifts of morphisms in Section 2.2, preadditive and linear categories in Section 2.3, and additive categories and additive closures in Section 2.4. In Section 2.5, we define **homomorphism structures**, a concept originally introduced in [Pos21a] to compute lifts in so-called **Freyd categories**. Consequently, we introduce Freyd categories, which can be used to model categories of finitely presented modules, in Section 2.6.

Afterwards, we introduce the constructions which we need for the quantum computing application in Chapter 5, where we use a categorical tower to model a foundational functional programming language for quantum computers. We introduce examples of limits and colimits in Section 2.7, and monoidal categories, including closed and rigid structures, in Section 2.8. In Section 2.9, we introduce **slice categories** and use them to model **decorations** of set-like structures like directed multigraphs. In Section 2.10, we introduce **categories of cospans**, which we use to model graphs with inputs and outputs in Chapter 5 and which can in general be used to model networks, circuits, processes, or systems with inputs and outputs [Cic18, FS19]. Finally, in Section 2.11, we view closed monoidal categories as so-called **typed generalized lambda calculi**, which in turn can be interpreted as foundational functional programming languages.

In the course of the chapter, we will see various **category constructors**, that is, functions constructing a category from some input. We will see category constructors for functor categories (Definition 2.1.6), rings interpreted as categories (Construction 2.3.7), additive closures (Construction 2.4.6), Freyd categories (Definition 2.6.1), slice categories (Definition 2.9.1), and categories of cospans (Definition 2.10.1). We will formally introduce category constructors in Section 3.1.

2.1 Functors and natural transformations

Definition 2.1.1 (Functors)**.** Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A **(covariant) functor** F **from** $\mathbf{C}$ **to** $\mathbf{D}$ is given by the following data:

- for every object A of $\mathbf{C}$, an object $F(A)$ of $\mathbf{D}$,
- for every morphism $f : A \to B$ of $\mathbf{C}$, a morphism $F(f) : F(A) \to F(B)$ of $\mathbf{D}$,

such that

- F is compatible with the equalities on morphisms, that is, for $f, g : A \to B$ in $\mathbf{C}$ with $f = g$ we have
$$F(f) = F(g),$$
- F is compatible with the composition, that is, for two composable morphisms $f : A \to B$ and $g : B \to C$ in $\mathbf{C}$ we have
$$F(f \cdot g) = F(f) \cdot F(g),$$
- F preserves identity morphisms, that is, for every object A of $\mathbf{C}$ we have
$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)}.$$

If F is not compatible with the composition in the above sense but in the sense that it swaps the arguments of the composition, that is,
$$F(f \cdot g) = F(g) \cdot F(f),$$
we call F a **contravariant functor**. In this case, F can be identified with a covariant functor from $\mathbf{C}^{\mathrm{op}}$ to $\mathbf{D}$.

We write $F : \mathbf{C} \to \mathbf{D}$ for a functor from $\mathbf{C}$ to $\mathbf{D}$. Two functors $F : \mathbf{C} \to \mathbf{D}$ and $G : \mathbf{D} \to \mathbf{E}$ can be composed by forming a functor $F \cdot G : \mathbf{C} \to \mathbf{E}$ which maps every object A of $\mathbf{C}$ to $G(F(A))$ and every morphism f of $\mathbf{C}$ to $G(F(f))$. The **identity functor** $\mathrm{Id}_{\mathbf{C}}$ **on** $\mathbf{C}$ maps every object A of $\mathbf{C}$ to A itself and every morphism f of $\mathbf{C}$ to f itself.

A functor $F : \mathbf{C} \to \mathbf{D}$ is called **full** (respectively **faithful**) if for every two objects A and B of $\mathbf{C}$ the map
$$\begin{aligned} \mathrm{Hom}_{\mathbf{C}}(A, B) &\to \mathrm{Hom}_{\mathbf{D}}(F(A), F(B)) \\ f &\mapsto F(f) \end{aligned}$$
is surjective (respectively injective). Moreover, a functor is called an **embedding of categories** if it is injective on objects (with regard to the meta-theoretical equality) and faithful.

Example 2.1.2 (Hom-functors)**.** Let $\mathbf{C}$ be a category and let A be an object of $\mathbf{C}$. We define the (covariant) hom-functor $\mathrm{Hom}_{\mathbf{C}}(A,-)$ as follows:

$$\begin{aligned} \mathrm{Hom}_{\mathbf{C}}(A,-) : \mathbf{C} &\to \mathbf{Sets} \\ B &\mapsto \mathrm{Hom}_{\mathbf{C}}(A,B) \\ f &\mapsto f_* \end{aligned}$$

where for $f : B \to C$ we set

$$\begin{aligned} f_* : \mathrm{Hom}_{\mathbf{C}}(A,B) &\to \mathrm{Hom}_{\mathbf{C}}(A,C) \\ g &\mapsto g \cdot f \end{aligned}$$

Moreover, for an object B of $\mathbf{C}$ we define the **contravariant** hom-functor $\mathrm{Hom}_{\mathbf{C}}(-,B)$ as follows:

$$\begin{aligned} \mathrm{Hom}_{\mathbf{C}}(-,B) : \mathbf{C} &\to \mathbf{Sets} \\ A &\mapsto \mathrm{Hom}_{\mathbf{C}}(A,B) \\ f &\mapsto f^* \end{aligned}$$

where for $f : A \to C$ we set

$$\begin{aligned} f^* : \mathrm{Hom}_{\mathbf{C}}(C,B) &\to \mathrm{Hom}_{\mathbf{C}}(A,B) \\ g &\mapsto f \cdot g \end{aligned}$$

Definition 2.1.3 (Natural transformations)**.** Let $\mathbf{C}$ and $\mathbf{D}$ be categories and let F, G be two functors from $\mathbf{C}$ to $\mathbf{D}$. A **natural transformation** η **from** F **to** G is given by the following data: for each object A of $\mathbf{C}$ we have a morphism

$$\eta_A : F(A) \to G(A)$$

of $\mathbf{D}$ such that for every morphism $f : A \to B$ of $\mathbf{C}$ the following diagram commutes:

$$\begin{tikzcd} F(A) \arrow[r, "\eta_A"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\ F(B) \arrow[r, "\eta_B"'] & G(B) \end{tikzcd}$$

If for every object A of $\mathbf{C}$ the component η_A is an isomorphism, that is, an invertible morphism, then η is called a **natural isomorphism**.

We write $\eta : F \Rightarrow G$ for a natural transformation from F to G. Two natural transformations $\eta : F \Rightarrow G$ and $\varepsilon : G \Rightarrow H$ can be composed componentwise, forming a natural transformation $\eta \cdot \varepsilon : F \to H$. The **identity natural transformation on** F is defined by having identity morphisms as components.

Definition 2.1.4 (Equivalences of categories)**.** Let $\mathbf{C}$ and $\mathbf{D}$ be categories. An **equivalence of C and D** is given by the following data:

- a functor $F : \mathbf{C} \to \mathbf{D}$,
- a functor $G : \mathbf{D} \to \mathbf{C}$,
- a natural isomorphism $G \cdot F \Rightarrow \mathrm{Id}_{\mathbf{D}}$,

- a natural isomorphism $\mathrm{Id}_{\mathbf{C}} \Rightarrow F \cdot G$.

Definition 2.1.5 (Isomorphisms of categories)**.** Let **C** and **D** be categories. An equivalence of **C** and **D** is called an **isomorphism of categories** if the natural isomorphisms $G \cdot F \Rightarrow \mathrm{Id}_{\mathbf{D}}$ and $\mathrm{Id}_{\mathbf{C}} \Rightarrow F \cdot G$ can be chosen as the identity natural transformations. In particular, for an object A of **C** and an object B of **D**, we must have

$$F(G(B)) \equiv B \qquad \text{and} \qquad G(F(A)) \equiv A,$$

where "≡" denotes the meta-theoretical equalities on objects.

Definition 2.1.6 (Functor categories)**.** Given two categories **C** and **D**, we construct the **functor category $\mathbf{D}^{\mathbf{C}}$ from C to D** as follows:

- its objects are the functors $\mathbf{C} \to \mathbf{D}$,
- the morphisms from F to G are the natural transformations $F \Rightarrow G$ with componentwise equality,[1]
- composition of morphisms is given by componentwise composition of natural transformations,
- identity morphisms are given by identity natural transformations.

One can easily check that this indeed defines a category.

2.2 Decidable lifts and colifts

Definition 2.2.1 (Lifts and colifts)**.** Let **C** be a category and let $\alpha : A \to B$ and $\tau : T \to B$ be two morphisms. A **lift** of τ along α is a morphism $\xi : T \to A$ making the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "\alpha"] & B \\
T \arrow[u, "\xi"] \arrow[ur, "\tau"']
\end{tikzcd}$$

By dualizing the definition, we obtain the definition of **colifts**.

Definition 2.2.2 (Categories with decidable lifts or colifts)**.** Let **C** be a category. We say that **C has decidable lifts** if

- given two morphisms $\alpha : A \to B$ and $\tau : T \to B$ we can decide if there exists a lift of τ along α, and
- we can construct a lift ξ of τ along α if such a lift exists.

If an analogous property for colifts holds, we say that **C has decidable colifts**.

[1] In a set-theoretic foundation, one restricts to the case of **C** being a small category to make sure the natural transformations $F \Rightarrow G$ form a small set.

2.3 Preadditive and linear categories

Definition 2.3.1 (Preadditive categories)**.** Let $\mathbf{C}$ be a category. We call $\mathbf{C}$ **preadditive** if

- every set of morphisms $\mathrm{Hom}_{\mathbf{C}}(A, B)$ is equipped with an addition operation "$+$" which turns $\mathrm{Hom}_{\mathbf{C}}(A, B)$ into an abelian group,
- composition distributes over the addition, that is, for morphisms $\alpha : A \to B$, $\beta_1, \beta_2 : B \to C$, and $\gamma : C \to D$ we have
$$\alpha \cdot (\beta_1 + \beta_2) = \alpha \cdot \beta_1 + \alpha \cdot \beta_2 \qquad \text{and} \qquad (\beta_1 + \beta_2) \cdot \gamma = \beta_1 \cdot \gamma + \beta_2 \cdot \gamma.$$

In particular, the addition and taking additive inverses must be compatible with the equality on morphisms, that is, for morphisms $\alpha_1, \alpha_2, \beta_1, \beta_2 : A \to B$ with $\alpha_1 = \alpha_2$ and $\beta_1 = \beta_2$ we have

$$\alpha_1 + \beta_1 = \alpha_2 + \beta_2 \qquad \text{and} \qquad -\alpha_1 = -\alpha_2.$$

Example 2.3.2 (Categories of matrices are preadditive categories)**.** Let R be a commutative ring. The usual addition of matrices endows $\mathbf{Mat}_R$ with a preadditive structure.

Definition 2.3.3 (Linear categories)**.** Let $\mathbf{C}$ be a preadditive category and let k be a commutative ring. We call $\mathbf{C}$ **k-linear** if

- every set of morphisms $\mathrm{Hom}_{\mathbf{C}}(A, B)$ is equipped with a scalar multiplication "$\cdot$" which turns $\mathrm{Hom}_{\mathbf{C}}(A, B)$ together with the addition "$+$" given by the preadditive structure into a k-module,
- composition is k-bilinear, that is, for morphisms $\alpha : A \to B$ and $\beta : B \to C$ and $\lambda \in k$ we have
$$(\lambda \cdot \alpha) \cdot \beta = \lambda \cdot (\alpha \cdot \beta) = \alpha \cdot (\lambda \cdot \beta).$$

In particular, the scalar multiplication must be compatible with the equality on morphisms, that is, for morphisms $\alpha_1, \alpha_2 : A \to B$ with $\alpha_1 = \alpha_2$ and $\lambda \in k$ we have
$$\lambda \cdot \alpha_1 = \lambda \cdot \alpha_2.$$

Moreover, if all sets of morphisms are **finitely generated free** k-modules, we say that $\mathbf{C}$ has **finitely generated free external homs**.

Example 2.3.4 (Categories of matrices over commutative rings are linear categories)**.** Let k be a commutative ring. We have already seen that the usual addition of matrices endows $\mathbf{Mat}_k$ with a preadditive structure. The usual scalar multiplication of matrices with elements of k endows $\mathbf{Mat}_k$ with a k-linear structure. Moreover, the $m \times n$ matrices with entries in k form a finitely generated free k-module of rank $m \cdot n$. Hence, $\mathbf{Mat}_k$ has finitely generated free external homs.

Remark 2.3.5 (Preadditive categories as linear categories)**.** Just as $\mathbb{Z}$-modules are just abelian groups, $\mathbb{Z}$-linear categories are just preadditive categories.

Remark 2.3.6 (Preadditivity and linearity of opposite cateogories)**.** The sets of morphisms of an opposite category $\mathbf{C}^{\mathrm{op}}$ are just given by the sets of morphisms of $\mathbf{C}$. Hence, if $\mathbf{C}$ is preadditive or linear, then so is $\mathbf{C}^{\mathrm{op}}$.

Prototypical examples of preadditive and linear categories appear when viewing rings or algebras over commutative rings as categories:

Construction 2.3.7 (Monoids and rings as categories with a single object)**.** Let M be a monoid, that is, a set with a multiplication and a neutral element. We can regard M as a category $\boldsymbol{\mathcal{C}}(M)$ as follows:

- It only has a single object, which we call $\star_{\boldsymbol{\mathcal{C}}(M)}$,
- the morphisms are the elements of M with the equality of the set underlying M,
- composition is given by the multiplication of M,
- the identity morphism is given by the neutral element of M.

Moreover, if M is the underlying multiplicative monoid of a ring R, the addition of R provides an abelian group structure on $\operatorname{Hom}_{\boldsymbol{\mathcal{C}}(M)}(\star_{\boldsymbol{\mathcal{C}}(M)}, \star_{\boldsymbol{\mathcal{C}}(M)})$, which turns $\boldsymbol{\mathcal{C}}(M)$ into a preadditive category.

Proof of correctness. We will use `CompilerForCAP` to prove the correctness of the construction in Section 7.1 and Section 7.4.1. ■

Construction 2.3.8 (Algebras over commutative rings define linear categories)**.** Let k be a commutative ring and let R be a (unital associative) k-algebra. That is, R is a ring with a k-module structure such that ring multiplication is k-bilinear. This turns $\boldsymbol{\mathcal{C}}(R)$ into a k-linear category. Moreover, if R is finitely generated free as a k-module, then $\boldsymbol{\mathcal{C}}(R)$ is a k-linear category with finitely generated free external homs.

As a special case, consider $R = k$. Since k is a free k-module of rank 1, $\boldsymbol{\mathcal{C}}(k)$ is a k-linear category with finitely generated free external homs.

Proof of correctness. We will use `CompilerForCAP` to prove the correctness of the construction in Section 7.4.1. ■

2.4 Additive categories and additive closures

Definition 2.4.1 (Direct sums)**.** Let $\mathbf{C}$ be a preadditive category and let $A_1, \dots, A_a$ for $a \in \mathbb{N}$ be a (finite) family of objects of $\mathbf{C}$. The **direct sum** of $A_1, \dots, A_a$ is an object $\bigoplus_{i=1}^a A_i$ together with

- morphisms $\pi_i : \bigoplus_{i=1}^a A_i \to A_i$ for all $i \in \{1, \dots, a\}$ called **projections**,
- morphisms $\iota_i : A_i \to \bigoplus_{i=1}^a A_i$ for all $i \in \{1, \dots, a\}$ called **injections**,
- for every family of morphisms $\tau_i : T \to A_i$ with $i \in \{1, \dots, a\}$ a morphism

$$(\tau_1, \dots, \tau_a)^{\oplus} : T \to \bigoplus_{i=1}^{a} A_i$$

  such that

$$(\tau_1, \dots, \tau_a)^{\oplus} \cdot \pi_i = \tau_i \quad \text{for all } i \in \{1, \dots, a\},$$

- for every family of morphisms $\sigma_i : A_i \to T$ with $i \in \{1, \dots, a\}$ a morphism

  $$\langle \sigma_1, \dots, \sigma_a \rangle^{\oplus} : \bigoplus_{i=1}^{a} A_i \to T$$

  such that

  $$\iota_i \cdot \langle \sigma_1, \dots, \sigma_a \rangle^{\oplus} = \sigma_i \quad \text{for all } i \in \{1, \dots, a\},$$

such that

$$\sum_{i=1}^{a} \pi_i \cdot \iota_i = \mathrm{id}_{\bigoplus_{i=1}^{a} A_i}$$

and

$$\iota_i \cdot \pi_j = \begin{cases} \mathrm{id}_{A_i} & \text{if } i = j, \\ 0_{A_i, A_j} & \text{else,} \end{cases} \quad \text{for all } i, j \in \{1, \dots, a\}.$$

One can show that the morphisms $(\tau_1, \dots, \tau_a)^{\oplus}$ and $\langle \sigma_1, \dots, \sigma_a \rangle^{\oplus}$ as above are uniquely determined.

We say that a category **has direct sums** if a direct sum of every finite family of objects $A_1, \dots, A_a$ exists. In the degenerate case $a = 0$, the direct sum is called a **zero object**.

Definition 2.4.2 (Additive categories)**.** Let $\mathbf{C}$ be a preadditive category. $\mathbf{C}$ is called **additive** if it has direct sums.

Remark 2.4.3 (Direct sums in opposite categories)**.** The definition of direct sums is self-dual. Hence, the opposite category of an additive category is additive.

Remark 2.4.4 (Morphisms between direct sums)**.** Let $\mathbf{C}$ be an additive category and let $A_1, \dots, A_a$ and $B_1, \dots, B_b$ be two families of objects of $\mathbf{C}$. To every morphism

$$\alpha : \bigoplus_{i=1}^{a} A_i \to \bigoplus_{j=1}^{b} B_j$$

we can associate an $a \times b$ matrix $M(\alpha)$ of morphisms of $\mathbf{C}$ with components

$$\big(M(\alpha)\big)_{ij} \coloneqq \iota_i \cdot \alpha \cdot \pi_j : A_i \to A_j.$$

Conversely, to every $a \times b$ matrix M of morphisms of $\mathbf{C}$ with components

$$M_{ij} : A_i \to B_j$$

we can associate a morphism

$$M^{\oplus} \coloneqq \big\langle (M_{ij})_j^{\oplus} \big\rangle_i^{\oplus} : \bigoplus_{i=1}^{a} A_i \to \bigoplus_{j=1}^{b} B_j.$$

One can check that the two constructions are mutually inverse. That is, morphisms between direct sums exactly correspond to matrices of morphisms between the factors. Moreover, one can check that the composition of morphisms between direct sums corresponds to a generalized version of matrix multiplication, that is, for

$$\alpha : \bigoplus_{i=1}^{a} A_i \to \bigoplus_{k=1}^{b} B_j \qquad \text{and} \qquad \beta : \bigoplus_{k=1}^{b} B_i \to \bigoplus_{j=1}^{c} C_j$$

we have

$$M(\alpha \cdot \beta) = M(\alpha) \cdot M(\beta) \quad \text{with} \quad \big(M(\alpha) \cdot M(\beta)\big)_{ij} \coloneqq \sum_{k=1}^{b} \alpha_{ik} \cdot \beta_{kj} : A_i \to C_j.$$

Remark 2.4.5 (Systems of linear equations)**.** Let $\mathbf{C}$ be a preadditive category. A **system of** n **linear equations with** m **unknowns** in $\mathbf{C}$ is given by

- morphisms $\alpha_{ij} : B_i \to C_j$ for $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$ (called **coefficients**),
- morphisms $\beta_j : A \to C_j$ for $j \in \{1, \dots, n\}$ (called **constants**),
- indeterminate morphisms $\xi_i : A \to B_i$ for $i \in \{1, \dots, m\}$ (the m **unknowns**),

where we want to determine the unknowns such that the following equations are fulfilled:

$$\begin{gathered} \xi_1 \cdot \alpha_{11} + \dots + \xi_m \cdot \alpha_{m1} = \beta_1, \\ \vdots \\ \xi_1 \cdot \alpha_{1n} + \dots + \xi_m \cdot \alpha_{mn} = \beta_n. \end{gathered}$$

If $\mathbf{C}$ is an additive category, we can use Remark 2.4.4 to rewrite this system of equations as a single equation of morphisms between direct sums:

$$\begin{pmatrix} \xi_1 & \dots & \xi_m \end{pmatrix}^{\oplus} \cdot \begin{pmatrix} \alpha_{11} & \dots & \alpha_{1n} \\ \vdots & \ddots & \vdots \\ \alpha_{m1} & \dots & \alpha_{mn} \end{pmatrix}^{\oplus} = \begin{pmatrix} \beta_1 & \dots & \beta_n \end{pmatrix}^{\oplus}.$$

In particular, this equation is just a lift equation in $\mathbf{C}$.

We now construct the **additive closure** of a preadditive category $\mathbf{C}$, which can be interpreted as adding formal direct sums to $\mathbf{C}$. Afterwards, we make this interpretation precise by showing that $\mathbf{C}$ can be embedded in its additive closure and that the additive closure indeed has a preadditive structure and direct sums.

Construction 2.4.6 (Additive closure of a preadditive category)**.** Let $\mathbf{C}$ be a preadditive category. We define the **additive closure $\mathbf{C}^{\oplus}$ of $\mathbf{C}$** as the following category:

- Its objects are tuples of objects of $\mathbf{C}$, where we write $\boxed{A}$ for an object given by an a-tuple $(A_1, \dots, A_a)$,
- a morphism from $\boxed{A}$ to $\boxed{B}$ in $\mathbf{C}^{\oplus}$ is an $a \times b$ matrix α of morphisms of $\mathbf{C}$ such that
$$\alpha_{ij} : A_i \to B_j,$$
where equality of morphisms in $\mathbf{C}^{\oplus}$ is checked entrywise,

- composition is given by generalized matrix multiplication, that is, the composite of two morphisms $\boxed{\alpha} : \boxed{A} \to \boxed{B}$ and $\boxed{\beta} : \boxed{B} \to \boxed{C}$ is given by the morphism $\boxed{\gamma} : \boxed{A} \to \boxed{C}$ with components

$$\gamma_{ij} = \sum_{k=1}^{b} \alpha_{ik} \cdot \beta_{kj} : A_i \to C_j,$$

- identity morphisms are given by generalized identity matrices, that is the identity on an object $\boxed{A}$ is given by the morphism $\boxed{\alpha} : \boxed{A} \to \boxed{A}$ with components

$$\alpha_{ij} = \begin{cases} \mathrm{id}_{A_i} & \text{if } i = j, \\ 0_{A_i,A_j} & \text{else.} \end{cases}$$

Proof of correctness. We will use `CompilerForCAP` to prove the correctness of the construction in Section 7.2. ■

Construction 2.4.7 (A preadditive structure on an additive closure)**.** Let $\mathbf{C}$ be a preadditive category. We can define the sum of two morphisms

$$\boxed{\alpha}, \boxed{\beta} : \boxed{A} \to \boxed{B}$$

in the additive closure $\mathbf{C}^{\oplus}$ entrywise, that is, as the morphism $\boxed{\gamma} : \boxed{A} \to \boxed{B}$ with components

$$\gamma_{ij} = \alpha_{ij} + \beta_{ij} : A_i \to B_j.$$

This gives a preadditive structure for $\mathbf{C}^{\oplus}$.

Proof of correctness. We will use `CompilerForCAP` to prove the correctness of the construction in Section 7.4.3. ■

Remark 2.4.8 (Embedding a preadditive category in its additive closure)**.** Let $\mathbf{C}$ be a preadditive category. We can embed $\mathbf{C}$ into its additive closure $\mathbf{C}^{\oplus}$ by mapping

- an object A of $\mathbf{C}$ to $\boxed{(A)}$ in $\mathbf{C}^{\oplus}$ and
- a morphism $\alpha : A \to B$ of $\mathbf{C}$ to $\boxed{(\alpha)} : \boxed{(A)} \to \boxed{(B)}$ in $\mathbf{C}^{\oplus}$.

Morphisms $\boxed{(A)} \to \boxed{(B)}$ in $\mathbf{C}^{\oplus}$ are matrices with a single entry and thus correspond exactly to morphisms $A \to B$ in $\mathbf{C}$, so this embedding is full. Moreover, since addition of morphisms in $\mathbf{C}^{\oplus}$ is defined entrywise, this embedding is compatible with the addition.

Construction 2.4.9 (Direct sums in an additive closure)**.** Let $\mathbf{C}$ be a preadditive category. We construct direct sums in the additive closure $\mathbf{C}^{\oplus}$, starting with binary direct sums. Let $\boxed{A}$ and $\boxed{B}$ be two objects in $\mathbf{C}^{\oplus}$. We construct a direct sum $\boxed{A} \oplus \boxed{B}$ together with the projections and injections as follows:

- We set $\boxed{A} \oplus \boxed{B} :\equiv \boxed{(A_1, \dots, A_a, B_1, \dots, B_b)}$.

- The projections $\boxed{\pi_1} : \boxed{A} \oplus \boxed{B} \to \boxed{A}$ and $\boxed{\pi_2} : \boxed{A} \oplus \boxed{B} \to \boxed{B}$ are given by "stacking identity morphisms and zero morphisms", that is,

$$(\pi_1)_{ij} \coloneqq \begin{pmatrix} \mathrm{id}_A \\ 0_{B,A} \end{pmatrix}_{ij} \coloneqq \begin{cases} \mathrm{id}_{A_i} & \text{if } 1 \leq i \leq a \text{ and } i = j, \\ 0_{A_i,A_j} & \text{if } 1 \leq i \leq a \text{ and } i \neq j, \\ 0_{B_i,A_j} & \text{if } a+1 \leq i \leq a+b, \end{cases}$$

  and

$$(\pi_2)_{ij} \coloneqq \begin{pmatrix} 0_{A,B} \\ \mathrm{id}_B \end{pmatrix}_{ij} \coloneqq \begin{cases} 0_{A_i,B_j} & \text{if } 1 \leq i \leq a, \\ \mathrm{id}_{B_i} & \text{if } a+1 \leq i \leq a+b \text{ and } i = j, \\ 0_{B_i,B_j} & \text{if } a+1 \leq i \leq a+b \text{ and } i \neq j. \end{cases}$$

- Similarly, the injections $\boxed{\iota_1} : \boxed{A} \to \boxed{A} \oplus \boxed{B}$ and $\boxed{\iota_2} : \boxed{B} \to \boxed{A} \oplus \boxed{B}$ are given by

$$(\iota_1)_{ij} \coloneqq \begin{pmatrix} \mathrm{id}_A & 0_{A,B} \end{pmatrix}_{ij} \coloneqq \begin{cases} \mathrm{id}_{A_i} & \text{if } 1 \leq j \leq a \text{ and } i = j, \\ 0_{A_i,A_j} & \text{if } 1 \leq j \leq a \text{ and } i \neq j, \\ 0_{A_i,B_j} & \text{if } a+1 \leq j \leq a+b, \end{cases}$$

  and

$$(\iota_2)_{ij} \coloneqq \begin{pmatrix} 0_{B,A} & \mathrm{id}_B \end{pmatrix}_{ij} \coloneqq \begin{cases} 0_{B_i,A_j} & \text{if } 1 \leq j \leq a, \\ \mathrm{id}_{B_i} & \text{if } a+1 \leq j \leq a+b \text{ and } i = j, \\ 0_{B_i,B_j} & \text{if } a+1 \leq j \leq a+b \text{ and } i \neq j. \end{cases}$$

Now, if we have an object $\boxed{T}$ with morphisms

$$\boxed{\tau_1} : \boxed{T} \to \boxed{A} \quad \text{and} \quad \boxed{\tau_2} : \boxed{T} \to \boxed{B},$$

we can choose $\boxed{u} = (\boxed{\tau_1}, \boxed{\tau_2})^{\oplus} : \boxed{T} \to \boxed{A} \oplus \boxed{B}$ as "the union of columns of the matrices τ_1 and τ_2", that is,

$$u_{ij} \coloneqq \begin{pmatrix} \tau_1 & \tau_2 \end{pmatrix}_{ij} \coloneqq \begin{cases} (\tau_1)_{ij} & \text{if } 1 \leq j \leq a, \\ (\tau_2)_{ij} & \text{if } a+1 \leq j \leq a+b. \end{cases}$$

Similarly, for two morphisms

$$\boxed{\tau_1} : \boxed{A} \to \boxed{T} \quad \text{and} \quad \boxed{\tau_2} : \boxed{B} \to \boxed{T}$$

we can choose $\boxed{u} = \langle \boxed{\sigma_1}, \boxed{\sigma_2} \rangle^{\oplus} : \boxed{A} \oplus \boxed{B} \to \boxed{T}$ as

$$u_{ij} \coloneqq \begin{pmatrix} \tau_1 \\ \tau_2 \end{pmatrix}_{ij} \coloneqq \begin{cases} (\sigma_1)_{ij} & \text{if } 1 \leq i \leq a, \\ (\sigma_2)_{ij} & \text{if } a+1 \leq i \leq a+b. \end{cases}$$

This construction can be generalized to arbitrary finite direct sums. This also includes the degenerate case of an empty direct sum, which constructs a zero object represented by an empty tuple.

Proof of correctness. We will use `CompilerForCAP` to prove the correctness of the construction in Section 7.4.3. ■

2.5 Homomorphism structures

In this section, we define **homomorphism structures**. The concept was introduced in [Pos21a] to compute lifts in so-called Freyd categories. We will see this application in Proposition 2.6.2. For a motivation for the definition, see Remark 4.1.2 and Remark 2.5.2. For a connection to enriched category theory, see Remark 2.5.3.

Definition 2.5.1 (Homomorphism structures, [Pos21a, Definition 6.4])**.** Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A **$\mathbf{D}$-homomorphism structure** on $\mathbf{C}$ consists of the following data:

- An object $1 \in \mathbf{D}$ called the **distinguished object**,
- a functor in two variables $H : \mathbf{C}^{\mathrm{op}} \times \mathbf{C} \to \mathbf{D}$ which is contravariant in the first component,[2]
- a natural isomorphism ν with components

$$\nu_{A,B} : \mathrm{Hom}_{\mathbf{C}}(A, B) \to \mathrm{Hom}_{\mathbf{D}}(1, H(A, B)).$$

Moreover, if we are in the context of preadditive categories, we require H to be additive in both components.

Remark 2.5.2 (The naturality of ν made explicit)**.** Let $\mathbf{C}$ be a category with a $\mathbf{D}$-homomorphism structure. The naturality of ν is given by the commutativity of the following diagram:

$$\begin{tikzcd}
\mathrm{Hom}_{\mathbf{C}}(A, B) \arrow[r, "\nu_{A,B}"] \arrow[d, "\mathrm{Hom}_{\mathbf{C}}(\alpha,\beta)"'] & \mathrm{Hom}_{\mathbf{D}}(1, H(A, B)) \arrow[d, "\mathrm{Hom}_{\mathbf{D}}(\mathrm{id}_1, H(\alpha,\beta))"] \\
\mathrm{Hom}_{\mathbf{C}}(C, D) \arrow[r, "\nu_{C,D}"] & \mathrm{Hom}_{\mathbf{D}}(1, H(C, D)
\end{tikzcd}$$

If we spell out the definitions of the hom-functors, we get for $\xi \in \mathrm{Hom}_{\mathbf{C}}(A, B)$ that

$$\nu_{C,D}(\alpha \cdot \xi \cdot \beta) = \mathrm{id}_1 \cdot \nu_{A,B}(\xi) \cdot H(\alpha, \beta) = \nu_{A,B}(\xi) \cdot H(\alpha, \beta).$$

For readers familiar with enriched category theory, we can relate homomorphism structures to enriched categories as follows:

Remark 2.5.3 (Connection to enriched categories, [Pos21a, Remark 6.5])**.** Let $\mathbf{C}$ be a category enriched over a locally small symmetric closed monoidal category $\mathbf{D}$. For two objects A and B of $\mathbf{C}$, let us write $H(A, B)$ for the hom object in $\mathbf{D}$. Moreover, let $\mathbf{C}_0$ be the underlying category of $\mathbf{C}$, that is, the category with the same objects as $\mathbf{C}$ and with the sets of morphisms defined by

$$\mathrm{Hom}_{\mathbf{C}_0}(A, B) \coloneqq \mathrm{Hom}_{\mathbf{D}}(1, H(A, B)),$$

where 1 denotes the tensor unit of $\mathbf{D}$. As stated in [Pos21a, Remark 6.5], we can extend H on objects to a functor $H : \mathbf{C}_0^{\mathrm{op}} \times \mathbf{C}_0 \to \mathbf{D}$ giving rise to a $\mathbf{D}$-homomorphism structure of $\mathbf{C}_0$.

Lemma 2.5.4. *If H is additive, then so is ν.*

[2] We write $\mathbf{C}^{\mathrm{op}}$ for the first component of H to signify the contravariance, but still apply H to objects and morphisms of $\mathbf{C}$ without an additional box in the first component.

Proof. Let $\mathbf{C}$ be a category with a $\mathbf{D}$-homomorphism structure. Moreover, let $\alpha_1, \alpha_2 : A \to B$ be two morphisms in $\mathbf{C}$. Using Remark 2.5.2, we compute:

$$\begin{aligned}\nu(\alpha_1 + \alpha_2) &= \nu((\alpha_1 + \alpha_2) \cdot \mathrm{id}_B \cdot \mathrm{id}_B) \\ &= \nu(\mathrm{id}_B) \cdot H(\alpha_1 + \alpha_2, \mathrm{id}_B) \\ &= \nu(\mathrm{id}_B) \cdot \big(H(\alpha_1, \mathrm{id}_B) + H(\alpha_2, \mathrm{id}_B)\big) \\ &= \big(\nu(\mathrm{id}_B) \cdot H(\alpha_1, \mathrm{id}_B)\big) + \big(\nu(\mathrm{id}_B) \cdot H(\alpha_2, \mathrm{id}_B)\big) \\ &= \nu(\alpha_1) + \nu(\alpha_2). \qquad \blacksquare\end{aligned}$$

Note that we could have used `CompilerForCAP` to prove this lemma, just as we use `CompilerForCAP` for many proofs in this section. However, since the proof is just a single sequence of equations, it is easier to simply give a manual proof.

Remark 2.5.5 (Extending homomorphism structures by full embeddings)**.** Let $\mathbf{C}$ be a category with a $\mathbf{D}$-homomorphism structure. Assume that we can embed $\mathbf{D}$ into a category $\mathbf{D}'$ by a full embedding $E : \mathbf{D} \hookrightarrow \mathbf{D}'$. Then $E(1)$, $H \cdot E$ and $\nu \cdot E$ define a $\mathbf{D}'$-homomorphism structure for $\mathbf{C}$.

We now give various examples of homomorphism structures of categories.

Construction 2.5.6 (Homomorphism structures on commutative rings)**.** Let k be a commutative ring. We can endow $\mathcal{C}(k)$ with a $\mathcal{C}(k)$-homomorphism structure as follows:

- The distinguished object 1 is the unique object $\star$ of $\mathcal{C}(k)$,
- we define

$$\begin{aligned} H : \mathcal{C}(k)^{\mathrm{op}} \times \mathcal{C}(k) &\to \mathcal{C}(k), \\ (\star, \star) &\mapsto \star, \\ (\boxed{r}, \boxed{s}) &\mapsto \boxed{r \cdot s}, \end{aligned}$$

- we define the only component $\nu_{\star,\star}$ of ν as the identity on

$$\mathrm{Hom}_{\mathcal{C}(k)}(\star, \star) \equiv \mathrm{Hom}_{\mathcal{C}(k)}(1, H(\star, \star)).$$

We provide a proof after the next construction, which is a generalization of the above.

Construction 2.5.7 (Homomorphism structures for linear categories, [Sal22, Lemma 4.11])**.** Let k be a commutative ring and let $\mathbf{C}$ be a k-linear category with finitely generated free external homs. For every set of morphisms $\mathrm{Hom}_{\mathbf{C}}(A, B)$, we fix a k-basis $\mathcal{B}_{A,B}$. Moreover, given a morphism $\varphi : A \to B$, we write λ_φ for the row vector of coefficients of φ with regard to the k-basis $\mathcal{B}_{A,B}$.

We can endow $\mathbf{C}$ with a $\mathbf{Mat}_k$-homomorphism structure as follows:

- The distinguished object is $\boxed{1}$.

- We define

$$
\begin{aligned}
H : \mathbf{C}^{\mathrm{op}} \times \mathbf{C} &\to \mathbf{Mat}_k \\
(A, B) &\mapsto \boxed{|\mathcal{B}_{A,B}|} \qquad \big(\text{with } |\mathcal{B}_{A,B}| \in \mathbb{N}\big) \\
(\alpha, \beta) &\mapsto \boxed{\begin{pmatrix} \lambda_{\alpha\cdot\varphi_1\cdot\beta} \\ \vdots \\ \lambda_{\alpha\cdot\varphi_n\cdot\beta} \end{pmatrix}} \qquad \begin{array}{l} \text{for } (\varphi_1, \dots, \varphi_n) = \mathcal{B}_{t(\alpha), s(\beta)} \\ \big(\text{with } \lambda_{\alpha\cdot\varphi_i\cdot\beta} \in k^{1\times|\mathcal{B}_{s(\alpha),t(\beta)}|}\big) \end{array}
\end{aligned}
$$

- We define

$$
\begin{aligned}
\nu_{A,B} : \mathrm{Hom}_{\mathbf{C}}(A, B) &\to \mathrm{Hom}_{\mathbf{Mat}_k}(\boxed{1}, H(A, B)), \\
\varphi &\mapsto \boxed{\lambda_\varphi} \qquad \big(\text{with } \lambda_\varphi \in k^{1\times|\mathcal{B}_{A,B}|}\big)
\end{aligned}
$$

Proof of correctness. A proof can be found in [Sal22, Lemma 4.11]. We will also use `CompilerForCAP` to prove the correctness of the construction in Section 7.4.2. ■

Proof of correctness of Construction 2.5.6. Let k be a commutative ring. As we have seen in Construction 2.3.8, $\mathcal{C}(k)$ is a k-linear category with finitely generated free external homs. Using Construction 2.5.7, we get a $\mathbf{Mat}_k$-homomorphism structure for $\mathcal{C}(k)$. However, we can do better: The only set of morphisms of $\mathcal{C}(k)$ is simply given by k, that is, is a free k-module **of rank** 1. Hence, in this case all matrices appearing in Construction 2.5.7 are 1×1 matrices. The subcategory of $\mathbf{Mat}_k$ given by the 1×1 matrices can be identified with $\mathcal{C}(k)$. Hence, $\mathcal{C}(k)$ has a $\mathcal{C}(k)$-homomorphism structure. This is just the homomorphism structure in Construction 2.5.6. ■

Construction 2.5.8 (Homomorphism structures of additive closures, [Pos21b, Construction 1.27])**.** Let $\mathbf{C}$ be a preadditive category with a $\mathbf{D}$-homomorphism structure, where $\mathbf{D}$ is an additive category. We construct a $\mathbf{D}$-homomorphism structure for $\mathbf{C}^{\oplus}$ as follows:

- We keep the distinguished object,
- we define

$$
\begin{aligned}
H : (\mathbf{C}^{\oplus})^{\mathrm{op}} \times \mathbf{C}^{\oplus} &\to \mathbf{D} \\
(\boxed{A}, \boxed{B}) &\mapsto \bigoplus_{i=1}^{a} \left(\bigoplus_{j=1}^{b} H(A_i, B_j) \right) \\
(\boxed{\alpha}, \boxed{\beta}) &\mapsto \left\langle \left(\left\langle \big(H(\alpha_{ij}, \beta_{st}) \big)_t^{\oplus} \right\rangle_s^{\oplus} \right)_i^{\oplus} \right\rangle_j^{\oplus}
\end{aligned}
$$

- we define

$$
\begin{aligned}
\nu_{\boxed{A}, \boxed{B}} : \mathrm{Hom}_{\mathbf{C}^{\oplus}}(\boxed{A}, \boxed{B}) &\to \mathrm{Hom}_{\mathbf{D}}(1, H(\boxed{A}, \boxed{B})) \\
\boxed{\alpha} &\mapsto \Big(\big(\nu(\alpha_{ij}) \big)_j^{\oplus} \Big)_i^{\oplus}
\end{aligned}
$$

Proof of correctness. We will use `CompilerForCAP` to prove the correctness of the construction in Section 7.4.3. ■

Construction 2.5.9 (Homomorphism structures of opposite categories)**.** Let $\mathbf{C}$ be a category with a $\mathbf{D}$-homomorphism structure. We can construct a $\mathbf{D}$-homomorphism structure for the opposite category $\mathbf{O} := \mathbf{C}^{\mathrm{op}}$ as follows:

- We keep the distinguished object,
- we swap the arguments of H, that is, we define

$$\begin{aligned} H : \mathbf{O}^{\mathrm{op}} \times \mathbf{O} &\to \mathbf{D}, \\ (\boxed{A}, \boxed{B}) &\mapsto H(B, A), \\ (\boxed{\alpha}, \boxed{\beta}) &\mapsto H(\beta, \alpha), \end{aligned}$$

- we "swap the components" of ν, that is, we define

$$\begin{aligned} \nu_{\boxed{A}, \boxed{B}} : \mathrm{Hom}_{\mathbf{O}}\big(\boxed{A}, \boxed{B}\big) &\to \mathrm{Hom}_{\mathbf{D}}\big(1, H(\boxed{A}, \boxed{B})\big) \equiv \mathrm{Hom}_{\mathbf{D}}\big(1, H(B, A)\big), \\ \boxed{\alpha} &\mapsto \nu_{B,A}(\alpha). \end{aligned}$$

Proof of correctness. We will use `CompilerForCAP` to prove the correctness of the construction at the end of Section 7.4. ■

2.6 Freyd categories

In this section, we introduce **Freyd categories**, a concept conceived by Peter Freyd in [Fre66]. We will use Freyd categories to model categories of finitely presented modules in Construction 4.2.5. As a motivation for the definition of Freyd categories, see Construction 4.2.4.

Definition 2.6.1 (Freyd categories)**.** Let $\mathbf{C}$ be a preadditive[3] category. We define the **Freyd category** $\mathbf{Freyd(C)}$ **of** $\mathbf{C}$ as follows:

- Its objects are the morphisms of $\mathbf{C}$.
- Let $\alpha : A \to B$ and $\beta : C \to D$ be morphisms in $\mathbf{C}$. A morphism from $\boxed{\alpha}_{\mathbf{Freyd(C)}}$ to $\boxed{\beta}_{\mathbf{Freyd(C)}}$ is a morphism $\varphi : B \to D$ in $\mathbf{C}$ such that there exists a morphism $\varphi' : A \to C$ which makes the following diagram commute:

$$\begin{tikzcd} A \arrow[r, "\varphi'"] \arrow[d, "\alpha"'] & C \arrow[d, "\beta"] \\ B \arrow[r, "\varphi"'] & D \end{tikzcd}$$

  We consider two morphisms $\boxed{\varphi}_{\mathbf{Freyd(C)}}$ and $\boxed{\psi}_{\mathbf{Freyd(C)}}$ as equal if and only if there exists a morphism $\kappa : B \to C$ in $\mathbf{C}$ which makes the triangle in the following diagram commute:

$$\begin{tikzcd} A \arrow[d, "\alpha"'] & C \arrow[d, "\beta"] \\ B \arrow[ur, "\kappa"] \arrow[r, "\varphi-\psi"'] & D \end{tikzcd}$$

[3] One usually defines Freyd categories over **additive** categories. With this, the Freyd category has more structure, for example cokernels. However, we do not need the additional structure in this thesis, so we try to stay as general as possible.

- Composition is given by composition in $\mathbf{C}$.
- Identity morphisms are given by identities in $\mathbf{C}$.

One can easily check that this indeed defines a category.

Proposition 2.6.2 (Decidable lifts in Freyd categories via homomorphism structures, [Pos21a, Corollary 6.11])**.** *Let* $\mathbf{D}$ *be an additive category with decidable lifts and let* $\mathbf{C}$ *be a preadditive category with a* $\mathbf{D}$*-homomorphism structure. Then* $\mathbf{Freyd}(\mathbf{C})$ *has decidable lifts.*

Proof. Let $\boxed{\varphi} : \boxed{\alpha} \to \boxed{\beta}$ and $\boxed{\psi} : \boxed{\gamma} \to \boxed{\beta}$ be two morphisms in $\mathbf{Freyd}(\mathbf{C})$. Assume that we want to compute a lift $\boxed{\xi} : \boxed{\gamma} \to \boxed{\alpha}$ such that the following diagram commutes:

$$\begin{tikzcd}
\boxed{\alpha} \arrow[r, "\boxed{\varphi}"] & \boxed{\beta} \\
\boxed{\gamma} \arrow[u, "\boxed{\xi}"] \arrow[ur, "\boxed{\psi}"'] &
\end{tikzcd}$$

In order for the morphism ξ in $\mathbf{C}$ to define a morphism $\boxed{\gamma} \to \boxed{\alpha}$, there has to exist a suitable morphism ξ' in $\mathbf{C}$ such that

$$\gamma \cdot \xi = \xi' \cdot \alpha. \tag{2.1}$$

Moreover, to make the diagram commute there has to exist a suitable morphism κ in $\mathbf{C}$ such that

$$\xi \cdot \varphi - \psi = \kappa \cdot \beta. \tag{2.2}$$

We would like to combine both equations (2.1) and (2.2) into a system of linear equations as defined in Remark 2.4.5. Unfortunately, in the first equation (2.1), the unknowns ξ and ξ' appear on different sides of the composition. This situation is not covered by Remark 2.4.5. However, we can use the naturality of ν of the homomorphism structure to rewrite the first equation (2.1) as the following equation in $\mathbf{D}$:

$$\nu(\xi) \cdot H(\gamma, \mathrm{id}_1) = \nu(\xi') \cdot H(\mathrm{id}_2, \alpha), \tag{2.3}$$

where id_1 and id_2 are suitable identity morphisms. Now, both indeterminates $\nu(\xi)$ and $\nu(\xi')$ appear on the left-hand side of a composition, and we can further rewrite equation (2.3) as

$$\nu(\xi) \cdot H(\gamma, \mathrm{id}_1) + \nu(\xi') \cdot \big(-H(\mathrm{id}_2, \alpha)\big) = 0, \tag{2.4}$$

where 0 is a suitable zero morphism.

Consequently, we also apply ν to (2.2) and obtain, using the naturality and additivity of ν:

$$\nu(\xi) \cdot H(\mathrm{id}_3, \varphi) - \nu(\psi) = \nu(\kappa) \cdot H(\mathrm{id}_4, \beta),$$

where id_3 and id_4 are suitable identity morphisms. We can rewrite this equation further as

$$\nu(\xi) \cdot H(\mathrm{id}_3, \varphi) + \nu(\kappa) \cdot \big(-H(\mathrm{id}_4, \beta)\big) = \nu(\psi). \tag{2.5}$$

Now, we can combine (2.4) and (2.5) into a system of linear equations in $\mathbf{D}$:

$$\begin{aligned}
&\nu(\xi) \cdot H(\gamma, \mathrm{id}_1) + \nu(\xi') \cdot \big(-H(\mathrm{id}_2, \alpha)\big) + \nu(\kappa) \cdot 0 &&= 0,\\
&\nu(\xi) \cdot H(\mathrm{id}_3, \varphi) + \nu(\xi') \cdot 0 \qquad\qquad\quad\; + \nu(\kappa) \cdot \big(-H(\mathrm{id}_4, \beta)\big) &&= \nu(\psi),
\end{aligned}$$

where the new zeros are again suitable zero morphisms. Since $\mathbf{D}$ is additive, we can use Remark 2.4.5 to rewrite this system of linear equations as a single equation of morphisms between direct sums:

$$\begin{pmatrix} \nu(\xi) & \nu(\xi') & \nu(\kappa) \end{pmatrix}^{\oplus} \cdot \begin{pmatrix} H(\gamma, \mathrm{id}_1) & H(\mathrm{id}_3, \varphi) \\ -H(\mathrm{id}_2, \alpha) & 0 \\ 0 & -H(\mathrm{id}_4, \beta) \end{pmatrix}^{\oplus} = \begin{pmatrix} 0 & \nu(\psi) \end{pmatrix}^{\oplus}.$$

Since $\mathbf{D}$ has decidable lifts and ν is bijective, we can solve this equation for $\begin{pmatrix} \nu(\xi) & \nu(\xi') & \nu(\kappa) \end{pmatrix}^{\oplus}$, extract the components $\nu(\xi)$, $\nu(\xi')$, and $\nu(\kappa)$, and apply the inverse of ν to obtain ξ, ξ', and κ. ■

2.7 Limits and colimits

In this section, we introduce some **limits** and **colimits**. Limits and colimits are defined by universal properties and are hence **essentially unique**, that is, unique up to a unique isomorphism.

Definition 2.7.1 (Coproducts)**.** Let $\mathbf{C}$ be a category and let A and B be two objects of $\mathbf{C}$. The (binary) **coproduct** of A and B is an object $A \sqcup B$ in $\mathbf{C}$ together with two morphisms

$$\iota_A^{\sqcup} : A \to A \sqcup B \qquad \text{and} \qquad \iota_B^{\sqcup} : B \to A \sqcup B$$

with the following universal property: For every two morphisms $\tau_A : A \to T$ and $\tau_B : B \to T$, there exists a unique morphism

$$u : A \sqcup B \to T$$

such that the triangles in the following diagram commute:

$$\begin{tikzcd}
A \arrow[dr, "\iota_A^{\sqcup}"] \arrow[ddr, bend right, "\tau_A"'] & & B \arrow[dl, "\iota_B^{\sqcup}"'] \arrow[ddl, bend left, "\tau_B"] \\
& A \sqcup B \arrow[d, dashed, "\exists_1 u"] & \\
& T &
\end{tikzcd}$$

The definition can be generalized to coproducts of arbitrary finite families of objects of $\mathbf{C}$. In the degenerate case of an empty family of objects, we obtain the definition of an **initial object**.

The definition can be extended to morphisms in a functorial way: Let $f : A \to B$ and $g : C \to D$ be two morphisms. We define $f \sqcup g : A \sqcup C \to B \sqcup D$ as the unique morphism making the following diagram commute:

$$\begin{tikzcd}
A \arrow[d, "f"'] \arrow[dr, "\iota^{\sqcup}_{A}"] & & C \arrow[d, "g"] \arrow[dl, "\iota^{\sqcup}_{C}"'] \\
B \arrow[dr, "\iota^{\sqcup}_{B}"'] & A \sqcup C \arrow[d, dashed, "f \sqcup g"] & D \arrow[dl, "\iota^{\sqcup}_{D}"] \\
& B \sqcup D &
\end{tikzcd}$$

One can check that this defines a functor in two arguments $\sqcup : \mathbf{C} \otimes \mathbf{C} \to \mathbf{C}$.

By dualizing the definition, we obtain the definition of **products**, with the degenerate case of an empty family of objects giving the definition of a **terminal object**.

Construction 2.7.2 (Coproducts in functor categories)**.** Let $\mathbf{C}$ be a category and let $\mathbf{D}$ be a category with coproducts. Then the functor category from $\mathbf{C}$ to $\mathbf{D}$ has coproducts:

Let $\boxed{F}, \boxed{G}$ be two objects of $\mathbf{D}^{\mathbf{C}}$, that is, $F, G : \mathbf{C} \to \mathbf{D}$ are two functors. We define a coproduct $\boxed{C} := \boxed{F} \sqcup \boxed{G}$ of $\boxed{F}$ and $\boxed{G}$ as follows:

$$\begin{aligned} C : \mathbf{C} &\to \mathbf{D} \\ A &\mapsto F(A) \sqcup G(A) \\ f &\mapsto F(f) \sqcup G(f) \end{aligned}$$

The injections $\iota^{\sqcup}_{\boxed{F}} : \boxed{F} \to \boxed{C}$ and $\iota^{\sqcup}_{\boxed{G}} : \boxed{G} \to \boxed{C}$ are given by the natural transformation whose components are the injections

$$\iota^{\sqcup}_{F(A)} : F(A) \to F(A) \sqcup G(A) \qquad \text{and} \qquad \iota^{\sqcup}_{G(A)} : G(A) \to F(A) \sqcup G(A),$$

respectively. Given two morphisms $\boxed{\tau_1} : \boxed{F} \to \boxed{T}$ and $\boxed{\tau_2} : \boxed{G} \to \boxed{T}$, the universal morphism $u : \boxed{C} \to \boxed{T}$ can be defined by taking the universal morphism in every component.

The construction can be generalized to coproducts of arbitrary finite families of objects of $\mathbf{D}^{\mathbf{C}}$.

Proof of correctness. The functoriality of $\sqcup : \mathbf{C} \otimes \mathbf{C} \to \mathbf{C}$ ensures that C is indeed a functor. Moreover, the construction of $\sqcup$ on morphisms ensures that the injections and the universal morphism are actually natural transformations. Finally, since the equality of morphisms in functor categories is given componentwise and all constructions are defined componentwise, the required properties follow immediately from the properties of coproducts in $\mathbf{D}$. ■

Definition 2.7.3 (Coequalizers)**.** Let $\mathbf{C}$ be a category and let $f, g : A \to B$ be two parallel morphisms of $\mathbf{C}$. The (binary) **coequalizer** of f and g is an object $\operatorname{coeq}(f, g)$ in $\mathbf{C}$ together with a morphism

$$\pi^{\text{coeq}} : B \to \operatorname{coeq}(f, g) \quad \text{such that} \quad f \cdot \pi^{\text{coeq}} = g \cdot \pi^{\text{coeq}}$$

with the following universal property: For every morphism $\tau : B \to T$ such that $f \cdot \tau = g \cdot \tau$, there exists a unique morphism

$$u : \operatorname{coeq}(f, g) \to T$$

such that the triangle in the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] & B \arrow[r, "\pi^{\mathrm{coeq}}"] \arrow[dr, "\tau"'] & \mathrm{coeq}(f,g) \arrow[d, dashed, "\exists_1 u"] \\
 & & T
\end{tikzcd}$$

The definition can be generalized to coequalizers of arbitrary finite families of parallel morphisms of **C**.

Similar to the situation for coproducts, the definition can be extended in a functorial way: Let $f, g : A \to B$ and $h, i : C \to D$ be two pairs of parallel morphisms and let $k : B \to D$ and $\ell : A \to C$ be two morphisms such that

$$f \cdot k = \ell \cdot h \qquad \text{and} \qquad g \cdot k = \ell \cdot i.$$

Moreover, consider the coequalizers $\mathrm{coeq}(f,g)$ with $\pi_1^{\mathrm{coeq}} : B \to \mathrm{coeq}(f,g)$ and $\mathrm{coeq}(h,i)$ with $\pi_2^{\mathrm{coeq}} : B \to \mathrm{coeq}(h,i)$. By a diagram chase, we get that

$$f \cdot (k \cdot \pi_2^{\mathrm{coeq}}) = g \cdot (k \cdot \pi_2^{\mathrm{coeq}}).$$

Hence, there exists a unique morphism $u_k^{\mathrm{coeq}} : \mathrm{coeq}(f,g) \to \mathrm{coeq}(h,i)$ such that

$$\pi_1^{\mathrm{coeq}} \cdot u_k^{\mathrm{coeq}} = k \cdot \pi_2^{\mathrm{coeq}}.$$

The situation can be visualized as follows:

$$\begin{tikzcd}
A \arrow[r, shift left, "f"] \arrow[r, shift right, "g"'] \arrow[d, "\ell"'] & B \arrow[r, "\pi_1^{\mathrm{coeq}}"] \arrow[d, "k"] & \mathrm{coeq}(f,g) \arrow[d, dashed, "\exists_1 u_k^{\mathrm{coeq}}"] \\
C \arrow[r, shift left, "h"] \arrow[r, shift right, "i"'] & D \arrow[r, "\pi_2^{\mathrm{coeq}}"'] & \mathrm{coeq}(h,i)
\end{tikzcd}$$

One can check that the construction is functorial in k, that is,

$$u_{\mathrm{id}_B}^{\mathrm{coeq}} = \mathrm{id}_{\mathrm{coeq}(f,g)} \qquad \text{and} \qquad u_{k_1 \cdot k_2}^{\mathrm{coeq}} = u_{k_1}^{\mathrm{coeq}} \cdot u_{k_2}^{\mathrm{coeq}}$$

for suitable coequalizer diagrams and morphisms k_1 and k_2.

By dualizing the definition, we obtain the definition of **equalizers**.

Example 2.7.4 (Coequalizers in the category of sets)**.** Let $\boxed{f}, \boxed{g} : \boxed{M} \to \boxed{N}$ be two parallel morphisms in **Sets**. Let $\sim^{\mathrm{cl}}$ be the smallest equivalence relation on N such that $f(m) \sim^{\mathrm{cl}} g(m)$ for all $m \in M$, that is, $\sim^{\mathrm{cl}}$ is the reflexive, symmetric, and transitive closure of the relation $f(m) \sim g(m)$ for all $m \in M$. Then the object $\boxed{B/{\sim^{\mathrm{cl}}}}$ given by the quotient set together with the morphism $\boxed{B} \to \boxed{B/{\sim^{\mathrm{cl}}}}$ given by the canonical projection $B \to B/{\sim^{\mathrm{cl}}}$ forms a coequalizer of $\boxed{f}$ and $\boxed{g}$. The universal property follows from the homomorphism theorem for sets. The construction can be generalized to coequalizers of arbitrary finite families of parallel morphisms of **Sets**.

Constructing the set of cosets $C := B/{\sim^{\mathrm{cl}}}$ can be made algorithmic as follows: We start with the finest possible choice of cosets $C := \{\{n\} \mid n \in N\}$. Then, we loop over the relation $\sim$: For $a \sim b$, if the cosets containing a and b are different, we merge them. By construction, the resulting cosets respect the relation $\sim$, and give indeed rise to the **smallest** equivalence relation respecting $\sim$.

On a computer, this process is relatively intricate and expensive: For every $a \sim b$, we have to loop over C and every coset of C to find the cosets C_a and C_b containing a and b. Afterwards, we possibly have to merge them, that is, remove C_a and C_b from C and add $C_a \cup C_b$ instead. The creation and manipulation of sets is usually a relatively expensive operation on a computer because it typically involves large memory allocations for large sets. Moreover, the situation gets even more complicated if we generalize to arbitrary finite families of parallel morphisms of **Sets**.

Construction 2.7.5 (Coequalizers in functor categories)**.** Let $\mathbf{C}$ be a category and let $\mathbf{D}$ be a category with coequalizers. Then the functor category from $\mathbf{C}$ to $\mathbf{D}$ has coequalizers:

Let $\boxed{\eta}, \boxed{\nu} : \boxed{F} \to \boxed{G}$ be two parallel morphisms of $\mathbf{D}^{\mathbf{C}}$. We define a coequalizer $\boxed{C} := \mathrm{coeq}(\boxed{\eta}, \boxed{\nu})$ of $\boxed{\eta}$ and $\boxed{\nu}$ as follows:

$$\begin{aligned} C : \mathbf{C} &\to \mathbf{D} \\ A &\mapsto \mathrm{coeq}(\eta_A, \nu_A) \\ f &\mapsto u_f^{\mathrm{coeq}} \end{aligned}$$

The morphism $\pi^{\mathrm{coeq}} : \boxed{G} \to \boxed{C}$ is given by the natural transformation whose components are the morphisms

$$\pi^{\mathrm{coeq}} : G(A) \to \mathrm{coeq}(\eta_A, \nu_A).$$

Given a morphism $\boxed{\tau} : \boxed{G} \to \boxed{T}$ such that

$$\boxed{\eta} \cdot \boxed{\tau} = \boxed{\nu} \cdot \boxed{\tau},$$

the universal morphism $u : \boxed{C} \to \boxed{T}$ can be defined by taking the universal morphism in every component.

The construction can be generalized to coequalizers of arbitrary finite families of parallel morphisms of $\mathbf{D}^{\mathbf{C}}$.

Proof of correctness. The functoriality of u^{coeq} ensures that C is indeed a functor. Moreover, the construction of u^{coeq} ensures that π^{coeq} and the universal morphism are actually natural transformations. Finally, since the equality of morphisms in functor categories is given componentwise and all constructions are defined componentwise, the required properties follow immediately from the properties of coequalizers in $\mathbf{D}$. ■

In a preadditive category, a coequalizer of a morphism with the zero morphism is called a **cokernel**. We explicitly spell out the definition:

Definition 2.7.6 (Cokernels)**.** Let $\mathbf{C}$ be a preadditive category and let $f : A \to B$ be a morphism of $\mathbf{C}$. The **cokernel** of f is an object $\mathrm{coker}(f)$ in $\mathbf{C}$ together with a morphism

$$\pi^{\mathrm{coker}} : B \to \mathrm{coker}(f) \quad \text{such that} \quad f \cdot \pi^{\mathrm{coker}} \text{ is zero}$$

with the following universal property: For every morphism $\tau : B \to T$ such that $f \cdot \tau$ is zero, there exists a unique morphism

$$u : \mathrm{coker}(f) \to T$$

such that the triangle in the following diagram commutes:

$$\begin{tikzcd}
A \arrow[r, "f"] & B \arrow[r, "\pi^{\mathrm{coker}}"] \arrow[rd, "\tau"'] & \mathrm{coker}(f) \arrow[d, dashed, "\exists_1 u"] \\
& & T
\end{tikzcd}$$

By dualizing the definition, we obtain the definition of **kernels**.

Remark 2.7.7 (Deriving binary coequalizers from cokernels and vice versa)**.** Let **C** be a preadditive category. Then all binary coequalizers exist in **C** if and only if all cokernels exists in **C**: To compute a cokernel of a morphism f, we can simply compute the coequalizer of f and 0. Conversely, to compute a coequalizer of two morphisms f and g, we can compute a cokernel of $f - g$ and use that

$$f - g \text{ is zero}$$

is equivalent to

$$f = g.$$

Definition 2.7.8 (Pushouts)**.** Let **C** be a category and let $f : C \to A$ and $g : C \to B$ be two morphisms in **C** with a common source. The (binary) **pushout** of f and g is an object P in **C** together with two morphisms

$$\iota_A : A \to P \quad \text{and} \quad \iota_B : B \to P \quad \text{such that} \quad f \cdot \iota_A = g \cdot \iota_B$$

with the following universal property: For every two morphisms $\tau_A : A \to T$ and $\tau_B : B \to T$ such that $f \cdot \tau_A = g \cdot \tau_B$, there exists a unique morphism

$$u : P \to T$$

such that the two triangles in the following diagram commute:

$$\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "g"'] & A \arrow[d, "\iota_A"] \arrow[rdd, bend left, "\tau_A"] & \\
B \arrow[r, "\iota_B"'] \arrow[rrd, bend right, "\tau_B"'] & P \arrow[rd, dashed, "\exists_1 u"] & \\
& & T
\end{tikzcd}$$

The definition can be generalized to pushouts of arbitrary finite families of morphisms in **C** with a common source. By dualizing the definition, we obtain the definition of **pullbacks**.

2.8 Monoidal categories

Definition 2.8.1 (Monoidal categories)**.** A **monoidal category** is a category **C** together with the following data:

- a functor $\otimes : \mathbf{C} \times \mathbf{C} \to \mathbf{C}$ called the **tensor product**,
- an object 1 in **C** called the **tensor unit**,[4]

[4] Note that we also use “1” to denote the distinguished object of a homomorphism structure. However, in this thesis we will not consider homomorphism structures and monoidal structures at the same time, so it is safe to reuse “1” for the tensor unit.

- a natural isomorphism α with components

$$\alpha_{A,B,C} : (A \otimes B) \otimes C \cong A \otimes (B \otimes C) \quad (\textbf{associators}),$$

- a natural isomorphism λ with components

$$\lambda_A : 1 \otimes A \cong A \quad (\textbf{left unitors}),$$

- a natural isomorphism ρ with components

$$\rho_A : A \otimes 1 \cong A \quad (\textbf{right unitors}),$$

such that for every two objects A and B the following **triangle identity** holds:

$$\begin{tikzcd}
(A \otimes 1) \otimes B \arrow[rr, "\alpha_{A,B,C}"] \arrow[dr, "\rho_A \otimes \mathrm{id}_B"'] & & A \otimes (1 \otimes B) \arrow[dl, "\mathrm{id}_A \otimes \lambda_B"] \\
& A \otimes B &
\end{tikzcd}$$

and for every four objects A, B, C, and D the following **pentagon identity** holds:

$$\begin{tikzcd}
& ((A \otimes B) \otimes C) \otimes D \arrow[dl, "\alpha_{A,B,C} \otimes \mathrm{id}_D"'] \arrow[ddr, "\alpha_{A \otimes B,C,D}"] & \\
(A \otimes (B \otimes C)) \otimes D \arrow[dd, "\alpha_{A,B \otimes C,D}"'] & & \\
& & (A \otimes B) \otimes (C \otimes D) \arrow[ddl, "\alpha_{A,B,C \otimes D}"] \\
A \otimes ((B \otimes C) \otimes D) \arrow[dr, "\mathrm{id}_A \otimes \alpha_{B,C,D}"'] & & \\
& A \otimes (B \otimes (C \otimes D)) &
\end{tikzcd}$$

If the associators and unitors are given by identity natural transformations, we call **C** a **strict** monoidal category.

Note that the pentagon identity can be seen in the same spirit as Figure 1.1: Both relate all possible ways of putting parentheses in a expression joining four things by a binary operator.

There are many equivalent definitions of **closed** monoidal categories. For our application in Chapter 5, the following definition is the most useful:

Definition 2.8.2 (Closed monoidal categories)**.** A **closed monoidal category** is a monoidal category **C** such that for every two objects B and C we have

- an object $\mathrm{hom}(B, C)$, called the **internal hom object of** B **and** C, and
- a morphism $\mathrm{ev}_{B,C} : \mathrm{hom}(B, C) \otimes B \to C$ called the **evaluation morphism on** B **and** C

such that for every morphism $f : A \otimes B \to C$ there exists a unique morphism $g : A \to \mathrm{hom}(B, C)$ which makes the following diagram commute:

$$\begin{tikzcd}
A \otimes B \arrow[rr, "f"] \arrow[dr, "g\otimes\mathrm{id}_B"'] & & C \\
& \hom(B,C) \otimes B \arrow[ur, "\mathrm{ev}_{B,C}"'] &
\end{tikzcd}$$

Note that the uniqueness of g ensures that mapping f to g as above is compatible with the equality on morphisms.

Definition 2.8.3 (Braided monoidal categories)**.** A **braided monoidal category** is a monoidal category together with a natural isomorphism γ with components

$$\gamma_{A,B} : A \otimes B \cong B \otimes A \quad (\textbf{braidings})$$

such that for every three objects A, B, and C the following two diagrams commute:

$$\begin{tikzcd}
& A \otimes (B \otimes C) \arrow[r, "\gamma_{A,B\otimes C}"] & (B \otimes C) \otimes A \arrow[dr, "\alpha_{B,C,A}"] & \\
(A \otimes B) \otimes C \arrow[ur, "\alpha_{A,B,C}"] \arrow[dr, "\gamma_{A,B}\otimes\mathrm{id}_C"'] & & & B \otimes (C \otimes A) \\
& (B \otimes A) \otimes C \arrow[r, "\alpha_{B,A,C}"'] & B \otimes (A \otimes C) \arrow[ur, "\mathrm{id}_B\otimes\gamma_{A,C}"'] &
\end{tikzcd}$$

$$\begin{tikzcd}
& (A \otimes B) \otimes C \arrow[r, "\gamma_{A\otimes B,C}"] & C \otimes (A \otimes B) \arrow[dr, "\alpha^{-1}_{C,A,B}"] & \\
A \otimes (B \otimes C) \arrow[ur, "\alpha^{-1}_{A,B,C}"] \arrow[dr, "\mathrm{id}_A\otimes\gamma_{B,C}"'] & & & (C \otimes A) \otimes B \\
& A \otimes (C \otimes B) \arrow[r, "\alpha^{-1}_{A,C,B}"'] & (A \otimes C) \otimes B \arrow[ur, "\gamma_{A,C}\otimes\mathrm{id}_B"'] &
\end{tikzcd}$$

Definition 2.8.4 (Symmetric monoidal categories)**.** A **symmetric monoidal category** is a braided monoidal category in which for each two objects A and B the inverse law

$$\gamma_{A,B} \cdot \gamma_{B,A} = \mathrm{id}_{A\otimes B}$$

holds.

Definition 2.8.5 (Cartesian and cocartesian monoidal categories)**.** Let $\mathbf{C}$ be a category with finite coproducts. We can define a symmetric monoidal structure on $\mathbf{C}$ as follows:

- the tensor product is the binary coproduct,
- the tensor unit is the initial object,
- the associators, unitors, and braidings can be constructed by the universal property of the coproduct.

It is easy to check that this indeed defines a symmetric monoidal structure on $\mathbf{C}$. We call $\mathbf{C}$ with this symmetric monoidal structure a **cocartesian monoidal category**.

The analogous construction for products gives rise to the notion of a **cartesian monoidal category**.

Definition 2.8.6 (Rigid symmetric monoidal categories)**.** A **rigid symmetric monoidal category** is a symmetric monoidal category $\mathbf{C}$ in which every object A has a **dual** A^*, that is, there exist two morphisms

$$\varepsilon_A : A^* \otimes A \to 1 \quad \text{and} \quad \eta_A : 1 \to A \otimes A^*$$

called the **evaluation** and the **coevaluation**, respectively, such that the following two diagrams commute:

$$\begin{tikzcd}
(A \otimes A^*) \otimes A \arrow[d, "\alpha_{A,A^*,A}"'] & 1 \otimes A \arrow[l, "\eta_A \otimes \mathrm{id}_A"'] \arrow[d, "\lambda_A"] \\
 & A \\
A \otimes (A^* \otimes A) \arrow[r, "\mathrm{id}_A \otimes \varepsilon_A"'] & A \otimes 1 \arrow[u, "\rho_A"']
\end{tikzcd} \quad \text{and} \quad \begin{tikzcd}
A^* \otimes (A \otimes A^*) \arrow[d, "\alpha^{-1}_{A^*,A,A^*}"'] & A^* \otimes 1 \arrow[l, "\mathrm{id}_{A^*} \otimes \eta_A"'] \arrow[d, "\rho_{A^*}"] \\
 & A^* \\
(A^* \otimes A) \otimes A^* \arrow[r, "\varepsilon_A \otimes \mathrm{id}_{A^*}"'] & 1 \otimes A^* \arrow[u, "\lambda_{A^*}"']
\end{tikzcd}$$

The following construction can, for example, be found in [KL80].

Construction 2.8.7 (Rigid symmetric monoidal categories are closed)**.** Let $\mathbf{C}$ be a rigid symmetric monoidal category. We can define a closed structure on $\mathbf{C}$ as follows: We set $\hom(B, C) :\equiv C \otimes B^*$. Let $f : A \otimes B \to C$ be a morphism. We define the corresponding morphism

$$g : A \to \hom(B, C) \equiv C \otimes B^*$$

as the composite of the following morphisms:

$$A \xrightarrow{\rho_A^{-1}} A \otimes 1 \xrightarrow{\mathrm{id}_A \otimes \eta_B} A \otimes (B \otimes B^*) \xrightarrow{\alpha_{A,B,B^*}} (A \otimes B) \otimes B^* \xrightarrow{f \otimes \mathrm{id}_{B^*}} C \otimes B^*.$$

Furthermore, we define the evaluation morphism

$$\mathrm{ev}_{B,C} : (C \otimes B^*) \otimes B \equiv \hom(B, C) \otimes B \to C$$

as the composite of the following morphisms:

$$(C \otimes B^*) \otimes B \xrightarrow{\alpha_{C,B^*,B}} C \otimes (B^* \otimes B) \xrightarrow{\mathrm{id}_C \otimes \varepsilon_B} C \otimes 1 \xrightarrow{\rho_C} C.$$

Remark 2.8.8 (The closed structure of **strict** rigid symmetric monoidal categories)**.** If $\mathbf{C}$ is a **strict** rigid symmetric monoidal category, then the definitions in Construction 2.8.7 can be simplified by dropping associators and unitors:

- We set $\hom(B, C) :\equiv C \otimes B^*$ as before.
- Let $f : A \otimes B \to C$ be a morphism. We define the corresponding morphism $g : A \to C \otimes B^*$ as the composite of the following morphisms:

$$A \xrightarrow{\mathrm{id}_A \otimes \eta_B} A \otimes B \otimes B^* \xrightarrow{f \otimes \mathrm{id}_{B^*}} C \otimes B^*.$$

- We define the evaluation morphism $\mathrm{ev}_{B,C}$ as the morphism

$$\mathrm{id}_C \otimes \varepsilon_B : C \otimes B^* \otimes B \to C.$$

2.9 Slice categories and decorations

Definition 2.9.1 (Slice categories)**.** Given a **base object** B in a category $\mathbf{C}$, the **slice category** $\mathbf{C}/B$ is defined as follows:

- its objects are morphisms in $\mathbf{C}$ with target B,
- a morphism from $\boxed{X \to B}$ to $\boxed{Y \to B}$ in $\mathbf{C}/B$ is a morphism $X \to Y$ in $\mathbf{C}$ such that

$$\begin{tikzcd}
X \arrow[rr, dashed] \arrow[dr] & & Y \arrow[dl] \\
& B &
\end{tikzcd}$$

  commutes, with the equality of morphisms inherited from $\mathbf{C}$,
- composition of morphisms in $\mathbf{C}/B$ is given by composition of morphisms in $\mathbf{C}$,
- identity morphisms in $\mathbf{C}/B$ are given by identity morphisms in $\mathbf{C}$.

One can easily check that this indeed defines a category.

Remark 2.9.2 (Decorations)**.** If a category $\mathbf{C}$ has a notion of "elements" of objects and a notion of "fibers" of such elements under morphisms, we can visualize the construction of slice categories as follows:

A morphism $f\colon X \to B$ in $\mathbf{C}$ into the base object B can be interpreted as a **decoration** of X by the elements of B. That is, for $b \in B$ we think of the fiber $f^{-1}(b)$ as being **decorated** by b. By construction, morphisms in $\mathbf{C}/B$ preserve the decoration: The commutativity of

$$\begin{tikzcd}
X \arrow[rr, dashed] \arrow[dr] & & Y \arrow[dl] \\
& B &
\end{tikzcd}$$

ensures that if an element in X is decorated by $b \in B$, then also its image in Y under a morphism is decorated by b.

Examples of categories with the required properties include the category of sets and the category of directed multigraphs (see Construction 5.3.1).

Construction 2.9.3 (Coproducts in slice categories)**.** Let $\mathbf{C}$ be a category with coproducts and let B be an object of $\mathbf{C}$. Then the slice category $\mathbf{C}/B$ has coproducts:

Let $\boxed{f}$ and $\boxed{g}$ be two objects in $\mathbf{C}/B$ with $f : X \to B$ and $g : Y \to B$ in $\mathbf{C}$. Their coproduct $\boxed{f} \sqcup \boxed{g}$ is given by the universal morphism u in the following diagram:

$$\begin{tikzcd}
X \arrow[dr, "\iota^{\sqcup}_{X}"] \arrow[ddr, "f"'] & & Y \arrow[dl, "\iota^{\sqcup}_{Y}"'] \arrow[ddl, "g"] \\
& X \sqcup Y \arrow[d, dashed, "u"] & \\
& B &
\end{tikzcd}$$

The injections of $\boxed{f}$ and $\boxed{g}$ into $\boxed{f} \sqcup \boxed{g}$ are simply given by the injections of X and Y into $X \sqcup Y$. Given an object $\boxed{t}$ in $\mathbf{C}/B$ with $t : T \to B$ and two morphisms $\boxed{\tau_1} : \boxed{f} \to \boxed{t}$ and $\boxed{\tau_2} : \boxed{g} \to \boxed{t}$, the universal morphism $u : \boxed{f} \sqcup \boxed{g} \to \boxed{t}$ is simply given by the universal morphism $X \sqcup Y \to T$.

The construction can be generalized to coproducts of arbitrary finite families of objects of $\mathbf{C}/B$.

Proof of correctness. With a diagram chase one can check that the injections and the universal morphism actually define morphisms in $\mathbf{C}/B$. Since the equality of morphisms in $\mathbf{C}/B$ is given by the equality of morphisms in $\mathbf{C}$ and all morphisms are defined by constructions in $\mathbf{C}$, the required properties follow immediately from the properties of coproducts in $\mathbf{C}$. ■

Construction 2.9.4 (Coequalizers in slice categories)**.** Let $\mathbf{C}$ be a category with coequalizers and let B be an object of $\mathbf{C}$. Then the slice category $\mathbf{C}/B$ has coequalizers:

Let $\boxed{p}, \boxed{q} : \boxed{f} \to \boxed{g}$ be two parallel morphisms in $\mathbf{C}/B$ with $f : X \to B$ and $g : Y \to B$ in $\mathbf{C}$. In particular, we have

$$p \cdot g = f = q \cdot g.$$

The coequalizer $\mathrm{coeq}(\boxed{p}, \boxed{q})$ is given by the universal morphism u in the following diagram:

$$\begin{tikzcd}
X \arrow[r, shift left, "p"] \arrow[r, shift right, "q"'] \arrow[rrd, bend right, "f"'] & Y \arrow[r, "\pi^{\mathrm{coeq}}"] \arrow[rd, "g"] & \mathrm{coeq}(p,q) \arrow[d, dashed, "\exists_1 u"] \\
 & & B
\end{tikzcd}$$

The morphism $\boxed{g} \to \mathrm{coeq}(\boxed{p}, \boxed{q})$ is simply given by $\pi^{\mathrm{coeq}} : Y \to \mathrm{coeq}(p, q)$. Given an object $\boxed{t}$ in $\mathbf{C}/B$ with $t : T \to B$ and a morphism $\boxed{\tau} : \boxed{g} \to \boxed{t}$ such that

$$\boxed{p} \cdot \boxed{t} = \boxed{q} \cdot \boxed{t},$$

the universal morphism $u : \mathrm{coeq}(\boxed{p}, \boxed{q}) \to \boxed{t}$ is simply given by the universal morphism $\mathrm{coeq}(p, q) \to T$.

The construction can be generalized to coequalizers of arbitrary finite families of parallel morphisms of $\mathbf{C}/B$.

Proof of correctness. With a diagram chase one can check that all constructed morphisms actually define morphisms in $\mathbf{C}/B$. Since the equality of morphisms in $\mathbf{C}/B$ is given by the equality of morphisms in $\mathbf{C}$ and all morphisms are defined by constructions in $\mathbf{C}$, the required properties follow immediately from the properties of coequalizers in $\mathbf{C}$. ■

2.10 Categories of cospans

Definition 2.10.1 (Categories of cospans)**.** Given a category $\mathbf{C}$ with binary pushouts, the **category of cospans $\mathbf{Csp}(\mathbf{C})$ of $\mathbf{C}$** is defined as follows:

- its objects are objects of $\mathbf{C}$,

- a morphism between objects $\boxed{X}$ and $\boxed{Y}$ in the category of cospans is a **cospan** $X \to A \leftarrow Y$ in $\mathbf{C}$, that is, a pair of morphisms $X \to A$ and $Y \to A$ in $\mathbf{C}$ for some object A in $\mathbf{C}$, where two morphisms $\boxed{X \to A \leftarrow Y}$ and $\boxed{X \to B \leftarrow Y}$ are considered equal if and only if there exists an isomorphism $A \to B$ in $\mathbf{C}$ which makes the triangles in the following diagram commute:

$$\begin{tikzcd}
 & A \arrow[dd, dashed] & \\
X \arrow[ur] \arrow[dr] & & Y \arrow[ul] \arrow[dl] \\
 & B &
\end{tikzcd}$$

- the composition of two morphisms $\boxed{X \to A \leftarrow Y}$ and $\boxed{Y \to B \leftarrow Z}$ is given by the pushout P together with the two dashed arrows in the following commutative diagram:

$$\begin{tikzcd}
X \arrow[dr] \arrow[ddrr, dashed, bend right] & & Y \arrow[dl] \arrow[dr] & & Z \arrow[dl] \arrow[ddll, dashed, bend left] \\
 & A \arrow[dr] & & B \arrow[dl] & \\
 & & P & &
\end{tikzcd}$$

- the identity morphism on an object $\boxed{X}$ is given by the cospan

$$X \xrightarrow{\mathrm{id}_X} X \xleftarrow{\mathrm{id}_X} X.$$

Note that the choice of the pushout in the composition does not matter: Any two pushouts are isomorphic and the isomorphism between two pushouts makes the corresponding morphisms equal.

One can check that this indeed defines a category by showing that various diagrams are pushout diagrams.

Construction 2.10.2 (Monoidal structures of categories of cospans)**.** Let $\mathbf{C}$ be a monoidal category with binary pushouts. Then the category of cospans of $\mathbf{C}$ inherits a monoidal structure as follows:

The tensor product on objects and the tensor unit are simply the tensor product on objects and the tensor unit in $\mathbf{C}$. The tensor product of two morphisms $\boxed{X \to A \leftarrow Y}$ and $\boxed{Z \to B \leftarrow W}$ uses the tensor product on morphisms in $\mathbf{C}$ twice to construct

$$\boxed{X \otimes Z \longrightarrow A \otimes B \longleftarrow Y \otimes W}.$$

The associators and unitors are obtained by taking the associators and unitors in $\mathbf{C}$ together with an identity:

$$\begin{aligned}
\alpha_{\boxed{X},\boxed{Y},\boxed{Z}} &:= \boxed{(X \otimes Y) \otimes Z \xrightarrow{\alpha_{X,Y,Z}} X \otimes (Y \otimes Z) \xleftarrow{\mathrm{id}_{X \otimes (Y \otimes Z)}} X \otimes (Y \otimes Z)}, \\
\lambda_{\boxed{X}} &:= \boxed{1 \otimes X \xrightarrow{\lambda_X} X \xleftarrow{\mathrm{id}_X} X}, \\
\rho_{\boxed{X}} &:= \boxed{X \otimes 1 \xrightarrow{\rho_X} X \xleftarrow{\mathrm{id}_X} X}.
\end{aligned}$$

Furthermore, if **C** is a braided or symmetric category, then the category of cospans of **C** is a braided or symmetric category, too, with braidings given by

$$\gamma_{\boxed{X},\boxed{Y}} \coloneqq \boxed{X \otimes Y \xrightarrow{\gamma_{X,Y}} Y \otimes X \xleftarrow{\mathrm{id}_{Y \otimes X}} Y \otimes X}.$$

Finally, if **C** is a cocartesian monoidal category, then the category of cospans of **C** is a rigid symmetric monoidal category with self-dual objects and evaluation and coevaluation given as

$$\varepsilon_{\boxed{X}} \coloneqq \boxed{X \otimes X \xrightarrow{\nabla} X \leftarrow 1},$$

$$\eta_{\boxed{X}} \coloneqq \boxed{1 \rightarrow X \xleftarrow{\nabla} X \otimes X},$$

where $\nabla : X \otimes X \equiv X \sqcup X \to X$ is the codiagonal morphism and $1 \to X$ is the unique morphism from the initial object.

Proof of correctness. A proof of the correctness can be found in [Cic18, Theorem A.6, Theorem A.8]. ■

2.11 Closed monoidal categories as typed generalized lambda calculi

There are many tight and rich connections between categories with certain structures, type theories, lambda calculi, and programming languages (see, for example, [Tur37], [See89], and [LS88]). For example, one can associate to categories an **internal logic**, a type theory which forms a formal **syntactic** language for the category. Conversely, one can endow type theories with **semantics** in categories.

In this section, we will see a concrete instance of such a connection: We will define a type of lambda calculus which, as we will see, corresponds to closed monoidal categories. This is similar to how certain typed lambda calculi correspond to cartesian closed categories in [LS88], and we make no claim of originality. In our case, closed monoidal categories in turn correspond to flavors of so called **linear type theories** [See89], which can be used to describe the logic of systems in quantum physics [Pra92]. Indeed, in Section 5.4 we will see a closed monoidal category arising in a quantum context. This category defines a lambda calculus which we can use as the foundation for a quantum programming language in Section 5.5.

We first clarify some terminology regarding the term “programming language”:

Remark 2.11.1 (Terminology)**.** Generally speaking, a **programming language** is a language for describing computations for execution on a computer. In a strict sense, such a language is a **formal language** given by a specification for the terms of the language. In this thesis, we use the term “programming language” in a more colloquial sense, namely for any notation which allows humans to specify computations for execution on a computer. Nevertheless, we still introduce a formal foundation by the notion of **typed generalized lambda calculi** in the next definition.

We now introduce **typed generalized lambda calculi** and show their connection to programming languages. We start with a definition without types and later add types in Definition 2.11.7. The following definition will contain some informal parts. We will give a proper formal categorical definition in Definition 2.11.11.

Definition 2.11.2 (Generalized lambda calculi)**.** A **generalized lambda calculus** is a formalism of so-called **lambda terms** built from a collection C of **constants**. Every lambda term is built in a **context**. A context is just a tuple $(x_1, \ldots, x_n)$ of symbols x_i which we call **free variables**. We introduce the following formation rules for building lambda terms in contexts:

1. in the empty context $()$, we have the lambda term c for every constant c,
2. in a context (x) with a single variable x, we have the lambda term x,
3. if we have a lambda term M in a context $(x_1, \ldots, x_n, x)$, we have the lambda term $\lambda x.M$ (an **abstraction**) in the context $(x_1, \ldots, x_n)$,
4. if we have a lambda term F in a context $(x_1, \ldots, x_n)$ and a lambda term N in a context $(y_1, \ldots, y_m)$, we have the lambda term FN (an **application**) in the context $(x_1, \ldots, x_n, y_1, \ldots, y_m)$.

Furthermore, we introduce the **substitution** of free variables: Let M and N be lambda terms in contexts $(x_1, \ldots, x_n)$ and $(y_1, \ldots, y_m)$, respectively. For $i \in \{1, \ldots, n\}$, we informally define the lambda term

$$M[x_i \coloneqq N] \quad \text{in the context} \quad (x_1, \ldots, x_{i-1}, y_1, \ldots, y_m, x_{i+1}, \ldots, x_n)$$

as follows: To form $M[x_i \coloneqq N]$, we repeat the construction of M, but use N instead of x_i. Of course, we require substitution to be compatible with all constructions. For example, for a lambda term F in a context $(x_1, \ldots, x_n)$ and two lambda terms N and M, we require that

$$(FN)[x_i \coloneqq M] = (F[x_i \coloneqq M])N.$$

We will properly formalize substitution in Definition 2.11.11.

Moreover, we introduce the following reduction operations:

1. β**-reduction** defines function evaluation: If M is a lambda term in the context $(x_1, \ldots, x_n, x)$, the term $(\lambda x.M)N$ can be reduced to $M[x \coloneqq N]$,
2. η**-reduction** introduces function extensionality: the term $\lambda x.Fx$ reduces to F, that is, functions are fully determined by their values.

We require an equality on lambda terms which is compatible with these reduction operations.

Remark 2.11.3 (Functions, values, and constants)**.** Note that a generalized lambda calculus does not distinguish between "functions" and "values". We have used the suggestive letter F when defining the application, but in fact every lambda term can be applied to every other lambda term. This flexibility is not always desired. To distinguish between functions and non-functions, we will later introduce **types**, and only elements of a function type will be applicable to other values. Nevertheless, we will still be able to use functions as input or output values of other functions.

Additionally, the term “constant” should neither be interpreted as “not being a function” nor as “being a constant function”. Instead, the term “constant” refers to the fact that constants cannot be used as parameters in function abstraction.

Remark 2.11.4 (Functional programming languages)**.** In a strict sense, a **functional programming language** is a programming language in which programs can be written solely by defining and applying pure functions. Moreover, in such a language there is no distinction between functions and values, that is, functions can be inputs and outputs of other functions, which are then called **higher-order functions**. Of course, a **practical** functional programming language usually provides syntactic sugar[5] to express some things more conveniently in a way which is not immediately recognizable as a function definition or application.

Example 2.11.5 (Connection between generalized lambda calculi and functional programming languages)**.** The central piece of any functional programming language is the construction and application of functions, and evaluating a program typically follows rules similar to β-reduction and η-reduction. Hence, a functional programming language typically embeds into a generalized lambda calculus. For example, in a functional subset of `GAP` we can do the following:

- We can take the integers together with the successor function `+ 1` as constants.
- We can form the expression `x`, where `x` is a free variable.
- We can apply the successor function to `x` to form `x + 1`.
- We can form the abstraction `x -> x + 1`, which is now a valid standalone `GAP` expression.
- We can form the application `(x -> x + 1)(1)`.

To evaluate an expression like `(x -> x + 1)(1)`, `GAP` replaces all occurrences of the function variable `x` in the function body `x + 1` by the argument `1` and evaluates the resulting expression. This corresponds to β-reduction. Moreover, in a functional subset of `GAP`, we can freely replace a function `F` by `x -> F(x)` and vice versa without changing the results of computations. This corresponds to η-reduction. Hence, if we view two expressions as equal if they evaluate to the same result, this functional subset of `GAP` embeds into a generalized lambda calculus.

Conversely, we can view β-reduction and η-reduction as computational rules for “evaluating” lambda terms. Hence, a generalized lambda calculus can be seen as a programming language.

In fact, the original lambda calculus (see Remark 2.11.6) is **Turing complete** as shown by Alan Turing in [Tur37]. That is, its computational power is equivalent to the computational power of a **Turing machine**, which, according to the **Church-Turing thesis**, in turn matches what humans intuitively regard as “computable” and which is typically computable in a programming language.

Remark 2.11.6. Lambda calculi are usually defined with less restrictions than the ones imposed above and hence typically include additional formation rules. For example, usually

[5] The term “syntactic sugar” describes syntactic constructs which do not introduce new functionality to a programming language but allow to express existing functionality in a more convenient way.

- one can form the lambda term x not only in the context (x) but in any context $(x_1, \ldots, x_n)$ with $x = x_i$ for some i, and
- the contexts of F and N in the fourth rule may coincide.

The original lambda calculus as introduced by Alonzo Church in [Chu32] is a free instance of a lambda calculus allowing these additional formation rules. We will see in Remark 5.5.2 why the additional formation rules do not appear in our application in Chapter 5. This is why we do not include these formation rules here and why we call a calculus as defined above a **generalized** lambda calculus.

We now introduce **types**.

Definition 2.11.7 (**Typed** generalized lambda calculi)**.** The basic idea of a **typed** generalized lambda calculus is to assign **types** to (some) lambda terms of a generalized lambda calculus.

To make this precise, we start with a generalized lambda calculus and a collection of **types** such that for each two types T_1 and T_2 there is a type $T_1 \Rightarrow T_2$ called the **function type from T_1 to T_2**.[6]

To assign types to lambda terms, first every constant in the collection C and every variable in a context $(x_1, \ldots, x_n)$ are given a type. We write $(x_1 : T_1, \ldots, x_n : T_n)$ for a context where the variable x_i is of type T_i. Then, for each formation rule of lambda terms we introduce a corresponding typing rule:

1. if a constant c is of type T, then so is the lambda term c,
2. in a context $(x : T)$ with a single variable x of type T, the lambda term x is of type T,
3. if a lambda term M if of type T_2 in a context $(\ldots, x : T_1)$, then the lambda term $\lambda x.M$ is of type $T_1 \Rightarrow T_2$,
4. if a lambda term F is of type $T_1 \Rightarrow T_2$ and a lambda term N is of type T_1, then the lambda term FN is of type T_2.

Furthermore, substitution does not change the type.

If we can derive a type of a lambda term M in a given context using these typing rules, we call M **typeable**. Similar to the situation of objects of categories, we need a meta-theoretical equality of types if we want to decide whether the fourth rule is applicable, that is, whether the type of F can be written as a function type from the type of N to some other type.

In a typed generalized lambda calculus, we

- only consider typeable lambda terms,
- only allow the substitution $M[x_i := N]$ of a free variable x_i by a lambda term N in case that x_i and N are of the same type,
- require the equality on lambda terms to preserve types.

[6] Usually, the function type is denoted by $T_1 \to T_2$. Here, we use the alternative notation $T_1 \Rightarrow T_2$ to avoid confusion with morphisms. Additionally, the internal hom of two objects A and B is sometimes denoted by $A \Rightarrow B$ and we will later indeed use internal homs as function types.

Note that β-reduction and η-reduction preserve the type. Hence, the equality on lambda terms can actually be compatible with β-reduction and η-reduction and the types.

Finally, we introduce the following notation: Given some lambda term M in a context, we write $M : T$ to express that M is of type T.

Example 2.11.8 (Connection to typed functional programming languages)**.** In Example 2.11.5, we have seen how generalized lambda calculi are connected to functional programming languages. This connection also extends to types: The types in a typed generalized lambda calculus correspond to data types in a functional programming language, which restrict to which values a function can be applied. To see this, we repeat Example 2.11.5 in a functional subset of `Julia`:

- Integers in `Julia` are of type `Integer`, and we would like to view the successor function `+ 1` as a function on integers.
- We can form the expression `x`, and view it as a free variable of type `Integer`.
- We can apply the successor function to `x` to form `x + 1`, which according to our typing rules must be of type `Integer`.
- When forming the abstraction, we can now include a type annotation: `x::Integer -> x + 1`. This type annotation tells `Julia` that we want to restrict to the case where the argument is of type `Integer`.
- When forming the application, we can again include a type annotation: `(x::Integer -> x + 1)(1)::Integer`. With this, `Julia` will check that the computed value indeed is an integer as expected by the typing rules. We have built this expression using the typing rules, so it is typeable and the type check will succeed.
- We could instead apply the function to the floating point number `1.0` and would get `(x::Integer -> x + 1)(1.0)::Integer`. This expression is not typeable according to our rules, and indeed we get an error when executing this expression in `Julia`.

Remark 2.11.9 (Termination of typed programs)**.** In this remark, we would like to explain why in type systems of typical programming languages all fully typed programs terminate. This means that a Turing complete programming language, in which non-terminating programs must exist, must permit some untyped expressions. This might be a surprising claim, given that many programming languages seem to require every expression to be typed. To explain how untyped expressions can hide in plain sight, consider the following typed `Julia` function `f`:

```
function f(x::Integer)::Integer
    if x == 0
        return 0
    else
        return f(x - 1)
    end
end
```

For non-negative integers, the function indeed returns an integer as the type annotation suggests. For negative integers, however, the function does not terminate.[7] We have specified types for all variables and return types, so `Julia` is certainly able to automatically type every subexpression. However, this does not mean that this function is fully typed, contrary to what one might expect. Instead, we are relying on a subtle feature built into many programming languages, which hides a missing type: The function `f` is not defined before we have constructed its body, yet we can already reference `f` in the body. So during construction of the body, `f` is just a placeholder. We can make this explicit by using the following function definition:

```
1  function f(g, x::Integer)::Integer
2      if x == 0
3          return 0
4      else
5          return g(g, x - 1)
6      end
7  end
```

Now, the placeholder in the body is a proper variable `g` which we have to provide manually when calling `f`. We do this by passing `f` to itself, for example as in

`f(f, -1).`

Now we see that there was an untyped variable `g` hidden in the original definition, and this variable is a function which is applied to itself (and to an integer) in line 5. Typing a function which should be applied to itself requires a function type $T_1 \Rightarrow T_2$ which is equal to the source type T_1. A typical type system does not include such a type because it would usually make type checking undecidable due to the infinite expansion

$$T_1 \Rightarrow T_2 \quad \rightsquigarrow \quad (T_1 \Rightarrow T_2) \Rightarrow T_2 \quad \rightsquigarrow \quad ((T_1 \Rightarrow T_2) \Rightarrow T_2) \Rightarrow T_2 \quad \rightsquigarrow \quad \ldots$$

Of course we can easily find examples where type checking is still decidable, for example in a type system with only a single type T, which is equal to all function types on itself. However, in a type system with only a single type, every term is typeable, so practically this is the same as having no type system at all, which makes this type system not particularly useful.

Summing up, in typical type systems all fully typed programs terminate, though the fact that a program is not fully typed is often hidden by the programming language.

In particular, if we view a typed generalized lambda calculus as a foundational functional programming language, all terms in this programming language will be fully typed, so this programming language will typically not be Turing complete.

We can now endow typed generalized lambda calculi with categorical semantics:

Remark 2.11.10 (Translation to category theory)**.** We would like to categorify the definition of typed generalized lambda calculi using strict monoidal categories. We do this using the following dictionary:

[7] More precisely, it throws a `StackOverflowError` because the recursion depth in `Julia` is limited.

non-categorical definition	**categorical definition**
types	objects
lambda terms of type T in a context $(x_1 : T_1, \dots, x_n : T_n)$	morphisms $T_1 \otimes \cdots \otimes T_n \to T$
constants of type T in the empty context	morphisms $1 \to T$ from the tensor unit
the free variable x in the context $(x : T)$	the identity morphism $\mathrm{id}_T : T \to T$
substitution of a free variable of type T	composition with a morphism into T
meta-theoretical equality on types	meta-theoretical equality on objects
equality on lambda terms	mathematical equality on morphisms

Using this translation, we now define **categories of lambda terms**:

Definition 2.11.11 (Categories of lambda terms)**.** A **category of lambda terms** is a strict monoidal category with the following additional structure:

- for every two objects T_1 and T_2 another object $T_1 \Rightarrow T_2$,
- for every morphism $M : C \otimes T_1 \to T_2$ a morphism

$$\lambda_{T_1} M : C \to (T_1 \Rightarrow T_2) \qquad (\textbf{abstraction})$$

- for every morphism $F : C_1 \to (T_1 \Rightarrow T_2)$ and every morphism $N : C_2 \to T_1$ a morphism

$$FN : C_1 \otimes C_2 \to T_2 \qquad (\textbf{application})$$

We require abstraction and application to be compatible with the equality on morphisms, and formalize substitution of free variables as follows: Given two morphisms

$$M : T_1 \otimes \cdots \otimes T_n \to T \qquad \text{and} \qquad N : C \to T_i,$$

we define

$$M[x_{T_i} \coloneqq N] : T_1 \otimes \cdots \otimes T_{i-1} \otimes C \otimes T_{i+1} \otimes \cdots \otimes T_n \to T$$

by

$$M[x_{T_i} \coloneqq N] \quad \coloneqq \quad (\mathrm{id}_{T_1 \otimes \cdots \otimes T_{i-1}} \otimes N \otimes \mathrm{id}_{T_{i+1} \otimes \cdots \otimes T_n}) \cdot M.$$

For a morphism $F : T_1 \otimes \cdots \otimes T_n \to (S \Rightarrow T)$, a morphism $N : C_1 \to S$, and a morphism $M : C_2 \to T_i$, we require the compatibility condition

$$(FN)[x_{T_i} \coloneqq M] = (F[x_{T_i} \coloneqq M])N.$$

Finally, for $M : C_1 \otimes T_1 \to T_2$ and $N : C_2 \to T_1$ we require

$$(\lambda_{T_1} M)N = M[x_{T_1} \coloneqq N] \qquad (\beta\text{-reduction})$$

and for $F : C \to (T_1 \Rightarrow T_2)$ and $x \coloneqq \mathrm{id}_{T_1}$ we require

$$\lambda_{T_1}(Fx) = F \qquad (\eta\text{-reduction})$$

We now show that the additional structure of categories of lambda terms beyond the strict monoidal structure is nothing else but a closed structure. This is similar to how certain typed lambda calculi correspond to cartesian closed categories in [LS88].

Construction 2.11.12. Categories of lambda terms coincide with strict closed monoidal categories:

If $\mathbf{C}$ is a category of lambda terms, then by definition it is a strict monoidal category, and we construct a closed structure as follows: For two objects B and C define

- $\hom(B,C) \coloneqq (B \Rightarrow C)$ and
- $\mathrm{ev}_{B,C} \coloneqq FN$ for $F \coloneqq \mathrm{id}_{\hom(B,C)}$ and $N \coloneqq \mathrm{id}_B$.

For a morphism $f : A \otimes B \to C$, define the corresponding morphism

$$g : A \to \hom(B,C)$$

via

$$g \coloneqq \lambda_B f : A \to \hom(B,C).$$

Conversely, if $\mathbf{C}$ is a strict closed monoidal category, we turn it into a category of lambda terms as follows: For objects T_1 and T_2, we set

$$T_1 \Rightarrow T_2 \quad \coloneqq \quad \hom(T_1, T_2).$$

Let $M : C \otimes T_1 \to T_2$ be a morphism. We define $\lambda_{T_1} M : C \to (T_1 \Rightarrow T_2)$ to be the morphism associated to M by the closed structure. Let $F : C_1 \to (T_1 \Rightarrow T_2)$ and $N : C_2 \to T_1$ be two morphisms. We define

$$FN \coloneqq (F \otimes N) \cdot \mathrm{ev}_{T_1,T_2} : C_1 \otimes C_2 \to T_2.$$

Proof of correctness. This is similar to how certain typed lambda calculi correspond to cartesian closed categories in [LS88], and we make no claim of originality. The proof is elementary but long-winded.

Let $\mathbf{C}$ be a category of lambda terms. By definition, it is a strict monoidal category. For verifying the closed structure, let $f : A \otimes B \to C$ be a morphism of $\mathbf{C}$. We have to show that $g \coloneqq \lambda_B f$ makes the diagram

$$\begin{tikzcd}
A \otimes B \arrow[rr, "f"] \arrow[dr, "g \otimes \mathrm{id}_B"'] & & C \\
& \hom(B,C) \otimes B \arrow[ur, "\mathrm{ev}_{B,C}"'] &
\end{tikzcd}$$

commute and is unique with this property. We first simplify the diagram for an arbitrary g using that substitution can be expressed as composition and is compatible with application:

$$\begin{aligned}
(g \otimes \mathrm{id}_B) \cdot \mathrm{ev}_{B,C} &= (FN)[x_{\hom_{B,C}} \coloneqq g] \\
&= (F[x_{\hom_{B,C}} \coloneqq g])N \\
&= (g \cdot F)N \\
&= (g \cdot \mathrm{id}_{B,C})N \\
&= gN
\end{aligned}$$

For $g = \lambda_B f$, we can simplify this further using β-reduction:

$$\begin{aligned} gN &= (\lambda_B f)N \\ &= f[x_B \coloneqq N] \\ &= (\mathrm{id}_A \otimes N) \cdot f \\ &= (\mathrm{id}_A \otimes \mathrm{id}_B) \cdot f \\ &= f \end{aligned}$$

Hence, the diagram above commutes.

To prove the uniqueness, let $h : A \to \hom(B, C)$ be a morphism such that

$$(g \otimes \mathrm{id}_B) \cdot \mathrm{ev}_{B,C} = (h \otimes \mathrm{id}_B) \cdot \mathrm{ev}_{B,C} .$$

From the general simplification above we get

$$gN = hN.$$

Now, we can apply abstraction with regard to B to both sides of the equation and use η-reduction with $N = \mathrm{id}_B$ to conclude

$$g = \lambda_B(gN) = \lambda_B(hN) = h.$$

Summing up, $\mathbf{C}$ is a strict closed monoidal category.

For the converse, let $\mathbf{C}$ be a strict closed monoidal category. In particular, it is a strict monoidal category. We have to show that

- abstraction and application (as defined in the construction) are compatible with the equality on morphisms,
- application is compatible with substitution, and
- the β- and η-reduction rules hold.

Abstraction and application are compatible with the equality on morphisms because they are defined via categorical constructions which are compatible with the equality on morphisms.

To show compatibility with the substitution, let $F : T_1 \otimes \cdots \otimes T_n \to (S \Rightarrow T)$, $N : C_1 \to S$, and $M : C_2 \to T_i$ be morphisms. Then, using the definitions of the application and the substitution, we have

$$\begin{aligned} (FN)[x_{T_i \coloneqq M}] &= ((F \otimes N) \cdot \mathrm{ev}_{S,T})[x_{T_i \coloneqq M}] \\ &= (\mathrm{id}_{T_1 \otimes \cdots \otimes T_{i-1}} \otimes M \otimes \mathrm{id}_{T_{i+1} \otimes \cdots \otimes T_n \otimes C_1}) \cdot (F \otimes N) \cdot \mathrm{ev}_{S,T} \\ &= (\mathrm{id}_{T_1 \otimes \cdots \otimes T_{i-1}} \otimes M \otimes \mathrm{id}_{T_{i+1} \otimes \cdots \otimes T_n} \otimes \mathrm{id}_{C_1}) \cdot (F \otimes N) \cdot \mathrm{ev}_{S,T} \\ &= (((\mathrm{id}_{T_1 \otimes \cdots \otimes T_{i-1}} \otimes M \otimes \mathrm{id}_{T_{i+1} \otimes \cdots \otimes T_n}) \cdot F) \otimes (\mathrm{id}_{C_1} \cdot N)) \cdot \mathrm{ev}_{S,T} \\ &= (((\mathrm{id}_{T_1 \otimes \cdots \otimes T_{i-1}} \otimes M \otimes \mathrm{id}_{T_{i+1} \otimes \cdots \otimes T_n}) \cdot F) \otimes N) \cdot \mathrm{ev}_{S,T} \\ &= (F[x_{T_i \coloneqq M}] \otimes N) \cdot \mathrm{ev}_{S,T} \\ &= (F[x_{T_i \coloneqq M}])N. \end{aligned}$$

To show that the β-reduction rule holds, let $M : C_1 \otimes T_1 \to T_2$ and $N : C_2 \to T_1$ be morphisms. Using the definition of the application, the functoriality of the tensor product, and the defining property of $\lambda_{T_1} M$, we get

$$\begin{aligned}(\lambda_{T_1} M)N &= ((\lambda_{T_1} M) \otimes N) \cdot \mathrm{ev}_{T_1,T_2} \\ &= (\mathrm{id}_{C_1} \otimes N) \cdot ((\lambda_{T_1} M) \otimes \mathrm{id}_{T_1}) \cdot \mathrm{ev}_{T_1,T_2} \\ &= (\mathrm{id}_{C_1} \otimes N) \cdot M \\ &= M[x_{T_1} \coloneqq N].\end{aligned}$$

To show that the η-reduction rule holds, let $F : C \to (T_1 \Rightarrow T_2)$ be a morphism and set $x \coloneqq \mathrm{id}_{T_1}$. By definition of the application, we have

$$Fx = (F \otimes x) \cdot \mathrm{ev}_{T_1,T_2} = (F \otimes \mathrm{id}_{T_1}) \cdot \mathrm{ev}_{T_1,T_2} \,.$$

By uniqueness, F must be the morphism associated to Fx by the closed structure, so we get

$$\lambda_{T_1}(Fx) = F$$

as desired. ■

Example 2.11.13 (Semantics of typed generalized lambda calculi)**.** We have seen that every strict closed monoidal category **C** defines a category of lambda terms, which corresponds to a typed generalized lambda calculus, which in turn can be interpreted as a foundational functional programming language. Of course the semantics of this programming language depends on **C**.

As a degenerate example, consider the terminal category with a single object and morphism, which trivially is a strict closed monoidal category. The corresponding generalized lambda calculus only has a single type and all lambda terms are equal, so any semantics would be very limited.

As a non-degenerate example, consider the category of sets with a strict version of its cartesian closed structure. In this case, the internal hom can be chosen to coincide with the external hom, which in turn is given by functions on sets. Moreover, the evaluation morphism $\hom(B, C) \otimes B \to C$ actually evaluates functions from B to C at elements of B to get elements of C. Hence, in this case we actually obtain a generalized lambda calculus with the semantics of function definition and application as we know it from sets.

Remark 2.11.14 (Generalized lambda calculi of **cartesian** closed categories, [LS88])**.** Let **C** be a strict cartesian closed category. In particular, **C** is a strict closed monoidal category and hence defines a typed generalized lambda calculus. The cartesian monoidal structure allows for additional formation rules for lambda terms: The projection $T_1 \times \cdots \times T_n \to T_i$ can be used to define a lambda term x_i of type T_i in the context $(x_1 : T_1, \ldots, x_n : T_n)$. Furthermore, if $F : C \to \hom(T_1, T_2)$ and $N : C \to T_1$ are two lambda terms defined in the same context, then the universal morphism into the product

$$(F, N) : C \to \hom(T_1, T_2) \times T_1$$

can be composed with the evaluation morphism

$$\mathrm{ev}_{T_1,T_2} : \hom(T_1, T_2) \times T_1 \to T_2$$

to obtain a lambda term $FN : C \to T_2$ in the same context as F and N. These additional formation rules can be used for defining the original lambda calculus in Remark 2.11.6.

Chapter 3

The concept of categorical towers

The goal of this chapter is to show how one can model categories as **categorical towers**. We will see diverse examples of categorical towers in Chapter 4 and Chapter 5. Our main example in this chapter is the categorical tower $R \mapsto \boldsymbol{\mathcal{C}}(R)^{\oplus}$ of height 2 obtained by composing the category constructor $R \mapsto \boldsymbol{\mathcal{C}}(R)$ for a ring R and the category constructor $\mathbf{C} \mapsto \mathbf{C}^{\oplus}$ for a preadditive category $\mathbf{C}$. In Construction 3.3.2, we will see that we can model categories of matrices $\mathbf{Mat}_R$ over a ring R via the categorical tower $R \mapsto \boldsymbol{\mathcal{C}}(R)^{\oplus}$ by constructing an isomorphism between $\mathbf{Mat}_R$ and $\boldsymbol{\mathcal{C}}(R)^{\oplus}$.

Modeling categories as categorical towers has many advantages: We can **reuse** concepts, have a **separation of concerns** between the different layers of the tower, can break down the **verification** of the tower into smaller pieces, and have a natural **emergence** of structures of the tower from the single layers. We examine those advantages in detail in Remark 3.2.1.

Categorical towers have been used before, for example in [Bar09], [BLH14], [Pos21a], and [Cic18]. However, in those applications the isomorphism relating a category to the categorical tower modeling it did not explicitly appear in algorithms. In this thesis, we introduce the notion of **reinterpretations of categorical towers** along isomorphisms, which makes the isomorphisms an explicit part of algorithms. We will make this precise in Section 3.3. The concept of reinterpretations was developed in the context of `CompilerForCAP` [Zic24a] as the central mechanism for generating efficient primitive implementations from categorical towers in `CAP`. We will see this application in Section 6.5.1. By using the concept of reinterpretations in conjunction with `CompilerForCAP`, we can get all of the advantages of categorical towers not only on a theoretical level but actually for implementations on a computer.

Categorical towers are our own interpretation of the term **compositionality**, which, for example, is used in [FS19] in the context of applied category theory.

The chapter is structured as follows: In Section 3.1, we introduce **category constructors**, which are the building blocks from which we build categorical towers in Section 3.2. In Section 3.3, we introduce **reinterpretations of categorical towers**. Finally, in Section 3.4, we look at an example of a computation in the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$. As we will see, computations in categorical towers naturally come with an overhead, which is the reason why we need a compiler like CompilerForCAP to actually get the advantages of categorical towers on a computer.

3.1 Category constructors

CAP allows users to implement **category constructors**. As the name suggests, a **category constructor** is a function which constructs a category from some input (or sometimes no input at all). There is no restriction on what this input can be. Often it is an algebraic or combinatorial data structure or another category. We have already seen many examples, including the following:

- Example 1.1.4: A function constructing **FinSets** is a category constructor without an input.
- Example 1.1.3: A function constructing $\mathbf{Mat}_R$ of a ring R is a category constructor $R \mapsto \mathbf{Mat}_R$ getting a ring as an input.
- Construction 2.3.7: A function constructing $\boldsymbol{\mathcal{C}}(R)$ of a ring R is a category constructor $\boldsymbol{\mathcal{C}}$ getting a ring as an input.
- Example 1.1.5: A function constructing the opposite category $\mathbf{C}^{\mathrm{op}}$ of a category $\mathbf{C}$ is a category constructor $\mathbf{C} \mapsto \mathbf{C}^{\mathrm{op}}$ getting an arbitrary category as an input.
- Construction 2.4.6: A function constructing the additive closure $\mathbf{C}^{\oplus}$ of a preadditive category $\mathbf{C}$ is a category constructor $\mathbf{C} \mapsto \mathbf{C}^{\oplus}$ getting a preadditive category as an input.
- Definition 2.1.6: A function constructing the functor category from $\mathbf{C}$ to $\mathbf{D}$ is a category constructor $(\mathbf{C}, \mathbf{D}) \mapsto \mathbf{D}^{\mathbf{C}}$ getting two categories as an input.

Moreover, we have seen category constructors for Freyd categories (Definition 2.6.1), slice categories (Definition 2.9.1), and categories of cospans (Definition 2.10.1).

3.2 Categorical towers

If a category constructor gets one or more categories as an input, we can apply it to the output of other category constructors, that is, we can compose category constructors. For example, we can compose $R \mapsto \boldsymbol{\mathcal{C}}(R)$ and $\mathbf{C} \mapsto \mathbf{C}^{\oplus}$ and obtain a category constructor $R \mapsto \boldsymbol{\mathcal{C}}(R)^{\oplus}$ constructing the additive closure of the category defined by the ring R. We call such a category constructor a **tower of category constructors** or **categorical tower**. To simplify the notation, we will also call the categories constructed by towers of category constructors **categorical towers**. For example, we would also call the category $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$ a categorical tower.

We sometimes say that a categorical tower has a certain **height** and talk about its **levels** or **layers**. For example, we could say that $R \mapsto \boldsymbol{\mathcal{C}}(R)^{\oplus}$ is a tower of height 2 with first level $R \mapsto \boldsymbol{\mathcal{C}}(R)$ and second level $\mathbf{C} \mapsto \mathbf{C}^{\oplus}$. We might also view the argument R as level 0 of the tower. However, those numbers do not have an intrinsic mathematical meaning. For example, we will see that

$\mathcal{C}(R)^{\oplus}$ is isomorphic to $\mathbf{Mat}_R$, and we would typically view $R \mapsto \mathbf{Mat}_R$ as a (degenerate) tower of height 1. Still, the height will always be clear from the context in our applications. To single out degenerate towers of height 1, we call them **primitive** category constructors and reserve the word **tower** for proper towers of height at least 2.

Remark 3.2.1 (Advantages of building categorical towers)**.** Each category constructor used for building a categorical tower is self-contained. Hence, the organization of categorical towers introduces a high degree of modularity. This has the following advantages:

- **Reusability**: The category constructors used to build one categorical tower can be reused for building categorical towers in other contexts.
- **Separation of concerns**: Every category constructor in a categorical tower can focus on a single concept, simplifying the definitions.
- **Verifiability**: The category constructors used to build a categorical tower can be verified independently of each other. Each category constructor has a limited scope and is hence relatively simple to comprehend and verify.
- **Emergence**: Simple and natural constructions at each level of the tower can lead to convoluted structures of the tower as a whole.

We will see a concrete example of these advantages in Remark 3.3.3 and many more examples in Chapter 4 and Chapter 5. On a computer, all these advantages lead to higher quality code:

- reusability of code leads to less code overall and better code coverage,
- the separation of concerns makes it possible to focus on one aspect at a time when implementing the various category constructors,
- better verifiability makes it easier to check code for correctness, and
- convoluted algorithms for the tower as a whole emerge from simple and natural algorithms at each level of the tower.

3.3 Reinterpretations of categorical towers

Reinterpretations of categorical towers allow to formally view categorical towers as primitive category constructors with possibly simplified data structures for objects and morphisms. The concept was developed in the context of `CompilerForCAP` as the central mechanism for generating efficient primitive implementations from categorical towers in `CAP`. We will see this application in Section 6.5.1.

Definition 3.3.1 (Reinterpretations of categorical towers)**.** Let $\mathcal{T}$ and $\mathcal{C}$ be category constructors with the same input specification, where $\mathcal{T}$ is a categorical tower and $\mathcal{C}$ is a primitive category constructor. If there exists an isomorphism[1] $\mathcal{R}_I : \mathcal{T}(I) \to \mathcal{C}(I)$ for all inputs I, we call

- $\mathcal{C}$ a **reinterpretation** of the categorical tower $\mathcal{T}$ via the functor $\mathcal{R}$, and
- $\mathcal{T}$ a **model** of $\mathcal{C}$.

[1] We will see in Remark 3.3.6 why a mere equivalence does not suffice for our applications.

We usually write $\mathcal{M}_I : \mathcal{C}(I) \to \mathcal{T}(I)$ for the inverse of $\mathcal{R}_I$. In contexts with additional structures like preadditive or monoidal structures, we assume that $\mathcal{R}$ and $\mathcal{M}$ are compatible with the additional structures.

As an example, we show how categories of matrices can be modeled as a reinterpretation of a categorical tower:

Construction 3.3.2 (Categories of matrices as categorical towers, [Pos21b, Example 1.17])**.** Let R be a ring and let $\star$ be the unique object of $\boldsymbol{\mathcal{C}}(R)$. We can model $\mathbf{Mat}_R$ as a reinterpretation of the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ via the functor $\mathcal{R}$ defined as

$$\begin{aligned} \mathcal{R} : \mathcal{C}(R)^{\oplus} &\to \mathbf{Mat}_R \\ \underbrace{\boxed{(\star,\dots,\star)}}_{m}{}_{\mathcal{C}(R)^{\oplus}} &\mapsto \boxed{m}_{\mathbf{Mat}_R} \\ \boxed{\left(\boxed{m_{ij}}_{\mathcal{C}(R)}\right)_{ij}}_{\mathcal{C}(R)^{\oplus}} &\mapsto \boxed{\left(m_{ij}\right)_{ij}}_{\mathbf{Mat}_R} \end{aligned}$$

together with its inverse $\mathcal{M}$ defined as

$$\begin{aligned} \mathcal{M} : \mathbf{Mat}_R &\to \mathcal{C}(R)^{\oplus} \\ \boxed{m}_{\mathbf{Mat}_R} &\mapsto \underbrace{\boxed{(\star,\dots,\star)}}_{m}{}_{\mathcal{C}(R)^{\oplus}} \\ \boxed{\left(m_{ij}\right)_{ij}}_{\mathbf{Mat}_R} &\mapsto \boxed{\left(\boxed{m_{ij}}_{\mathcal{C}(R)}\right)_{ij}}_{\mathcal{C}(R)^{\oplus}} \end{aligned}$$

Note that $\mathcal{R}$ simplifies the data structure of objects: Objects in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ are given by tuples of objects of $\boldsymbol{\mathcal{C}}(R)$, that is, tuples containing $\star$, while the objects in $\mathbf{Mat}_R$ are just given by integers. Moreover, $\mathcal{R}$ eliminates the inner boxes of morphisms. On paper, where we only view boxes as a notation, this just simplifies the notation. However, on the computer, where boxes are an explicit part of the data structures, this actually simplifies the data structure of morphisms.

A primitive implementation of $\mathbf{Mat}_R$ generated from $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ and optimized by `CompilerForCAP` is available via the category constructor `CategoryOfRows`[2] in the package `FreydCategoriesForCAP` [BPZ24].

Proof of correctness. The equalities on morphisms in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ and $\mathbf{Mat}_R$ are both given entrywise by the equality of ring elements, so $\mathcal{R}$ and $\mathcal{M}$ respect the equality on morphisms. Moreover, $\mathcal{R}$ and $\mathcal{M}$ are mutually inverse both on objects and morphisms. Composition and identities in the additive closure are just given by generalized matrix multiplication and identity matrices, which in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ specialize to the usual matrix multiplication and identity matrices over rings, which in turn defines composition and identities in $\mathbf{Mat}_R$.

Summing up, $\mathcal{R}$ indeed defines an isomorphism of categories with inverse $\mathcal{M}$, as required for a reinterpretation. ■

Remark 3.3.3 (Advantages of modeling categories of matrices as categorical towers)**.** Modeling $\mathbf{Mat}_R$ as the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ exhibits all advantages mentioned in Remark 3.2.1:

[2] The term **category of rows** is a synonym of **category of matrices**.

- **Reusability**: The constructor for additive closures can be reused in other contexts, for example for constructing free abelian categories in [Pos22].
- **Separation of concerns**: The ring algorithms (for example, multiplication of ring elements) are strictly separate from the high-level rules of matrix calculus.
- **Verifiability**: Verifying the construction of $\mathcal{C}(R)$ and the construction of the additive closure is straightforward. We will use `CompilerForCAP` to prove the correctness of both constructions in Section 7.1 and Section 7.2.
- **Emergence**: In Section 4.1, we will see how the natural constructions of homomorphism structures of $\mathcal{C}(R)$ and additive closures give a convoluted homomorphism structure of $\mathbf{Mat}_R$.

To take full advantage of categorical towers, we would actually like to **define** primitive category constructors as reinterpretations of categorical towers. For this, we note that in the context of a reinterpretation, the categorical structure of the primitive category is already uniquely determined by the categorical structure of the categorical tower:

Remark 3.3.4. Let $\mathcal{T}$ and $\mathcal{C}$ be category constructors as in Definition 3.3.1 and let I be some input. Set $\mathbf{T} \coloneqq \mathcal{T}(I)$, $\mathbf{C} \coloneqq \mathcal{C}(I)$, $\mathcal{R} \coloneqq \mathcal{R}_I$, and $\mathcal{M} \coloneqq \mathcal{M}_I$. For morphisms f and g in $\mathbf{C}$, we have

$$f \cdot g = \mathcal{R}(\mathcal{M}(f \cdot g)) = \mathcal{R}(\mathcal{M}(f) \cdot \mathcal{M}(g)). \tag{3.1}$$

Moreover, for an object A in $\mathbf{C}$, we have

$$\mathrm{id}_A = \mathcal{R}(\mathcal{M}(\mathrm{id}_A)) = \mathcal{R}(\mathrm{id}_{\mathcal{M}(A)}). \tag{3.2}$$

This shows that the composition and the identity morphisms of $\mathbf{C}$ are uniquely determined by the composition and the identity morphisms of $\mathbf{T}$ together with the values of $\mathcal{R}$ and $\mathcal{M}$ on objects and morphisms.

In contexts with additional structures like preadditive or monoidal structures, we have assumed that $\mathcal{R}$ and $\mathcal{M}$ are compatible with the additional structures. Hence, also the additional structures are uniquely determined by the structures of $\mathbf{T}$ together with the values of $\mathcal{R}$ and $\mathcal{M}$ on objects and morphisms.

This motivates the following construction:

Construction 3.3.5 (Defining categories via reinterpretations)**.** Let $\mathcal{T}$ and $\mathcal{C}$ be category constructors as in Definition 3.3.1 and let I be some input. Set $\mathbf{T} \coloneqq \mathcal{T}(I)$, $\mathbf{C} \coloneqq \mathcal{C}(I)$, $\mathcal{R} \coloneqq \mathcal{R}_I$, and $\mathcal{M} \coloneqq \mathcal{M}_I$. We ignore the existing categorical structure of $\mathbf{C}$, that is, composition and identity morphisms, as well as the fact that $\mathcal{R}$ and $\mathcal{M}$ are compatible with the composition and the identities. Instead, we define a categorical structure on $\mathbf{C}$ via

$$f \cdot g \coloneqq \mathcal{R}(\mathcal{M}(f) \cdot \mathcal{M}(g)) \qquad \text{and} \qquad \mathrm{id}_A \coloneqq \mathcal{R}(\mathrm{id}_{\mathcal{M}(A)}).$$

With this definition, $\mathcal{R}$ and $\mathcal{M}$ are automatically compatible with the composition and the identities, and thus isomorphisms of categories again. Hence, $\mathbf{C}$ is again a reinterpretation of $\mathbf{T}$ via $\mathcal{R}$ and $\mathcal{M}$.

Additional structures like preadditive or monoidal structures are defined in a similar way via $\mathcal{R}$, $\mathcal{M}$, and the structure of $\mathbf{T}$. Again, $\mathcal{R}$ and $\mathcal{M}$ are automatically compatible with the additional structures.

Proof of correctness. First note that for $f : A \to B$ and $g : B \to C$, the composite $\mathcal{R}(\mathcal{M}(f)\cdot\mathcal{M}(g))$ is a morphism from $\mathcal{R}(\mathcal{M}(A)) \equiv A$ to $\mathcal{R}(\mathcal{M}(C)) \equiv C$ as required. Similarly, id_A is a morphism from A to A as required. Equations (3.1) and (3.2) allow to check the axioms required for the newly defined categorical structure on $\mathbf{C}$ by the corresponding axioms of $\mathbf{T}$. Moreover, the two equations are equivalent to $\mathcal{R}$ and $\mathcal{M}$ being compatible with the composition and preserving identities. Additional structures like preadditive or monoidal structures are handled analogously. ■

This constructions allows us to **define** primitive category constructors as reinterpretations of categorical towers. Remark 3.3.4 ensures that this construction is compatible with any existing categorical structure.

Remark 3.3.6. Construction 3.3.5 would not work if $\mathcal{R}$ and $\mathcal{M}$ would not be isomorphisms but merely equivalences of categories: In this case, setting

$$f \cdot g := \mathcal{R}(\mathcal{M}(f) \cdot \mathcal{M}(g)) \qquad \text{and} \qquad \mathrm{id}_A := \mathcal{R}(\mathrm{id}_{\mathcal{M}(A)})$$

would not define morphisms with the required source and targets because in general

$$\mathcal{R}(\mathcal{M}(A)) \not\equiv A$$

if $\mathcal{R}$ and $\mathcal{M}$ are mere equivalences.

Let us try to properly account for the natural isomorphisms coming with equivalences: Let $\eta : \mathcal{M} \cdot \mathcal{R} \Rightarrow \mathrm{Id}_{\mathbf{C}}$ be the natural isomorphism in $\mathbf{C}$. We consider an equation analogous to equation (3.1): For morphisms $f : A \to B$ and $g : B \to C$ in $\mathbf{C}$, we have

$$f \cdot g = \eta_A^{-1} \cdot \mathcal{R}(\mathcal{M}(f \cdot g)) \cdot \eta_C = \eta_A^{-1} \cdot \mathcal{R}(\mathcal{M}(f) \cdot \mathcal{M}(g)) \cdot \eta_C.$$

Note that the right-hand side of this equations still uses the composition in $\mathbf{C}$. Hence, this equation cannot be used for defining a composition in $\mathbf{C}$. Thus, we cannot proceed if $\mathcal{R}$ and $\mathcal{M}$ are mere equivalences.

As an example for Construction 3.3.5, we show how to construct direct sums in $\mathbf{Mat}_R$ from the direct sums in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$.

Construction 3.3.7 (Direct sums in categories of matrices)**.** Let R be a ring and let $\boxed{m}$ and $\boxed{n}$ be two objects of $\mathbf{Mat}_R$. We construct a direct sum $\boxed{m} \oplus \boxed{n}$ together with the projections and injections as follows:

- We set $\boxed{m} \oplus \boxed{n} :\equiv \boxed{m+n}$.
- The projections $\boxed{P_1} : \boxed{m} \oplus \boxed{n} \to \boxed{m}$ and $\boxed{P_2} : \boxed{m} \oplus \boxed{n} \to \boxed{n}$ are given by stacking identity matrices and zero matrices as follows:

$$P_1 := \begin{pmatrix} I_m \\ 0_{n,m} \end{pmatrix} \quad \text{and} \quad P_2 := \begin{pmatrix} 0_{m,n} \\ I_n \end{pmatrix}.$$

- Similarly, the injections $\boxed{J_1} : \boxed{m} \to \boxed{m} \oplus \boxed{n}$ and $\boxed{J_2} : \boxed{n} \to \boxed{m} \oplus \boxed{n}$ are given by

$$J_1 := \begin{pmatrix} I_m & 0_{m,n} \end{pmatrix} \quad \text{and} \quad J_2 := \begin{pmatrix} 0_{n,m} \\ I_n \end{pmatrix}.$$

Now, if we have an object $\boxed{t}$ with morphisms

$$\boxed{T_1} : \boxed{t} \to \boxed{m} \quad \text{and} \quad \boxed{T_2} : \boxed{t} \to \boxed{n},$$

we can choose $\boxed{U} = (\boxed{T_1}, \boxed{T_2})^{\oplus} : \boxed{t} \to \boxed{m} \oplus \boxed{n}$ as the union of columns of the matrices T_1 and T_2, that is,

$$U \coloneqq \begin{pmatrix} T_1 & T_2 \end{pmatrix}.$$

Similarly, for two morphisms

$$\boxed{S_1} : \boxed{m} \to \boxed{t} \quad \text{and} \quad \boxed{S_2} : \boxed{n} \to \boxed{t}$$

we can choose $\boxed{U} = \langle \boxed{S_1}, \boxed{S_2} \rangle^{\oplus} : \boxed{m} \oplus \boxed{n} \to \boxed{t}$ as

$$U \coloneqq \begin{pmatrix} T_1 \\ T_2 \end{pmatrix}.$$

Proof of correctness. Let R be a ring. The tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ has direct sums by Construction 2.4.9. Using Construction 3.3.2 and Construction 3.3.5, we get direct sums for $\mathbf{Mat}_R$ as follows:

Let $\boxed{m}$ and $\boxed{n}$ be two objects of $\mathbf{Mat}_R$. By Construction 3.3.5, we have

$$\boxed{m} \oplus \boxed{n} :\equiv \mathcal{R}(\mathcal{M}(\boxed{m}) \oplus \mathcal{M}(\boxed{n})).$$

Let us expand the definitions of $\mathcal{M}$ and of the direct sum in the additive closure:

$$\begin{aligned} \mathcal{R}\Big(\mathcal{M}(\boxed{m}) \oplus \mathcal{M}(\boxed{n})\Big) &\equiv \mathcal{R}\Big(\underbrace{\boxed{(\star, \dots, \star)}}_{m} \oplus \underbrace{\boxed{(\star, \dots, \star)}}_{n}\Big) \\ &\equiv \mathcal{R}\Big(\boxed{(\underbrace{\star, \dots, \star,}_{m} \underbrace{\star, \dots, \star}_{n})}\Big). \end{aligned}$$

We can now turn an algorithm on tuples (the concatenation of two tuples) into an algorithm on integers (the sum of two integers):

$$\mathcal{R}\Big(\boxed{(\underbrace{\star, \dots, \star,}_{m} \underbrace{\star, \dots, \star}_{n})}\Big) \equiv \mathcal{R}\Big(\underbrace{\boxed{(\star, \dots, \star)}}_{m+n}\Big).$$

Now the last term is simply equal to $\boxed{m+n}$. Summing up, we do indeed get

$$\boxed{m} \oplus \boxed{n} \equiv \boxed{m+n}.$$

Note that the steps above can be seen as a manual compilation of the expression

$$\mathcal{R}(\mathcal{M}(\boxed{m}) \oplus \mathcal{M}(\boxed{n})).$$

In Chapter 6 and in particular in Section 6.3.3, we will see how this compilation can be automated using `CompilerForCAP`.

The constructions of the projections, injections, and universal morphisms can be obtained from $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ in a similar way. ■

3.4 Computations in categorical towers

In this section, we look at an example of a computation in a categorical tower. We will see that such computations come with an overhead, which is the reason why we need a compiler like `CompilerForCAP` to actually get the advantages of categorical towers on a computer.

Let us compare a computation in the primitive category $\mathbf{Mat}_{\mathbb{Z}}$ to a corresponding computation in the categorical tower $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$. For this, consider three parallel morphisms in $\mathbf{Mat}_{\mathbb{Z}}$:

$$\boxed{(x_{ij})_{ij}}, \boxed{(y_{ij})_{ij}}, \boxed{(z_{ij})_{ij}} : \boxed{m} \to \boxed{n}.$$

The sum of the three morphisms can be computed as follows:

$$\begin{aligned}\Big(\boxed{(x_{ij})_{ij}} + \boxed{(y_{ij})_{ij}}\Big) + \boxed{(z_{ij})_{ij}} &= \boxed{(x_{ij} + y_{ij})_{ij}} + \boxed{(z_{ij})_{ij}} \\ &= \boxed{\big((x_{ij} + y_{ij}) + z_{ij}\big)_{ij}}.\end{aligned}$$

Analogously, consider three parallel morphisms in $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$:

$$\boxed{\big(\boxed{x_{ij}}\big)_{ij}}, \boxed{\big(\boxed{y_{ij}}\big)_{ij}}), \boxed{\big(\boxed{z_{ij}}\big)_{ij}} : \underbrace{\boxed{(\star, \dots, \star)}}_{m} \to \underbrace{\boxed{(\star, \dots, \star)}}_{n}.$$

The sum of the three morphisms can be computed as follows:

$$\begin{aligned}\Big(\boxed{\big(\boxed{x_{ij}}\big)_{ij}} + \boxed{\big(\boxed{y_{ij}}\big)_{ij}}\Big) + \boxed{\big(\boxed{z_{ij}}\big)_{ij}} &= \boxed{\big(\boxed{x_{ij}} + \boxed{y_{ij}}\big)_{ij}} + \boxed{\big(\boxed{z_{ij}}\big)_{ij}} \\ &= \boxed{\big(\boxed{x_{ij} + y_{ij}}\big)_{ij}} + \boxed{\big(\boxed{z_{ij}}\big)_{ij}} \\ &= \boxed{\big(\boxed{x_{ij} + y_{ij}} + \boxed{z_{ij}}\big)_{ij}} \\ &= \boxed{\big(\boxed{(x_{ij} + y_{ij}) + z_{ij}}\big)_{ij}}.\end{aligned}$$

We see that we need twice as many steps for the computation in $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$ than for the computation in $\mathbf{Mat}_{\mathbb{Z}}$. This is due to the fact that $\mathbf{Mat}_{\mathbb{Z}}$ is a primitive category while $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$ is a tower of height 2. In general, a computation in a tower of height n requires n times as many steps as an analogous computation in a primitive category. Consequently, on a computer, this results in a major performance hit for computations in higher towers. Moreover, on a computer, boxing is not a free operation. If m and n in the above example are large, many inner boxes have to be created for the computation in $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$, resulting in an additional major performance hit. We will see more detailed analyses and benchmarks in Section 6.2 and Section 6.3.

Due to this performance overhead, formerly in many cases large computations in categorical towers were not feasible on a computer. In particular, the advantages of building categorical towers given in Remark 3.2.1 could not be fully exploited on a computer. This was particularly unfortunate because concepts like reusability are even more useful on the computer, where all constructions have to be fully executed, than they are on paper, where constructions can more easily be abbreviated.

`CompilerForCAP` [Zic24a] was started to avoid the performance overhead of categorical towers, hence allowing us to make full use of the advantages of building categorical towers on a computer. `CompilerForCAP` can avoid redundant computations inside categorical towers and categorical computations in general, and can even generate primitive implementations of reinterpretations of categorical towers. With this, large computations in categorical towers are finally feasible, often even beating the performance of implementations which have formerly been optimized by hand. We will see all of this in great detail in Chapter 6.

Chapter 4

Mathematical applications of categorical towers

In this chapter, we will see diverse mathematical applications of categorical towers. All categorical towers appearing in this chapter are implemented in various packages in the `CAP` ecosystem and available as primitive implementations completely or partially generated and optimized by `CompilerForCAP` [Zic24a].

In Section 4.1, we endow categories of matrices over certain rings with homomorphism structures using the categorical tower $\mathcal{C}(R)^{\oplus}$ seen in the previous chapter. In Section 4.2, we show how one can use these homomorphism structures to compute lifts in categories of finitely presented modules by modeling such categories as Freyd categories. In Section 4.3, we see an application of categorical towers which is more technical than the previous examples but will become very useful in the context of automatic code generation in Section 6.5.2: We use the categorical tower $\mathbf{C} \mapsto (\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ to formally dualize algorithms, that is, replace all concepts in an algorithm by the dual concept in the opposite category. On paper, the construction is straightforward, but when implemented on a computer, some subtle issues arise, which we solve in the course of the section. Finally, in Section 4.4, we introduce the **category of quivers Quiv**. We model **Quiv** as a functor category to show the existence of coproducts, coequalizers, and pushouts, which we need for the quantum computing application in the next chapter.

4.1 Homomorphism structures of categories of matrices

Let R be a ring. In Construction 3.3.2, we have seen that $\mathbf{Mat}_R$ can be modeled as a reinterpretation of the categorical tower $\mathcal{C}(R)^{\oplus}$. We will use this to endow $\mathbf{Mat}_R$ with a homomorphism structure with little effort if R is

- a commutative ring or, more generally,
- a k-algebra which is finitely generated free as a k-module, for some commutative ring k.

Homomorphism structures of $\mathbf{Mat}_R$ can be used to compute lifts in categories of finitely presented modules, as we will see in Section 4.2.

We start with the situation for commutative rings. In Construction 2.5.6, we have seen that for a commutative ring k, the category $\boldsymbol{\mathcal{C}}(k)$ has a homomorphism structure. This homomorphism structure can be lifted to a homomorphism structure for $\boldsymbol{\mathcal{C}}(k)^{\oplus}$. With the help of Construction 3.3.5, this homomorphism structure defines a homomorphism structure for $\mathbf{Mat}_k$. As we will see in Remark 4.1.2, this homomorphism structure is a formalization of a computational trick for solving two-sided matrix equations which is known as the “vec trick”.

Construction 4.1.1 (Homomorphism structures on categories of matrices over commutative rings, [Pos21b, Example 1.29])**.** Let k be a commutative ring. We can endow $\mathbf{Mat}_k$ with a $\mathbf{Mat}_k$-homomorphism structure as follows:

- The distinguished object is the object $\boxed{1}$.
- We define

$$\begin{aligned} H : {\mathbf{Mat}_k}^{\mathrm{op}} \times \mathbf{Mat}_k &\to \mathbf{Mat}_k \\ (\boxed{m}, \boxed{n}) &\mapsto \boxed{m \cdot n}, \\ (\boxed{M}, \boxed{N}) &\mapsto \boxed{M^T \otimes N}, \end{aligned}$$

  where “$\otimes$” denotes the Kronecker product of matrices.
- The components

$$\nu_{\boxed{m},\boxed{n}} : \mathrm{Hom}_{\mathbf{Mat}_k}(\boxed{m}, \boxed{n}) \to \mathrm{Hom}_{\mathbf{Mat}_k}(1, \boxed{m \cdot n})$$

  of the natural isomorphism ν unfold a given matrix row-wise into a matrix with a single row, that is,

$$\nu_{\boxed{m},\boxed{n}}\big((a_{ij})_{ij}\big) = (a_{11}, \dots, a_{1n}, a_{21}, \dots, a_{2n}, \dots, a_{m1}, \dots, a_{mn}).$$

A primitive implementation of this homomorphism structure of $\mathbf{Mat}_k$ generated and optimized by `CompilerForCAP` is available via the category constructor `CategoryOfRows` in the package `FreydCategoriesForCAP` [BPZ24].

Proof of correctness. We can endow $\boldsymbol{\mathcal{C}}(k)^{\oplus}$ with a $\boldsymbol{\mathcal{C}}(k)^{\oplus}$-homomorphism structure obtained as follows:

- $\boldsymbol{\mathcal{C}}(k)$ has a $\boldsymbol{\mathcal{C}}(k)$-homomorphism structure by Construction 2.5.6,
- using the full embedding $\boldsymbol{\mathcal{C}}(k) \hookrightarrow \boldsymbol{\mathcal{C}}(k)^{\oplus}$ in Remark 2.4.8 together with Remark 2.5.5, we obtain a $\boldsymbol{\mathcal{C}}(k)^{\oplus}$-homomorphism structure for $\boldsymbol{\mathcal{C}}(k)$,
- using that $\boldsymbol{\mathcal{C}}(k)^{\oplus}$ is an additive category, we can apply Construction 2.5.8 and obtain a $\boldsymbol{\mathcal{C}}(k)^{\oplus}$-homomorphism structure for $\boldsymbol{\mathcal{C}}(k)^{\oplus}$.

Using Construction 3.3.2 and Construction 3.3.5, we can turn this homomorphism structure into a $\mathbf{Mat}_k$-homomorphism structure for $\mathbf{Mat}_k$. Unfolding the definitions of the tower can be automated using `CompilerForCAP`, as we will see in Chapter 6. ■

This homomorphism structure formalizes a computational trick for solving two-sided matrix equations, which is known as the “vec trick”:

Remark 4.1.2. Let k be a commutative ring for which we can solve a system of linear equations $x \cdot A = b$ with $A \in k^{m \times n}$ and $b \in k^{1 \times n}$. Examples for such rings include fields with a Gaussian elimination algorithm or polynomial rings over a field with a Gröbner basis algorithm. Now assume that we want so solve an equation of the form

$$A \cdot X \cdot B = C$$

for X with given matrices A, B, and C of suitable sizes. There exists a computational trick known as the "vec trick" (see, for example, [Neu69], [Bre78], [ZL02], [BLH11]) which makes it possible to rewrite such an equation as a usual system of linear equations again: The key idea of this trick is to introduce an operation ν, also known as "vec", which vectorizes a given matrix by unfolding it row-wise into a matrix with a single row.[1] That is, for $A = (a_{ij})_{ij} \in k^{m \times n}$ we have

$$\nu\big((a_{ij})_{ij}\big) := \mathrm{vec}\big((a_{ij})_{ij}\big) := (a_{11}, \dots, a_{1n}, a_{21}, \dots, a_{2n}, \dots, a_{m1}, \dots, a_{mn}).$$

By spelling out all the definitions and using the commutativity of k, one can check that

$$\nu(A \cdot X \cdot B) = \nu(X) \cdot (A^T \otimes B),$$

where "$\otimes$" denotes the Kronecker product of matrices. This construction exactly matches the homomorphism structure in Construction 4.1.1 and the representation of the naturality in Remark 2.5.2, and was the motivation for the definition of homomorphism structures in [Pos21a].

Since ν is bijective, solving the equation

$$A \cdot X \cdot B = C$$

for X is equivalent to solving the equation

$$\nu(X) \cdot (A^T \otimes B) = \nu(C)$$

for $\nu(X)$, which is a usual system of linear equations. This matches the way in which we have used homomorphism structures in Proposition 2.6.2 for converting two-sided equations to usual lifts.

We can generalize the result for commutative rings to k-algebras which are finitely generated free as k-modules:

Construction 4.1.3 (Homomorphism structures on categories of matrices over finite-free k-algebras)**.** Let k be a commutative ring and let R be a **finite-free** k-algebra, that is, a k-algebra which is finitely generated free as a k-module. We first introduce some notation: Let b be the k-rank of R and fix a k-basis $\mathcal{B}$ of R, which we write as a column vector in $R^{b \times 1}$. We define an operation $\lambda : R \to k^{1 \times b}$ which maps an element $r \in R$ to the row vector of coefficients of r with regard to the k-basis $\mathcal{B}$. We extend this operation to an operation $\Lambda : R^{m \times n} \to k^{m \times (b \cdot n)}$ on matrices which applies λ entrywise.

Furthermore, we introduce a dual version of the Kronecker product for matrices: The normal Kronecker product of two matrices $A = (a_{ij})_{ij} \in R^{m \times n}$

[1] Dually, the vectorization could also be performed by unfolding a given matrix column-wise into a matrix with a single column.

and $B = (b_{ij})_{ij} \in R^{r \times s}$ is defined as follows:

$$A \otimes B := \begin{pmatrix} a_{11} \cdot B & \dots & a_{1n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m1} \cdot B & \dots & a_{mn} \cdot B \end{pmatrix}.$$

Now, we define the dual version as follows:

$$A \oslash B := \begin{pmatrix} A \cdot b_{11} & \dots & A \cdot b_{1s} \\ \vdots & \ddots & \vdots \\ A \cdot b_{r1} & \dots & A \cdot b_{rs} \end{pmatrix}.$$

For consistency of the notation, we define $A \obslash B := A \otimes B$.

With this, we can endow $\mathbf{Mat}_R$ with a $\mathbf{Mat}_k$-homomorphism structure as follows:

- The distinguished object is the object $\boxed{1}$.
- We define
$$\begin{aligned} H : {\mathbf{Mat}_R}^{\mathrm{op}} \times \mathbf{Mat}_R &\to \mathbf{Mat}_k \\ (\boxed{m}, \boxed{n}) &\mapsto \boxed{m \cdot b \cdot n} \\ (\boxed{M}, \boxed{N}) &\mapsto \boxed{\Lambda\big(M^T \obslash (\mathcal{B} \oslash N)\big)} \end{aligned}$$
- The components
$$\nu_{\boxed{m},\boxed{n}} : \mathrm{Hom}_{\mathbf{Mat}_k}(\boxed{m}, \boxed{n}) \to \mathrm{Hom}_{\mathbf{Mat}_k}(1, \boxed{m \cdot b \cdot n})$$
of the natural isomorphism ν unfold a given matrix row-wise into a matrix with a single row and apply Λ, that is,
$$\nu_{\boxed{m},\boxed{n}}\big((a_{ij})_{ij}\big) = \Lambda\big((a_{11}, \dots, a_{1n}, a_{21}, \dots, a_{2n}, \dots, a_{m1}, \dots, a_{mn})\big).$$

Proof of correctness. We can endow $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ with a $\mathbf{Mat}_k$-homomorphism structure as follows:

- $\boldsymbol{\mathcal{C}}(R)$ is a k-linear category with finitely generated free external homs by Construction 2.3.8,
- hence, $\boldsymbol{\mathcal{C}}(R)$ has a $\mathbf{Mat}_k$-homomorphism structure by Construction 2.5.7,
- using that $\mathbf{Mat}_k$ is an additive category, we can apply Construction 2.5.8 and obtain a $\mathbf{Mat}_k$-homomorphism structure for $\boldsymbol{\mathcal{C}}(R)^{\oplus}$.

Using Construction 3.3.2 and Construction 3.3.5, we can turn this homomorphism structure into a $\mathbf{Mat}_k$-homomorphism structure for $\mathbf{Mat}_R$. Unfolding the definitions of the tower can be automated using `CompilerForCAP`, as we will see in Chapter 6. ■

A common example of algebras which are finitely generated free as modules are **exterior algebras** over a field K:

Example 4.1.4 (Exterior algebras)**.** Let K be a field and let V be a K-vectorspace with basis $e_1, \dots, e_n$. The **exterior algebra** $\bigwedge V$ as defined in Definition B.1.1 is a finite dimensional K-algebra with dimension

$$\dim_K(\bigwedge V) = \sum_{k=0}^{n} \binom{n}{k} = 2^n.$$

Hence, we can apply Construction 4.1.3 to obtain a $\mathbf{Mat}_K$-homomorphism structure of $\mathbf{Mat}_{\bigwedge V}$. A primitive implementation of this homomorphism structure generated and optimized by `CompilerForCAP` is available via the category constructor `CategoryOfRows` in the package `FreydCategoriesForCAP`.

As an example, consider $K = \mathbb{Q}$ and $V = \mathbb{Q}^2$ with standard basis e_1, e_2. Then $\bigwedge V$ has $\mathbb{Q}$-dimension $b = 4$ with a possible basis given by the entries of

$$\mathcal{B} := \begin{pmatrix} 1 \\ e_1 \\ e_2 \\ e_1 \wedge e_2 \end{pmatrix}.$$

Consider two 1×2 matrices

$$M := \begin{pmatrix} 2 & 3e_1 \end{pmatrix} \qquad \text{and} \qquad N := \begin{pmatrix} 5e_2 & 7(e_1 \wedge e_2) \end{pmatrix}.$$

By construction of the $\mathbf{Mat}_{\mathbb{Q}}$-homomorphism structure of $\mathbf{Mat}_{\bigwedge \mathbb{Q}^2}$, we have

$$H(\boxed{M}, \boxed{N}) = \boxed{\Lambda\big(M^T \obslash (\mathcal{B} \oslash N)\big)}.$$

We compute

$$\begin{aligned} M^T \obslash (\mathcal{B} \oslash N) &= \begin{pmatrix} 2 \\ 3e_1 \end{pmatrix} \obslash \begin{pmatrix} 5e_2 & 7(e_1 \wedge e_2) \\ 5(e_1 \wedge e_2) & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 10e_2 & 14(e_1 \wedge e_2) \\ 10(e_1 \wedge e_2) & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 15(e_1 \wedge e_2) & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}. \end{aligned}$$

If we apply Λ to the result, we obtain

$$\left(\begin{array}{cccc|cccc} 0 & 0 & 10 & 0 & 0 & 0 & 0 & 14 \\ 0 & 0 & 0 & 10 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 15 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right),$$

where the left half (respectively the right half) of the matrix comes from the first (respectively the second column) of the original matrix. As we see, the matrices quickly become large, especially if the dimension n of V grows, because $\bigwedge V$ has exponential dimension 2^n. In Remark B.1.3, we discuss possible solutions for this problem by viewing the exterior algebra as an algebra over its center instead of as an algebra over K. This discussion involves additional terminology not introduced in this thesis.

4.2 Decidable lifts in categories of finitely presented modules

Decidable lifts are an important tool in category theory, for example for deciding whether monomorphisms and epimorphisms are split and for computing inverses of isomorphisms or more generally retractions and sections of split monomorphisms and split epimorphisms. In this section, we look at categories of finitely presented modules and see how we can decide and compute lifts there. We will again use categorical towers to tackle the problem.

Convention 4.2.1. Let R be a ring and $n \in \mathbb{N}$. In this section, we think of the elements of R^n as row vectors.

Definition 4.2.2 (Categories of finitely presented left modules)**.** Let R be a ring. We define the **category ${}_R\mathbf{FPres}$ of finitely presented left modules over** R as follows:

- Its objects are finitely presented left modules given by the quotient of some R^n by finitely many relations between the elements of the standard basis,
- its morphisms are linear maps with pointwise equality,
- composition is given by the usual composition of linear maps,
- identity morphisms are given by the identity maps.

One can easily check that this indeed defines a category.

For now, we only consider **left** modules and hence omit the word **left** in the following. Once we have modeled ${}_R\mathbf{FPres}$ as a categorical tower, we can easily switch to **right** modules by simply inserting an opposite category in the tower, see Construction 4.2.7.

Remark 4.2.3. Let R be a ring. Defining a meta-theoretical equality on the objects of ${}_R\mathbf{FPres}$ is difficult. For example, consider the following finitely presented modules:

$$R^2/\langle e_1 = e_2\rangle, \qquad R^2/\langle e_2 = e_1\rangle, \qquad R^2/\langle -e_1 = -e_2\rangle, \qquad R^2/\langle e_1 - e_2 = 0\rangle.$$

On paper, we would probably view all these finitely presented modules as equal. Moreover, consider the following finitely presented modules:

$$R^3/\langle e_1 = e_2, e_2 = e_3\rangle, \qquad R^3/\langle e_2 = e_3, e_1 = e_2\rangle, \qquad R^3/\langle e_1 = e_3, e_2 = e_3\rangle,$$
$$R^3/\langle e_1 = e_2, e_1 = e_3, e_2 = e_3\rangle.$$

Again, one paper we would possibly view all these finitely presented modules as equal. We might even view them as equal to the finitely presented modules above, as all are isomorphic to R^1. However, doing so possibly requires us to manually insert appropriate isomorphism in computations.

In the following, we will define a data structure for relations and simply use the technical equality coming with this data structure as the meta-theoretical equality on objects to avoid any ambiguity. As explained in Remark 1.4.23, this does not change the category theory.

Construction 4.2.4 (Data structures for categories of finitely presented modules, [BLH11, Section 3])**.** Let R be a ring. We want to provide data structures for objects and morphisms in ${}_R\mathbf{FPres}$. We start with a data structure for objects: Every relation between standard basis vectors of R^n can be expressed as an equation $\sum_{i=1}^n \lambda_i e_i = 0$ with $\lambda_i \in R$ by bringing all summands of the relation to one side and by explicitly multiplying standard basis vectors not appearing in the relation by 0. Hence, we can encode relations as row vectors $(\lambda_1, \dots, \lambda_n)$. To encode multiple relations, we simply stack the corresponding row vectors to form a matrix M with n columns. Hence, an object in ${}_R\mathbf{FPres}$ can be encoded as a matrix.

Determining whether this encoding is one-to-one depends on the meta-theoretical equality on objects in ${}_R\mathbf{FPres}$, which is difficult to define, as explained in Remark 4.2.3. To avoid ambiguity, we simply use the usual equality of matrices as the meta-theoretical equality on objects in ${}_R\mathbf{FPres}$.

We continue with a data structure for morphisms. Let $\boxed{M}$ and $\boxed{N}$ be two objects, that is, let $M \in R^{k,m}$ and $N \in R^{\ell,n}$ be two matrices to which we associate two finitely presented modules by viewing the rows of M and N as relations. We write $R^m/\langle M\rangle$ and $R^n/\langle N\rangle$ for these finitely presented modules. Let $(e_1, \dots, e_m)$ and $(f_1, \dots, f_n)$ be the standard bases of R^m and R^n, respectively. A morphism $\boxed{M} \to \boxed{N}$ is a linear map

$$\varphi : R^m/\langle M\rangle \to R^n/\langle N\rangle.$$

Such a map can be encoded as a matrix $P \in R^{m\times n}$ by choosing P such that

$$\varphi(\overline{e_i}) = \sum_{j=1}^{n} P_{ij}\overline{f_j}.$$

Hence, matrices in $R^{m\times n}$ provide a suitable data structure for morphisms. However, not every matrix in $R^{m\times n}$ defines a linear map $R^m/\langle M\rangle \to R^n/\langle N\rangle$ because the relations of the source might not be respected, so we have to restrict to a subset of matrices in $R^{m\times n}$. To make this precise, let $(\lambda_1, \dots, \lambda_m)$ be a row of M, that is, a relation of the source. Then

$$0 = \varphi(0) = \varphi\Big(\sum_{i=1}^{m} \lambda_i \overline{e_i}\Big) = \sum_{i=1}^{m} \lambda_i \sum_{j=1}^{n} P_{ij}\overline{f_j},$$

that is,

$$\sum_{i=1}^{m} \lambda_i \sum_{j=1}^{n} P_{ij} f_j = 0 \quad \text{modulo the relations of the target}$$

or equivalently

$$(\lambda_1, \dots, \lambda_m) \cdot P = \Big(\sum_{i=1}^{m} \lambda_i P_{ij}\Big)_{j=1,\dots,n} = 0 \quad \text{modulo the rows of } N.$$

The last line in turn is equivalent to the existence of a row vector $(\mu_1, \dots, \mu_\ell)$ such that

$$(\lambda_1, \dots, \lambda_m) \cdot P = (\mu_1, \dots, \mu_\ell) \cdot N.$$

This can be repeated for every row of M, so we get a matrix $P' \in R^{k \times \ell}$ such that

$$M \cdot P = P' \cdot N.$$

Hence, we get the existence of P' as a necessary condition for P to define a linear map $R^m/\langle M\rangle \to R^n/\langle N\rangle$. Using the same arguments and the homomorphism theorem one can show that the existence of P' is also a sufficient condition for P to define a linear map $R^m/\langle M\rangle \to R^n/\langle N\rangle$.

Finally, we would like to examine when two matrices in $R^{m\times n}$ define the same linear map $R^m/\langle M\rangle \to R^n/\langle N\rangle$. For this, let $P, Q \in R^{m\times n}$ be matrices and let φ and ψ be the corresponding linear maps $R^m/\langle M\rangle \to R^n/\langle N\rangle$. We can check equality on generators, so φ and ψ are equal if and only if for all $i \in \{1, \dots, m\}$ we have

$$\varphi(\overline{e_i}) = \psi(\overline{e_i}),$$

that is,

$$\sum_{j=1}^{n} P_{ij}\overline{f_j} = \sum_{j=1}^{n} Q_{ij}\overline{f_j}$$

or equivalently

$$\sum_{j=1}^{n}(P_{ij} - Q_{ij})f_j = 0 \quad \text{modulo the relations of the target.}$$

As above this is equivalent to the existence of a matrix $K \in R^{m\times\ell}$ such that

$$P - Q = K \cdot N.$$

Summing up, a morphism $\boxed{M} \to \boxed{N}$ is exactly given by a matrix $P \in R^{m\times n}$ such that there exists a matrix $P' \in R^{k\times\ell}$ with

$$M \cdot P = P' \cdot N,$$

and we view two morphisms $\boxed{P}, \boxed{Q} : \boxed{M} \to \boxed{N}$ as equal if and only if there exists a matrix $K \in R^{m\times\ell}$ such that

$$P - Q = K \cdot N.$$

Composition of morphisms can then simply be given by matrix multiplication and identity morphisms can be given by identity matrices.

We have seen that both objects and morphisms of ${}_R\mathbf{FPres}$ can be encoded as matrices over R, where the matrices defining morphisms have to fulfill certain conditions. These matrices can be seen as morphisms in $\mathbf{Mat}_R$, and the conditions on morphisms of ${}_R\mathbf{FPres}$ can be expressed as equations in $\mathbf{Mat}_R$. Hence, Construction 4.2.4 can be interpreted as a construction on $\mathbf{Mat}_R$. This construction is a special case of the construction of Freyd categories in Definition 2.6.1, which allows us to model ${}_R\mathbf{FPres}$ as a categorical tower in the following proposition.

Construction 4.2.5 (Categories of finitely presented modules as categorical towers, [Pos21a, Example 4.2])**.** Let R be a ring. We can model ${}_R\mathbf{FPres}$ as a reinterpretation of the categorical tower $\mathbf{Freyd}(\mathbf{Mat}_R)$ via the functor $\mathcal{R}$ defined as

$$\begin{aligned} \mathcal{R} : \mathbf{Freyd}(\mathbf{Mat}_R) &\to {}_R\mathbf{FPres} \\ \boxed{\boxed{M}_{\mathbf{Mat}_R}}_{\mathbf{Freyd}(\mathbf{Mat}_R)} &\mapsto \boxed{M}_{{}_R\mathbf{FPres}} \\ \boxed{\boxed{P}_{\mathbf{Mat}_R}}_{\mathbf{Freyd}(\mathbf{Mat}_R)} &\mapsto \boxed{P}_{{}_R\mathbf{FPres}} \end{aligned}$$

together with its inverse $\mathcal{M}$ defined as

$$\begin{aligned} \mathcal{M} : {}_R\mathbf{FPres} &\to \mathbf{Freyd}(\mathbf{Mat}_R) \\ \boxed{M}_{{}_R\mathbf{FPres}} &\mapsto \boxed{\boxed{M}_{\mathbf{Mat}_R}}_{\mathbf{Freyd}(\mathbf{Mat}_R)} \\ \boxed{P}_{{}_R\mathbf{FPres}} &\mapsto \boxed{\boxed{P}_{\mathbf{Mat}_R}}_{\mathbf{Freyd}(\mathbf{Mat}_R)} \end{aligned}$$

A primitive implementation of ${}_R\mathbf{FPres}$ partially generated from $\mathbf{Freyd}(\mathbf{Mat}_R)$ and optimized by `CompilerForCAP` is available via the category constructor `LeftPresentations` in the package `ModulePresentationsForCAP` [GPZ24b].

Proof of correctness. The technical equality on objects and the mathematical equality on morphisms in $\mathbf{Freyd}(\mathbf{Mat}_R)$ exactly correspond to the equalities in ${}_R\mathbf{FPres}$ given in Construction 4.2.4, so in particular $\mathcal{R}$ and $\mathcal{M}$ respect the equality on morphisms. Moreover, $\mathcal{R}$ and $\mathcal{M}$ are mutually inverse both on objects and morphisms. Finally, composition and identity morphisms are given by matrix multiplication and identity matrices on both sides.

Summing up, $\mathcal{R}$ indeed defines an isomorphism of categories with inverse $\mathcal{M}$, as required for a reinterpretation. ■

We can immediately make use of the description of ${}_R\mathbf{FPres}$ as a categorical tower:

Corollary 4.2.6 (Decidable lifts in categories of finitely presented modules via homomorphism structures)**.** *Let R be a ring. Assume that we can endow $\mathbf{Mat}_R$ with a $\mathbf{D}$-homomorphism structure for an additive category $\mathbf{D}$ with decidable lifts. Then ${}_R\mathbf{FPres}$ has decidable lifts.*

Proof. This follows from Construction 4.2.5 and Proposition 2.6.2. ■

We can now define categories of finitely presented **right** modules by dualizing the category of matrices over R in the following construction.

Construction 4.2.7 (Categories of finitely presented **right** modules, [Pos21a, Example 4.2])**.** Let R be a ring. We define the category $\mathbf{FPres}_R$ of finitely presented **right** modules over R as follows:

- Its objects are matrices over R,
- a morphism from $M \in R^{m\times k}$ to $N \in R^{n\times \ell}$ is a matrix $P \in R^{n\times m}$ such that there exists a matrix $P' \in \ell \times k$ with

$$P \cdot M = N \cdot P',$$

where we view two morphisms $\boxed{P}, \boxed{Q} : \boxed{M} \to \boxed{N}$ as equal if and only if there exists a matrix $K \in R^{\ell \times m}$ such that

$$P - Q = N \cdot K.$$

The categorical structure is defined using Construction 3.3.5 via a reinterpretation of the categorical tower $\mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}})$ such that we get an isomorphism of categories $\mathcal{R}$ defined as

$$\begin{aligned} \mathcal{R} : \mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}}) &\to \mathbf{FPres}_R \\ \boxed{\boxed{\boxed{M}_{\mathbf{Mat}_R}}_{\mathbf{Mat}_R{}^{\mathrm{op}}}}_{\mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}})} &\mapsto \boxed{M}_{\mathbf{FPres}_R} \\ \boxed{\boxed{\boxed{P}_{\mathbf{Mat}_R}}_{\mathbf{Mat}_R{}^{\mathrm{op}}}}_{\mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}})} &\mapsto \boxed{P}_{\mathbf{FPres}_R} \end{aligned}$$

together with its inverse $\mathcal{M}$ defined as

$$\begin{aligned} \mathcal{M} : \mathbf{FPres}_R &\to \mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}}) \\ \boxed{M}_{\mathbf{FPres}_R} &\mapsto \boxed{\boxed{\boxed{M}_{\mathbf{Mat}_R}}_{\mathbf{Mat}_R{}^{\mathrm{op}}}}_{\mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}})} \\ \boxed{P}_{\mathbf{FPres}_R} &\mapsto \boxed{\boxed{\boxed{P}_{\mathbf{Mat}_R}}_{\mathbf{Mat}_R{}^{\mathrm{op}}}}_{\mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}})} \end{aligned}$$

A primitive implementation of $\mathbf{FPres}_R$ partially generated from $\mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}})$ and optimized by `CompilerForCAP` is available via the category constructor `RightPresentations` in the package `ModulePresentationsForCAP`.

Corollary 4.2.8 (Decidable lifts in categories of finitely presented **right** modules via homomorphism structures)**.** *Let R be a ring. Assume that we can endow $\mathbf{Mat}_R$ with a $\mathbf{D}$-homomorphism structure for an additive category $\mathbf{D}$ with decidable lifts. Then $\mathbf{FPres}_R$ has decidable lifts.*

Proof. The opposite category $\mathbf{Mat}_R{}^{\mathrm{op}}$ is a preadditive category (Remark 2.3.6) with a $\mathbf{D}$-homomorphism structure due to Construction 2.5.9. Hence, the claim follows from Construction 4.2.7 and Proposition 2.6.2. ■

Example 4.2.9 (Categories of finitely presented modules over exterior algebras have decidable lifts)**.** Let K be a field and let V be a finite dimensional K-vectorspace. We consider the exterior algebra $E \coloneqq \bigwedge V$. In Example 4.1.4, we have seen that $\mathbf{Mat}_E$ has a $\mathbf{Mat}_K$-homomorphism structure. The category of matrices $\mathbf{Mat}_K$ is an additive category. Moreover, using Gaussian elimination, we can solve systems of linear equations $X \cdot A = B$ over K, that is, $\mathbf{Mat}_K$ has decidable lifts. Hence, ${}_E\mathbf{FPres}$ and $\mathbf{FPres}_E$ have decidable lifts.

We again see some of the advantages mentioned in Remark 3.2.1:

- **Reusability**: The constructor for categories of matrices and the constructor for Freyd categories can be reused for both left and right modules. Moreover, the constructor for Freyd categories can, for instance, be reused for modeling finitely presented functors in [Pos21a] and for constructing free abelian categories in [Pos22].
- **Separation of concerns**: The intricate equality of morphisms in Freyd categories, dualization as a generic concept, and the matrix algorithms are strictly separated from each other.

4.3 Dualizing algorithms

We now consider an application of categorical towers which is more technical than the previous examples but will become very useful in the context of automatic code generation in Section 6.5.2: The automatic dualization of algorithms. Many of the concepts defined in Chapter 2 have dual versions in the opposite category, either by definition, for example products and coproducts, or implicitly because they are self-dual, for example direct sums. For every algorithm involving such concepts, there also exists a dual algorithm using the dual concepts. For instance, the algorithm computing binary pushouts from coproducts and coequalizers in Example 1.2.6 has a dual version which computes binary pullbacks from products and equalizers. On paper, we often do not spell out the dual version of an algorithm. On a computer, however, we would like to have implementations of the dual versions of algorithms and would like to avoid having to dualize implementations by hand. Hence, in this section, we formalize the dualization of algorithms with the help of categorical towers. In Section 6.5.2, we will use this formalization to automatically generate dual versions of implementations with the help of `CompilerForCAP`.

Remark 4.3.1 (The categorical tower $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$)**.** The central idea for computing the dual version of an algorithm is to pass from a category $\mathbf{C}$ to its opposite $\mathbf{C}^{\mathrm{op}}$, apply the algorithm there, and then pass to $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$. On paper, one usually views $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ and $\mathbf{C}$ as the same category, so we get the intended result in $\mathbf{C}$. On the computer, however, $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ and $\mathbf{C}$ are different categories because they are endowed with different algorithms: The algorithms of $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ are all built on algorithms of $\mathbf{C}^{\mathrm{op}}$, while the algorithms of $\mathbf{C}$ can never be built on algorithms of $\mathbf{C}^{\mathrm{op}}$ because this would introduce infinite recursions. Hence, formally we should distinguish between $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ and $\mathbf{C}$.

To formalize that $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ is mathematically just $\mathbf{C}$ but has different algorithms, we create a "copy" $\mathbf{C}'$ of $\mathbf{C}$ which has the same objects and morphisms as $\mathbf{C}$ but is endowed with a categorical structure via a reinterpretation of $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$.

Construction 4.3.2 ("Copies" of categories)**.** Let $\mathbf{C}$ be a category. We create a "copy" $\mathbf{C}'$ of $\mathbf{C}$ as follows: $\mathbf{C}'$ has the same objects and morphisms as $\mathbf{C}$. This allows us to introduce two mappings on objects and morphisms of $\mathbf{C}$ and $\mathbf{C}'$:

$$\begin{aligned}\mathcal{N} : \mathbf{C} &\to \mathbf{C}'\\ \boxed{X}_{\mathbf{C}} &\mapsto \boxed{X}_{\mathbf{C}'}\\ \boxed{x}_{\mathbf{C}} &\mapsto \boxed{x}_{\mathbf{C}'}\end{aligned}$$

and its inverse

$$\begin{aligned}\mathcal{O} : \mathbf{C}' &\to \mathbf{C}\\ \boxed{X}_{\mathbf{C}'} &\mapsto \boxed{X}_{\mathbf{C}}\\ \boxed{x}_{\mathbf{C}'} &\mapsto \boxed{x}_{\mathbf{C}}\end{aligned}$$

where "$\mathcal{N}$" stands for "new" and "$\mathcal{O}$" for "original". Now, we can use Construction 3.3.5 to define a categorical structure on $\mathbf{C}'$ as a reinterpretation of the categorical tower $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ via

$$\begin{aligned} \mathcal{R} : (\mathbf{C}^{\mathrm{op}})^{\mathrm{op}} &\to \mathbf{C}' \\ \boxed{\boxed{A}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}} &\mapsto \mathcal{N}(A) \\ \boxed{\boxed{f}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}} &\mapsto \mathcal{N}(f) \end{aligned}$$

with inverse

$$\begin{aligned} \mathcal{M} : \mathbf{C}' &\to (\mathbf{C}^{\mathrm{op}})^{\mathrm{op}} \\ A &\mapsto \boxed{\boxed{\mathcal{O}(A)}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}} \\ f &\mapsto \boxed{\boxed{\mathcal{O}(f)}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}} \end{aligned}$$

This categorical structure on $\mathbf{C}'$ ultimately comes from the categorical structure of $\mathbf{C}$. Hence, $\mathcal{N}$ and $\mathcal{O}$ are functors, and since they are mutually inverse, they are even isomorphisms.

Note that on paper, we only view boxes as a notation, and if we drop the boxes all the functors are just given by identity maps. On the computer, however, the functors properly translate between the various boxed data structures and concepts.

Using this construction, we can hide the subtleties of the categorical tower $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$:

Remark 4.3.3. Let $\mathbf{C}$ be a category in which we want to compute a dual version of an algorithm. As seen before, by applying the algorithm in $\mathbf{C}^{\mathrm{op}}$, the categorical tower $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ can compute the dual version of the algorithm. We can hide this tower by creating a copy $\mathbf{C}'$ of $\mathbf{C}$ as above. Now, we can compute the dual version of the algorithm in $\mathbf{C}$ by passing to $\mathbf{C}'$ via $\mathcal{N}$, computing the dual version of the algorithm there, and returning to $\mathbf{C}$ via $\mathcal{O}$.

This shows another kind of **separation of concerns**: We get access to the dual version of an algorithm without actually seeing opposite categories because those are hidden by the reinterpretation.

The construction of the copy $\mathbf{C}'$ of $\mathbf{C}$ as above depends on the data structures of objects and morphisms of $\mathbf{C}$. However, we can still make the dualization of algorithms independent of the concrete category $\mathcal{C}$. To see how this can be done, we first consider a concrete example.

Example 4.3.4. Consider an algorithm which computes binary coequalizers of two morphisms by computing the cokernel of the difference of the two morphisms as in Remark 2.7.7. Moreover, consider a preadditive category $\mathbf{C}$ endowed with an algorithm for computing kernels but a priori without an algorithm for computing binary equalizers. Naturally, we would like to endow $\mathbf{C}$ with an algorithm for computing binary equalizers by dualizing the algorithm for binary coequalizers in Remark 2.7.7. According to the above, we proceed as follows:

- The opposite category $\mathbf{C}^{\mathrm{op}}$ is a preadditive category which is naturally endowed with an algorithm for computing cokernels by computing kernels in $\mathbf{C}$.

- Hence, by Remark 2.7.7, $\mathbf{C}^{\mathrm{op}}$ can be endowed with an algorithm for computing binary coequalizers.
- Therefore, the double opposite category $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ is naturally endowed with an algorithm for computing binary equalizers by computing binary coequalizers in $\mathbf{C}^{\mathrm{op}}$.
- Now, we reinterpret the categorical tower $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$ as a copy $\mathbf{C}'$ of $\mathbf{C}$ as above.
- With this, we can computing binary equalizers in $\mathbf{C}$ by passing to $\mathbf{C}'$ via $\mathcal{N}$, computing binary equalizers there, and returning to $\mathbf{C}$ via $\mathcal{O}$.

Now, let us consider a computation of the equalizer of two parallel morphisms $f, g : A \to B$ in $\mathbf{C}$:

$$
\begin{aligned}
\mathrm{eq}(f,g) &\coloneqq \mathcal{O}\Big(\mathrm{eq}\Big(\mathcal{N}(f), \mathcal{N}(g)\Big)\Big) \\
&\equiv \mathcal{O}\Big(\mathcal{R}\Big(\mathrm{eq}\Big(\mathcal{M}(\mathcal{N}(f)), \mathcal{M}(\mathcal{N}(g))\Big)\Big)\Big) \\
&\equiv \mathcal{O}\Big(\mathcal{R}\Big(\mathrm{eq}\Big(\boxed{\boxed{\mathcal{O}(\mathcal{N}(f))}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}}, \boxed{\boxed{\mathcal{O}(\mathcal{N}(g))}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}}\Big)\Big)\Big) \\
&\equiv \mathcal{O}\Big(\mathcal{R}\Big(\mathrm{eq}\Big(\boxed{\boxed{f}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}}, \boxed{\boxed{g}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}}\Big)\Big)\Big) \\
&\equiv \mathcal{O}\Big(\mathcal{R}\Big(\boxed{\mathrm{coeq}\big(\boxed{f}_{\mathbf{C}^{\mathrm{op}}}, \boxed{g}_{\mathbf{C}^{\mathrm{op}}}\big)}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}}\Big)\Big) \\
&\equiv \mathcal{O}\Big(\mathcal{R}\Big(\boxed{\mathrm{coker}\big(\boxed{f}_{\mathbf{C}^{\mathrm{op}}} - \boxed{g}_{\mathbf{C}^{\mathrm{op}}}\big)}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}}\Big)\Big) \\
&\equiv \mathcal{O}\Big(\mathcal{R}\Big(\boxed{\mathrm{coker}\big(\boxed{f - g}_{\mathbf{C}^{\mathrm{op}}}\big)}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}}\Big)\Big) \\
&\equiv \mathcal{O}\Big(\mathcal{R}\Big(\boxed{\boxed{\mathrm{ker}\big(f - g\big)}_{\mathbf{C}^{\mathrm{op}}}}_{(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}}\Big)\Big) \\
&\equiv \mathcal{O}\Big(\mathcal{N}\Big(\mathrm{ker}\big(f - g\big)\Big)\Big) \\
&\equiv \mathrm{ker}\big(f - g\big)
\end{aligned}
$$

Note that we have not applied the definitions of $\mathcal{N}$ and $\mathcal{O}$ at all, but have simply used that they are mutually inverse. In particular, we have not seen a value being boxed as an object or morphism in $\mathbf{C}$ or $\mathbf{C}'$. Hence, the computation can be performed symbolically independent of the data structures of objects and morphisms of $\mathbf{C}$. Summing up the computation, we get the categorical algorithm

$$\mathrm{eq}(f,g) \coloneqq \mathrm{ker}\big(f - g\big)$$

for two parallel morphisms $f, g : A \to B$ in an arbitrary preadditive category $\mathbf{C}$ with kernels. This is exactly the expected dualization of computing binary coequalizers via cokernels.

The previous example shows that we can perform the required computations symbolically and independent of the category, which allows us to extract the desired dual algorithm. In Section 6.5.2, we will use this to automatically generate dual versions of implementations of categorical algorithms with the help of `CompilerForCAP`.

We give an example of the dual version of a more complex algorithm:

Example 4.3.5. The dual version of the algorithm in Example 1.2.6 computes binary pullbacks from products and equalizers as follows:

Let $\mathbf{C}$ be a category and let $f : A \to C$ and $g : B \to C$ be two morphisms in $\mathbf{C}$ with a common source. We can compute a pullback P of f and g from products and equalizers as follows:

- compute the product $A \times B$ together with the projections $\pi_A^\times : A \times B \to A$ and $\pi_B^\times : A \sqcup B \to B$,
- define two auxiliary morphisms $d \coloneqq \pi_A^\times \cdot f$ and $e \coloneqq \pi_B^\times \cdot g$,
- define $P \coloneqq \mathrm{eq}(d, e)$ with morphisms $\pi_A \coloneqq \iota^{\mathrm{eq}} \cdot \pi_A^\times$ and $\pi_B \coloneqq \iota^{\mathrm{eq}} \cdot \pi_B^\times$.

Given two morphisms $\tau_A : T \to A$ and $\tau_B : T \to B$ such that $\tau_A \cdot f = \tau_B \cdot g$, we can compute $u : T \to P$ with the required properties by first applying the universal property of the product and then the universal property of the equalizer.

Of course, on paper we do not actually obtain this dual algorithm by symbolic computations in the categorical tower $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$. On the computer, however, an implementation for the dual algorithm will indeed be generated via $(\mathbf{C}^{\mathrm{op}})^{\mathrm{op}}$.

Remark 4.3.6 (Advantages of using a categorical tower)**.** We again see some of the advantages of categorical towers mentioned in Remark 3.2.1:

- **Reusability**: We can reuse the constructor for opposite categories, which we have already used in Construction 4.2.7.
- **Verifiability**: Verifying computations involving dualization is notoriously difficult. Here, even two layers of dualization are involved. Thanks to the categorical tower, we only have to verify the construction of the opposite category once.
- **Emergence**: The dual version of the algorithm emerges naturally from the original algorithm and the use of the opposite categories.

4.4 Pushouts in the category of quivers

In this section, we introduce the **category of quivers** and model it as a functor category into **Sets**. Since limits and colimits in functor categories can be computed pointwise in the target category, the category of quivers inherits limits and colimits from **Sets**. We specifically look at coproducts, coequalizers, and pushouts, which we will need for the quantum computing application in Chapter 5.

Definition 4.4.1 (Quivers)**.** A **quiver** (or **directed multigraph**) consists of the following data:

- a set of **vertices** V,
- a set of **arrows** A,
- two maps $s, t : A \to V$.

We call $a \in A$ an **arrow from** $s(a)$ **to** $t(a)$. Moreover, we call a tuple of arrows $(a_1, \dots, a_n)$ a **path from** $s(a_1)$ **to** $t(a_n)$ if $t(a_i) = s(a_{i+1})$ for all $i \in 1, \dots, n-1$.

A morphism η between quivers (V_1, A_1, s_1, t_1) and (V_2, A_2, s_2, t_2) is given by two maps of sets $\eta_V : V_1 \to V_2$ and $\eta_A : A_1 \to A_2$ compatible with s_1, s_2, t_1, and t_2 in the sense that the following two diagrams commute:

$$\begin{tikzcd} A_1 \arrow[r, "\eta_A"] \arrow[d, "s_1"'] & A_2 \arrow[d, "s_2"] \\ V_1 \arrow[r, "\eta_V"'] & V_2 \end{tikzcd} \qquad \begin{tikzcd} A_1 \arrow[r, "\eta_A"] \arrow[d, "t_1"'] & A_2 \arrow[d, "t_2"] \\ V_1 \arrow[r, "\eta_V"'] & V_2 \end{tikzcd}$$

To a quiver Q, we can associate a category:

Definition 4.4.2 (Free categories)**.** Given a quiver Q, we construct the **free category $\mathcal{C}(Q)$ generated by Q** as follows:

- its objects are the vertices of the quiver,
- a morphism from $\boxed{v}$ to $\boxed{w}$ is a path from v to w, where two paths are equal if they have the same length and are given by the same arrows in the same order,
- composition of morphisms is given by the concatenation of paths,
- identity morphisms are given by empty paths.

One can easily check that this indeed defines a category.

Moreover, we see that quivers and their morphisms themselves form a category **Quiv**:

Definition 4.4.3 (The category of quivers)**.** The category of quivers **Quiv** is defined as follows:

- its objects are quivers,
- its morphisms are morphisms of quivers with equality given by the pointwise equality of the maps on vertices and arrows,
- composition of two morphisms η and ε is given by composing the maps on vertices and arrows separately,
- identity morphisms are given by the identity maps on vertices and arrows.

One can easily check that this indeed defines a category.

If we restrict to quivers with finite sets of vertices and arrows, we obtain the **category of finite quivers FinQuiv**.

For the application in Section 5.4, we want to show that the category of quivers **Quiv** has binary pushouts. To do so, we again build a categorical tower.

Construction 4.4.4 (The category of quivers as a categorical tower)**.** We start with the following quiver[2] $\mathfrak{Q}$:

$$\begin{tikzcd} \mathfrak{A} \arrow[r, shift left, "\mathfrak{s}"] \arrow[r, shift right, "\mathsf{t}"'] & \mathfrak{V} \end{tikzcd}$$

That is, the quiver $\mathfrak{Q}$ has vertex set $V := \{\mathfrak{V}, \mathfrak{A}\}$ and arrow set $A := \{\mathfrak{s}, \mathsf{t}\}$ with $s(\mathfrak{s}) = s(\mathsf{t}) = \mathfrak{A}$ and $t(\mathfrak{s}) = t(\mathsf{t}) = \mathfrak{V}$. The reason for the initially confusing notation will become clear soon.

[2] We use a quiver to model the category of quivers, so this looks like a cyclic definition at first. We will explain why this is not the case in Remark 4.4.5.

We can model **Quiv** as a reinterpretation of the categorical tower $\mathbf{Sets}^{\mathcal{C}(\mathfrak{Q})}$, the functor category from $\mathcal{C}(\mathfrak{Q})$ to **Sets**, via the functor $\mathcal{R}$ defined as

$$\begin{aligned} \mathcal{R} : \mathbf{Sets}^{\mathcal{C}(\mathfrak{Q})} &\to \mathbf{Quiv} \\ F &\mapsto \big(F(\boxed{\mathfrak{V}}), F(\boxed{\mathfrak{A}}), F(\boxed{\mathfrak{s}}), F(\boxed{\mathfrak{t}})\big) \\ \eta &\mapsto \big(\eta_{\boxed{\mathfrak{V}}}, \eta_{\boxed{\mathfrak{A}}}\big) \end{aligned}$$

together with its inverse $\mathcal{M}$ defined as

$$\begin{aligned} &\mathcal{M} : \mathbf{Quiv} \to \mathbf{Sets}^{\mathcal{C}(\mathfrak{Q})} \\ &(V, A, s, t) \mapsto F \text{ with } F(\boxed{\mathfrak{V}}) := V,\ F(\boxed{\mathfrak{A}}) := A,\ F(\boxed{\mathfrak{s}}) := s, \text{ and } F(\boxed{\mathfrak{t}}) := t \\ &\quad (\eta_V, \eta_A) \mapsto \eta \text{ with } \eta_{\boxed{\mathfrak{V}}} := \eta_V \text{ and } \eta_{\boxed{\mathfrak{A}}} := \eta_A \end{aligned}$$

Analogously, we can model **FinQuiv** as a reinterpretation of the categorical tower $\mathbf{FinSets}^{\mathcal{C}(\mathfrak{Q})}$. A primitive implementation of **FinQuiv** generated from $\mathbf{FinSets}^{\mathcal{C}(\mathfrak{Q})}$ and partially optimized by `CompilerForCAP` is available via the category `FinQuivers` in the package `FunctorCategories` [BS24].

Proof of correctness. The category $\mathcal{C}(\mathfrak{Q})$ has only two objects $\boxed{\mathfrak{V}}$ and $\boxed{\mathfrak{A}}$ and four morphisms: the two identity morphisms given by the empty paths on $\boxed{\mathfrak{V}}$ and $\boxed{\mathfrak{A}}$ and two morphisms $\boxed{(\mathfrak{s})}$ and $\boxed{(\mathfrak{t})}$ each given by a path containing a single arrow. This shows that $\mathcal{M}$ actually specifies all the data required for uniquely defining a functor from $\mathcal{C}(\mathfrak{Q})$ to **Sets** and a natural transformation between such functors.

One can easily check that $\mathcal{R}$ actually defines quivers and morphisms of quivers and that $\mathcal{M}$ actually defines functors and natural transformations. For example, the naturality of the natural transformations exactly corresponds to the diagrams in Definition 4.4.1.

The equalities of morphisms in $\mathbf{Sets}^{\mathcal{C}(\mathfrak{Q})}$ and **Quiv** are both given by the componentwise equality of maps of sets, so $\mathcal{R}$ and $\mathcal{M}$ respect the equality on morphisms. Moreover, with the canonical meta-theoretical equalities on functors and quivers, $\mathcal{R}$ and $\mathcal{M}$ are mutually inverse on objects and morphisms.

Finally, composition and identity morphisms are given by componentwise composition of maps of sets and identity maps of sets on both sides.

Summing up, $\mathcal{R}$ indeed defines an isomorphism of categories with inverse $\mathcal{M}$, as required for a reinterpretation. ■

Remark 4.4.5. Note that for constructing the functor category from $\mathcal{C}(\mathfrak{Q})$ to **Sets**, we need a data structure for quivers to encode $\mathfrak{Q}$. However, we neither need a data structure for morphisms of quivers nor any algorithm on $\mathfrak{Q}$. Still, the categorical tower has data structures for objects and morphisms and algorithms on objects and morphisms. Those are obtained from the data structures and algorithms for functors, natural transformations, and (maps of) sets. Hence, the construction of the tower is not cyclic at all.

We can use the categorical tower to see that the category of quivers has coproducts and coequalizers:

Remark 4.4.6 (The category of quivers has coproducts and coequalizers)**.** As noted in Remark 1.2.3, the category of sets has coproducts. Moreover, in

Example 2.7.4 we have constructed coequalizers in the category of sets. Hence, by Construction 2.7.2 and Construction 2.7.5, the functor category from $\mathcal{C}(\mathfrak{Q})$ to **Sets** has coproducts and coequalizers. Thus, also the category of quivers **Quiv** has coproducts and coequalizers. If we unfold all the definitions, the coproduct of two quivers is given by the disjoint union of the quivers and the coequalizer of two morphisms of quivers is given by a quotient of quivers.

We could have constructed coproducts and coequalizers of quivers on paper without using a categorical tower. However, implementing disjoint unions and quotients of quivers together with functions on such structures on a computer would be tedious and error-prone: We would have to develop suitable data structures, and since quivers are given by two sets, we basically would have to implement all algorithms for sets twice, while ensuring consistency between the two parts. Hence, being able to reuse implementations of sets while the consistency is ensured by the functor category is a huge advantage.

By Example 1.2.6, we get:

Corollary 4.4.7. $\mathbf{Quiv}/S_{\mathbf{ZX}}$ *has pushouts.*

Remark 4.4.8 (Advantages of constructing the category of quivers as a tower)**.** In the preceding remarks, we have see all advantages of categorical towers mentioned in Remark 3.2.1:

- **Reusability**: We have seen that reusing algorithms for sets on a computer is a huge advantage.
- **Separation of concerns**: We have seen that separating the algorithms for sets from ensuring the consistency allows us to focus on one aspect at a time.
- **Verifiability**: We can verify the concrete algorithms for sets and the abstract algorithms for functor categories independently.
- **Emergence**: We have seen that we only need a data structure for quivers to encode $\mathfrak{Q}$, but thanks to the categorical tower automatically obtain data structures for morphisms and algorithms on objects and morphisms.

Chapter 5

A quantum computing application of a categorical tower

In this chapter, we have a look at an application of categorical towers in the field of quantum computing: Using a categorical tower, we can model a foundational functional programming language[1] for quantum computers. To this end, we proceed as follows: In Section 2.11, specifically in Example 2.11.5, Example 2.11.8, Remark 2.11.10, and Construction 2.11.12, we have seen correspondences between typed functional programming languages, typed generalized lambda calculi, categories of lambda terms, and closed monoidal categories. Hence, to define a functional **quantum** programming language, we introduce a closed monoidal category whose morphisms are generalized versions of **quantum circuits**, which in turn model computations on a quantum computer. This category is called the **category of ZX-diagrams**. The closed monoidal structure of the category of ZX-diagrams emerges naturally when modeling it as a categorical tower as described in [Cic18]. We follow the construction in [Cic18] but refine some details which are needed for making the tower fully algorithmic, as we will see in Remark 5.3.4.

For a short introduction to classical and quantum computations and circuits representing such computations see Section B.2. More details can, for example, be found in [JAA$^+$22] and [NC10].

The chapter is structured as follows: In Section 5.1, we introduce the **ZX-calculus**, a graphical language for describing quantum computations using **ZX-diagrams**. In Section 5.2, we show that ZX-diagrams form a category, which we model as a categorical tower in Section 5.3. In Section 5.4, we see how a rigid symmetric closed monoidal structures of the category of ZX-diagrams naturally emerges from the categorical tower. Finally, in Section 5.5, we use the correspondence between closed monoidal categories, categories of lambda terms, typed generalized lambda calculi, and functional programming languages to introduce a foundational functional programming language for quantum computers.

[1] See Remark 2.11.1 for some notes regarding the term “programming language”.

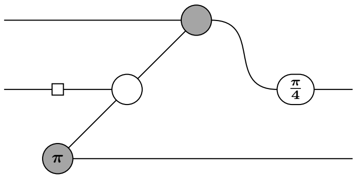


Figure 5.1: A ZX-diagram

5.1 The ZX-calculus

The **ZX-calculus** is a graphical language for describing quantum computations. It is visualized using **ZX-diagrams** like the one in Figure 5.1. As we will see, ZX-diagrams form a closed monoidal category. Using the connections indicated at the beginning of Section 2.11, the ZX-calculus can also be seen as a graphical interpretation of a so-called **linear type theory** describing quantum logic [See89, Pra92].

As a concrete application, quantum circuits can be turned into ZX-diagrams, which then can be simplified using the rules of the ZX-calculus. Afterwards, one can extract a quantum circuit from the resulting ZX-diagram which is often simpler than the original quantum circuit.

Definition 5.1.1 (ZX-diagrams)**. ZX-diagrams** are finite vertex-labeled multigraphs with five main types of nodes: inputs nodes, output nodes, Z-spiders, X-spiders, and Hadamard nodes. For an example of a ZX-diagram see Figure 5.1. Input and outputs nodes are drawn as dangling edges from the left and to the right, respectively. The order of input nodes matters, that is, the input nodes are formally labeled consecutively starting from 1, which is visualized by the order in which the dangling edges are drawn. The same holds for the output nodes. Z-spiders and X-spiders are represented by green (or light) and red (or dark) circles, respectively. Spiders are additionally labeled with a **phase** $\alpha \in [0, 2\pi)$, which is omitted for $\alpha = 0$ when drawing the graph. Hadamard nodes are represented by a (yellow) rectangle.

Input and output nodes are called **boundary nodes**, while all other nodes are called **inner nodes**.

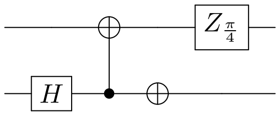
There are some restrictions on the **degree** of nodes, that is, on the number of edges connected to a node: Boundary nodes must have degree 1 and Hadamard nodes must have degree 2. Spiders can have arbitrary degree.

Example 5.1.2. A quantum circuit can be translated to a ZX-diagram as follows:

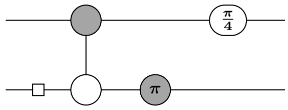
- a Hadamard gate is converted to a Hadamard node,
- a phase shift gate Z_α is converted to a Z-spider with phase α,
- a NOT gate is converted to an X-spider with phase π,
- a CNOT gate is written as a Z-spider on the controlling qubit connected to an X-spider on the target qubit, with both spiders having phase 0.

Since Hadamard gates, phase shift gates, and CNOT gates form a universal set of quantum gates, these rules allow to convert every quantum circuit to a

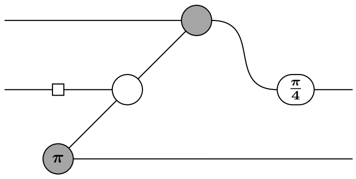
ZX-diagram. For example, the quantum circuit



can be written as a ZX-diagram as follows:



While we have retained the structure of the quantum circuit, most of this structure is not a formal part of the ZX-diagram. In particular, the "flow" from left to right and the mapping of gates to qubits is not encoded in the ZX-diagram and we could, for example, also draw the diagram as follows:



Of course drawing inputs and outputs as dangling edges from the left and to the right, respectively, still defines a general "direction" in the diagram, but this is only a convention and not part of the theory.

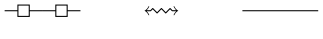
The process of translating a ZX-diagram to a quantum circuit is known as **circuit extraction**. Not every ZX-diagram can be translated to a quantum circuit, and even if one restricts to the cases where a translation exists in principle, it is speculated that there is no efficient general algorithm for circuit extraction, see [DKPvdW20, Section 8].

For more details about the ZX-calculus in general and in particular the extraction of quantum circuits from ZX-diagrams, see [DKPvdW20].

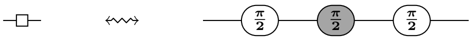
Remark 5.1.3 (Simplification rules for ZX-diagrams)**.** ZX-diagrams can be simplified using certain rules. If the ZX-diagram corresponds to a quantum circuit, then also the simplified ZX-diagram corresponds to a quantum circuit, and the simplification does not change the corresponding quantum function [DKPvdW20].

We list some of the simplifications rules for ZX-diagrams. For a complete list, see [DKPvdW20, Figure 1, Remark 2.5]. In the presentation of the rules, dangling edges do not necessarily have to correspond to boundary nodes but symbolize edges connected to arbitrary nodes. In particular, the diagrams in the rules do not have to be read from left to right but can appear in arbitrary orientation in larger diagrams. Moreover, while we call the rules "simplification rules", they can actually be applied in both directions.

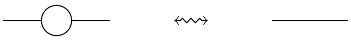
- Two adjacent Hadamard nodes can be canceled:



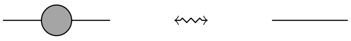
- A Hadamard node can be expanded to three spiders:



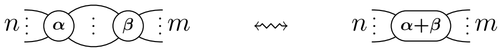
- A spider of degree 2 with phase 0 can be removed:



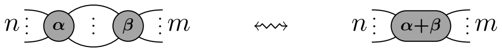
and



- Adjacent spiders of the same color can be fused, adding up their phases:

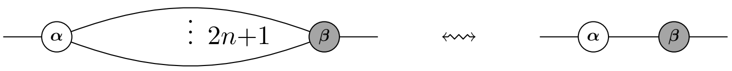


and

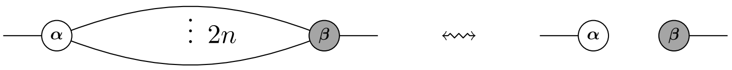


Phases are always reduced modulo 2π.

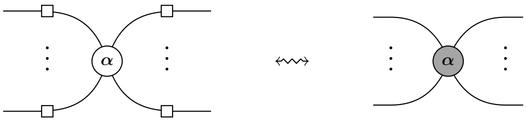
- An even number of edges between spiders of different colors cancels:



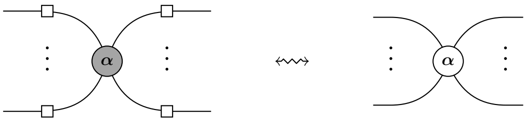
and



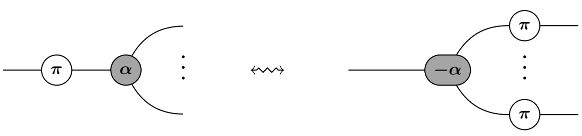
- A spider surrounded by Hadamard nodes can change its color by consuming the Hadamard nodes:



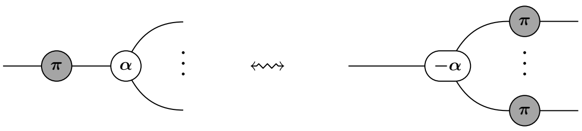
and



- A spider of degree 2 with phase π copies through an adjacent spider of a different color, flipping the phase of that spider:



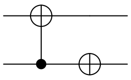
and



While this does not look like a simplification at first, it can be used together with other rules to actually simplify diagrams.

We first give a simple, detailed example showing how one can simplify quantum circuits using the ZX-calculus. Afterwards, we use the ZX-calculus to prove that the quantum circuit describing quantum teleportation can be simplified to an identity.

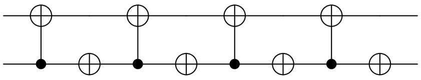
Example 5.1.4 (Simplification of a quantum circuit via the ZX-calculus)**.** We give an example of a simplification of a quantum circuit using ZX-diagrams. For this, we consider the circuit



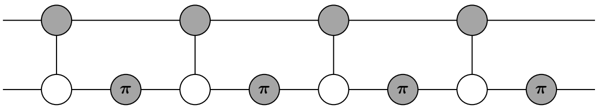
This circuit computes the following quantum function:

$$\begin{aligned}(\mathbb{C}^2)^{\otimes 2} &\to (\mathbb{C}^2)^{\otimes 2}\\ |00\rangle &\mapsto |01\rangle\\ |01\rangle &\mapsto |10\rangle\\ |10\rangle &\mapsto |11\rangle\\ |11\rangle &\mapsto |00\rangle\end{aligned}$$

As we see, the function just maps the first basis vector to the second, the second basis vector to the third, and so on. Hence, we expect that repeating the quantum circuit 4 times gives a quantum circuit which just computes the identity:

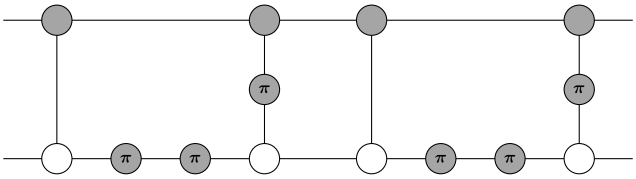


We check this using the ZX-calculus. First, we translate the quantum circuit to a ZX-diagram:

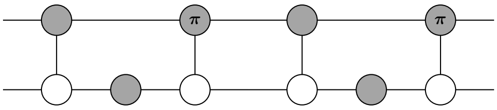


Then, we simplify the ZX-diagram according to the rules above:

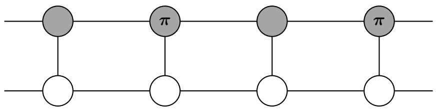
First, we copy the second and the fourth of the lower red spiders through the green spiders on their left:



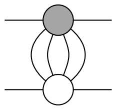
Next, we fuse some of the red spiders, reducing the phases modulo 2π:



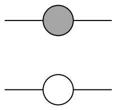
Spiders of degree 2 with phase 0 can be removed:



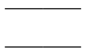
Now, we can fuse all the red spiders and the green spiders, respectively, again reducing the phases modulo 2π:



We can cancel the even number of edges between the red and the green spider:

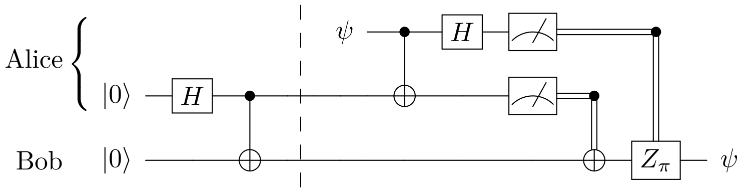


Finally, we can remove the spiders of degree 2 with phase 0 and obtain an ZX-diagram without nodes:

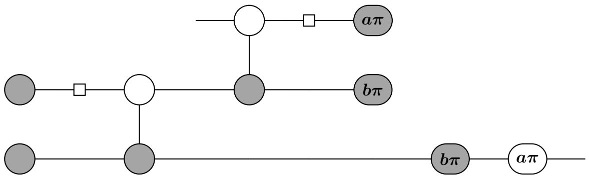


This ZX-diagram cannot be simplified anymore, so we extract a quantum circuit. Since the ZX-diagram has no inner nodes, the corresponding quantum circuit has no gates. Such a circuit does not modify the inputs at all and thus represents the identity, as expected. Hence, the simplification rules for ZX-diagrams indeed allowed us to simplify the circuit we started with.

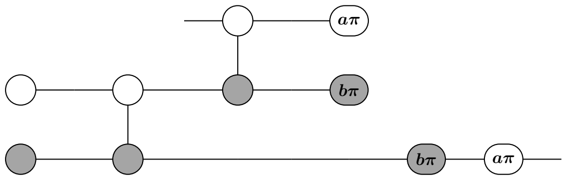
Example 5.1.5 (Quantum teleportation in the ZX-calculus)**.** We consider the circuit describing quantum teleportation in Example B.2.11:



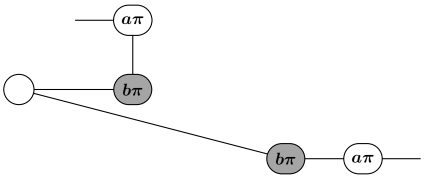
We can also encode the initialization of qubits and measurements in ZX-diagrams, which gives the following ZX-diagram:



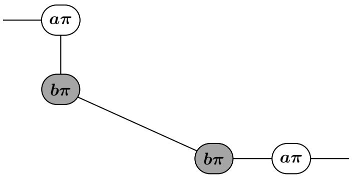
The parameters $a, b \in \{0, 1\}$ link the outcomes of the measurements to the application of the NOT gate and/or the phase shift gate. To simplify this ZX-diagram, we first change the colors of the dangling spiders connected to Hadamard nodes:



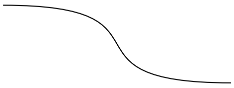
Afterwards, we fuse adjacent spiders of the same color:



We can remove the Z-spider with phase 0:



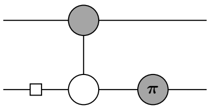
Now, for all possible values of $a, b \in \{0, 1\}$, the remaining spiders successively fuse do identities. Summing up, we get



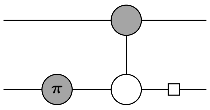
which corresponds to an identity. This shows that the input quantum state is indeed teleported without modification from Alice to Bob.

Remark 5.1.6 (Inverting ZX-diagrams)**.** Quantum circuits given by Hadamard gates, phase shift gates, and CNOT gates can be inverted by reversing the circuit and negating the phases of the phase shift gates. This property carries over to ZX-diagrams: ZX-diagrams coming from quantum circuits can be inverted by reversing the diagram, that is, turning inputs into outputs and vice versa, and negating all phases of spiders. In particular, if all spiders have phase $\alpha = 0$ or $\alpha = \pi$, reversing the diagram coincides with inverting it.

For example, composing the ZX-diagram



with the reversed ZX-diagram



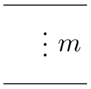
can easily be simplified to a ZX-diagram without nodes. Hence, as expected, the reversed diagram indeed describes the inverse because all spiders have phase $\alpha = 0$ or $\alpha = \pi$.

5.2 The category of ZX-diagrams

We now see how ZX-diagrams form a category. We give a direct definition of the category of ZX-diagrams in this section and model it as a categorical tower in the next section.

Definition 5.2.1 (The category of ZX-diagrams)**.** The **category ZX of ZX-diagrams** is defined as follows:

- its objects are natural numbers,
- a morphism from m to n is a ZX-diagrams with m inputs and n outputs, where we consider two ZX-diagrams as equal if they are isomorphic as multigraphs via an isomorphism preserving the labels,
- composition is given by glueing the outputs of the first diagram to the inputs of the second diagram,[2]
- the identity morphism on m is given by the ZX-diagram with m input nodes and m output nodes, connected by m parallel edges:



One can easily check that this indeed defines a category.

Definition 5.2.2 (The category of ZX-diagrams with relations)**.** The **category $\mathbf{ZX}/{\sim}$ of ZX-diagrams with relations** is defined as the quotient of **ZX** by the simplification rules of the ZX-calculus, that is, we construct $\mathbf{ZX}/{\sim}$ in the same way as **ZX** but view two morphisms as equal if they are equal up to the simplification rules of the ZX-calculus. The equality up to the simplification rules is compatible with the composition, so this indeed defines a category.

Remark 5.2.3. We will see that the category of ZX-diagrams **without** relations is sufficient for our aim of modeling a foundational functional quantum programming language. Hence, we only define $\mathbf{ZX}/{\sim}$ for the sake of completeness.

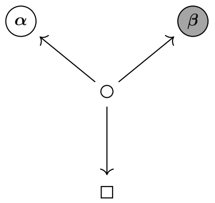
5.3 The category of ZX-diagrams as a categorical tower

In this section, we show how the category of ZX-diagrams can be constructed as a categorical tower. We follow the construction in [Cic18] but refine some details, which we will see in Remark 5.3.4. The category constructors used in the construction of the tower are implemented in various packages in the `CAP` ecosystem and combined in the package `ZXCalculusForCAP` [Zic24j].

For building the categorical tower, we proceed in two steps: First, we consider a slice category of **Quiv**, which we can view as a category of **decorated quivers**. Afterwards, we pass to a category of cospans, which allows us to formalize the glueing of outputs and inputs in the composition of morphisms of **ZX**.

Construction 5.3.1 (Decorated quivers)**.** Quivers are given by two sets and morphisms of quivers are given by two maps on these sets, so we naturally have notions of “elements”, “fibers”, and “images” (of elements) in **Quiv** compatible with the interpretations in Remark 2.9.2. Hence, we can think of a morphism $\boxed{X} \to \boxed{B}$ in **Quiv** as a quiver X whose vertices and arrows are decorated by the vertices and arrows of B, respectively.

[2] Formally, we glue the output nodes of the first diagram to the input nodes of the second diagram, which gives a node of degree 2. We then remove this node and merge the two edges connected to it into a single edge.



where α and β run through $[0, 2\pi)$

Figure 5.2: The quiver $S_{\mathbf{ZX}}$

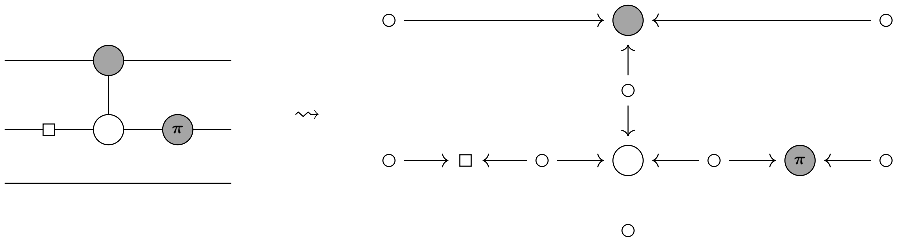


Figure 5.3: Encoding of a ZX-diagram as a quiver decorated over $S_{\mathbf{ZX}}$

As a preparation for modeling the category of ZX-diagrams as a categorical tower, we show how to encode ZX-diagrams as decorated quivers:

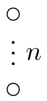
Construction 5.3.2 (Encoding ZX-diagrams as decorated quivers)**.** Let $S_{\mathbf{ZX}}$ be the quiver in Figure 5.2, that is, the quiver with vertices

$$\begin{array}{ll} \textcircled{\alpha} & \text{for } \alpha \in [0, 2\pi), \\ \textcircled{\beta} & \text{for } \beta \in [0, 2\pi), \\ \square & \\ \circ & \end{array}$$

where the so-called **neutral vertex** $\circ$ connects to all other vertices by an arrow. We choose $S_{\mathbf{ZX}}$ as the base quiver for forming the slice category $\mathbf{Quiv}/S_{\mathbf{ZX}}$, that is, the category of quivers decorated by $S_{\mathbf{ZX}}$. Now, we want to encode a ZX-diagram D as a quiver Q_D decorated by $S_{\mathbf{ZX}}$. An example showing how we do this can be seen in Figure 5.3. Specifically, we use following rules:

- Boundary nodes of D are encoded as neutrally decorated vertices of Q_D.
- Inner nodes of D are encoded as vertices of Q_D decorated in the obvious way.
- An edge connecting a boundary node and an inner node of D is encoded as an arrow of Q_D pointing away from the corresponding neutrally decorated vertex.
- An edge connecting two inner nodes of D is encoded as two arrows of Q_D pointing away from a neutrally decorated vertex which is newly added.
- An edge connecting two boundary nodes of D is handled in a special way: The edge together with the two boundary nodes is encoded as a single neutrally decorated vertex of Q_D.

Finally, we also associate a decorated quiver Q_n to natural numbers n, whose role will become clear in the next construction: We define Q_n as a quiver with n neutrally decorated vertices, which can be visualized as follows:



With this, we can model the category of ZX-diagrams as a category of cospans:

Construction 5.3.3 (The category of ZX-diagrams as a categorical tower)**.** We use the notation of Construction 5.3.2. We define an embedding $\mathcal{R}$ of $\mathbf{ZX}$ into $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$ as follows:

$$\begin{aligned} \mathcal{R} : \mathbf{ZX} &\to \mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}}), \\ \boxed{n} &\mapsto \boxed{Q_n}, \\ \boxed{m} \xrightarrow{\boxed{D}} \boxed{n} &\mapsto \boxed{Q_m \to Q_D \leftarrow Q_n}, \end{aligned}$$

where the morphisms of the cospan map the i-th vertex of Q_m (respectively Q_n) to the vertex of Q_D coming from the i-th input (respectively output) of D. With this, $\mathcal{R}$ defines an isomorphism to a subcategory of $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$. Hence, we can model $\mathbf{ZX}$ as a reinterpretation of a subcategory of the categorical tower $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$ via $\mathcal{R}$. The subcategory can be characterized as follows: Its only objects are $\boxed{Q_n}$ for natural numbers n. Its morphisms are only those cospans where the central object is given by a decorated quiver whose vertices fulfill the degree restrictions in the definition of ZX-diagrams, that is:

- a vertex decorated as "Hadamard" must have degree 2,
- the vertices representing inputs or outputs must have degree 1, with the degenerate case that a vertex might respresent two boundary nodes at the same time and has degree 0.

Proof of correctness. First, we show that $\mathcal{R}$ is compatible with the equalities on morphisms and injective on morphisms. We spell out the equalities in $\mathbf{ZX}$ and $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$: Two morphisms $\boxed{D}, \boxed{E} : \boxed{m} \to \boxed{n}$ in $\mathbf{ZX}$ are equal if and only if the ZX-diagrams D and E are isomorphic as multigraphs via an isomorphism preserving the labels. Moreover, the two morphisms

$$\boxed{Q_m \to Q_D \leftarrow Q_n} \quad \text{and} \quad \boxed{Q_m \to Q_E \leftarrow Q_n}$$

in $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$ are equal if and only if there exists an isomorphism of decorated quivers which makes the triangles in the following diagram commute:

$$\begin{tikzcd} & Q_D \arrow[dd, dashed, "f"] & \\ Q_m \arrow[ur] \arrow[dr] & & Q_n \arrow[ul] \arrow[dl] \\ & Q_E & \end{tikzcd}$$

Since Q_m and Q_n are mapped to the inputs and outputs of Q_D and Q_E, the triangles in the diagram commute if and only if f preserves inputs and outputs, including their order. Moreover, an isomorphism of decorated quivers is just an isomorphism of quivers preserving decorations, in particular Hadamard nodes and spiders including their colors and phases. Furthermore, we have normalized the directions of the arrows to always point away from neutrally decorated vertices. Hence, there is no difference between isomorphisms of quivers, that is, **directed** multigraphs, and isomorphisms of **undirected** multigraphs in this context. Summing up, $\boxed{D}$ and $\boxed{E}$ are equal in **ZX** if and only if

$$\boxed{Q_m \to Q_D \leftarrow Q_n} \qquad \text{and} \qquad \boxed{Q_m \to Q_E \leftarrow Q_n}$$

are equal in $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$. Hence, $\mathcal{R}$ is compatible with the equality on morphisms and injective on morphisms. Moreover, we can recover the number m from Q_m by counting the vertices, so $\mathcal{R}$ is also injective on objects.

Next, we consider the composition. Let $\boxed{D} : \boxed{m} \to \boxed{n}$ and $\boxed{E} : \boxed{n} \to \boxed{\ell}$ be two morphisms in **ZX**. To compose

$$\boxed{Q_m \to D \leftarrow Q_n} \qquad \text{and} \qquad \boxed{Q_n \to E \leftarrow Q_\ell}$$

in $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$, we form the pushout

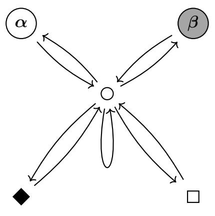
$$\begin{tikzcd}
& Q_n \arrow[ld] \arrow[rd] & \\
D \arrow[rd] & & E \arrow[ld] \\
& P &
\end{tikzcd}$$

This is a formalization of glueing the images of Q_n in D and E, that is, the outputs of D and the inputs of E. Hence, the composition in $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$ and the composition in **ZX** coincide.

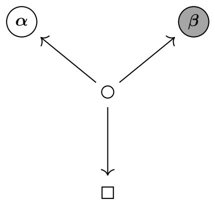
Finally, it is easy to see that the identities in **ZX** map to the identities in $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$.

Summing up, we have shown that $\mathcal{R}$ is an embedding. ■

Remark 5.3.4 (Deviations from [Cic18])**.** We deviate in various points from [Cic18], which can be seen by comparing the decorating quiver used there to our decorating quiver:



compared to

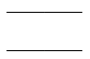


We look at the differences:

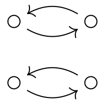
- In the decorating quiver on the left, we see an additional kind of node: the so-called **diamond node**. This is due to [Cic18] using a presentation of the ZX-calculus accounting for so-called **scalar factors**, see [DKPvdW20, Remark 2.2]. We could easily include additional kinds of nodes if needed by simply adding them to our decorating quiver.

- For every arrow in the decorating quiver on the left there also exists a reversed arrow. Probably the intention in [Cic18] is to model undirected graphs as directed graphs by encoding every undirected edge by two directed arrows covering both possible directions. However, this is not actually done in the paper and it seems like the problem is not properly addressed. We avoid this problem by simply normalizing the direction of arrows to always point away from neutrally decorated vertices.
- In the decorating quiver on the left, the neutral vertex has a self-loop. To see the significance of this loop, recall the following special case in the construction of the directed quiver Q_D correponding to a ZX-diagram D in Construction 5.3.2: An edge connecting two boundary nodes of D is handled in a special way: The edge together with the two boundary nodes is encoded as a single neutrally decorated vertex of Q_D.

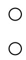
  If we would endow the neutral vertex with a self-loop, we could avoid this special case and instead encode an edge connecting two boundary nodes in D by an actual arrow in Q_D. For example, we could encode the ZX-diagram D given as

  as a decorated quiver Q'_D as follows:

  However, with this, $\mathcal{R}$ would not map identities of **ZX** to identities of **Csp**(**Quiv**/$S_{\mathbf{ZX}}$) anymore: The diagram D above is just the diagram defining $\mathrm{id}_{\boxed{2}}$ in **ZX**. However, Q'_D is not isomorphic to

  which is the central object of the cospan defining the identity on $\mathcal{R}(\boxed{2})$ in **Csp**(**Quiv**/$S_{\mathbf{ZX}}$):

  $$Q_2 \xrightarrow{\mathrm{id}_{Q_2}} Q_2 \xleftarrow{\mathrm{id}_{Q_2}} Q_2.$$

  To solve this problem, [Cic18] passes to a quotient of **Csp**(**Quiv**/$S_{\mathbf{ZX}}$) where the images of the identity morphisms of **ZX** are identified with the corresponding identity morphisms of the tower. However, working with such a quotient in a computer implementation is cumbersome. Hence, we prefer to solve the problem right in the encoding process.

Remark 5.3.5 (Advantages of constructing the category of ZX-diagrams as a categorical tower)**.** We again see some of the advantages of categorical towers mentioned in Remark 3.2.1: At first, modeling the category of ZX-diagrams as a categorical tower seems to be much more cumbersome than just working with the category of ZX-diagrams as in the original definition. This is true as long as one stays on a purely visual level. However, as soon as one wants to implement the category on a computer, one quickly notices that defining a suitable data structure for morphisms and formalizing the notion of "glueing" is difficult. When working with the categorical tower, though, the data structures and formalizations arise quite naturally. This is a perfect example of **emergence**.

Moreover, the theory of functor categories, which we use to model quivers, and slice categories is well developed, so we can actually build on existing theory for the mathematics and on existing implementations on the computer. We will see this in the next section, where we construct a closed monoidal structure on the category of ZX-diagrams via the tower $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$. This is a nice example of **reusability**.

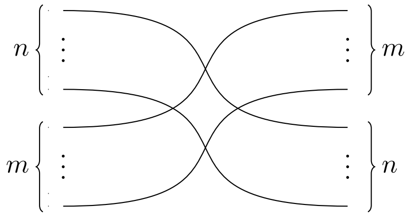
Remark 5.3.6 (Implementation of the tower $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$)**.** The decorating quiver $S_{\mathbf{ZX}}$ has infinitely many vertices which complicates the implementation on a computer. However, since ZX-diagrams are finite, only finitely many decorations appear in a given computation. Hence, a possible solution for a computer implementation is to initialize $S_{\mathbf{ZX}}$ with only the neutral vertex at the beginning and afterwards extend it by new vertices (and corresponding arrows) on demand.

5.4 A closed monoidal structure on the category of ZX-diagrams

In this section, we see how a rigid symmetric closed monoidal structure of the category of ZX-diagrams naturally emerges from the categorical tower by lifting the cocartesian monoidal structure of the category of quivers to a rigid symmetric monoidal structure of the tower $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$.

Construction 5.4.1 (A strict rigid symmetric monoidal structure on the category of ZX-diagrams, [Cic18])**.** We can define a strict rigid symmetric monoidal structure on the category of ZX-diagrams $\mathbf{ZX}$ as follows:

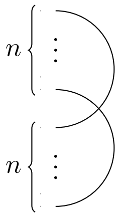
- the tensor product of two objects is just their sum,
- the tensor product of two morphisms stacks the diagrams,
- the tensor unit is the object $\boxed{0}$.
- The braiding from $\boxed{n} \otimes \boxed{m}$ to $\boxed{m} \otimes \boxed{n}$ connects the first n inputs to the last n outputs and the last m inputs to the first m outputs:



- For an object $\boxed{n}$, we set $\boxed{n}^* := \boxed{n}$, that is, objects are self-dual.
- For an object $\boxed{n}$, we define the evaluation

$$\varepsilon_{\boxed{n}} : \boxed{n+n} \equiv \boxed{n}^* \otimes \boxed{n} \to 1 \equiv \boxed{0}$$

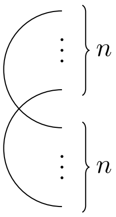
by connecting the i-th input to the $n+i$-th input for all $i \in \{1, \ldots, n\}$:



- For an object $\boxed{n}$, we define the coevaluation

$$\eta_{\boxed{n}} : \boxed{0} \equiv 1 \to \boxed{n} \otimes \boxed{n}^* \equiv \boxed{n+n}$$

  by connecting the i-th output to the $n+i$-th output for all $i \in \{1, \dots, n\}$:



Proof of correctness. We obtain a rigid symmetric monoidal structure of the categorical tower $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$ as follows:

- the category of quivers **Quiv** has coproducts and coequalizers, see Remark 4.4.6,
- hence, the slice category $\mathbf{Quiv}/S_{\mathbf{ZX}}$ has coproducts and coequalizers by Construction 2.9.3 and Construction 2.9.4,
- by Example 1.2.6, $\mathbf{Quiv}/S_{\mathbf{ZX}}$ has binary pushouts,
- the coproducts turn $\mathbf{Quiv}/S_{\mathbf{ZX}}$ into a cocartesian monoidal category, see Definition 2.8.5,
- hence, $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$ has a rigid symmetric monoidal structure by Construction 2.10.2.

Using Construction 5.3.3 and Construction 3.3.5, we can turn this rigid symmetric monoidal structure into a rigid symmetric monoidal structure of the category of ZX-diagrams **ZX**. When modeling **Quiv** as a functor category into a version of the category of sets with strict coproducts, the strictness is preserved throughout the tower. In particular, stacking diagrams will be associative and stacking a morphism with the identity on the tensor unit $\boxed{0}$ will do nothing, just as on paper.

Unfolding the definitions of the tower can be automated with the help of `CompilerForCAP` [Zic24a], as we will see in Chapter 6. ■

We highlight a situation which is implicit in the previous construction:

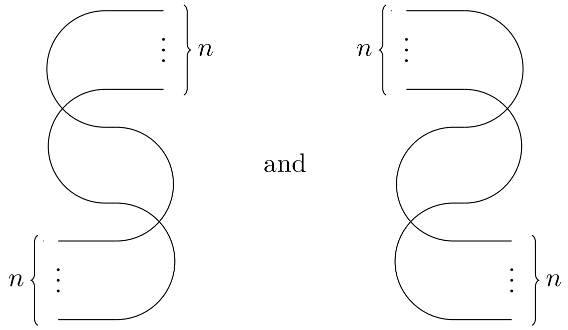
Remark 5.4.2 (Yanking)**.** Recall that the definition of the rigid structure contains two commutativity conditions: For every object A, the following two diagrams have to commute:

$$\begin{tikzcd}
(A \otimes A^*) \otimes A \arrow[d, "\alpha_{A,A^*,A}"'] & 1 \otimes A \arrow[l, "\eta_A \otimes \mathrm{id}_A"'] \arrow[d, "\lambda_A"] \\
 & A \\
A \otimes (A^* \otimes A) \arrow[r, "\mathrm{id}_A \otimes \varepsilon_A"'] & A \otimes 1 \arrow[u, "\rho_A"']
\end{tikzcd}
\quad \text{and} \quad
\begin{tikzcd}
A^* \otimes (A \otimes A^*) \arrow[d, "\alpha^{-1}_{A^*,A,A^*}"'] & A^* \otimes 1 \arrow[l, "\mathrm{id}_{A^*} \otimes \eta_A"'] \arrow[d, "\rho_{A^*}"] \\
 & A^* \\
(A^* \otimes A) \otimes A^* \arrow[r, "\varepsilon_A \otimes \mathrm{id}_{A^*}"'] & 1 \otimes A^* \arrow[u, "\lambda_{A^*}"']
\end{tikzcd}$$

In a strict setup, these two diagrams can be simplified to the following two equations:

$$(\eta_A \otimes \mathrm{id}_A) \cdot (\mathrm{id}_A \otimes \varepsilon_A) = \mathrm{id}_A \qquad \text{and} \qquad (\mathrm{id}_{A^*} \otimes \eta_A) \cdot (\varepsilon_A \otimes \mathrm{id}_{A^*}) = \mathrm{id}_{A^*}.$$

If we express these equations for $A := \boxed{n}$ in **ZX**, we get that the following two ZX-diagrams must be equal to identity diagrams:



In **ZX**, this statement is trivial: Just as the identities, both diagrams connect the i-th input to the i-th output and have no additional nodes, so they must be equal to the identities. However, when transporting the constituents of the diagrams to the categorical tower $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$, a non-trivial proof is required for showing that the cospans actually glue to an identity cospan.[3] In this sense, the ZX-calculus can also be seen as a language for doing visual proofs. This particular case of a proof which "pulls bent edges tight" is called **yanking**.

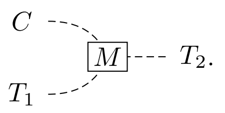
Using Construction 2.8.7, we can construct a closed structure from the rigid structure:

Construction 5.4.3 (A strict closed monoidal structure on the category of ZX-diagrams)**.** We obtain a closed structure of the strict monoidal category **ZX** as follows:

We define the internal hom of two objects $\boxed{T_1}$ and $\boxed{T_2}$ in **ZX** as

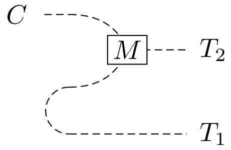
$$\mathrm{hom}(\boxed{T_1}, \boxed{T_2}) :\equiv \boxed{T_2 + T_1}.$$

Then, let $\boxed{M} : \boxed{C} \otimes \boxed{T_1} \to \boxed{T_2}$ be a morphism, which we visualize as follows:



[3] This proof was an implicit part of Construction 5.4.1, arising from the proof of Construction 2.10.2 applied to the concrete situation of decorated quivers.

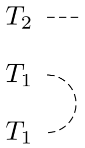
We use dashed lines to illustrate that a line represents multiple inputs and outputs of a given ZX-diagram like M at once. Now, we define the corresponding morphism $\boxed{C} \to \hom(\boxed{T_1}, \boxed{T_2}) \equiv \boxed{T_2 + T_1}$ via the following diagram:



Finally, the evaluation morphism

$$\mathrm{ev}_{\boxed{T_1}, \boxed{T_2}} : \boxed{T_2 + T_1 + T_1} \to \boxed{T_2}$$

is defined via the following diagram:



Proof of correctness. By Construction 5.4.1, the category of ZX-diagrams is a strict rigid symmetric monoidal category. Hence, we can apply Construction 2.8.7 and obtain the above closed structure. ∎

5.5 A foundational functional quantum programming language

The category of ZX-diagrams is a strict closed monoidal category and thus defines a typed generalized lambda calculus by Construction 2.11.12 and Remark 2.11.10. For understanding the semantics of the corresponding foundational functional programming language[4], we now take a closer look at this typed generalized lambda calculus.

Construction 5.5.1 (The typed generalized lambda calculus **ZX**)**.** We interpret **ZX** as a typed generalized lambda calculus as in Construction 2.11.12 and Remark 2.11.10.

The types are objects of **ZX**, that is, natural numbers. These numbers correspond to the number of inputs and outputs of ZX-diagrams. If the ZX-diagram corresponds to a quantum circuit and in particular has the same number of inputs and outputs, this number is the number of qubits on which the quantum circuit acts. Hence, in this case the types correspond to a number of qubits, just like, for example, the default type of integers on a classical 64-bit computer usually corresponds to 64 classical bits.

Function types are given by the internal hom. Using Construction 5.4.3, we see that

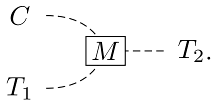
$$\hom(\boxed{T_1}, \boxed{T_2}) \equiv \boxed{T_1 + T_2}.$$

Using the above interpretation of types as numbers of qubits, this means that a function on T qubits can be encoded using $2T$ qubits.

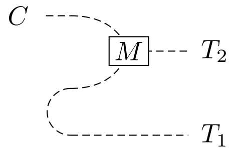
[4] See Remark 2.11.1 for some notes regarding the term “programming language”.

The lambda terms of type $\boxed{T}$ in a context $(x_1 : \boxed{T_1}, \dots, x_n : \boxed{T_n})$ are morphisms $\boxed{T_1} \otimes \dots \otimes \boxed{T_n} \to \boxed{T}$, that is, ZX-diagrams with $\sum_i T_i$ inputs and T outputs. In particular, a lambda term of type $\boxed{T}$ in a context $(x : \boxed{T})$ potentially corresponds to a quantum functions on T qubits.

For the abstraction, consider a lambda term $\boxed{M}$ of type $\boxed{T_2}$ in a context $(\dots, x : \boxed{T_1})$, that is, a morphism $\boxed{C} \otimes \boxed{T_1} \to \boxed{T_2}$ given by a ZX-diagram M with $C + T_1$ inputs and T_2 outputs, which we visualize as follows:



As seen in Construction 5.4.3, the abstraction $\lambda_{T_1}\boxed{M} : \boxed{C} \to \boxed{T_2 + T_1}$ can be visualized as follows:

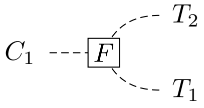


That is, the last T_1 inputs are "bent" to the right to form outputs.

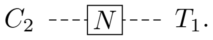
For the application, consider a lambda term $\boxed{F}$ of type $\boxed{T_1} \Rightarrow \boxed{T_2}$ and a lambda term $\boxed{N}$ of type T_1, that is, two morphisms

$$\boxed{F} : \boxed{C_1} \to \boxed{T_2 + T_1} \qquad \text{and} \qquad \boxed{N} : \boxed{C_2} \to \boxed{T_1}.$$

We can visualize this setup as follows:



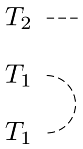
and



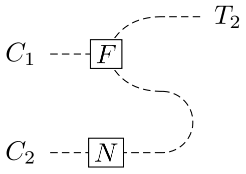
The application $\boxed{F}\boxed{N} : \boxed{C_1 + C_2} \to \boxed{T_2}$ is formed by tensoring the two morphisms and composing with the evaluation morphism

$$\mathrm{ev}_{\boxed{T_1},\boxed{T_2}} : \boxed{T_2 + T_1 + T_1} \to \boxed{T_2}$$

given by

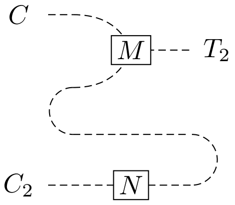


Hence, we can visualize $\boxed{F}\boxed{N}$ as follows:

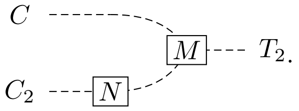


Now, if $\boxed{F}$ is an abstraction $\lambda_{T_1}\boxed{M}$ as above,[5] we get:

[5] Note that this is exactly the situation where β-reduction can be applied.



that is,[6]



If the ZX-diagrams M and N correspond to quantum circuits C_M and C_N which compute quantum functions F_M and F_N, this diagram corresponds to a quantum circuit which applies F_M to the outputs of F_N (and possible some other arguments). Hence, our notion of "application" actually corresponds the application of quantum functions.

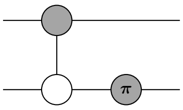
Summing up, we have seen that if the ZX-diagrams correspond to quantum circuits, we indeed get the semantics expected for a quantum programming language:

the typed generalized lambda calculus defined by ZX	**interpretation as a quantum programming language**
a type $\boxed{T}$	T qubits
a lambda term of type $\boxed{T}$ in a context $(x : \boxed{T})$	a quantum function on T qubits
application of lambda terms	application of quantum functions

Remark 5.5.2. The monoidal structure of **ZX** is not cartesian monoidal, so we do not get the additional formation rules in Remark 2.11.14. In fact, in the quantum setup this situation is expected (see [BS11, end of Section 2.2.3]): A projection $T_1 \times \cdots \times T_n \to T_i$ could be interpreted as forgetting information about T_j for $j \neq i$, which would contradict the **no-deletion theorem** of quantum information theory. Similarly, the diagonal morphism $T \to T \times T$ obtained via the universal morphism could be interpreted as duplicating information, which would contradict the **no-cloning theorem** of quantum information theory.

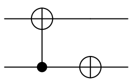
We show how programs in the functional programming language modeled by **ZX** could look like.

Example 5.5.3 (Functional quantum programs)**.** We start by introducing some constants, called **library functions**, into our programming language. For example, we consider the ZX-diagram



[6] See Remark 5.4.2 for why the two ZX-diagrams are trivially equal but why the corresponding transformation in the tower $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$ requires a non-trivial proof.

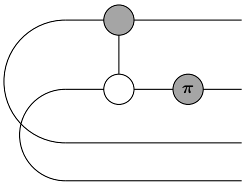
This ZX-diagram corresponds to the quantum circuit



which encodes the quantum function

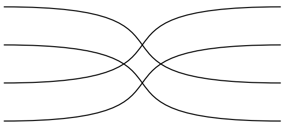
$$
\begin{aligned}
|00\rangle &\mapsto |01\rangle \\
|01\rangle &\mapsto |10\rangle \\
|10\rangle &\mapsto |11\rangle \\
|11\rangle &\mapsto |00\rangle
\end{aligned}
$$

We interpret the basis vectors as the binary representations of the numbers 0, 1, 2, and 3, just as we would with classical bit strings. Then the above function encodes the successor function on $\mathbb{Z}/4\mathbb{Z}$ given by two qubits, corresponding to the type $\boxed{2}$. The corresponding ZX-diagram is a morphism $\boxed{D} : \boxed{2} \to \boxed{2}$ in **ZX** and hence a lambda term in the context $(x : \boxed{2})$ with one variable. We define $\mathtt{succ} := \lambda_{\boxed{2}}\boxed{D} : \boxed{0} \to \boxed{2} \Rightarrow \boxed{2}$, which is given by the following diagram:



That is, `succ` is a constant of the function type $\boxed{2} \Rightarrow \boxed{2}$.

Furthermore, we consider the following ZX-diagram:



This encodes a higher-order function: If we compose a constant of type $\boxed{2} \Rightarrow \boxed{2}$, for example `succ` above, with this ZX-diagram, we effectively swap the inputs and the outputs, thus reversing the diagram. We have seen in Remark 5.1.6 that reversing a diagram gives the inverse if all phases appearing in the diagram are integral multiples of π. Again, we interpret this ZX-diagram as a constant $\mathtt{reverse} : \boxed{0} \to (\boxed{2} \Rightarrow \boxed{2}) \Rightarrow (\boxed{2} \Rightarrow \boxed{2})$.

We call constants as the ones defined above **library functions**, from which we now build programs in an ad-hoc syntax. We start with the following program:

```
main : 2 -> 2
main x = succ (succ x)
```

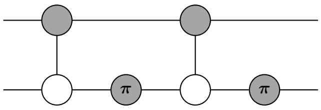
We go through this program step by step and show how it can be realized in the typed generalized lambda calculus given by the category of ZX-diagrams. The first line simply declares that the main program is a lambda term of type $\boxed{2}$ in a context corresponding to $\boxed{2}$. The second line explicitly constructs this lambda term in the context $(\mathtt{x} : \boxed{2})$:

We start with the innermost expression `x` on the right-hand side of the equality sign. This is just the free variable `x` of type $\boxed{2}$ in the context $(\mathtt{x} : \boxed{2})$.

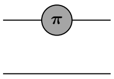
The next expression `succ x` is the application of `succ` to `x`, which is a lambda term of type $\boxed{2}$ in the context $(\mathtt{x} : \boxed{2})$. The final expression `succ (succ x)` is the application of `succ` to `succ x`, which again is a lambda term of type $\boxed{2}$ in the context $(\mathtt{x} : \boxed{2})$. The program is given by this final expression.

Regarding the semantics, we expect the program to apply the successor function on $\mathbb{Z}/4\mathbb{Z}$ twice, that is, we expect the program to add 2.

All constructions of lambda terms above correspond to constructions in the category of ZX-diagrams. If we perform those constructions, we see that the program is given by the following ZX-diagram:



Using the simplification rules for ZX-diagrams, we can simplify this to



which corresponds to a NOT gate on the first qubit, which indeed corresponds to adding 2.

Next, we consider the following program:

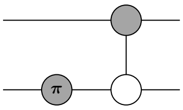
```
main : 2 -> 2
main x = pred (pred x)
            where pred = reverse succ
```

Again, the main program is a lambda term of type $\boxed{2}$ in the context $(\mathtt{x} : \boxed{2})$. Here, we use a named variable `pred`, which is just a syntactic construct: Equivalently, we could have written

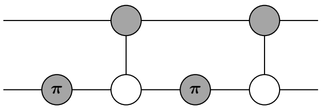
```
main : 2 -> 2
main x = (reverse succ) ((reverse succ) x)
```

We first create the lambda term `pred` $\coloneqq$ `reverse succ` as the application of `reverse` to `succ`, which is a constant of type $\boxed{2} \Rightarrow \boxed{2}$. Afterwards, we apply `pred` to the free variable `x` twice, similar to the applications in the first program. The resulting lambda term forms the program.

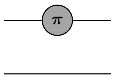
All phases of spiders in the ZX-diagram representing `succ` are integral multiples of π, so we expect `pred` to actually be the inverse of `succ`, that is, to compute the predecessor. Moreover, we expect the whole program to subtract 2 (which modulo 4 is equivalent to adding 2). Indeed, `pred x` is given by the ZX-diagram



which computes the predecessor. Moreover, the whole program is given by the ZX-diagram



which again can easily be simplified to



which in turn corresponds to subtracting (or adding) 2 as expected.

In the last program, we have first reversed `succ` and then applied it twice. Now, we first form a function which applies `succ` twice, and then revert this function:

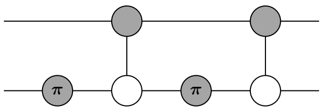
```
main : 2 -> 2
main x = (reverse ((y : 2) -> succ (succ y))) x
```

Again, the main program is a lambda term of type $\boxed{2}$ in the context (`x` : $\boxed{2}$). We first look at the expression

`(y : 2) -> succ (succ y).`

This is an anonymous function in one variable `y` of type $\boxed{2}$. To create the body of this anonymous function, we consider a context (`y` : $\boxed{2}$) having `y` as a free variable. In this context, we can build the lambda term `succ (succ y)` of type $\boxed{2}$. Then, we can form `(y : 2) -> succ (succ y)` as the abstraction of `succ (succ y)` with regard to `y`, giving us a constant of type $\boxed{2} \Rightarrow \boxed{2}$. Finally, we can apply `reverse` to this abstraction and then apply the result to `x`. The resulting lambda term forms the program.

Taking the inverse commutes with composition, so we expect this program to ultimately perform the same computation as the previous program. Indeed, performing all the constructions in the category of ZX-diagrams, we get the ZX-diagram



which is the same diagram as we got for the previous program.

Construction 5.5.4 (A universal functional quantum programming language)**.** Summing up, we have actually created a functional quantum programming language:

We can import existing quantum circuits as ZX-diagrams and view them as constants, which we call **library functions**. We can then write typed programs using these library functions, (free) variables, and abstraction and application of (higher-order) functions.

Such a program is represented by a ZX-diagram. If we can extract a quantum circuit from the diagram, we can hence execute the program on a quantum computer. Moreover, the semantics of lambda terms and application of lambda terms in our programs actually corresponds to quantum functions and application of quantum functions. Hence, the quantum circuit extracted from the ZX-diagram representing our program will actually compute what we formulated abstractly in our program. In particular, we actually get a semantics of higher-order quantum functions, as one would expect in a functional quantum programming language.

Assuming that we include the ZX-diagrams corresponding to a universal set of quantum gates, we can create every quantum circuit in the programming language. Hence, every quantum function can be modeled in the programming language, that is, the programming language is a **universal** functional quantum programming language.

Chapter 6

Using CompilerForCAP for code optimization

In this chapter, we introduce CompilerForCAP [Zic24a] and see how it can be used for optimizing algorithms in categorical towers. We have already seen the need for a compiler in Section 3.4: Computations in categorical towers naturally come with a significant performance overhead, which is why formerly in many cases large computations in categorical towers were not feasible on a computer. As we will see in this chapter, by using CompilerForCAP we can avoid this performance overhead, which allows us to make full use of the advantages of building categorical towers on a computer.

This chapter is structured as follows: In Section 6.1, we see an example showing how the compilation process is carried out by CompilerForCAP. In Section 6.2, we look at benchmarks showing the performance gains for the two main towers in Chapter 5 and Chapter 4: For an implementation of the category of ZX-diagrams, which at the time of writing has not been optimized for use with CompilerForCAP yet, we already get an improvement by more than a factor of 5. For an implementation of the category of finitely presented right modules over an exterior algebra, which uses CompilerForCAP to its full potential, we can get an arbitrarily large improvement for large examples and see that CompilerForCAP can make the difference between "finishes in seconds" and "will never finish". In Section 6.3, we describe various situations in which CompilerForCAP can optimize code during compilation, and also explain the corresponding compiler techniques. In Section 6.4, we compare CompilerForCAP to other compilers to see why developing a special compiler was necessary. In Section 6.5, we see two examples of how CompilerForCAP can be used for generating categorical code. The examples show that **reusability** is not merely a theoretical advantage of categorical towers but actually allows us to write less code overall. Finally, in Section 6.6 we conclude by summarizing the advantages of building categorical towers on a computer which are now accessible thanks to CompilerForCAP.

6.1 Introduction to CompilerForCAP

We first clarify some terminology regarding compilers. For more general background on compilers, see, for example, [ALSU06].

Remark 6.1.1 (Terminology)**.** Generally speaking, a **compiler** is a computer program which translates a computer program written in one language (the **source** language) into an equivalent[1] program in another (or the same) language (the **target** language). More specifically, the term is often used when translating from higher-level languages like C to lower-level languages like assembly languages. A program which translates between languages of similar level is then called a **source-to-source** translator.

In our context, we distinguish between CAP code and GAP code:[2] CAP code extensively uses the categorical interfaces but only uses some very basic GAP functionality, for example for the manipulation of lists. GAP code uses no categorical interfaces but uses advanced functionality of GAP itself and other GAP packages, for example the operation RREF which computes a reduced row echelon form of a matrix. The categorical interfaces are usually more abstract than the GAP interfaces, so we view CAP code as a higher-level code than GAP code.

CompilerForCAP typically acts on CAP code and, depending on the application, outputs CAP code or GAP code. Hence, CompilerForCAP can act either as a compiler or as a source-to-source translator, though the distinction between the two cases is purely semantic: The underlying compiler techniques are exactly the same in both cases.

To see how CompilerForCAP simplifies implementations, we look at a simple but instructive example:

Example 6.1.2. We look at an algorithm computing the fourth power of a morphism in $\mathbf{Mat}_{\mathbb{Z}}$ by squaring twice:

$$\begin{aligned}&\text{Given a morphism } \boxed{M} : \boxed{m} \to \boxed{m} \text{ in } \mathbf{Mat}_{\mathbb{Z}},\\ &\text{compute } \boxed{N} \cdot \boxed{N} \text{ with } \boxed{N} := \boxed{M} \cdot \boxed{M}.\end{aligned} \tag{6.1}$$

We can use the implementation Mat_ZZ of $\mathbf{Mat}_{\mathbb{Z}}$ in Section 1.3.2 and express the algorithm in CAP as follows:

```
function ( M_boxed )
  local N_boxed;

    N_boxed := PreCompose( Mat_ZZ, M_boxed, M_boxed );

    return PreCompose( Mat_ZZ, N_boxed, N_boxed );
end
```

If we apply CompilerForCAP to such an algorithm, it proceeds in three phases, which we now illustrate:

In the first phase, CompilerForCAP expands the implementations of categorical operations. Here, the categorical operation PreCompose expands to the implementation of PreCompose in Section 1.3.2:

```
function ( M_boxed )
  local M1, M2, M_times_M, N_boxed, N1, N2, N_times_N;

```

[1] Here, the meaning of term "equivalence" depends on the context and may be difficult to define or may not be formally defined at all.

[2] Since CAP is written in GAP, of course technically speaking CAP code is just GAP code, so the distinction is purely semantic.

```
    M1 := AsPrimitiveValue( M_boxed );
    M2 := AsPrimitiveValue( M_boxed );
    M_times_M := M1 * M2; # matrix multiplication
    N_boxed := AsCapCategoryMorphism( Mat_ZZ, ..., M_times_M, ... );

    N1 := AsPrimitiveValue( N_boxed );
    N2 := AsPrimitiveValue( N_boxed );
    N_times_N := N1 * N2; # matrix multiplication
    return AsCapCategoryMorphism( Mat_ZZ, ..., N_times_N, ... );
end
```

Here, for reasons of brevity, we omit sources and targets of morphisms.

In the second phase, CompilerForCAP applies rules. It comes with an included set of general rules which can be extended by the user for more specific cases, as we will see in Section 6.3.3. To simplify finding matches of the rules, CompilerForCAP **inlines** all variable assignments in this phase, that is, it replaces references to variables by their assignments. This produces a far less readable version of the implementation:

```
function ( M_boxed )
    return AsCapCategoryMorphism( Mat_ZZ,
        ...,
        AsPrimitiveValue( AsCapCategoryMorphism( Mat_ZZ,
            ...,
            AsPrimitiveValue( M_boxed ) * AsPrimitiveValue( M_boxed ),
            ...
        ) ) * AsPrimitiveValue( AsCapCategoryMorphism( Mat_ZZ,
            ...,
            AsPrimitiveValue( M_boxed ) * AsPrimitiveValue( M_boxed ),
            ...
        ) ),
        ...
    );;
end
```

Note that whenever a variable is referenced more than once, the inlining introduces some duplication. This will be dealt with later on. The advantage of this inlined representation is that we now see that

AsPrimitiveValue(AsCapCategoryMorphism(..., matrix, ...)),

where matrix is given by

AsPrimitiveValue(M_boxed) * AsPrimitiveValue(M_boxed),

occurs in the computation, that is, matrix is boxed and immediately unboxed again. Unboxing a boxed value just gives the value again, so we can simplify the above expression to just matrix. As we will see in Section 6.3.1, this is a situation which occurs regularly, and CompilerForCAP has a built-in rule which performs the desired simplification. Hence, at the end of the second phase, the code looks much simpler:

```
function ( M_boxed )
    return AsCapCategoryMorphism( Mat_ZZ,
        ...,
        (AsPrimitiveValue( M_boxed ) * AsPrimitiveValue( M_boxed ))
                * (AsPrimitiveValue( M_boxed ) * AsPrimitiveValue( M_boxed )),
        ...
    );;
end
```

In the third and last phase, CompilerForCAP does some postprocessing. For example, it deduplicates expressions which appear multiple times, as we will see in detail in Section 6.3.4. Here, we see that the expression

```
AsPrimitiveValue( M_boxed ) * AsPrimitiveValue( M_boxed )
```

occurs twice.[3] CompilerForCAP will detect this and deduplicate the expression as follows:

```
function ( M_boxed )
  local deduped_1;

    deduped_1 := AsPrimitiveValue( M_boxed ) * AsPrimitiveValue( M_boxed );

    return AsCapCategoryMorphism( Mat_ZZ,
        ...,
        deduped_1 * deduped_1,
        ...
    );;
end
```

With this, the compilation is finished. If we translate the result back to mathematical notation, we get the algorithm:

$$\begin{aligned}&\text{Given a morphism } \boxed{M} : \boxed{m} \to \boxed{m} \text{ in } \mathbf{Mat}_{\mathbb{Z}},\\ &\text{compute } \boxed{N \cdot N} \text{ with } N := M \cdot M.\end{aligned}$$

This algorithm is equivalent to the algorithm (6.1) but has fewer boxes. On paper, where we view boxes only as a notation, this is only a minor simplification. On the computer, though, where objects and morphisms have to be boxed and unboxed explicitly, this simplification can have a significant impact, especially when building high categorical towers. We will see this in detail in Section 6.3.1. Moreover, we will see many more situations in which CompilerForCAP can optimize code during compilation in Section 6.3.

Remark 6.1.3. In Convention 1.4.19, we have required that all implementations of CAP operations must be pure functions. In the context of CompilerForCAP, this requirement is tightened further: CompilerForCAP requires all functions appearing inside the implementations of CAP operations to be pure.[4] This ensures that all of the above code transformations are valid, that is, they do not change the result of the function up to the technical equality. For example, the purity ensures that applying a function twice to equal inputs gives equal outputs, which allows us to deduplicate expressions appearing multiple times.

Remark 6.1.4 (Termination, confluence, and correctness)**.** We discuss some questions which arise naturally:

- **Termination**: Does the compilation process always terminate?
- **Confluence**: Does the compilation process always produce the same code independent of the order in which rules are applied?

[3] Here, the duplication only appears due to the previous inlining of variables, but in Section 6.3.4 we will see an example where duplicate computations actually appear independent of the inlining.

[4] Since composing pure functions gives a pure function by Proposition 1.4.8, this implies the original requirement in Convention 1.4.19. The converse implication does not hold because a pure function can be constructed from impure functions, see Example 1.4.5.

- **Correctness**: Does the compiled code actually produce the same results as the uncompiled code?

In general, neither termination nor confluence are guaranteed:

- One can trivially create an infinite loop in the compilation process by adding a rule together with its inverse. For example, we can add both the rule rewriting the expression `1 + 1` to `2` and the inverse rule rewriting `2` to `1 + 1`. If the expression `1 + 1` or the expression `2` occurs in the code, the compilation process will apply these two rules alternately ad infinitum.
- One can trivially break confluence by adding two rules rewriting the same expression to different results. For example, we can add a rule rewriting the expression `1 + 1` to `2` and a rule rewriting `1 + 1` to `3 - 1`. Which rule will be applied first is an undefined implementation detail.

Despite these theoretical limitations, in concrete applications these points have not caused problems until now.

Regarding correctness, the purity of the code ensures that the code transformation done by `CompilerForCAP` are valid, as explained in Remark 6.1.3. However, this does not rule out implementation flaws in `CompilerForCAP` or wrong rules added by the user. As a countermeasure, some package run their tests both with and without compiled code to allow to spot possible differences. With this, there have only been few flaws in the implementation of `CompilerForCAP` which have not been immediately discovered: Due to the transformations done by `CompilerForCAP` being very generic and due to the large existing codebase to which `CompilerForCAP` is applied, any issues show up very quickly.

In the outlook in Chapter 8 we sketch an idea of how `CompilerForCAP` could be implemented as a rewriting system modeled by a categorical tower itself. This would allow `CompilerForCAP` to optimize and verify itself, and could make it possible to find assumptions guaranteeing termination and confluence of the compilation process.

Remark 6.1.5 (Infrastructure for handling compiled code)**.** We describe some of the infrastructure handling compiled code: `CompilerForCAP` can output all compiled `CAP` operations of a category to a file, which can automatically be read in later sessions. That is, compilation can be done **ahead of time**.[5] In particular, the cost of compilation is irrelevant at runtime. We nevertheless provide some rough numbers: The categories in this thesis can be compiled in minutes. For higher categorical towers and more complex operations, the compilation can sometimes also take hours or days.

For categories where compilation finishes in minutes, the compilation is run as part of the package tests. This ensures that the stored compilation results are always up-to-date.

6.2 Benchmarking the towers in this thesis

Before we go into more details regarding the optimization strategies used by `CompilerForCAP`, we look at some benchmarks showing the performance gains

[5] `CompilerForCAP` was started as a **just-in-time compiler**, which compiles expressions upon the first execution. This is still reflected in the name of some functions in `CompilerForCAP` which start with `CapJit...`.

for the two main towers in Chapter 4 and Chapter 5:

- the category of ZX-diagrams $\mathbf{ZX}$ modeled as a reinterpretation of the categorical tower $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$, where the category of quivers $\mathbf{Quiv}$ is itself modeled via the categorical tower $\mathbf{Sets}^{\mathcal{C}(\mathbb{Q})}$, and
- the category of finitely presented right modules $\mathbf{FPres}_R$ modeled via the categorical tower $\mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}})$, where the category of matrices $\mathbf{Mat}_R$ is itself modeled via the categorical tower $\mathcal{C}(R)^{\oplus}$.

For some background regarding benchmarks in GAP, see Remark 6.3.2. First, we show a modest example to get a baseline what can be expected from using CompilerForCAP by default. Afterwards, we show an example in which CompilerForCAP is used to its full potential and where using CompilerForCAP makes the difference between “finishes in seconds” and “will never finish”.

Example 6.2.1 (Benchmark of a computation in **ZX**)**.** A primitive implementation of $\mathbf{ZX}$ generated from $\mathbf{Csp}(\mathbf{Quiv}/S_{\mathbf{ZX}})$ by CompilerForCAP is available via the category constructor CategoryOfZXDiagrams in the package ZXCalculusForCAP [Zic24j]. At the time of writing, this package has not been optimized by the author yet, that is, the package only contains a few very basic rules for CompilerForCAP. Hence, it can serve as a baseline what can be expected from using CompilerForCAP by default, that is, without writing additional complex rules.

Let $\boxed{m}$ be an object in $\mathbf{ZX}$. As an example of a simple computation in $\mathbf{ZX}$, we compute

$$(\eta_{\boxed{m}} \otimes \mathrm{id}_{\boxed{m}}) \cdot (\mathrm{id}_{\boxed{m}} \otimes \varepsilon_{\boxed{m}})$$

for various values of m to check that this indeed yields the identity as expected in Remark 5.4.2. We get the following timings in milliseconds averaged over three runs:[6]

m	without compiled code	with compiled code
1	2269	1
125	2680	65
250	4619	240
500	11283	992
1000	36899	3939
2000	139106	15769

The computation without compiled code has a constant overhead coming from some static checks inside the tower which are dropped during compilation. Hence, we normalize the timings of the computation without compiled code by subtracting the time for $m = 1$ from all values and ignoring the case $m = 1$:

m	without compiled code (normalized)	with compiled code	factor
125	411	65	6.3
250	2350	240	9.8
500	9014	992	9.1
1000	34630	3939	8.8
2000	136837	15769	8.7

[6] The source code of the benchmark can be found in the file ZX.g in [Zic24b].

We see that the improvement varies with the value of m but is significantly above a factor of 5 in all cases. As mentioned above, at the time of writing the implementation of **ZX** has not been optimized by the author yet, but we still get a welcome improvement simply by applying `CompilerForCAP` as is. In the next example, we will see the case of a highly optimized computation.

Example 6.2.2 (Benchmark of a computation in $\mathbf{FPres}_R$)**.** A primitive implementation of $\mathbf{FPres}_R$ which is partially generated from $\mathbf{Freyd}(\mathbf{Mat}_R{}^{\mathrm{op}})$ and optimized by `CompilerForCAP` is available via the category constructor `RightPresentations` in the package `ModulePresentationsForCAP` [GPZ24b]. As an example of a computation in $\mathbf{FPres}_R$, we consider the exterior algebra $R = \bigwedge \mathbb{Q}^2$ and compute a simple lift: Let I_m be the identity matrix with m rows and let id_m be the identity morphism on the object $\boxed{I_m}$ in $\mathbf{FPres}_R$. We compute the lift of id_m along id_m, which of course is just id_m again. However, in the timings we see the categorical tower struggles with this simple computation:[7]

m	**without compiled code**	**with compiled code**	**factor**
1	221	45	5
2	2369	58	41
3	19107	72	265
4	118893	93	1278
5	584488	119	4912
10	N/A	345	N/A
20	N/A	1344	N/A
30	N/A	3529	N/A

We see that the runtime of the computation without compiled code increases so fast that we cannot even do sensible timings for two-digit numbers anymore. Moreover, we start out with a improvement by a factor of 5 for $m = 1$ which increases if m gets larger, reaching already a factor of nearly 5000 for $m = 5$. As we can see, we get significant speedups which for larger examples make the difference between "finishes in seconds" and "will never finish". In the next section we see which optimization strategies lead to this speedup.

6.3 Optimization strategies

In this section, we describe various situations in which `CompilerForCAP` can optimize code during compilation, and also explain the corresponding compiler techniques. To assess the potential performance gains, we analyze the optimizations theoretically and also look at benchmarks showing the gains in practice.

Remark 6.3.1 (Optimizations spanning multiple layers of categorical towers)**.** We will see that during the compilation of categorical towers there is a huge potential for optimizations spanning multiple layers of the tower. To manually implement such an optimization, we would have to mix up those layers, losing the advantages of organizing code as categorical towers as explained in Remark 3.2.1. Additionally, we would have to interweave the categorical code and the optimizations, producing code which would be increasingly difficult to

[7] The source code of the benchmark can be found in the file `FPres.g` in [Zic24b].

maintain if more and more optimizations were added. Using CompilerForCAP, we never have to mix up categorical layers by hand and can keep the optimizations strictly separate from the categorical code. This way, we have another kind of **separation of concerns**, which keeps both the categorical code and the rules easy to comprehend and check.

Of course, CompilerForCAP itself has to break this separation of concerns during the compilation. However, since compilation can be automated, this does not hamper maintainability: If there is a change in one of the layers of the categorical tower or if new optimization are added, we can simply trigger a recompilation of the code.

Before we start, we make some technical remarks regarding benchmarks in GAP:

Remark 6.3.2 (Benchmarks in GAP)**.** GAP is single-threaded and thus rarely reaches thermal limits or power limits of a system even if other light tasks are running on the system in parallel. This makes benchmarks in CPU-bound scenarios quite repeatable. However, GAP has automated memory management with a garbage collector. The dynamic nature of garbage collection introduces variance in benchmarks bound by memory access. Additionally, some of our benchmarks will access Singular [DGPS23] as an external computer algebra system. In this scenario, computations can be bound by the I/O between GAP and Singular, introducing further variance. Hence, the benchmarks in this section should not be seen as scientific experiments on their own, but as complements to the theoretical analysis, showing that the trends found theoretically can indeed also be found in practice.

The specifications of the system used for the benchmarks can be found in Appendix C. All benchmarks are obtained by averaging over three runs.

6.3.1 Reducing the overhead of boxing

As we have already seen in Example 6.1.2, the most obvious place for optimizations of computations in CAP is to avoid superfluous boxing and unboxing.

The problem

In Example 6.1.2, we have seen that when composing categorical operations, the computation produces boxes which are immediately unboxed again. That is, using boxes introduces overhead in computations. This overhead gets larger for higher categorical towers, especially if the objects or morphisms of a category are formed by multiple objects or morphisms in another category. A prime example for such a category is an additive closure: Let us consider the additive closure $\mathcal{C}(\mathbb{Z})^\oplus$. Its objects are given by tuples of the unique object $\star$ of $\mathcal{C}(\mathbb{Z})$, and its morphisms are given by matrices of morphisms of $\mathcal{C}(\mathbb{Z})$:

$$\boxed{\left(\boxed{x_{ij}}_{\mathcal{C}(\mathbb{Z})}\right)_{ij}}_{\mathcal{C}(\mathbb{Z})^\oplus} : {\underbrace{\boxed{(\star,\dots,\star)}}_{m}}_{\mathcal{C}(\mathbb{Z})^\oplus} \to {\underbrace{\boxed{(\star,\dots,\star)}}_{n}}_{\mathcal{C}(\mathbb{Z})^\oplus}.$$

Creating such a morphism in CAP creates $m \cdot n$ inner boxes and one outer box. This happens for every intermediate result of a computation in $\mathcal{C}(\mathbb{Z})^\oplus$. We have

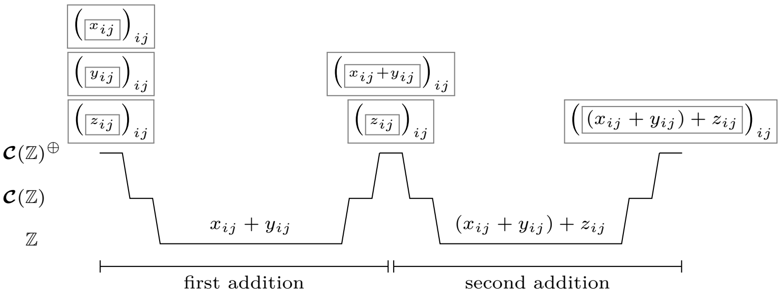

$\left(\boxed{x_{ij}}\right)_{ij}$
$\left(\boxed{y_{ij}}\right)_{ij}$
$\left(\boxed{z_{ij}}\right)_{ij}$
$\left(\boxed{x_{ij}+y_{ij}}\right)_{ij}$
$\left(\boxed{z_{ij}}\right)_{ij}$
$\left(\boxed{(x_{ij}+y_{ij})+z_{ij}}\right)_{ij}$
$\mathcal{C}(\mathbb{Z})^\oplus$
$\mathcal{C}(\mathbb{Z})$
$\mathbb{Z}$
$x_{ij}+y_{ij}$
$(x_{ij}+y_{ij})+z_{ij}$
first addition
second addition

Figure 6.1: Visualization of boxing and unboxing in a computation

already seen an example for this in Section 3.4, where we have computed the sum of three parallel morphisms in $\mathcal{C}(\mathbb{Z})^\oplus$

$$\boxed{\left(\boxed{x_{ij}}\right)_{ij}}, \boxed{\left(\boxed{y_{ij}}\right)_{ij}}), \boxed{\left(\boxed{z_{ij}}\right)_{ij}} : \underbrace{\boxed{(\star,\dots,\star)}}_{m} \to \underbrace{\boxed{(\star,\dots,\star)}}_{n}$$

as follows:

$$\begin{aligned}\left(\boxed{\left(\boxed{x_{ij}}\right)_{ij}} + \boxed{\left(\boxed{y_{ij}}\right)_{ij}}\right) + \boxed{\left(\boxed{z_{ij}}\right)_{ij}} &= \boxed{\left(\boxed{x_{ij}} + \boxed{y_{ij}}\right)_{ij}} + \boxed{\left(\boxed{z_{ij}}\right)_{ij}} \\ &= \boxed{\left(\boxed{x_{ij}+y_{ij}}\right)_{ij}} + \boxed{\left(\boxed{z_{ij}}\right)_{ij}} \\ &= \boxed{\left(\boxed{x_{ij}+y_{ij}} + \boxed{z_{ij}}\right)_{ij}} \\ &= \boxed{\left(\boxed{(x_{ij}+y_{ij})+z_{ij}}\right)_{ij}}.\end{aligned}$$

The first two steps compute the first addition and introduce the boxes in the intermediate result

$$\boxed{\left(\boxed{x_{ij}} + \boxed{y_{ij}}\right)_{ij}}$$

while the last two steps compute the second addition and introduce the boxes in the final result

$$\boxed{\left(\boxed{(x_{ij}+y_{ij})+z_{ij}}\right)_{ij}}.$$

Figure 6.1 visualizes the corresponding computation in CAP.

Boxing of the final expression is unavoidable because the boxes are part of the expected data structure of the result. However, boxing the integers $x_{ij}+y_{ij}$ to obtain the intermediate result and immediately unboxing them again is unnecessary. If we cancel the intermediate boxing and unboxing, we obtain a simplified algorithm, which is visualized in Figure 6.2. Only the required boxes are kept: The inputs are unboxed in two steps, the integers x_{ij}, y_{ij}, and z_{ij} are summed up, and the results are boxed in two steps to form the final result.

Necessity of an automated solution

The boxing happens in the implementation of the categories, in this case in the implementation of the category constructor $\mathcal{C}$ for rings and in the implementation of the additive closure. Hence, a categorical algorithm computing the sum of

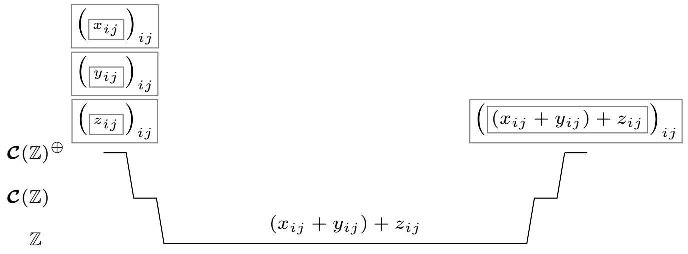

$\left(\boxed{x_{ij}}\right)_{ij}$
$\left(\boxed{y_{ij}}\right)_{ij}$
$\left(\boxed{z_{ij}}\right)_{ij}$
$\left(\boxed{(x_{ij}+y_{ij})+z_{ij}}\right)_{ij}$
$\mathcal{C}(\mathbb{Z})^{\oplus}$
$\mathcal{C}(\mathbb{Z})$
$\mathbb{Z}$
$(x_{ij}+y_{ij})+z_{ij}$

Figure 6.2: Visualization of the simplified version of the computation in Figure 6.1

three morphisms in an arbitrary category cannot avoid the superfluous boxing and immediate unboxing of the intermediate result. That is, the possibility of canceling boxing and unboxing only arises once we compose categorical operations in a concrete categorical tower. As explained in Remark 6.3.1, we cannot perform such optimizations by hand without severe downsides. Instead, we rely on `CompilerForCAP` for this.

The compiler technique

`CompilerForCAP` includes a rule which simplifies expressions of the form

```
AsPrimitiveValue( AsCapCategoryObject( cat, value ) )
```

to the expression

```
value
```

and a similar rule for morphisms. Hence, when applying `CompilerForCAP` to the algorithm visualized in Figure 6.1, it will cancel the unboxing and boxing and output the simplified algorithm visualized in Figure 6.2.

This is an important special case of a compiler technique called **peephole optimization**, which we will discuss in more generality in Section 6.3.3.

Performance analysis

We analyze the theoretical performance gain from this optimization.[8] In general, if we chain n categorical operations, we can cancel $n-1$ cases of boxing and only have to retain the final one. Hence, the optimization can reduce the overhead of boxing by a factor of $\frac{n-1}{n}$. In particular, the overhead is not linear in the number of categorical operations n anymore but constant.

Benchmarks

As a benchmark, we add three identity morphisms on

$$\underbrace{\boxed{(\star,\dots,\star)}}_{m}{}_{\mathcal{C}(\mathbb{Z})^{\oplus}}$$

[8] For simplicity, we ignore the case of having lists of objects or morphisms as inputs or outputs of categorical operations. In such a case, the improvement would be further amplified by the length of the lists. For the degenerate case of empty lists, we would not get an improvement at all. However, in this case there simply is no overhead, even without the optimization.

for various sizes of m, once compiled without the rule which cancels boxing and once compiled including the rule. Since we chain two additions, that is, two categorical operations, we expect the overhead of boxing to be halved by the optimization. In `CAP`, integer addition is cheap compared to boxing, so when looking at the overall runtime, we actually expect a speedup by a factor of almost 2. We get the following timings in milliseconds:[9]

m	**without canceling boxing**	**with canceling boxing**	**factor**
1	0	0	-
125	103	53	1.94
250	412	209	1.97
500	1817	926	1.96
1000	7722	3846	2.01
2000	31072	15750	1.97

We perfectly see that we get the expected speedup by a factor of almost 2.

In a complex categorical algorithm, where we might chain tens or hundreds of categorical operations, we get a two-digit or three-digit reduction of the overhead of boxing and unboxing. If this overhead is one of the bottlenecks of an algorithm, as was the case in our benchmark above, we can consequently get a significant improvement of the overall runtime.

6.3.2 Simplification of data structures

In the previous section, we have successfully removed intermediate boxing, but have not touched the boxing of the final result, which still causes some overhead. To also reduce the amount of boxing in the final result, we have to manually introduce data structures with less boxing.

The problem

We have already seen how we can simplify data structures of categorical towers using reinterpretations. For example, in Construction 3.3.2 we have modeled the category of matrices $\mathbf{Mat}_R$ as a reinterpretation of the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ via the functor $\mathcal{R}$ defined as

$$\begin{aligned} \mathcal{R} : \mathcal{C}(R)^{\oplus} &\to \mathbf{Mat}_R \\ \underbrace{\boxed{(\star,\dots,\star)}}_{m}{}_{\mathcal{C}(R)^{\oplus}} &\mapsto \boxed{m}_{\mathbf{Mat}_R} \\ \boxed{\left(\boxed{m_{ij}}_{\mathcal{C}(R)}\right)_{ij}}_{\mathcal{C}(R)^{\oplus}} &\mapsto \boxed{(m_{ij})_{ij}}_{\mathbf{Mat}_R} \end{aligned}$$

together with its inverse $\mathcal{M}$ defined as

$$\begin{aligned} \mathcal{M} : \mathbf{Mat}_R &\to \mathcal{C}(R)^{\oplus} \\ \boxed{m}_{\mathbf{Mat}_R} &\mapsto \underbrace{\boxed{(\star,\dots,\star)}}_{m}{}_{\mathcal{C}(R)^{\oplus}} \\ \boxed{(m_{ij})_{ij}}_{\mathbf{Mat}_R} &\mapsto \boxed{\left(\boxed{m_{ij}}_{\mathcal{C}(R)}\right)_{ij}}_{\mathcal{C}(R)^{\oplus}} \end{aligned}$$

[9] The source code of the benchmark can be found in the file `boxing.g` in [Zic24b].

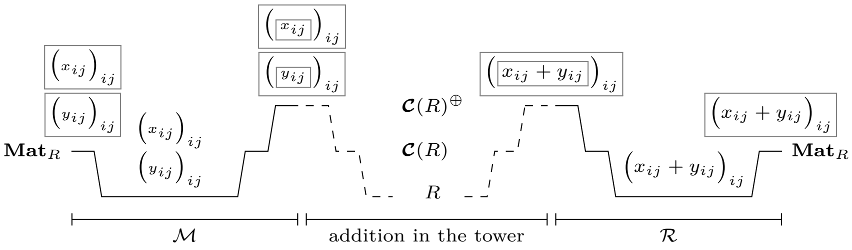

$\boxed{\left(\boxed{\boxed{x_{ij}}}\right)_{ij}}$
$\boxed{\left(\boxed{\boxed{y_{ij}}}\right)_{ij}}$
$\boxed{\left(\boxed{\boxed{x_{ij}+y_{ij}}}\right)_{ij}}$
$\boxed{(x_{ij})_{ij}}$
$\boxed{(y_{ij})_{ij}}$
$\boxed{(x_{ij}+y_{ij})_{ij}}$
$\mathbf{Mat}_R$
$(x_{ij})_{ij}$
$(y_{ij})_{ij}$
$\boldsymbol{\mathcal{C}}(R)^{\oplus}$
$\boldsymbol{\mathcal{C}}(R)$
R
$(x_{ij}+y_{ij})_{ij}$
$\mathbf{Mat}_R$
$\mathcal{M}$
addition in the tower
$\mathcal{R}$

Figure 6.3: Visualization of boxing and unboxing in a computation involving a reinterpretation

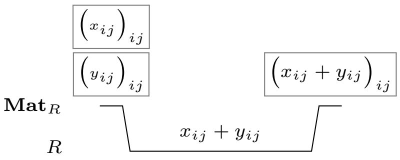


Figure 6.4: Visualization of the simplified version of the computation in Figure 6.3

Clearly, $\mathcal{R}$ simplifies the data structures of objects and morphisms. In particular, $\mathcal{R}$ eliminates the inner boxes of morphisms.

Recall that by Construction 3.3.5, we can obtain categorical operations of $\mathbf{Mat}_R$ as follows: We map the inputs to the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ via $\mathcal{M}$, apply the categorical operation there, and map the result back to $\mathbf{Mat}_R$. A visualization of an example of this can be found in Figure 6.3. We see that to add two morphisms

$$\boxed{(x_{ij})_{ij}} \quad \text{and} \quad \boxed{(y_{ij})_{ij}}$$

in $\mathbf{Mat}_R$, we first apply $\mathcal{M}$, which unboxes and re-boxes the inputs as morphisms in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$. Next, we perform the addition of the two morphisms in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$. Finally, we apply $\mathcal{R}$, which unboxes the result and re-boxes it as

$$\boxed{(x_{ij}+y_{ij})_{ij}}$$

in $\mathbf{Mat}_R$.

We see that by using the reinterpretation, a priori we introduce even more boxing and unboxing. However, we can now cancel the boxing at the end of $\mathcal{M}$ with the unboxing at the beginning of the addition in the tower. Similarly, we can cancel the boxing at the end of the addition in the tower with the unboxing at the beginning of $\mathcal{R}$. With this, we obtain the simplified algorithm visualized in Figure 6.4. Note that the simplified algorithm has no reference to the tower anymore and hence indeed avoids all inner boxes of morphisms in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$. In particular, we have generated a primitive implementation of the addition in the category of matrices from the categorical tower, which we previously would have had to write by hand. We will stress this point more in Section 6.5.1, where we will see an example of a package which does not contain any manual implementations of `CAP` operations anymore but only implementations generated by `CompilerForCAP`.

Necessity of an automated solution

Clearly, this optimization affects the categorical tower as a whole. As explained in Remark 6.3.1, we cannot perform such optimizations by hand without severe downsides. Instead, we rely on `CompilerForCAP` for this.

The compiler technique

From a technical perspective, there is no difference to the cancellation of boxing and unboxing in Section 6.3.1, so `CompilerForCAP` will again use the rules which cancel boxing immediately followed by unboxing.

Performance analysis

We analyze the theoretical performance gains. Recall that in the previous section we got a speedup which was linear in the length of the chain of categorical operations. Here, we add exactly two additional simplifications to such a chain, one at the beginning and one at the end of the computation, so the additional speedup will be constant. Furthermore, the speedup will heavily depend on the original data structure of the tower and the new simplified data structure. For this reason, we will now only analyze the concrete tower in this example for $R = \mathbb{Z}$ and simply compare the cost of creating a morphism in $\mathbf{Mat}_{\mathbb{Z}}$ and the cost of creating a morphism in $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$.

Creating a morphism in $\mathbf{Mat}_{\mathbb{Z}}$ creates three boxes: one for the source, one for the target, and one for the morphism itself. Creating a morphism in $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$ also creates three boxes for the source, the target, and the morphism itself, but additionally creates a box for every matrix entry. For large matrices, creating three boxes is negligible. Hence, we expect that the overhead of creating a morphism in $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$ compared to creating a morphism in $\mathbf{Mat}_{\mathbb{Z}}$ can become arbitrarily large. More specifically, we expect the overhead to be linear in the number of matrix entries.

Benchmarks

A good example for benchmarks of the creation of morphisms is the creation of zero morphisms, as for those typically no additional computation is needed at all. We create the zero morphism $\boxed{1000} \to \boxed{m}$ in $\mathbf{Mat}_{\mathbb{Z}}$ for various values of m as well as the corresponding zero morphism in $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$. To get a baseline, we also create $1000 \times m$ zero matrices without any boxing at all.[10]

Following the above analysis, we expect that the creation of a zero morphism in $\mathbf{Mat}_{\mathbb{Z}}$ has a negligible overhead compared to creating a zero matrix without any boxing at all. Indeed, get the following timings in milliseconds:

m	$\mathbf{Mat}_{\mathbb{Z}}$	**zero matrix**	**difference**
1	1	0	1
500	38	37	1
1000	74	72	2
2000	146	144	2
4000	360	357	3
8000	803	797	6

[10] The source code of the benchmark can be found in the file `data_structures.g` in [Zic24b].

As expected, the overhead of creating a zero morphism in $\mathbf{Mat}_{\mathbb{Z}}$ compared to creating a zero matrix without any boxing at all is only a few milliseconds.

Now, for $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$, we get the following timings in milliseconds:

m	$\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$	$\mathbf{Mat}_{\mathbb{Z}}$	**difference**	**difference per 1k entries**
1	4	1	3	3.0
500	1701	38	1663	3.3
1000	3594	74	3520	3.5
2000	7164	146	7018	3.5
4000	14398	360	14038	3.5
8000	29986	803	29183	3.6

We find the expected overhead of the creation of a morphism in $\boldsymbol{\mathcal{C}}(\mathbb{Z})^{\oplus}$ compared to the creation of a morphism in $\mathbf{Mat}_{\mathbb{Z}}$: The benchmarks show that the overhead is linear in the number of matrix entries, amounting to about 3.5 milliseconds per thousand entries.

Summing up, we see that by using a reinterpretation, we can avoid the overhead of boxing almost completely. If this overhead is one of the bottlenecks of an algorithm, as was the case in our benchmark above, we can consequently get a significant improvement of the overall runtime.

6.3.3 Simplification of algorithms

In the previous section, we have seen how we can simplify data structures to reduce the overhead of boxing. Using simplified data structures often also allows us to simplify algorithms. In this section, we will see various examples for such situations.

The problem

If we model the category of matrices $\mathbf{Mat}_R$ via the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$, the reinterpretation simplifies the data structure of objects: Objects in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ are given by tuples (containing the unique object of $\boldsymbol{\mathcal{C}}(R)$), while objects of $\mathbf{Mat}_R$ are given by (non-negative) integers. Now, let us consider the computation of direct sums in $\mathbf{Mat}_R$ via the categorical tower as in Construction 3.3.7. For two objects $\boxed{m}$ and $\boxed{n}$ of $\mathbf{Mat}_R$, we have:

$$\begin{aligned}
\boxed{m} \oplus \boxed{n} &\equiv \mathcal{R}\Big(\mathcal{M}(\boxed{m}) \oplus \mathcal{M}(\boxed{n})\Big) \\
&\equiv \mathcal{R}\Big(\underbrace{\boxed{\boxed{(\star,\dots,\star)}}}_{m} \oplus \underbrace{\boxed{\boxed{(\star,\dots,\star)}}}_{n}\Big) \\
&\equiv \mathcal{R}\Big(\boxed{\boxed{(\underbrace{\star,\dots,\star}_{m},\underbrace{\star,\dots,\star}_{n})}}\Big) \\
&\equiv \mathcal{R}\Big(\underbrace{\boxed{\boxed{(\star,\dots,\star)}}}_{m+n}\Big) \\
&\equiv \boxed{m+n}
\end{aligned}$$

Here, we see an example of a simplification of an algorithm induced by the simplification of the data structure of objects: The second to last equality turns an algorithm on tuples (the concatenation of two tuples) into an algorithm on integers (the sum of two integers).

Let us see how we can perform this simplification with CompilerForCAP. An implementation of $\mathbf{Mat}_R$ via the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ is available via the category constructor CategoryOfRowsAsAdditiveClosureOfRingAsCategory[11] in the package FreydCategoriesForCAP [BPZ24]. We can use this constructor to create the category of matrices Mat_ZZ over the integers. Then, computing the direct sum of two objects in Mat_ZZ is straightforward:

```
function ( m_boxed, n_boxed )
    return DirectSum( Mat_ZZ, m_boxed, n_boxed );
end
```

If we apply CompilerForCAP to this function, a priori it will generate the following implementation:

```
function ( m_boxed, n_boxed )
  local m, n;

    m := AsPrimitiveValue( m_boxed );
    n := AsPrimitiveValue( n_boxed );

    return AsCapCategoryObject( Mat_ZZ,
        Length( Concatenation(
            ListWithIdenticalEntries( m, star ),
            ListWithIdenticalEntries( n, star )
        ) )
    );
end
```

where star is a global variable pointing to the unique object of $\boldsymbol{\mathcal{C}}(\mathbb{Z})$. This implementation is the algorithm on tuples: It creates two lists[12] of length m and n containing the unique object of $\boldsymbol{\mathcal{C}}(\mathbb{Z})$, concatenates them, and boxes the length of the resulting list as an object in $\mathbf{Mat}_{\mathbb{Z}}$. To turn this algorithm into an algorithm on integers, we can add **logic templates** to CompilerForCAP as follows:

```
CapJitAddLogicTemplate(
    rec(
        variable_names := [ "list1", "list2" ],
        variable_filters := [ IsList, IsList ],
        src_template := "Length( Concatenation( list1, list2 ) )",
        dst_template := "Length( list1 ) + Length( list2 )",
    )
);
```

and

```
CapJitAddLogicTemplate(
    rec(
        variable_names := [ "n", "obj" ],
        variable_filters := [ IsInt, IsObject ],
        src_template := "Length( ListWithIdenticalEntries( n, obj ) )",
        dst_template := "n",
    )
);
```

[11] The term **category of rows** is a synonym of **category of matrices**.

[12] We use lists to model tuples in GAP.

The first logic template teaches `CompilerForCAP` that it may replace any occurrence of an expression of the form

```
Length( Concatenation( list1, list2 ) )
```

in the code by

```
Length( list1 ) + Length( list2 )
```

where `list1` and `list2` can be arbitrary expressions computing lists. After application of this logic template, the implementation looks as follows:

```
function ( m_boxed, n_boxed )
  local m, n;

    m := AsPrimitiveValue( m_boxed );
    n := AsPrimitiveValue( n_boxed );

    return AsCapCategoryObject( Mat_ZZ,
        Length( ListWithIdenticalEntries( m, star ) ) +
        Length( ListWithIdenticalEntries( n, star ) )
    );
end
```

Applying the second logic template will simplify this implementation to

```
function ( m_boxed, n_boxed )
  local m, n;

    m := AsPrimitiveValue( m_boxed );
    n := AsPrimitiveValue( n_boxed );

    return AsCapCategoryObject( Mat_ZZ, m + n );
end
```

This is the expected simplified algorithm on integers.

A special case of this situation is that we can reuse existing optimized algorithms. For example, if we compile the composition of `Mat_ZZ` as above, we get a naive algorithm for matrix multiplication:

```
function ( M_boxed, N_boxed )
    ...
    product :=
        List( [ 1 .. nr_rows ], i ->
            List( [ 1 .. nr_cols ], j ->
                Sum( List( [ 1 .. inner_dim ], k -> M[i][k] * N[k][j] ) )
            )
        );
    ...
end
```

This algorithm simply implements the mathematical definition of matrix multiplication by looping over the number of rows and columns of the result and taking the corresponding sum. There exist many strategies for improving this naive algorithm, see, for example, [ALSU06, Section 11.2]. Using a logic template, we can simplify the implementation to

```
function ( M_boxed, N_boxed )
    ...
    product := M * N;
    ...
end
```

where "*" calls the matrix multiplication in GAP or other computer algebra systems like Singular, where highly optimized algorithms for matrix multiplication are already implemented.

A similar example is the use of the Kronecker product of matrices and its dual version in Construction 4.1.3, which can certainly be computed more efficiently than by naively looping over the entries of the matrices. This is one for the reasons for the large improvement of the computation of lifts in $\mathbf{FPres}_R$ in Example 6.2.2.

Summing up, using logic templates we can provide rules to CompilerForCAP which allow to simplify algorithms. Possibilities for such simplifications arise, for example, after simplification of data structures by reinterpretations of categorical towers.

Necessity of an automated solution

Many simplifications of algorithms span multiple layers of the categorical tower or are only applicable after the simplification of data structures by a reinterpretation, which affects the categorical tower as a whole. As explained in Remark 6.3.1, we cannot perform such optimizations by hand without severe downsides. Moreover, even if an optimization can be implemented in a single layer of the categorical tower, we would still like to have a **separation of concerns** between the categorical implementation and the optimizations, as explained in Remark 6.3.1. Hence, we rely on CompilerForCAP to perform the optimizations.

The compiler technique

Locally replacing algorithms by more efficient ones is called **peephole optimization** [ALSU06, Section 8.7]. CompilerForCAP comes with a set of built-in peephole optimization rules, which can be extended by the user by logic templates as we have seen above. Note that when applying such a rule, we often trigger a cascading effect, that is, applying one rule makes it possible to apply another rule, which again makes it possible to apply a third rule, and so on. We have seen this above: the second logic template only triggers after the first logic template is applied. Hence, CompilerForCAP repeats peephole optimization until no further rules can be applied anymore.

Performance analysis

The speedup obtained by replacing an algorithm on a suboptimal data structure by an optimized algorithm can be arbitrarily large. Hence, we cannot perform a generic analysis and instead analyze and benchmark the concrete example above, where we have simplified

```
function ( m_boxed, n_boxed )
  local m, n;

    m := AsPrimitiveValue( m_boxed );
    n := AsPrimitiveValue( n_boxed );

    return AsCapCategoryObject( Mat_ZZ,
        Length( Concatenation(
            ListWithIdenticalEntries( m, star ),
            ListWithIdenticalEntries( n, star )
        ) )
    );
end
```

to

```
function ( m_boxed, n_boxed )
  local m, n;

    m := AsPrimitiveValue( m_boxed );
    n := AsPrimitiveValue( n_boxed );

    return AsCapCategoryObject( Mat_ZZ, m + n );
end
```

Creating long lists is a relatively expensive operation due to the required memory allocation. Compared to this, the runtime of a single addition of integers is completely minuscule. Hence, we expect to run into limitations of the algorithm on tuples long before we can even measure the runtime of the algorithm on integers.

Benchmarks

We benchmark the implementations given in the performance analysis above, excluding the boxing, for $m = n$ of various sizes and get the following timings in milliseconds:[13]

m **(in million)**	**algorithm on tuples**	**algorithm on integers**
1	69	0
25	892	0
50	1525	0
100	2801	0
200	5556	0
400	out of memory	0
10^{10^9}	N/A[a]	152

[a] On 64-bit systems, `ListWithIdenticalEntries` only accepts integers up to $2^{60} - 1$.

We see that the algorithm on tuples has a significant runtime and runs out of memory long before we even get measurable results for the algorithm on integers. This is a typical situation where applying a simple optimization makes the difference between an algorithm finishing in no time and not finishing at all. In practice, the performance gains of course depend on the concrete optimization.

6.3.4 Avoiding irrelevant or redundant computations

Another common source of overhead appearing in categorical towers are redundant computations. In this section, we look at three instances of this: Computations which are not needed at all, computations which appear in the algorithm multiple times, and computations inside loops which are actually independent of the loop variable but are still recomputed for every iteration.

[13] The source code of the benchmark can be found in the file `algorithms.g` in [Zic24b].

The problem

Example 6.3.3 (Duplicate computations)**.** An example of a duplicate computation can be seen in the algorithm in Example 1.3.2: The implementation contains the following two lines:

```
    ...
    iA := InjectionOfCofactorOfCoproduct( cat, [ A, B ], 1 );
    iB := InjectionOfCofactorOfCoproduct( cat, [ A, B ], 2 );
    ...
```

Computing `InjectionOfCofactorOfCoproduct` automatically computes the target of the injection, that is, the coproduct

`Coproduct( cat, [ A, B ] )`

on both cases. Hence, when compiling the above algorithm in a concrete category, the expression computing `Coproduct( cat, [ A, B ] )` would appear twice. To avoid this, we can store the result of the expression computing `Coproduct( cat, [ A, B ] )` in a variable, and replace all occurrences of the expression by references to this variable. We have already seen a concrete example of such a deduplication in Example 6.1.2.

Example 6.3.4 (Irrelevant computations)**.** We look at an example of an irrelevant computation, that is, a computation which is not needed at all: By Remark 2.4.4, we can compute the sum of morphisms $\alpha_1, \dots, \alpha_n : A \to B$ in an additive category by composing two morphisms between direct sums, expressed as a row vector and a column vector:

$$\sum_{i=1}^{n} \alpha_i = \begin{pmatrix} \alpha_1 & \dots & \alpha_n \end{pmatrix}^{\oplus} \cdot \begin{pmatrix} \mathrm{id}_B \\ \vdots \\ \mathrm{id}_B \end{pmatrix}^{\oplus}.$$

The target of the morphism between direct sums given by the row vector and the source of the morphism between direct sums given by the column vector both are the n-fold direct sum of the object B. Hence, when computing the right-hand side of the equation, the direct sum $\bigoplus_{i=1}^{n} B$ will be computed as the central object of the composition. However, most categories forget the central object during composition. For example, the composition of two morphisms $\boxed{\varphi} : \boxed{\alpha} \to \boxed{\beta}$ and $\boxed{\psi} : \boxed{\beta} \to \boxed{\gamma}$ in a Freyd category simply composes φ and ψ and does not use the central object $\boxed{\beta}$ at all. In such a case, the central object $\bigoplus_{i=1}^{n} B$ will not actually be used at all, so we would like to avoid computing it.

Example 6.3.5 (Redundant computations in loops)**.** Another important example of redundant computations appears in the context of loops: Often, parts of the body of a loop are actually independent of the loop variable but are still recomputed for every iteration.

As an example, let k be a commutative ring, let R be a k-algebra which is finitely generated free as a k-module and consider the $\mathbf{Mat}_k$-homomorphism structure of $\mathbf{Mat}_R$ in Construction 4.1.3. This homomorphism structure is constructed via the categorical tower $\mathcal{C}(R)^{\oplus}$ and maps a pair of morphisms $\boxed{M}$ and $\boxed{N}$ in $\mathbf{Mat}_R$ to

$$\boxed{\Lambda\Big(\Big(M_{ij} \cdot \big((\mathcal{B} \cdot N_{st})_{st}\big)\Big)_{ji}\Big)}$$

in $\mathbf{Mat}_k$, where we have expanded the Kronecker product of matrices and its dual version. In an implementation, this corresponds to four nested loops coming from the homomorphism structure on morphisms of additive closures in Construction 2.5.8: First, we loop over j and i to get the element M_{ij}, and inside these two loops we loop over s and t to build the inner matrix

$$(\mathcal{B} \cdot N_{st})_{st}.$$

Now, note that this inner matrix is actually independent of j and i, but still recomputed in every iteration of the two outer loops. To avoid this, we can compute the inner matrix before the two outer loops, store the result in a variable, and reference this variable inside the two outer loops, analogous to the following mathematical expression:

$$\boxed{\Lambda\Big(\big(M_{ij} \cdot L\big)_{ji}\Big)} \quad \text{for} \quad L := (\mathcal{B} \cdot N_{st})_{st}.$$

Necessity of an automated solution

In all examples above, the redundancy only shows up once we expand definitions of a concrete categorical tower:

- The computation of the coproduct object is a part of the implementation of `InjectionOfCofactorOfCoproduct`.
- Whether the central object of the composition is used in the implementation of the composition depends on the concrete category.
- The four nested loops come from the homomorphism structure on morphisms of an additive closure $\mathbf{C}^{\oplus}$ as in Construction 2.5.8, which maps a pair of morphisms $\boxed{\alpha}$ and $\boxed{\beta}$ to

  $$\Big\langle \Big(\Big\langle \big(H(\alpha_{ij}, \beta_{st})\big)_t^{\oplus} \Big\rangle_s^{\oplus} \Big)_i^{\oplus} \Big\rangle_j^{\oplus}$$

  This implementation cannot know whether some parts of the implementation of H in $\mathbf{C}$ are independent of the first argument and hence independent of i and j.

Hence, the optimizations are specific to a concrete category or span multiple layers of a categorical tower. As explained in Remark 6.3.1, we cannot perform such optimizations by hand without severe downsides. Instead, we rely on `CompilerForCAP` for this.

The compiler technique

A piece of code which computes a value which is never used is called **dead code**. `CompilerForCAP` will detect and remove dead code, which is a compiler technique known as **dead code elimination** [ALSU06, Section 9.1.6]. The purity of the code guarantees that the result of a computation cannot be affected by whether dead code is executed or not.

During compilation, `CompilerForCAP` searches for subexpressions appearing multiple times and replaces them by a single variable. This compiler technique is called **common subexpression elimination** (CSE) [ALSU06, Section 9.1.4]. The purity of the code ensures that CSE does not change the result with regard to the technical equality. For example, the value of the subexpression $\bigoplus_{i=1}^{n} B$ above only depends on B and computing it does not have side effects, so the result of the whole expression does not change with when and how often we compute this subexpression.

Parts of a loop body which are both independent of the loop variable and independent of any variables introduced in the loop body are called **loop-invariant**. Similar to the situation for CSE, the purity of the code ensures that we can move loop-invariant code in front of the loop, store the result in a variable, and simply reference the variable inside the loop. This compiler technique is called **loop-invariant code motion** (LICM) or, less formally, **hoisting** [ALSU06, Section 9.1.7].

Performance analysis

Eliminating dead code always gives a speedup, though the performance gain of course depends on the dead code which is removed. For CSE and LICM, the situation is more complicated: We have to consider the additional cost of storing the value of an expression in a new variable and referencing this variable later. In `GAP`, this cost is negligible compared to the cost of more complex tasks like calling functions.

Hence, eliminating non-trivial common subexpressions always gives a speedup. Trivial subexpressions like integer literals can easily be excluded from the optimization. Of course, the performance gain ultimately depends on the cost of the subexpression and scales with the number of occurrences of the subexpression.

For LICM, if the hoisted subexpression dominates the runtime of the loop body, we expect a speedup by a factor of almost the number of iterations of the loop. For loops over large ranges or nested loops, where the sizes of the ranges multiply, this is a significant win. For loops with a single iteration, we have the negligible overhead of populating and referencing the additional variable. For empty loops, we actually lose performance because the hoisted expressions is now computed but would not have been computed without hoisting. However, empty loops usually occur as simple edge cases where performance is not relevant.

Summing up, in general all the optimizations give a speedup. The performance gain of course depends on the runtime of the expression which is eliminated, deduplicated, or hoisted. Significant gains can especially be expected for LICM in the case of deeply nested loops over large ranges.

Benchmarks

Here, we cannot provide sensible benchmarks because dead code elimination, common subexpression elimination, and loop-invariant code motion are essential parts of the compilation process: As we have seen in Example 6.1.2, `CompilerForCAP` inlines all assignment to variables. This automatically drops unused variables, so we cannot simply turn off dead code elimination for benchmarking purposes. Moreover, inlining might duplicate expressions if a variable is referenced more than once and might move computations inside loops. Hence, a compilation without common subexpression elimination or loop-invariant code motion might actually produce code which is worse than the original code. Hence, we cannot provide sensible benchmarks with common subexpression elimination or loop-invariant code motion turned off. Instead, we refer to the theoretical performance analysis above.

6.4 Comparison to other compilers

We now compare CompilerForCAP to other compilers to see why developing a special compiler was necessary. Experiments in Julia, Rust, and Haskell indicate that their compilers do not optimize as aggressively as CompilerForCAP does. We look at two examples in Julia.

Example 6.4.1 (Peephole optimization in Julia)**.** Consider the following function in Julia:

```
function (m)
    return length(fill(0, m))
end
```

The Julia function fill is the equivalent of GAP's ListWithIdenticalEntries with swapped arguments: Here, we create a list of length m filled with zeros. Afterwards, we compute the length of this list, which of course is just m. Julia uses a **just-in-time compiler**, which compiles expressions upon the first execution. We hope that the Julia compiler is able to simplify the above function to

```
function (m)
    return m
end
```

just as we did using CompilerForCAP in Section 6.3.3. When benchmarking the two functions in Julia, we get the following timings in milliseconds:

m (in million)	**return length(fill(0, m))**	**return m**
1	1	0
125	129	0
250	250	0
500	869	0
1000	out of memory	0

Clearly, the Julia compiler did not perform the optimization we hoped for.

Example 6.4.2 (Hoisting and dead code elimination in Julia)**.** We consider the following function in Julia:

```
function ()
    for x in 1:10
        y = factorial(big(2^22))
    end
end
```

The code loops over the computation of a large factorial, while the factorial is independent of the loop variable x. We hope that the Julia compiler is able to hoist the computation of the factorial to produce

```
function ()
    val = factorial(big(2^22))
    for x in 1:10
        y = val
    end
end
```

or that it even notes that the function does neither return a value nor have a side effect and hence can actually be simplified to the empty function

```
function ()
end
```

However, executing the first function takes about 10.8 seconds, executing the second function takes about 1 second, and executing the empty function takes less than a millisecond. Again, clearly the `Julia` compiler did not perform the optimizations we hoped for.

The previous examples show that the `Julia` compiler is less aggressive than `CompilerForCAP`, and similar experiments show the same for the compilers of Rust and Haskell. One explanation for this might be that organizing code via categorical towers is particularly inefficient and thus requires more aggressive optimizations than typical code in `Julia`, Rust, or Haskell. For example, one would typically not model non-negative integers via tuples, as we did for $\mathbf{Mat}_R$ and the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$. Similarly, one would typically refrain from separating a loop header from its body in performance critical places, so one would be able to hoist expressions manually. In implementations of categorical towers, however, we have seen in Section 6.3.4 that this would require to mix up different layers of the tower, which would come with severe downsides, as explained in Remark 6.3.1. This can explain why we need a compiler which is more aggressive than the compilers of other programming languages.

6.5 Generating categorical code

`CompilerForCAP` can be used for generating categorical code. In this section, we present two examples of this. The examples show that **reusability** is not a merely theoretical advantage of categorical towers but actually allows us to write less code overall.

6.5.1 Generating primitive implementations from categorical towers

In the previous section we have seen how we can generate primitive implementations of categories as reinterpretations of categorical towers: In Section 6.3.2, we have seen how applying `CompilerForCAP` to a reinterpretation cancels all boxes of the categorical tower, so no references to the tower remain in the data structures, as can for example be seen in Figure 6.4. Afterwards, we can also adapt algorithms to the new data structures as explained in Section 6.3.3. For example, we have generated the following algorithm for computing direct sums in $\mathbf{Mat}_R$ from the categorical tower $\boldsymbol{\mathcal{C}}(R)^{\oplus}$:

$$\boxed{m} \oplus \boxed{n} \coloneqq \boxed{m+n}.$$

This algorithm has no reference to the categorical tower anymore, and matches the implementation of direct sums in $\mathbf{Mat}_R$ which we would previously have written by hand. CompilerForCAP can output all compiled CAP operations of a category to a file. Hence, with all of the above, we can actually generate a standalone implementation of $\mathbf{Mat}_R$ from $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ which is completely independent of $\boldsymbol{\mathcal{C}}(R)^{\oplus}$. This can be seen in the package LinearAlgebraForCAP [GPZ24a], which provides an implementation of $\mathbf{Mat}_K$ for a field K: The package does not contain any manual implementations of CAP operations anymore but only implementations generated by CompilerForCAP.

Primitive implementations of categories generated from categorical towers exist for all categorical towers in Chapter 4 and Chapter 5, and if a newly considered category can be modeled as a categorical tower using existing category constructors, we only have to provide the functors defining a reinterpretation to generate the complete implementation. Often, the generated implementation can even beat the performance of previous manually optimized implementations, because CompilerForCAP performs optimizations more consistently and thoroughly.

6.5.2 Generating dual algorithms

We can also apply the above ideas to contexts where the category is not yet fixed. In Example 4.3.4, we have seen how by using a categorical tower we can dualize the algorithm

$$\mathrm{coeq}(f, g) :\equiv \mathrm{coker}(f - g)$$

for arbitrary preadditive categories with cokernels to

$$\mathrm{eq}(f, g) :\equiv \ker(f - g)$$

for arbitrary preadditive categories with kernels. To get the dual algorithm, we performed a symbolic computation which essentially just expanded definitions of categorical operations and canceled two mutually inverse functors. CompilerForCAP automatically expands definitions during compilation, and the cancellation rule for the two concrete functors can be provided as a logic template. Hence, CompilerForCAP can automatically generate dual versions of categorical algorithms. This is actually used in CAP to automatically generate dual versions of derivations, which drastically reduces the implementation time for new derivations and avoids errors due to wrong dualization by hand.

Note that here the output of the compilation is again high-level CAP code. Hence, following the terminology introduced in Remark 6.1.1, this is an example of a situation where CompilerForCAP acts as a source-to-source translator.

6.6 Conclusion

In this chapter, we have seen that by using CompilerForCAP we can avoid the performance overhead naturally appearing in computations in categorical towers, which allows us to make full use of the advantages of building categorical towers on a computer. Specifically, we have seen the following advantages:

- **Reusability**: We can reuse category constructors to build different categorical towers and can actually use this to generate optimized primitive implementations.

- **Separation of concerns**: Different concepts inside the tower are strictly separate from each other and the optimizations given by logic templates are strictly separate from the implementation.
- **Emergence:** Implementations for complex algorithms like homomorphism structures of categories of matrices emerge from implementations of relatively simple algorithms in every layer of the categorical tower, together with a series of small logic template.

In the next chapter, we will also look at the aspect of **verifiability**. Specifically, we will show how to use `CompilerForCAP` as a proof assistant to prove the correctness of implementations of categories in `CAP`. Summing up, thanks to `CompilerForCAP`, we get all of the advantages of categorical towers given in Remark 3.2.1 not only on a theoretical level but actually for implementations on a computer.

Chapter 7

Using `CompilerForCAP` for verification

In this chapter, we show how `CompilerForCAP` [Zic24a] can be used for code verification. As an application, we prove the correctness of the constructions of the categorical tower used for endowing categories of matrices with a homomorphism structure in Construction 4.1.3. For example, given a ring R in `GAP`, we can use `CompilerForCAP` to verify that an implementation of the composition in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ is associative. In our constructive context, this automatically means that `CompilerForCAP` also acts as a **proof assistant**, that is, as a tool for formalizing and verifying proofs on a computer: Verifying that the implementation of the composition is associative also proves the mathematical statement that the composition in $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ is associative, assuming that the implementation of $\boldsymbol{\mathcal{C}}(R)^{\oplus}$ is faithful to the mathematics. In fact, we will see that the steps used for verifying the implementation correspond to the steps in a proof of the mathematical statement on paper.

Note that while `CompilerForCAP` has some built-in rules, it has no engine for finding general proof tactics and is not verified itself. Moreover, compared to purpose built proof assistants like Coq, Agda, or Lean, `CompilerForCAP` is far less rigorous and requires more manual checks. The reason for this is that the primary goal of `CompilerForCAP` is to optimize computations in categorical towers, as we have seen in Chapter 6, while the proof assistant mode is an additional feature added on top. The advantage of this approach is that `CompilerForCAP` can not only verify categorical towers, but can also generate efficient primitive implementations from the verified categorical towers, as we have seen in Chapter 6. In this sense, Coq, Agda, or Lean are **proof-centric**, while `CAP` and `CompilerForCAP` are **algorithm-centric**.

This chapter is structured as follows: In Section 7.1, we introduce the basic techniques for using `CompilerForCAP` as a proof assistant using a simple guiding example. In Section 7.2, we see a more advanced example of a proof using `CompilerForCAP`. In Section 7.3, we see that `CompilerForCAP` can verify degenerate cases without manual input at all. This is of interest because issues in computer implementations disproportionately often appear in degenerate cases. In the last section Section 7.4, we prove the correctness of the constructions of the categorical tower used for endowing categories of matrices with a homomorphism structure in Construction 4.1.3. We conclude with Example 7.4.1, an example of an actual flaw in an implementation discovered by using CompilerForCAP as a proof assistant. This example is of interest because the flaw is typical for the kind of issues we expect to find by formally verifying code.

7.1 Every monoid defines a category with a single object

In this section, we give a simple example demonstrating in detail how one can use CompilerForCAP as a proof assistant. Our aim is to prove the following proposition:

Proposition 7.1.1. *Let M be a monoid. Then $\boldsymbol{\mathcal{C}}(M)$ is a category.*

The manual proof, which we will give in Section 7.1.2, will be at a first semester level. This allows us to first focus on the technical subtleties of using CompilerForCAP as a proof assistant before we use the same techniques for a mathematically more advanced proof in Section 7.2.

7.1.1 An implementation of $\boldsymbol{\mathcal{C}}(\mathbb{Z})$

For proving Proposition 7.1.1 in CAP, we first need an implementation of $\boldsymbol{\mathcal{C}}(M)$ for a monoid M in CAP. To make things as concrete as possible, we first start with $M = (\mathbb{Z}, \cdot, 1)$ and generalize to arbitrary monoids in Section 7.1.6. In GAP, the ring of integers is available in the global variable Integers.

For the implementation, we first load CAP and create a CAP category using the operations explained in Section 1.3.2. Here, we use a more elaborate version of CreateCapCategory than in Section 1.3.2. This version is called CreateCapCategoryWithDataTypes and allows us to specify additional filters for the category, objects, morphisms, and 2-cells[1], as well as for the unboxed values. Here, the only relevant information for this example are the filters IsUnicodeCharacter and IsInt: We want to define the unique object of $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ via the unicode character "$\star$", and want unboxed morphisms to be given by integers.

→ D.1.1

```
gap> LoadPackage( "CAP", false );
true
gap> monoid_as_category_ZZ := CreateCapCategoryWithDataTypes(
>        "MonoidAsCategory( ZZ )", IsCapCategory,
>        IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryTwoCell,
>        IsUnicodeCharacter, IsInt, fail
>    );
MonoidAsCategory( ZZ )
```

Note that the listing is marked by gray lines on the left and right and has a reference attached at the top right corner. This reference links to a listing in the appendix which contains a copyable version of the code.

Next, we create the unique object using the unicode character "$\star$":

[1] 2-cells are "morphisms of morphisms" in higher category theory and do not appear in this thesis.

→ D.1.2

```
gap> star := UChar( 8902 ); # '⋆' has code point 8902 in decimal notation
'⋆'
gap> unique_object := AsCapCategoryObject( monoid_as_category_ZZ, star );
<An object in MonoidAsCategory( ZZ )>
```

Then, we add the required operations as in Section 1.3.2. Note that we can use the operation `AsInteger` instead of the generic `AsPrimitiveValue` for unboxing morphisms since we have provided the information that morphisms are given by integers to `CAP`.

→ D.1.3

```
gap> AddIsWellDefinedForMorphisms( monoid_as_category_ZZ,
>      { cat, mor } -> AsInteger( mor ) in Integers
>    );
gap> AddIsCongruentForMorphisms( monoid_as_category_ZZ,
>      { cat, mor1, mor2 } -> AsInteger( mor1 ) = AsInteger( mor2 )
>    );
gap> AddIsEqualForObjects( monoid_as_category_ZZ,
>      { cat, obj1, obj2 } -> true
>    );
gap> AddPreCompose( monoid_as_category_ZZ,
>      { cat, mor1, mor2 } -> AsCapCategoryMorphism( cat,
>        Source( mor1 ), AsInteger( mor1 ) * AsInteger( mor2 ), Target( mor2 )
>      )
>    );
gap> AddIdentityMorphism( monoid_as_category_ZZ,
>      { cat, obj } -> AsCapCategoryMorphism( cat,
>        obj, One( Integers ), obj
>      )
>    );
```

Now, we can finalize the category and introduce some LaTeX symbols which will be used later when producing LaTeX output:

→ D.1.4

```
gap> Finalize( monoid_as_category_ZZ );;
gap> Integers!.LaTeXSymbol := "\\mathbb{Z}";;
gap> monoid_as_category_ZZ!.LaTeXSymbol
>                            := "\\boldsymbol{\\mathcal{C}}(\\mathbb{Z})";;
gap> monoid_as_category_ZZ!.VariableLaTeXSymbols := rec(
>        alpha := "m",
>        beta := "n",
>        gamma := "l"
>    );;
```

Since we want to apply `CompilerForCAP` to this implementation, we have to make sure that all functions appearing in the implementation are pure. By Convention 1.4.19, the technical equalities are chosen in such a way that boxing and unboxing, that is, `AsCapCategory` and `AsInteger`, are pure functions. Moreover, checking membership in the integers with "`in`" and the integer multiplication "$*$" are pure functions. The result of `One( Integers )` is just the integer 1, which we can view as a constant, so we do not have to consider the purity of `One`. Finally, the mathematical equality and the technical equality on integers coincide, so testing the mathematical equality on integers is a pure function. Summing up, all functions appearing in the implementation are pure.

Furthermore, note that the mathematical equality and the technical equality on morphisms coincide. Thus, the purity of the composition implies that composition is compatible with the technical equality and hence also with the mathematical equality on morphisms. This will be important later: `CAP` has no generic mechanism for specifying certificates for the mathematical equality of two morphisms, so we cannot formalize the compatibility of the composition with the mathematical equality.

Remark 7.1.2 (The importance of trivialities)**.** Above, checking the purity of the code and the compatibility of the composition with the mathematical equality on morphisms was easy and might seem trivial. This is one of the advantages of building categorical towers: The goal is to split the construction of a category into many simple or even trivial steps. However, to properly support the resulting categorical tower, each level of the tower must be checked with the same diligence as one would check the tower as a whole.

7.1.2 A manual proof

We first give a manual proof for Proposition 7.1.1 and later repeat the proof with assistance of `CompilerForCAP`. The manual proof will be at a first semester level because all the details which one would dismiss as trivial at a more advanced level will actually show up in the formalized proof. In particular, the only proper mathematical rules we will encounter are the axioms of M. The simplicity of the example allows us to first focus on the technical subtleties of using `CompilerForCAP` as a proof assistant before we use the same techniques for a mathematically more advanced proof in Section 7.2.

Manual proof of Proposition 7.1.1. We construct $\mathcal{C}(M)$ as in Construction 2.3.7 with a unique object $\star$.

First, we check that composition actually constructs a morphism from $\star$ to $\star$. For this, let $\boxed{m}, \boxed{n} : \star \to \star$ be two composable morphisms. The composite is constructed as $\boxed{m \cdot n} : \star \to \star$. We note that

- we can actually apply the multiplication of M to m and n,
- the result of this multiplication is an element of M again,

so the composition actually constructs a morphism from $\star$ to $\star$. Here, the fact that we can apply the multiplication of M to m and n might seem trivial, but in more complex examples this point is important: For example, in the context of matrix multiplication we would have to check that the number of columns of the first matrix coincides with the number of rows of the second matrix, which might involve a non-trivial proof. For this reason, we already stress this point here.

Next, we check that composition is compatible with the equality on the sets of morphisms. For this, let $\boxed{m_1}, \boxed{m_2}, \boxed{n_1}, \boxed{n_2} : \star \to \star$ be four morphisms with $\boxed{m_1} = \boxed{m_2}$ and $\boxed{n_1} = \boxed{n_2}$. We unfold the definition of the equality on morphisms and see that $m_1 = m_2$ and $n_1 = n_2$ as elements of M. Hence, we have $m_1 \cdot n_1 = m_2 \cdot n_2$ and thus $\boxed{m_1 \cdot n_1} = \boxed{m_2 \cdot n_2}$ as desired.

Next, we check that composition is associative. For this, let $\boxed{m}, \boxed{n}, \boxed{\ell} : \star \to \star$ be three morphisms. We have to show that

$$(\boxed{m} \cdot \boxed{n}) \cdot \boxed{\ell} = \boxed{m} \cdot (\boxed{n} \cdot \boxed{\ell}).$$

Unfolding the definitions of the composition and the equality on morphisms, we see that this is equivalent to

$$(m \cdot n) \cdot \ell = m \cdot (n \cdot \ell).$$

This equality of elements of M is just the associativity axiom of M.

Next, note that the neutral element of M is indeed an element of M, so the identity morphism actually is a morphism from $\star$ to $\star$.

Finally, we check that the identity morphisms are neutral elements of the composition. For this, let $\boxed{m} : \star \to \star$ be a morphism. We have to show that

$$\mathrm{id}_\star \cdot \boxed{m} = \boxed{m} \qquad \text{and} \qquad \boxed{m} \cdot \mathrm{id}_\star = \boxed{m}.$$

Unfolding the definitions of the composition, the identity morphisms, and the equality on morphisms, we see that this is equivalent to

$$1_M \cdot m = m \qquad \text{and} \qquad m \cdot 1_M = m.$$

These equalities of elements of M are just the identity axiom of M. ■

7.1.3 Formalizing proofs in `CAP`

We would like to formalize the manual proof of Proposition 7.1.1 in `CAP` with the help of `CompilerForCAP`. As the first example of a formalization, we consider the check that composition in $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ is associative. For three morphisms $\boxed{m} : \star \to \star$, $\boxed{n} : \star \to \star$, and $\boxed{\ell} : \star \to \star$, this check can be formalized in a straightforward way as the following function:

Listing 7.1.1: Formalization of the associativity of the composition

```
function ( m, n, l )
  local m_n, left, n_l, right;

    m_n := PreCompose( m, n );
    left := PreCompose( m_n, l );

    n_l := PreCompose( n, l );
    right := PreCompose( m, n_l );

    return IsCongruentForMorphisms( left, right );
end
```

This function

- gets three morphisms `m`, `n`, and `l` as inputs,
- composes the inputs once with the parentheses on the left and once with the parenthesis on the right, and
- returns whether both compositions give equal results.

That is, this function can be used to check if the composition of three given morphisms in $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ is associative. More generally, composition in $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ is associative if and only if this function returns `true` for every three morphisms in $\boldsymbol{\mathcal{C}}(\mathbb{Z})$.

To show this general statement, we first expand the definitions of the categorical operations `PreCompose` and `IsCongruentForMorphisms`, where for reasons of readability, we omit sources and targets:

```
function ( m, n, l )
  local m_n, left, n_l, right;

```

```
    m_n := AsCapCategoryMorphism(..., AsInteger( m ) * AsInteger( n ), ...);
    left := AsCapCategoryMorphism(..., AsInteger( m_n ) * AsInteger( l ), ...);

    n_l := AsCapCategoryMorphism(..., AsInteger( n ) * AsInteger( l ), ...);
    right := AsCapCategoryMorphism(..., AsInteger( m ) * AsInteger( n_l ), ...);

    return AsInteger( left ) = AsInteger( right );
end
```

We note that `m_n` is a boxed morphism which is immediately unboxed as an integer again in the computation of left. The same happens for `left` itself and `n_l` and `right` as well. Purity of boxing and unboxing ensures that we can cancel boxing immediately followed by unboxing without changing the result:

```
function ( m, n, l )
  local m_n_unboxed, left, n_l, right;

    m_n_unboxed := AsInteger( m ) * AsInteger( n );
    left_unboxed := m_n_unboxed * AsInteger( l );

    n_l_unboxed := AsInteger( n ) * AsInteger( l );
    right_unboxed := AsInteger( m ) * n_l_unboxed;

    return left_unboxed = right_unboxed;
end
```

Now, purity ensures that the order of evaluation is irrelevant, so we can avoid storing intermediate results and simplify the function further to

```
function ( m, n, l )
    return (AsInteger( m ) *  AsInteger( n )) * AsInteger( l ) =
            AsInteger( m ) * (AsInteger( n )  * AsInteger( l ));
end
```

Now, we can use associativity of the multiplication of integers to move the parentheses on the right-hand side of the equation from the right to the left without changing the result and obtain:

```
function ( m, n, l )
    return (AsInteger( m ) * AsInteger( n )) * AsInteger( l ) =
           (AsInteger( m ) * AsInteger( n )) * AsInteger( l );
end
```

Here, we treat the associativity of the multiplication of integers in `GAP` as an axiom for the following reasons: We certainly cannot verify a program down to the level of processor instructions, so we have to stop at some level. Since we want to generalize to arbitrary monoids later, where associativity of the multiplication is an axiom anyway, it makes sense to stop at the level of this axiom.

Finally, both sides of the equation are now given by the same expression. Purity of the code ensures that evaluation of both sides will return the same value, so the equality will always evaluate to `true` and we can simplify the function to

```
function ( m, n, l )
    return true;
end
```

This function certainly returns `true` for every input. Since no transformation step has changed the result, also the original function must return `true` for every input. This proves the statement.

The transformation steps above can be automated using `CompilerForCAP`: `CompilerForCAP` automatically

- expands the implementations of `CAP` operations,
- cancels boxing immediately followed by unboxing,
- avoids storing intermediate results, and
- simplifies equalities with the same expression on both sides.

Moreover, mathematical equalities like the associativity of the integer multiplication, which typically come from axioms or previously proven lemmata, can be taught to `CompilerForCAP`. We will see how this works in the next section.

7.1.4 Automating proofs using `CompilerForCAP`

We now show how the steps in the previous section can be automated using `CompilerForCAP`. Using the implementation `monoid_as_category_ZZ` of $\mathcal{C}(\mathbb{Z})$ in Section 7.1.1, we proceed as follows:

First, we start `CompilerForCAP` in its proof assistant mode:

→ D.1.5

```
gap> LoadPackage( "CompilerForCAP", false );
true
gap> CapJitEnableProofAssistantMode( );
```

Next, we modify the formalization of the associativity of the composition in Listing 7.1.1 to also include variables for the sources and targets of the three morphisms:

→ D.1.6

```
gap> statement := function ( cat, A, B, C, D, m, n, l )
>      local m_n, left, n_l, right;
>
>        m_n := PreCompose( m, n );
>        left := PreCompose( m_n, l );
>
>        n_l := PreCompose( n, l );
>        right := PreCompose( m, n_l );
>
>        return IsCongruentForMorphisms( left, right );
>    end;;
```

We do not actually need the variables for source and targets in $\mathcal{C}(\mathbb{Z})$ because there only is a single object, but for demonstration purposes we already provide the general case. Then we can turn this formalization into a lemma using the function `StateLemma`:

→ D.1.7

```
gap> StateLemma(
>        "composition is associative",
>        statement,
>        monoid_as_category_ZZ,
>        [ "category", "object", "object", "object", "object",
>          "morphism", "morphism", "morphism" ],
>        [
>            rec( src_template := "Source( m )", dst_template := "A" ),
>            rec( src_template := "Target( m )", dst_template := "B" ),
>            rec( src_template := "Source( n )", dst_template := "B" ),
>            rec( src_template := "Target( n )", dst_template := "C" ),
>            rec( src_template := "Source( l )", dst_template := "C" ),
>            rec( src_template := "Target( l )", dst_template := "D" ),
>        ]
>    );
```

→ D.1.7

The following arguments are given to `StateLemma`:

1. a human readable description of the statement,
2. the formalized statement,
3. the category for which we want to prove the statement,
4. a list of strings describing the data types of the arguments of the statement,
5. a list of preconditions.

Here, the preconditions assign source and targets of morphisms to encode the condition that the morphisms are composable. Again, in $\mathcal{C}(\mathbb{Z})$ this would not be necessary because there only is a single object, but for demonstration purposes we already provide the general case.

In an interactive `GAP` session, `StateLemma` displays the stated lemma as a plain text output. In this thesis, for demonstration purposes we use a modified version of `StateLemma` which outputs LaTeX code.[2] The LaTeX code generated by the above call of `StateLemma` produces the following paragraph marked by lines on the left and right:

Lemma 7.1.3. *In $\mathcal{C}(\mathbb{Z})$, composition is associative: For four objects $\boxed{A}$, $\boxed{B}$, $\boxed{C}$, and $\boxed{D}$ and three morphisms $\boxed{m} : \boxed{A} \to \boxed{B}$, $\boxed{n} : \boxed{B} \to \boxed{C}$, and $\boxed{\ell} : \boxed{C} \to \boxed{D}$ we have*

$$(\boxed{m} \cdot \boxed{n}) \cdot \boxed{\ell} = \boxed{m} \cdot \big(\boxed{n} \cdot \boxed{\ell}\big).$$

→ D.1.8

Note that `CompilerForCAP` has already eliminated storing the intermediate results explicitly and instead generates a single formula.

Proof. `CompilerForCAP` automatically

- expands the definitions of the composition and the mathematical equality on morphisms and
- cancels boxing immediately followed by unboxing

in the background. We can now simply use the command

```
gap> PrintLemma( );
```

→ D.1.9

to get the current state of the lemma as LaTeX code. We see that, just as in the manual proof of Proposition 7.1.1 in Section 7.1.2, `CompilerForCAP` requires us to show that

$$(m \cdot n) \cdot \ell = m \cdot (n \cdot \ell).$$

→ D.1.10

To show this, we want to use the associativity of the integer multiplication as an axiom, which we can teach to `CompilerForCAP` as follows:

[2] This version of `StateLemma` is not an official part of `CompilerForCAP` because it is difficult to generate a readable LaTeX output in more complex examples.

→ D.1.11

```
gap> ApplyLogicTemplate( rec(
>       variable_names := [ "a", "b", "c" ],
>       variable_filters := [ IsInt, IsInt, IsInt ],
>       src_template := "a * (b * c)",
>       dst_template := "(a * b) * c",
>    ) );
```

This introduces a logic template which teaches `CompilerForCAP` that it may replace any occurrence of an expression of the form

`a * (b * c)`

in the code by

`(a * b) * c`,

where `a`, `b`, and `c` can be arbitrary expressions computing an integer. The command `ApplyLogicTemplate` applies this logic template a single time, that is, it looks for some expression matching `a * (b * c)` in the current state of the proof and replaces it by `(a * b) * c`. If no occurrence can be found, an error is raised. If multiple occurrences appear, only a single occurrence is replaced.[3] Again, for demonstration purposes we use a version of `ApplyLogicTemplate` which translates the logic template to more readable LaTeX code, which looks as follows:

For integers a, b, and c we have:

→ D.1.12

$$\text{(rule)} \qquad a \cdot (b \cdot c) = (a \cdot b) \cdot c.$$

`CompilerForCAP` detects that now both sides of the equation we had to show are given by the same expression and automatically simplifies the statement. We again use the command

→ D.1.13

```
gap> PrintLemma( );
```

and see that `CompilerForCAP` has simplified the statement to

→ D.1.14

$$\blacksquare.$$

We use ■ to denote the boolean value True. Hence, the claim is proven and we end the proof using the command

→ D.1.15

```
gap> AssertLemma( );
```

which produces:

→ D.1.16

With this, `CompilerForCAP` has verified the lemma. ■

In the next section we will not explicitly list all commands used to generate the paragraphs marked by lines on the left and right anymore. The commands can still be found by following the references in the top right corners of the paragraphs.

[3] For an example why `ApplyLogicTemplate` only replaces a single occurrence, consider an expression of the form $a + b = b + a$ which we want to simplify using commutativity: If we apply commutativity only a single time, for example on the left-hand side of the equation, we obtain $b + a = b + a$, which trivially holds true. If we would use commutativity on both sides of the equation, we would obtain $b + a = a + b$, which is just an alternative form of the original equation.

7.1.5 $\mathcal{C}(\mathbb{Z})$ is a category

In the last section, we have seen how we can state and prove a single lemma using CompilerForCAP. In this section, we want to use CompilerForCAP to prove a larger proposition: We want to prove that $\mathcal{C}(\mathbb{Z})$ is a category. For convenience, CompilerForCAP includes a library of lemmata corresponding to such propositions. We can state the proposition that $\mathcal{C}(\mathbb{Z})$ is a category as follows:

```
gap> StateProposition( monoid_as_category_ZZ, "is_category" );
```
→ D.1.17

This produces the following output:

Proposition 7.1.4. *$\mathcal{C}(\mathbb{Z})$ is a category.* → D.1.18

Now, we can use

```
gap> StateNextLemma( );
```
→ D.1.19

to state the next lemma needed for the current proposition. Here, the lemmata correspond to the substatements of the manual proof of Proposition 7.1.1 in Section 7.1.2. The first lemma reads:

Lemma 7.1.5. *In $\mathcal{C}(\mathbb{Z})$, the composite of two morphisms is a morphism: For three objects $\boxed{A}$, $\boxed{B}$, and $\boxed{C}$ and two morphisms $\boxed{m} : \boxed{A} \to \boxed{B}$ and $\boxed{n} : \boxed{B} \to \boxed{C}$ we have* → D.1.20

$$\boxed{m} \cdot \boxed{n} \text{ defines a morphism in } \mathcal{C}(\mathbb{Z}) \text{ from } \boxed{A} \text{ to } \boxed{C}.$$

Note that CAP does not know that $\mathcal{C}(\mathbb{Z})$ only has a single object, so the lemma is stated with generic sources and targets.

Proof. We have to show that

$$m \cdot n \in \mathbb{Z}.$$ → D.1.21

Just as in the manual proof of Proposition 7.1.1 in Section 7.1.2, this statement can be broken down into two parts:

- m and n form valid inputs for the function "$\cdot$", and
- the result of the function application is an integer.

For deciding the first part automatically, CompilerForCAP would need a formalization of which inputs are valid for the function "$\cdot$". While such a library of preconditions could be implemented in the future, currently we have to manually check such conditions and notify CompilerForCAP once we have done so. Here, the multiplication of integers is certainly applicable to the integers m and n, so m and n indeed form valid inputs for the function "$\cdot$". We tell CompilerForCAP that we have checked all inputs via

```
gap> AttestValidInputs( );
```
→ D.1.22

which produces:

We let CompilerForCAP assume that all inputs are valid. → D.1.23

Now, `CompilerForCAP` has a built-in typing rule saying that the product of integers is an integer, which covers the second part of the statement.

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.24

We continue with the next lemma:

Lemma 7.1.6. *In $\boldsymbol{\mathcal{C}}(\mathbb{Z})$, composition is associative: For four objects $\boxed{A}$, $\boxed{B}$, $\boxed{C}$, and $\boxed{D}$ and three morphisms $\boxed{m} : \boxed{A} \to \boxed{B}$, $\boxed{n} : \boxed{B} \to \boxed{C}$, and $\boxed{\ell} : \boxed{C} \to \boxed{D}$ we have* → D.1.25

$$\left(\boxed{m} \cdot \boxed{n}\right) \cdot \boxed{\ell} = \boxed{m} \cdot \left(\boxed{n} \cdot \boxed{\ell}\right).$$

Proof. We have to show that

$$(m \cdot n) \cdot \ell = m \cdot (n \cdot \ell).$$ → D.1.26

We have done this before in Section 7.1.4, so we can reuse the following rule: For integers a, b, and c we have:

$$\text{(rule)} \qquad a \cdot (b \cdot c) = (a \cdot b) \cdot c.$$ → D.1.27

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.28

Lemma 7.1.7. *In $\boldsymbol{\mathcal{C}}(\mathbb{Z})$, the identity on an object defines a morphism: For an object $\boxed{A}$ we have* → D.1.29

$$\mathrm{id}_{\boxed{A}} \textit{ defines a morphism in } \boldsymbol{\mathcal{C}}(\mathbb{Z}) \textit{ from } \boxed{A} \textit{ to } \boxed{A}.$$

Proof. We have to show that

$$1_{\mathbb{Z}} \in \mathbb{Z}.$$ → D.1.30

The integers have a multiplicative identity, so:

We let CompilerForCAP assume that all inputs are valid. → D.1.31

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.32

Lemma 7.1.8. *In $\boldsymbol{\mathcal{C}}(\mathbb{Z})$, identity morphisms are left neutral: For two objects $\boxed{A}$ and $\boxed{B}$ and a morphism $\boxed{m} : \boxed{A} \to \boxed{B}$ we have* → D.1.33

$$\mathrm{id}_{\boxed{A}} \cdot \boxed{m} = \boxed{m}.$$

Proof. We have to show that

$$1_{\mathbb{Z}} \cdot m = m.$$ → D.1.34

We use the first part of the identity axiom of $\mathbb{Z}$, that is, for every integer $a \in \mathbb{Z}$ we have:

$$\text{(rule)} \qquad 1_{\mathbb{Z}} \cdot a = a.$$ → D.1.35

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.36

Lemma 7.1.9. *In $\boldsymbol{\mathcal{C}}(\mathbb{Z})$, identity morphisms are right neutral: For two objects $\boxed{A}$ and $\boxed{B}$ and a morphism $\boxed{m} : \boxed{A} \to \boxed{B}$ we have* → D.1.37

$$\boxed{m} \cdot \mathrm{id}_{\boxed{B}} = \boxed{m}.$$

Proof. We have to show that

$$m \cdot 1_{\mathbb{Z}} = m.$$ → D.1.38

We use the second part of the identity axiom of $\mathbb{Z}$, that is, for every integer $a \in \mathbb{Z}$ we have:

$$\text{(rule)} \qquad a \cdot 1_{\mathbb{Z}} = a.$$ → D.1.39

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.40

This completes the proof of the proposition. We can use the command `AssertProposition` to see that `CompilerForCAP` has indeed verified the proposition:

Summing up, we have shown: $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ is a category. ■ → D.1.41

Remark 7.1.10 (Compatibility of the composition with the mathematical equality on morphisms)**.** Almost all substatements of the manual proof of Proposition 7.1.1 in Section 7.1.2 are fully algorithmic and appear in the formalization above. The only exception is checking that the composition is compatible with the mathematical equality on morphisms: We cannot properly formalize this statement because `CAP` has no generic mechanism for specifying certificates for the mathematical equality of two morphisms. Hence, we have to prove that the composition is compatible with the mathematical equality on morphisms by hand. We have already done this for our implementation of $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ in Section 7.1.1 right after the implementation.

7.1.6 Generalization to arbitrary monoids with the help of dummy implementations

For technical reasons, `CompilerForCAP` currently needs a concrete category when expanding categorical operations, for example the category $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ above. Hence, when using `CompilerForCAP`, a priori we can only prove Proposition 7.1.1 for a concrete monoid, for example for $M = \mathbb{Z}$ as above. This is a technical limitation of `CAP`: `CAP` currently has no mechanism for specifying algorithms for a “family” of categories $\boldsymbol{\mathcal{C}}(M)$ for an arbitrary monoid M in a declarative way. However, we can at least easily generalize the implementation of $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ in Section 7.1.1 to any monoid in `GAP` whose elements can be multiplied by the operation “$*$” and which supports the operation `One`: To do so, we simply replace `Integers` by the monoid and `IsInt` by a filter for the elements of the monoids. We can also generalize the rules used in the proofs in a similar way because we have only used the monoid axioms and no implementation details of the integers in `GAP`. Hence, all of the above would work for an arbitrary monoid in `GAP`. In fact, we never actually use the implementation of the monoid in the proof and neither execute any code, so the monoid does not even have to be properly implemented in `GAP`. Therefore, we can manually conclude that our proof for $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ generalizes to a proof of Proposition 7.1.1.

To formalize this, we can use a **dummy implementation** of a monoid. A **dummy implementation** of a concept seems to provide an interface for the concept, but calling any operation in this interface will simply signal an error. Hence, when using a dummy implementation in a proof, we can be sure that we only rely on the abstract interface but not on any implementation details, for the simple reason that there is no actual implementation.

We could implement a **dummy monoid** in `GAP` as follows: We first declare new filters `IsDummyMonoid` and `IsDummyMonoidElement` for our dummy monoid and its elements, inheriting from the generic filters for monoids:

```
DeclareFilter( "IsDummyMonoid", IsMonoid );
DeclareFilter( "IsDummyMonoidElement", IsMultiplicativeElementWithOne );
```

Afterwards, we provide dummy implementations for the equality and multiplication of monoid elements as well as for the neutral element of the monoid. These dummy implementations simply signal errors.

```
# implementation of the equality of elements of the dummy monoid
InstallMethod( \=, [ IsDummyMonoidElement, IsDummyMonoidElement ],
    function ( m, n )
        Error( "not implemented" );
    end
);

# implementation of the multiplication of elements of the dummy monoid
InstallMethod( \*, [ IsDummyMonoidElement, IsDummyMonoidElement ],
    function ( m, n )
        Error( "not implemented" );
    end
);

# implementation of the neutral element of the monoid
InstallMethod( One, [ IsDummyMonoid ],
    function ( M )
        Error( "not implemented" );
    end
);
```

Finally, we create a dummy monoid lying in the filter `IsDummyMonoid`. Creating a value in a concrete filter in `GAP` is a bit involved and can be done as follows:

```
dummy_monoid := Objectify( NewType( NewFamily( "" ), IsDummyMonoid ), rec( ) );
```

If we now repeat the proof in Section 7.1.5 using `dummy_monoid` instead of the integers, we can be sure that we do not rely on any details of the implementation simply because `dummy_monoid` has no actual implementation. Hence, we can be sure that we have actually proven Proposition 7.1.1 in full generality.

For dummy implementations of categories, `CAP` provides the constructor `DummyCategory`. Like any dummy implementation, such dummy categories have no actual implementation but can still be used as inputs for category constructors. We will see proofs using a dummy category in the next section.

7.2 The additive closure of a preadditive category is a category

Let $\mathbf{C}$ be a preadditive category. In this section, we want to use `CompilerForCAP` for a proof in a more complex setting: We want to prove the correctness of Construction 2.4.6, that is, we want to show that the additive closure $\mathbf{C}^{\oplus}$ of $\mathbf{C}$ is a category. This example exhibits some concepts and subtleties which we have not encountered until now:

- The setup is more complex: We use a dummy category and introduce two global logic templates.
- When showing that the composite of two morphisms is a morphism, we actually have to do some work to show that the inputs for the categorical operations in the construction are valid.
- The rules we use in the proofs are much more complex. For example, in lemmata dealing with identity morphisms, we will see rules handling case distinctions.

For the setup, we use an implementation of `AdditiveClosure` available in the package `FreydCategoriesForCAP` [BPZ24]. This implementation encodes matrices in **row-major order**, that is, a morphism $\boxed{\alpha} : \boxed{A} \to \boxed{B}$ in $\mathbf{C}^{\oplus}$ is encoded as an a-tuple of b-tuples of morphisms in $\mathbf{C}$ with components $(\alpha_i)_j : A_i \to B_j$. As always, the implementation only uses pure functions. For $\mathbf{C}$, we use a preadditive dummy category with the required operations.

→ D.1.42

```
gap> LoadPackage( "FreydCategoriesForCAP", false );
true
gap> dummy := DummyCategory( rec(
>       name := "a preadditive category",
>       list_of_operations_to_install := [
>           "IsEqualForObjects",
>           "IsWellDefinedForObjects",
>           "IsWellDefinedForMorphismsWithGivenSourceAndRange",
>           "IsCongruentForMorphisms",
>           "PreCompose",
>           "IdentityMorphism",
>           "SumOfMorphisms",
>           "ZeroMorphism",
>           "AdditiveInverseForMorphisms",
>       ],
>       properties := [
>           "IsAbCategory", # another name for a preadditive category
>       ],
>   ) );;
gap> dummy!.LaTeXSymbol := "\\mathbf{C}";;
gap> additive_closure := AdditiveClosure( dummy );
AdditiveClosure( a preadditive category )
gap> additive_closure!.LaTeXSymbol := "\\mathbf{C}^\\oplus";;
gap> LoadPackage( "CompilerForCAP", false );
true
gap> CapJitEnableProofAssistantMode( );
```

The dummy category has no actual implementation, so we have to prevent `CompilerForCAP` from trying to expand the dummy implementations of the `CAP` operations:

→ D.1.43

```
gap> StopCompilationAtPrimitivelyInstalledOperationsOfCategory( dummy );
```

Next, we introduce two global logic templates, which tell `CompilerForCAP` that for a morphism $\boxed{\alpha} : \boxed{A} \to \boxed{B}$, the component $(\alpha_i)_j$ is a morphism from A_i to B_j:

→ D.1.44

```
gap> CapJitAddLogicTemplate( rec(
>       variable_names := [ "alpha", "i", "j" ],
>       variable_filters := [ IsAdditiveClosureMorphism, IsInt, IsInt ],
>       src_template := "Source( MorphismMatrix( alpha )[i][j] )",
>       dst_template := "ObjectList( Source( alpha ) )[i]",
>   ) );
gap> CapJitAddLogicTemplate( rec(
>       variable_names := [ "alpha", "i", "j" ],
>       variable_filters := [ IsAdditiveClosureMorphism, IsInt, IsInt ],
>       src_template := "Target( MorphismMatrix( alpha )[i][j] )",
>       dst_template := "ObjectList( Target( alpha ) )[j]",
>   ) );
```

→ D.1.44

Here, `ObjectList` and `MorphismMatrix` unbox objects and morphisms in `AdditiveClosure`.

Finally, note that the mathematical equality on morphisms in $\mathbf{C}^{\oplus}$ is given by the entrywise mathematical equality on morphisms in $\mathbf{C}$. All morphisms in $\mathbf{C}^{\oplus}$ are constructed by applying operations in $\mathbf{C}$ which respect the mathematical equality on morphisms in $\mathbf{C}$. Hence, all constructions in $\mathbf{C}^{\oplus}$ respect the mathematical equality on morphisms in $\mathbf{C}^{\oplus}$.

Now we can use `CompilerForCAP` to state:

Proposition 7.2.1. *$\mathbf{C}^{\oplus}$ is a category.* → D.1.45

Lemma 7.2.2. *In $\mathbf{C}^{\oplus}$, the composite of two morphisms is a morphism: For three objects $\boxed{A}$, $\boxed{B}$, and $\boxed{C}$ and two morphisms $\boxed{\alpha} : \boxed{A} \to \boxed{B}$ and $\boxed{\beta} : \boxed{B} \to \boxed{C}$ we have*

$$\boxed{\alpha} \cdot \boxed{\beta} \textit{ defines a morphism in } \mathbf{C}^{\oplus} \textit{ from } \boxed{A} \textit{ to } \boxed{C}.$$

→ D.1.46

Proof. We have to show that

$$\left(\left(\sum_{k=1}^{b} \underbrace{(\alpha_i)_k \cdot (\beta_k)_j}_{A_i \to C_j} \right)_{j=1}^{c} \right)_{i=1}^{a} \text{ is a tuple}$$

$$\text{and } \bigwedge_{i=1}^{a} \left(\sum_{k=1}^{b} \underbrace{(\alpha_i)_k \cdot (\beta_k)_j}_{A_i \to C_j} \right)_{j=1}^{c} \text{ is a tuple}$$

$$\text{and } \bigwedge_{i=1}^{a} \bigwedge_{j=1}^{c} \sum_{k=1}^{b} \underbrace{(\alpha_i)_k \cdot (\beta_k)_j}_{A_i \to C_j} \text{ defines a morphism in } \mathbf{C} \text{ from } A_i \text{ to } C_j.$$

→ D.1.47

Note that to save horizontal space in automatically generated formulas, we write

$$\bigwedge_{x=1}^{\ell} P(x) \qquad \text{instead of} \qquad \forall x \in \{1, \dots, \ell\} \colon P(x).$$

This is the first time we actually have to check some conditions to show that all inputs are valid: We have to show that

- $(\alpha_i)_k$ and $(\beta_k)_j$ are composable and are hence valid inputs for the composition, and
- the composites $(a_i)_k \cdot (\beta_k)_j$ are morphisms from A_i to C_j as indicated by the brace, and are hence valid inputs for the sum.

By assumption, $\boxed{\alpha}$ has source $\boxed{A}$ and target $\boxed{B}$, so $(\alpha_i)_k$ has source A_i and target B_k. Similarly, $\boxed{\beta}$ has source $\boxed{B}$ and target $\boxed{C}$, so $(\beta_k)_j$ has source B_k and target C_j. Hence, $(\alpha_i)_k$ and $(\beta_k)_j$ are composable and the composite $(a_i)_k \cdot (\beta_k)_j$ has source A_i and target C_j. The construction of tuples has no preconditions. Hence:

We let CompilerForCAP assume that all inputs are valid. → D.1.48

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.49

Lemma 7.2.3. *In $\mathbf{C}^{\oplus}$, composition is associative: For four objects $\boxed{A}$, $\boxed{B}$, $\boxed{C}$, and $\boxed{D}$ and three morphisms $\boxed{\alpha} : \boxed{A} \to \boxed{B}$, $\boxed{\beta} : \boxed{B} \to \boxed{C}$, and $\boxed{\gamma} : \boxed{C} \to \boxed{D}$ we have* → D.1.50

$$\left(\boxed{\alpha} \cdot \boxed{\beta}\right) \cdot \boxed{\gamma} = \boxed{\alpha} \cdot \left(\boxed{\beta} \cdot \boxed{\gamma}\right).$$

Proof. We have to show that

→ D.1.51

$$\bigwedge_{i=1}^{a} \bigwedge_{j=1}^{d} \sum_{k_1=1}^{c} \underbrace{\sum_{k_2=1}^{b} \underbrace{(\alpha_i)_{k_2} \cdot (\beta_{k_2})_{k_1}}_{A_i \to C_{k_1}} \cdot (\gamma_{k_1})_j}_{A_i \to D_j} = \underbrace{\sum_{k_1=1}^{b} (\alpha_i)_{k_1} \cdot \sum_{k_2=1}^{c} \underbrace{(\beta_{k_1})_{k_2} \cdot (\gamma_{k_2})_j}_{B_{k_1} \to D_j}}_{A_i \to D_j}.$$

First, we use that composition in $\mathbf{C}$ is additive in both components, that is, for a morphism $\varphi : X \to Y$ in $\mathbf{C}$ and two families of morphisms $\psi_k : W \to X$ and $\chi_k : Y \to Z$ in $\mathbf{C}$ we have

→ D.1.52

$$\text{(rule)} \qquad \sum_{k=1}^{\ell} \underbrace{\psi_k}_{W \to X} \cdot \varphi = \sum_{k=1}^{\ell} \underbrace{\psi_k \cdot \varphi}_{W \to Y}$$

and

→ D.1.53

$$\text{(rule)} \qquad \varphi \cdot \sum_{k=1}^{\ell} \underbrace{\chi_k}_{Y \to Z} = \sum_{k=1}^{\ell} \underbrace{\varphi \cdot \chi_k}_{X \to Z}.$$

With this, it remains to show that

→ D.1.54

$$\bigwedge_{i=1}^{a} \bigwedge_{j=1}^{d} \sum_{k_1=1}^{c} \underbrace{\sum_{k_2=1}^{b} \underbrace{\left((\alpha_i)_{k_2} \cdot (\beta_{k_2})_{k_1}\right) \cdot (\gamma_{k_1})_j}_{A_i \to D_j}}_{A_i \to D_j} = \sum_{k_1=1}^{b} \underbrace{\sum_{k_2=1}^{c} \underbrace{(\alpha_i)_{k_1} \cdot \left((\beta_{k_1})_{k_2} \cdot (\gamma_{k_2})_j\right)}_{A_i \to D_j}}_{A_i \to D_j}.$$

Note that the parentheses on the left-hand side and on the right-hand side differ and that the ranges of the sums are swapped.

To continue, we use that composition in $\mathbf{C}$ is associative, that is, for three composable morphisms φ, ψ and χ in $\mathbf{C}$ we have

→ D.1.55

$$\text{(rule)} \qquad \varphi \cdot (\psi \cdot \chi) = (\varphi \cdot \psi) \cdot \chi.$$

With this, the parentheses on the left-hand side and on the right-hand side agree:

$$\bigwedge_{i=1}^{a}\bigwedge_{j=1}^{d}\sum_{k_1=1}^{c}\underbrace{\sum_{k_2=1}^{b}\underbrace{\left((\alpha_i)_{k_2}\cdot(\beta_{k_2})_{k_1}\right)\cdot(\gamma_{k_1})_j}_{A_i\to D_j}}_{A_i\to D_j}=\sum_{k_1=1}^{b}\underbrace{\sum_{k_2=1}^{c}\underbrace{\left((\alpha_i)_{k_1}\cdot(\beta_{k_1})_{k_2}\right)\cdot(\gamma_{k_2})_j}_{A_i\to D_j}}_{A_i\to D_j}.$$ → D.1.56

Finally, we use that finite sums of morphisms $\varphi_{ij} : X \to Y$ in $\mathbf{C}$ can be interchanged:

$$\text{(rule)}\qquad \sum_{i=1}^{m}\underbrace{\sum_{j=1}^{n}\underbrace{\varphi_{ij}}_{X\to Y}}_{X\to Y}=\sum_{j=1}^{n}\underbrace{\sum_{i=1}^{m}\underbrace{\varphi_{ij}}_{X\to Y}}_{X\to Y}.$$ → D.1.57

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.58

Lemma 7.2.4. *In $\mathbf{C}^{\oplus}$, the identity on an object defines a morphism: For an object $\boxed{A}$ we have* → D.1.59

$$\mathrm{id}_{\boxed{A}} \textit{ defines a morphism in } \mathbf{C}^{\oplus} \textit{ from } \boxed{A} \textit{ to } \boxed{A}.$$

Proof. We have to show that

$$\begin{aligned}
&\left(\left(\begin{Bmatrix}\mathrm{id}_{A_i} & \text{if } i=j,\\ 0_{A_i,A_j} & \text{else.}\end{Bmatrix}\right)_{j=1}^{a}\right)_{i=1}^{a} \text{ is a tuple}\\
\text{and } &\bigwedge_{i=1}^{a}\left(\begin{Bmatrix}\mathrm{id}_{A_i} & \text{if } i=j,\\ 0_{A_i,A_j} & \text{else.}\end{Bmatrix}\right)_{j=1}^{a} \text{ is a tuple}\\
\text{and } &\bigwedge_{i=1}^{a}\bigwedge_{j=1}^{a}\begin{Bmatrix}\mathrm{id}_{A_i} & \text{if } i=j,\\ 0_{A_i,A_j} & \text{else.}\end{Bmatrix} \text{ defines a morphism in } \mathbf{C} \text{ from } A_i \text{ to } A_j.
\end{aligned}$$ → D.1.60

Every object of $\mathbf{C}$ has an identity morphism and there is a zero morphism between every two objects of $\mathbf{C}$. Hence, no preconditions have to be checked, so:

We let CompilerForCAP assume that all inputs are valid. → D.1.61

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.62

Lemma 7.2.5. *In $\mathbf{C}^{\oplus}$, identity morphisms are left neutral: For two objects $\boxed{A}$ and $\boxed{B}$ and a morphism $\boxed{\alpha} : \boxed{A} \to \boxed{B}$ we have* → D.1.63

$$\mathrm{id}_{\boxed{A}}\cdot\boxed{\alpha}=\boxed{\alpha}.$$

Proof. We have to show that

$$\bigwedge_{i=1}^{a}\bigwedge_{j=1}^{b}\underbrace{\sum_{k=1}^{a}\left(\begin{Bmatrix}\mathrm{id}_{A_i} & \text{if } i=k,\\ 0_{A_i,A_k} & \text{else.}\end{Bmatrix}\right)\cdot(\alpha_k)_j}_{A_i\to B_j}=(\alpha_i)_j.$$ → D.1.64

First, we use that we can pull the right factor of the composition into the case distinction, that is, for morphisms $\varphi_1, \varphi_2 : X \to Y$ and $\psi : Y \to Z$ in $\mathbf{C}$ and a predicate P we have:

$$\text{(rule)} \qquad \left(\begin{Bmatrix} \varphi_1 & \text{if } P, \\ \varphi_2 & \text{else.} \end{Bmatrix}\right) \cdot \psi = \begin{Bmatrix} \varphi_1 \cdot \psi & \text{if } P, \\ \varphi_2 \cdot \psi & \text{else.} \end{Bmatrix}.$$ → D.1.65

With this, we still have to show:

$$\bigwedge_{i=1}^{a} \bigwedge_{j=1}^{b} \sum_{k=1}^{a} \underbrace{\begin{Bmatrix} \mathrm{id}_{A_i} \cdot (\alpha_k)_j & \text{if } i = k, \\ 0_{A_i,A_k} \cdot (\alpha_k)_j & \text{else.} \end{Bmatrix}}_{A_i \to B_j} = (\alpha_i)_j.$$ → D.1.66

Now, we can use that for the underlying category the identity is a neutral element and the zeros are absorbing elements with regard to composition, that is, for a morphisms $\varphi : X \to Y$ and an object W in $\mathbf{C}$, we have:

$$\text{(rule)} \qquad \mathrm{id}_X \cdot \varphi = \varphi$$ → D.1.67

and

$$\text{(rule)} \qquad 0_{W,X} \cdot \varphi = 0_{W,Y}.$$ → D.1.68

With this, we still have to show

$$\bigwedge_{i=1}^{a} \bigwedge_{j=1}^{b} \sum_{k=1}^{a} \underbrace{\begin{Bmatrix} (\alpha_k)_j & \text{if } i = k, \\ 0_{A_i,B_j} & \text{else.} \end{Bmatrix}}_{A_i \to B_j} = (\alpha_i)_j.$$ → D.1.69

Now, we use that we can drop zeros from sums, that is, for a family of morphisms $\varphi_k : X \to Y$ in $\mathbf{C}$ we have

$$\text{(rule)} \qquad \sum_{k=1}^{\ell} \underbrace{\begin{Bmatrix} \varphi_k & \text{if } i = k, \\ 0_{X,Y} & \text{else.} \end{Bmatrix}}_{X \to Y} = \varphi_i$$ → D.1.70

if the case $i = k$ actually occurs. Above, this is the case because i and k run over the same range. We usually try to avoid rules with unformalized preconditions like $i = k$ occurring here, but sometimes the required information simply is not available locally at the point where the rule should be applied.

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.71

Lemma 7.2.6. *In $\mathbf{C}^{\oplus}$, identity morphisms are right neutral: For two objects $\boxed{A}$ and $\boxed{B}$ and a morphism $\boxed{\alpha} : \boxed{A} \to \boxed{B}$ we have* → D.1.72

$$\boxed{\alpha} \cdot \mathrm{id}_{\boxed{B}} = \boxed{\alpha}.$$

Proof. This is completely analogous to the previous lemma.

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.73

Summing up, we have shown: $\mathbf{C}^{\oplus}$ is a category. ■ → D.1.74

7.3 The additive closure of a preadditive category has a zero object

Let $\mathbf{C}$ be a preadditive category. In Construction 2.4.9, we have seen that the additive closure $\mathbf{C}^{\oplus}$ of $\mathbf{C}$ has a zero object. We now show the correctness of this construction. This is just a special case of showing that $\mathbf{C}^{\oplus}$ has finite direct sums, but the proof is interesting nevertheless: The zero object is given by the empty tuple, so the constructions do not involve any objects or morphisms in $\mathbf{C}$ and are purely symbolic. We will see that, as a result, we do not have to provide any rules to `CompilerForCAP` at all in the proof. Nevertheless, verifying degenerate cases is important because issues in computer implementations disproportionately often appear in such cases.

We reuse the setup at the beginning of Section 7.2 and state:

Proposition 7.3.1. *$\mathbf{C}^{\oplus}$ has a zero object.* → D.1.75

Lemma 7.3.2. *In $\mathbf{C}^{\oplus}$, the zero object is an object: We have* → D.1.76

$$0_{\mathbf{C}^{\oplus}} \text{ defines an object in } \mathbf{C}^{\oplus}.$$

Proof. We have to show that

$$() \text{ is a tuple.}$$ → D.1.77

There are no preconditions to check, so:

We let CompilerForCAP assume that all inputs are valid. → D.1.78

With this, `CompilerForCAP` has verified the lemma. ∎ → D.1.79

Lemma 7.3.3. *In $\mathbf{C}^{\oplus}$, the universal morphism into the zero objects defines a morphism: For an object $\boxed{A}$ we have* → D.1.80

$$u^{\to 0_{\mathbf{C}^{\oplus}}}(\boxed{A}) \text{ defines a morphism in } \mathbf{C}^{\oplus} \text{ from } \boxed{A} \text{ to } 0_{\mathbf{C}^{\oplus}}.$$

Proof. We have to show that

$$\begin{aligned} (())_{i=1}^{a} &\text{ is a tuple} \\ \text{and } \bigwedge_{i=1}^{a} () &\text{ is a tuple.} \end{aligned}$$ → D.1.81

There are no preconditions to check, so:

We let CompilerForCAP assume that all inputs are valid. → D.1.82

With this, `CompilerForCAP` has verified the lemma. ∎ → D.1.83

Lemma 7.3.4. *In $\mathbf{C}^{\oplus}$, the universal morphism into the zero object is unique: For an object $\boxed{A}$ and a morphism $\boxed{u} : \boxed{A} \to 0_{\mathbf{C}^{\oplus}}$ we have* → D.1.84

$$u^{\to 0_{\mathbf{C}^{\oplus}}}(\boxed{A}) = \boxed{u}.$$

Proof. This is immediate from the construction. ∎

Remark 7.3.5. The reason why this is immediate from the construction is that all morphisms into the zero object have a zero number of columns. Hence, the entrywise comparison ranges over empty tuples and thus trivially holds true.

Lemma 7.3.6. *In $\mathbf{C}^\oplus$, the universal morphism from the zero objects defines a morphism: For an object $\boxed{B}$ we have* → D.1.85

$$u^{\leftarrow 0_{\mathbf{C}^\oplus}}(\boxed{B}) \textit{ defines a morphism in } \mathbf{C}^\oplus \textit{ from } 0_{\mathbf{C}^\oplus} \textit{ to } \boxed{B}.$$

Proof. We have to show that

$$() \text{ is a tuple.}$$ → D.1.86

There are no preconditions to check, so:

We let CompilerForCAP assume that all inputs are valid. → D.1.87

With this, `CompilerForCAP` has verified the lemma. ■ → D.1.88

Lemma 7.3.7. *In $\mathbf{C}^\oplus$, the universal morphism from the zero object is unique: For an object $\boxed{B}$ and a morphism $\boxed{u} : 0_{\mathbf{C}^\oplus} \to \boxed{B}$ we have* → D.1.89

$$u^{\leftarrow 0_{\mathbf{C}^\oplus}}(\boxed{B}) = \boxed{u}.$$

Proof. This is immediate from the construction. ■

Remark 7.3.8. The reason why this is immediate from the construction is that all morphisms from the zero object have a zero number of rows. Hence, the entrywise comparison ranges over an empty tuple and thus trivially holds true.

Summing up, we have shown: $\mathbf{C}^\oplus$ has a zero object. ■ → D.1.90

7.4 Homomorphism structures of categories of matrices

In this section, we use `CompilerForCAP` to prove the correctness of the constructions of the categorical tower used for endowing categories of matrices with a homomorphism structure in Construction 4.1.3. We will not provide the proofs with the same level of detail as in the previous section for two reasons:

- Some of the proofs mainly apply the obvious axioms of the underlying data structures. These proofs follow exactly the same pattern as the proof that $\boldsymbol{\mathcal{C}}(M)$ is a category in Section 7.1.5, which quickly becomes repetitive.
- The more complicated proofs contain complex statements and logic templates which are difficult to visualize as comprehensible LaTeX code. For example, when proving that additive closures have direct sums in Construction 2.4.9, we have only made the construction explicit for **binary** direct sums to keep the notation relatively simple. The proof using `CompilerForCAP` will, however, involve arbitrary direct sums, which means we would have to add even more layers of indices into the notation.

Hence, instead of giving the details in the thesis, we provide links to the code of the proofs, which are part of the test suite of `CompilerForCAP`.

The proof of Construction 4.1.3 constructs a $\mathbf{Mat}_k$-homomorphism structure of $\boldsymbol{\mathcal{C}}(R)^\oplus$ in three steps, which we follow in the next sections.

7.4.1 Algebras over commutative rings define linear categories

The first step in the proof of Construction 4.1.3 notices that $\boldsymbol{\mathcal{C}}(R)$ is a k-linear category with finitely generated free external homs. We have already proven in Section 7.1 that the underlying multiplicative monoid of R defines a category $\boldsymbol{\mathcal{C}}(R)$ with a single object. Now, we use `CompilerForCAP` to show that the addition of R turns $\boldsymbol{\mathcal{C}}(R)$ into a preadditive category (Construction 2.3.7). For this, we can use an implementation of `RingAsCategory` available in the package `FreydCategoriesForCAP`. The basic parts of this implementation match our implementation of $\boldsymbol{\mathcal{C}}(\mathbb{Z})$ in Section 7.1.1. Again, the implementation only uses pure functions, and the mathematical equality and the technical equality on morphisms coincide, so purity also ensures that all operations on morphisms are compatible with the mathematical equality. The proof simply uses the ring axioms and can be found in [Zic24h].

Next, we have to show that $\boldsymbol{\mathcal{C}}(R)$ has a k-linear structure (Construction 2.3.8). The proof uses the k-algebra axioms and can be found in [Zic24i].

Lastly, the fact that R has finitely generated free external homs is just a reformulation of the fact that R is finitely generated free as a k-module.

7.4.2 Homomorphism structures for linear categories

The second step in the proof of Construction 4.1.3 notices that $\boldsymbol{\mathcal{C}}(R)$ has a $\mathbf{Mat}_k$-homomorphism structure (Construction 2.5.7). The proof uses the same techniques but much more complicated rules than the previous proofs and can be found in [Zic24e].

7.4.3 Additive closures

The third step in the proof of Construction 4.1.3 applies Construction 2.5.8 to obtain a $\mathbf{Mat}_k$-homomorphism structure for $\boldsymbol{\mathcal{C}}(R)^{\oplus}$. Beforehand, we should show that additive closures actually have preadditive structures (Construction 2.4.7) and direct sums (Construction 2.4.9).

The proof for the preadditive structures uses ideas which are similar to the ones we have seen in Section 7.2. It can be found in [Zic24d]. The proof showing the existence of direct sums uses the same techniques but much more complicated rules and can be found in [Zic24c].

Finally, we can prove the correctness of the construction of homomorphism structures of additive closures in Construction 2.5.8. The proof again uses complex rules and can be found in [Zic24f].

Summing up, we have used `CompilerForCAP` to prove the correctness of all constructions of the categorical tower used for endowing categories of matrices with a homomorphism structure in Construction 4.1.3.

Finally, to also cover opposite categories of categories of matrices, we use `CompilerForCAP` for proving the correctness of Construction 2.5.9. The proof simply reduces all statements in the opposite category to the axioms in the original category and can be found in [Zic24g].

Example 7.4.1 (Implementation flaws discovered by formal verification)**.** Using `CompilerForCAP` for proving the correctness of Construction 2.5.8 actually turned up some flaws in the implementation of `AdditiveClosure` in the package `FreydCategoriesForCAP`. These flaws are typical for the kind of issues we expect to find by formally verifying code, so we give some details here:

In Construction 2.5.8, the functor H of the homomorphism structure for additive closures was defined as follows:

$$\begin{aligned} H : (\mathbf{C}^{\oplus})^{\mathrm{op}} \times \mathbf{C}^{\oplus} &\to \mathbf{D} \\ (\boxed{A}, \boxed{B}) &\mapsto \bigoplus_{i=1}^{a} \left(\bigoplus_{j=1}^{b} H(A_i, B_j) \right) \\ (\boxed{\alpha}, \boxed{\beta}) &\mapsto \left\langle \left(\left\langle \big(H(\alpha_{ij}, \beta_{st}) \big)_t^{\oplus} \right\rangle_s^{\oplus} \right)_i^{\oplus} \right\rangle_j^{\oplus} \end{aligned}$$

In particular, two objects $\boxed{A}$ and $\boxed{B}$ in $\mathbf{C}^{\oplus}$ are mapped to a direct sum of a direct sums of b objects each:

$$\big(H(A_1, B_1) \oplus \cdots \oplus H(A_1, B_b) \big) \oplus \cdots \oplus \big(H(A_a, B_1) \oplus \cdots \oplus H(A_a, B_b) \big). \tag{7.1}$$

Previously however, the implementation omitted the parentheses and simply computed a single direct sum of $a \cdot b$ objects:

$$H(A_1, B_1) \oplus \cdots \oplus H(A_1, B_b) \oplus \cdots \oplus H(A_a, B_1) \oplus \cdots \oplus H(A_a, B_b). \tag{7.2}$$

With this wrong implementation, H is not a functor, which can be seen as follows: Let $\boxed{\alpha} : \boxed{C} \to \boxed{A}$ and $\boxed{\beta} : \boxed{B} \to \boxed{D}$ be two morphisms in $\mathbf{C}^{\oplus}$. Then for H to be a functor, $H(\boxed{\alpha}, \boxed{\beta})$ has to define a morphism from $H(\boxed{A}, \boxed{B})$ to $H(\boxed{C}, \boxed{D})$. By construction, we have

$$\left\langle \left(\left\langle \big(H(\alpha_{ij}, \beta_{st}) \big)_t^{\oplus} \right\rangle_s^{\oplus} \right)_i^{\oplus} \right\rangle_j^{\oplus} : \bigoplus_{i=1}^{a} \left(\bigoplus_{j=1}^{b} H(A_i, B_j) \right) \to \bigoplus_{i=1}^{c} \left(\bigoplus_{j=1}^{d} H(C_i, D_j) \right).$$

In particular, source and target of $H(\boxed{\alpha}, \boxed{\beta})$ are of the form (7.1). Hence, with the wrong implementation of H on objects, `CompilerForCAP` would require us to show that the object in (7.1) is equal to the object in (7.2) with regard to the meta-theoretical equality. However, in general, direct sums are only associative up to a natural isomorphism, which means in general, the objects in (7.1) and (7.2) are different with regard to the meta-theoretical equality.

Spotting such an issue is difficult because we are used to identifying objects on paper even if computations require to insert a natural isomorphism. Moreover, this issue was not discovered by software tests because in all current applications of this implementation, direct sums are actually strictly associative. This shows that a form of formal verification is required for systematically catching such issues.

The changes to the code required to fix these inconsistencies can be found in [Zic23a] and [Zic23b].

Chapter 8

Conclusion and outlook

In this thesis, we have seen the advantages of modeling categories as **categorical towers**, both on paper and on a computer: We can **reuse** concepts, have a **separation of concerns** between the different layers of the tower, can break down the **verification** of the tower into smaller pieces, and have a natural **emergence** of structures of the tower from the single layers.

However, we have seen that a performance overhead appears naturally in computations in categorical towers and that due to this performance overhead, formerly in many cases large computations in categorical towers were not feasible on a computer. Sources of such overhead include:

- superfluous boxing and unboxing,
- suboptimal data structures of the categorical towers, and
- structural barriers, like the separation of loop headers from their bodies.

To rewrite the data structures of a categorical tower to desired and more efficient data structures, we have introduced the concept of **reinterpretations** of categorical towers along isomorphisms. With this, `CompilerForCAP` is able to avoid the overhead completely, which makes large computations in categorical towers finally feasible. As we have seen, `CompilerForCAP` can make the difference between "finishes in seconds" and "will never finish". Moreover, `CompilerForCAP` can also be used as a **proof assistant** for the verification of categorical algorithms and implementations.

In summary, `CompilerForCAP` can generate efficient and verified implementations, allowing us to make full use of the advantages of building categorical towers on a computer.

A future direction of the development of `CompilerForCAP` could be to implement `CompilerForCAP` (or parts of it) as a rewriting system modeled by a categorical tower itself. Computer programs can be represented as **syntax trees** [ALSU06], that is, as graphs. Moreover, categories of cospans of decorated quivers, which we have seen in Chapter 5, are closely related to **double pushout graph rewriting (DPO)** [BHK23, Cic18, EEGT06]. This shows a possible path how (parts of) `CompilerForCAP` could be modeled as a categorical tower. With this, `CompilerForCAP` could optimize and verify itself. Creating such a self-compiling compiler is a technique known as **bootstrapping**. With this, `CompilerForCAP` could profit from the advantages of categorical towers itself, completely closing the loop.

Bibliography

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*, Addison-Wesley, 2006. 121, 136, 137, 140, 141, 169

[Bar09] Mohamed Barakat, *The homomorphism theorem and effective computations*, Habilitation thesis, Department of Mathematics, RWTH-Aachen University, April 2009. 71

[BEKS17] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah, *Julia: A fresh approach to numerical computing*, SIAM Review **59** (2017), no. 1, 65–98. 17, 191

[BGK$^+$24] Mohamed Barakat, Sebastian Gutsche, Tom Kuhmichel, Sebastian Posur, and Fabian Zickgraf, *`MonoidalCategories` – Monoidal and monoidal (co)closed categories*, 2018–2024, `https://homalg-project.github.io/pkg/MonoidalCategories` (Retrieved: January 11, 2024), part of doi: 10.5281/zenodo.10927504. 17

[BHK23] Nicolas Behr, Russ Harmer, and Jean Krivine, *Fundamentals of compositional rewriting theory*, Journal of Logical and Algebraic Methods in Programming **135** (2023), 100893. 169

[BLH11] Mohamed Barakat and Markus Lange-Hegermann, *An axiomatic setup for algorithmic homological algebra and an alternative approach to localization*, Journal of Algebra and Its Applications **10** (2011), no. 2, 269–293. 83, 87

[BLH14] Mohamed Barakat and Markus Lange-Hegermann, *Gabriel morphisms and the computability of Serre quotients with applications to coherent sheaves*, 2014, `arXiv:1409.2028`. 71

[BMZ24] Mohamed Barakat, Julia Mickisch, and Fabian Zickgraf, *`FinSetsForCAP` – The elementary topos of (skeletal) finite sets*, 2017–2024, `https://homalg-project.github.io/pkg/FinSetsForCAP` (Retrieved: January 11, 2024), doi: 10.5281/zenodo.10927604. 17, 191

[BPZ24] Martin Bies, Sebastian Posur, and Fabian Zickgraf, *`FreydCategoriesForCAP` – Formal (co)kernels for additive* *categories*, 2018–2024, https://homalg-project.github.io/pkg/FreydCategoriesForCAP (Retrieved: January 21, 2024), part of doi: 10.5281/zenodo.10927504. 74, 82, 135, 160

[Bre78] J. Brewer, *Kronecker products and matrix calculus in system theory*, IEEE Transactions on Circuits and Systems **25** (1978), no. 9, 772–781. 83

[BS11] J. Baez and M. Stay, *Physics, Topology, Logic and Computation: A Rosetta Stone*, New Structures for Physics (Bob Coecke, ed.), Lecture Notes in Physics, vol. 813, Springer Berlin Heidelberg, 2011, pp. 95–172. 116

[BS24] Mohamed Barakat and Kamal Saleh, *FunctorCategories – Categories of functors*, 2017–2024, https://homalg-project.github.io/pkg/FunctorCategories (Retrieved: January 21, 2024), part of doi: 10.5281/zenodo.10927625. 96

[CAP24] CAP project authors, *The* CAP *project – Categories, Algorithms, and Programming*, https://homalg-project.github.io/prj/CAP_project (Retrieved: January 23, 2024), doi: 10.5281/zenodo.10927504, 2015–2024. 191

[Cat24] CategoricalTowers authors, *CategoricalTowers – Towers of category constructors*, https://homalg-project.github.io/prj/CategoricalTowers (Retrieved: January 23, 2024), doi: 10.5281/zenodo.10927625, 2023–2024. 191

[Chu32] Alonzo Church, *A Set of Postulates for the Foundation of Logic*, Annals of Mathematics **33** (1932), no. 2, 346–366. 64

[Cic18] Daniel Cicala, *Categorifying the ZX-calculus*, Electronic Proceedings in Theoretical Computer Science **266** (2018), 294–314. 1, 35, 61, 71, 99, 106, 109, 110, 111, 169

[DGPS23] Wolfram Decker, Gert-Martin Greuel, Gerhard Pfister, and Hans Schönemann, Singular *4-3-2 — A computer algebra system for polynomial computations*, http://www.singular.uni-kl.de, 2023. 128, 191

[DKPvdW20] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering, *Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus*, Quantum **4** (2020), 279. 101, 109, 189

[EEGT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Golas, and Gabriele Taentzer, *Fundamentals of Algebraic Graph Transformation*, Springer Berlin Heidelberg, 2006. 169

[Fre66] Peter Freyd, *Representations in Abelian Categories*, Proceedings of the Conference on Categorical Algebra (Berlin, Heidelberg) (S. Eilenberg, D. K. Harrison, S. MacLane, and H. Röhrl, eds.), Springer Berlin Heidelberg, 1966, pp. 95–120. 48

[FS19] Brendan Fong and David I. Spivak, *An Invitation to Applied Category Theory: Seven Sketches in Compositionality*, Cambridge University Press, 2019. 35, 71

[GAP22] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.12.2*, 2022, `https://www.gap-system.org` (Retrieved: January 11, 2024). 10, 17, 191

[GPSZ24] Sebastian Gutsche, Sebastian Posur, Øystein Skartsæterhagen, and Fabian Zickgraf, *`CAP` – Categories, Algorithms, Programming*, 2013–2024, `https://homalg-project.github.io/pkg/CAP` (Retrieved: January 11, 2024), part of doi: 10.5281/zenodo.10927504. 1, 5, 16

[GPZ24a] Sebastian Gutsche, Sebastian Posur, and Fabian Zickgraf, *`LinearAlgebraForCAP` – Category of Matrices over a Field for CAP*, 2015–2024, `https://homalg-project.github.io/pkg/LinearAlgebraForCAP` (Retrieved: January 25, 2024), part of doi: 10.5281/zenodo.10927504. 144

[GPZ24b] Sebastian Gutsche, Sebastian Posur, and Fabian Zickgraf, *`ModulePresentationsForCAP` – Category R-pres for CAP*, 2015–2024, `https://homalg-project.github.io/pkg/ModulePresentationsForCAP` (Retrieved: January 21, 2024), part of doi: 10.5281/zenodo.10927504. 89, 127

[Gut17] Sebastian Gutsche, *Constructive category theory and applications to algebraic geometry*, Ph.D. thesis, Universität Siegen, 2017. 23

[Hag87] Tatsuya Hagino, *A Categorical Programming Language*, Ph.D. thesis, University of Edinburgh, 1987. 15

[JAA$^+$22] Abhijith J., Adetokunbo Adedoyin, John Ambrosiano, et al., *Quantum Algorithm Implementations for Beginners*, ACM Transactions on Quantum Computing **3** (2022), no. 4, 18. 99, 183

[KL80] G. M. Kelly and M. L. Laplaza, *Coherence for compact closed categories*, Journal of Pure and Applied Algebra **19** (1980), 193–213. 57

[KvR21] Nicolai Kraus and Jakob von Raumer, *Path Spaces of Higher Inductive Types in Homotopy Type Theory*, Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '19, IEEE Press, 2021. 8

[Lon15] Brian Lonsdorf, *Professor Frisby's Mostly Adequate Guide to Functional Programming*, `https://mostly-adequate.gitbook.io/mostly-adequate-guide/` (Retrieved: January 14, 2024), 2015. 23

[LS88] J. Lambek and P. J. Scott, *Introduction to Higher-Order Categorical Logic*, Cambridge Studies in Advanced Mathematics, Cambridge University Press, 1988. 61, 67, 68, 70

[NC10] Michael A. Nielsen and Isaac L. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*, Cambridge University Press, 2010. 99, 183, 184, 185, 187, 189, 190

[Neu69] H. Neudecker, *Some Theorems on Matrix Differentiation with Special Reference to Kronecker Matrix Products*, Journal of the American Statistical Association **64** (1969), no. 327, 953–963. 83

[Pac24] Package authors of the `CAP` project, *List of `CAP`-based packages*, 2013–2024, `https://homalg-project.github.io/docs/CAP_project-based` (Retrieved: January 11, 2024). 17

[Pal19] Erik Palmgren, *On Equality of Objects in Categories in Constructive Type Theory*, 23rd International Conference on Types for Proofs and Programs (TYPES 2017) (Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, eds.), Leibniz International Proceedings in Informatics (LIPIcs), vol. 104, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019, pp. 7:1–7:7. 9

[Pos21a] Sebastian Posur, *A Constructive Approach to Freyd Categories*, Applied Categorical Structures **29** (2021), no. 1, 171–211. 1, 35, 45, 49, 71, 83, 89, 90

[Pos21b] Sebastian Posur, *Methods of constructive category theory*, Representations of Algebras, Geometry and Physics, Contemporary Mathematics, vol. 769, American Mathematical Society, 2021, pp. 157–208. 47, 74, 82

[Pos22] Sebastian Posur, *On free abelian categories for theorem proving*, Journal of Pure and Applied Algebra **226** (2022), no. 7, 106994. 75, 90

[Pra92] V. Pratt, *Linear Logic For Generalized Quantum Mechanics*, Workshop on Physics and Computation (Dallas, TX, USA), IEEE, 1992, pp. 166–180. 61, 100

[Pł22] Michał Płachta, *Grokking Functional Programming*, Manning Publications Co., Shelter Island, New York, 2022. 23

[Sal22] Kamal Saleh, *Constructive category theory and tilting equivalences via strong exceptional sequences*, Ph.D. thesis, Universität Siegen, 2022. 46, 47

[See89] R. A. G. Seely, *Linear logic, $*$-autonomous categories and cofree coalgebras*, Categories in Computer Science and Logic, Contemporary Mathematics, vol. 92, American Mathematical Society, 1989, pp. 371–382. 61, 100

[Tur37] A. M. Turing, *Computability and λ-definability*, Journal of Symbolic Logic **2** (1937), no. 4, 153–163. 61, 63

[Uni13] The Univalent Foundations Program, *Homotopy Type Theory: Univalent Foundations of Mathematics*, `https://homotopytypetheory.org/book` (Retrieved: January 11, 2024), Institute for Advanced Study, 2013. 9, 28, 180

[Zic23a] Fabian Zickgraf, *Fix of a flaw in the implementation of* *`HomomorphismStructureOnObjects`* *in* `AdditiveClosure`, `https://github.com/homalg-project/CAP_project/commit/61ffc050b9bf3d2f8fb979f60be2a0e0a2ada8fa` (Retrieved: January 04, 2024), 2023. 168

[Zic23b] Fabian Zickgraf, *Fix of a flaw in the implementation of* *`InterpretMorphismFromDistinguishedObjectToHomomorphismStructureAsMorphism`* *in* `AdditiveClosure`, `https://github.com/homalg-project/CAP_project/commit/a9e9425931e137a923a9121ad24210f7064a6495` (Retrieved: January 04, 2024), 2023. 168

[Zic24a] Fabian Zickgraf, *`CompilerForCAP` – Speed up and verify categorical algorithms*, 2020–2024, `https://homalg-project.github.io/pkg/CompilerForCAP` (Retrieved: January 11, 2024), part of doi: 10.5281/zenodo.10927504. 2, 35, 71, 78, 81, 112, 121, 147

[Zic24b] Fabian Zickgraf, *Collection of benchmarks used in the thesis*, `https://github.com/homalg-project/CAP_project/tree/93b73c93ecea4157ceeb5b20342171c940b0a363/CompilerForCAP/benchmarks` (Retrieved: April 05, 2024), 2023–2024. 126, 127, 131, 133, 138

[Zic24c] Fabian Zickgraf, *Formalized proof: Additive closures have direct sums*, `https://github.com/homalg-project/CAP_project/blob/4b901d7729c7a03df3e13759a5e7f127b29bc01d/CompilerForCAP/tst/300_proof_AdditiveClosure_has_direct_sums.tst` (Retrieved: January 04, 2024), 2023–2024. 167

[Zic24d] Fabian Zickgraf, *Formalized proof: Additive closures have preadditive structures*, `https://github.com/homalg-project/CAP_project/blob/4b901d7729c7a03df3e13759a5e7f127b29bc01d/CompilerForCAP/tst/300_proof_AdditiveClosure_has_preadditive_structure.tst` (Retrieved: January 04, 2024), 2023–2024. 167

[Zic24e] Fabian Zickgraf, *Formalized proof: Homomorphism structures for linear categories*, `https://github.com/homalg-project/CAP_project/blob/4b901d7729c7a03df3e13759a5e7f127b29bc01d/CompilerForCAP/tst/300_proof_hom_structure_for_linear_categories.tst` (Retrieved: January 04, 2024), 2023–2024. 167

[Zic24f] Fabian Zickgraf, *Formalized proof: Homomorphism structures of additive closures*, `https://github.com/homalg-project/CAP_project/blob/4b901d7729c7a03df3e13759a5e7f127b29bc01d/CompilerForCAP/tst/300_proof_AdditiveClosure_hom_structure.tst` (Retrieved: January 04, 2024), 2023–2024. 167

[Zic24g] Fabian Zickgraf, *Formalized proof: Homomorphism structures of opposite categories*, `https://github.com/homalg-project/CAP_project/blob/af98d02ed2c3e3f16eb7bbd1df20c7de07a94ca3/CompilerForCAP/tst/300_proof_OppositeCategory_hom_structure.tst` (Retrieved: January 19, 2024), 2023–2024. 167

[Zic24h] Fabian Zickgraf, *Formalized proof: $\boldsymbol{\mathcal{C}}(R)$ has a preadditive structure*, `https://github.com/homalg-project/CAP_project/blob/4b901d7729c7a03df3e13759a5e7f127b29bc01d/CompilerForCAP/tst/300_proof_RingAsCategory_has_preadditive_structure.tst` (Retrieved: January 04, 2024), 2023–2024. 167

[Zic24i] Fabian Zickgraf, *Formalized proof: $\boldsymbol{\mathcal{C}}(R)$ of a k-algebra R has a k-linear structure*, `https://github.com/homalg-project/CAP_project/blob/4b901d7729c7a03df3e13759a5e7f127b29bc01d/CompilerForCAP/tst/300_proof_RingAsCategory_of_k_algebra_has_linear_structure.tst` (Retrieved: January 04, 2024), 2023–2024. 167

[Zic24j] Fabian Zickgraf, *`ZXCalculusForCAP` – The category of ZX-diagrams*, 2023–2024, `https://homalg-project.github.io/pkg/ZXCalculusForCAP` (Retrieved: January 21, 2024), doi: 10.5281/zenodo.10927642. 106, 126, 191

[ZL02] E. Zerz and V. Lomadze, *A Constructive Solution to Interconnection and Decomposition Problems with Multidimensional Behaviors*, SIAM Journal on Control and Optimization **40** (2002), no. 4, 1072–1086. 83

Appendix A

Additional background for mathematical foundations

A.1 Categories with a single set of morphisms

One can also define the notion **category** in a different way than in Definition 1.1.2 in the special case of a **set** of objects. We state this definition and discuss its advantages and disadvantages because this opens up an additional perspective on Notation 1.1.16.

Definition A.1.1 (Categories with a single set of morphisms)**.** A **category with a single set of morphisms $\mathbf{C}$** consists of the following data:

- a set $\mathrm{Obj}_{\mathbf{C}}$ of **objects**,
- a (single) set $\mathrm{Mor}_{\mathbf{C}}$ of **morphisms**,
- for each morphism f two objects $s(f)$ and $t(f)$ called **source** and **target**, respectively,
- for each two morphisms f and g with $t(f) = s(g)$ a morphism $f \cdot g$ (called the **composite of** f **and** g) with $s(f \cdot g) = s(f)$ and $t(f \cdot g) = t(g)$,
- for each object A a morphism id_A (called **identity morphism of** A) with $s(\mathrm{id}_A) = A$ and $t(\mathrm{id}_A) = A$

fulfilling the following conditions:

- taking source and target is compatible with the equalities on objects and morphisms, that is, for morphisms f and g such that $f = g$ we have $s(f) = s(g)$ and $t(f) = t(g)$,
- composition is compatible with the equality on the set of morphisms, that is, for morphisms f_1, f_2, g_1 and g_2 with $f_1 = f_2$, $g_1 = g_2$ and $t(f_1) = t(f_2) = s(g_1) = s(g_2)$ we have $f_1 \cdot g_1 = f_2 \cdot g_2$,
- composition is associative, that is, for morphisms f, g, and h with $t(f) = s(g)$ and $t(g) = s(h)$ we have $(f \cdot g) \cdot h = f \cdot (g \cdot h)$,
- identity morphisms are neutral elements of the composition, that is, for each object A, each morphism f with $s(f) = A$ and each morphism g with $t(g) = A$ we have $\mathrm{id}_A \cdot f = f$ and $g \cdot \mathrm{id}_A = g$.

If we want to explicitly distinguish between this definition and the definition of **category** in Definition 1.1.2, we call the latter a **category with a family of sets of morphisms**.

Example A.1.2 (The category of matrices over $\mathbb{Z}$ with a single set of morphisms)**.** We define the category of matrices over $\mathbb{Z}$ **with a single set of morphisms** as follows:

- its objects are natural numbers with the usual equality,
- its morphisms are matrices over $\mathbb{Z}$ with the equality given by checking if the dimensions match and if they do, checking entrywise equality of integers,
- source and target of a matrix are given by its number of rows and columns, respectively,
- composition of two matrices is given by matrix multiplication,
- identity morphisms are given by identity matrices.

One can easily check that this indeed defines a category with a single set of morphisms.

Remark A.1.3. The major advantage of defining categories with single sets of morphisms is that one does not always have to explicitly specify source and target of morphisms in the context. This simplifies the notation on paper and the interface of an implementation on the computer. For example, we can write

$$\text{For two morphisms } f \text{ and } g \text{ with } t(f) = s(g)\text{, we consider } f \cdot g.$$

without giving explicit names to objects, or compose two composable morphisms `f` and `g` in `CAP` by

```
PreCompose( f, g )
```

without passing sources and targets explicitly. However, the translation between the two definitions, which we will explain in a moment, is a non-trivial process in some cases. Hence, we instead use Notation 1.1.16 which provides this convenience also for categories with families of sets of morphisms.

Let us now see how a category with a single set of morphisms can be translated into a category with a family of morphisms and vice versa:

- Let $\mathbf{C}$ be a category with a single set of morphisms. For each two objects A and B we can define

  $$\mathrm{Hom}_{\mathbf{C}}(A, B) \coloneqq \{f \in \mathrm{Mor}_{\mathbf{C}} \mid s(f) = A \text{ and } t(f) = B\}$$

  with the equality inherited from $\mathrm{Mor}_{\mathbf{C}}$. One can easily check that with this we obtain a category with a family of sets of morphisms.
- Now let $\mathbf{C}$ be a category with a family of sets of morphisms. To turn $\mathbf{C}$ into a category $\mathbf{D}$ with a single set of morphisms we have to proceed as follows: First, we have to construct a suitable notion of equality on objects, which beforehand was implicit in the segmentation of morphisms into a family of sets. Second, we have to construct the set $\mathrm{Mor}_{\mathbf{D}}$ from the sets $\mathrm{Hom}_{\mathbf{C}}(A, B)$. In general, the sets $\mathrm{Hom}_{\mathbf{C}}(A, B)$ are not disjoint, so taking

their union would not be sufficient. Instead, we have to construct $\mathrm{Mor}_{\mathbf{D}}$ as a coproduct, that is, a **disjoint** union, of the sets $\mathrm{Hom}_{\mathbf{C}}(A, B)$:

$$\mathrm{Mor}_{\mathbf{D}} \coloneqq \bigsqcup_{A,B \in \mathrm{Obj}_{\mathbf{C}}} \mathrm{Hom}_{\mathbf{C}}(A, B)$$

Then, we can simply define s and t on the components $\mathrm{Hom}_{\mathbf{C}}(A, B)$ in the obvious way. This means that in general a morphism of $\mathbf{C}$ is not literally a morphism of $\mathbf{D}$ anymore because we have to take the embedding into the disjoint union into account. See Example A.1.5 for such a situation.

We now construct an example where using categories with single sets of morphisms does **not** provide an advantage.

Example A.1.4 (Preorders as categories with a family of sets of morphisms)**.** Given a set P with a preorder $\leq$ and a singleton set S, we can define a category $\mathbf{C}$ with a family of sets of morphisms as follows:

- $\mathrm{Obj}_{\mathbf{C}} \coloneqq P$,
- $\mathrm{Hom}_{\mathbf{C}}(a, b) \coloneqq S$ if $a \leq b$, $\mathrm{Hom}_{\mathbf{C}}(a, b) \coloneqq \varnothing$ else, both with the trivial equalities,
- composition of two morphisms is given by the transitivity of $\leq$,
- identity morphisms are given by the reflexivity of $\leq$.

One can easily check that this indeed defines a category with a family of sets of morphisms.

Example A.1.5 (The preorder $(\mathbb{Z}, \leq)$ as a category with a single set of morphisms)**.** Consider the category $\mathbf{C}$ defined by the integers $\mathbb{Z}$ with the usual preorder $\leq$ and a singleton set $S = \{\star\}$ as in Example A.1.4. We want to turn this category into a category $\mathbf{D}$ with a single set of morphisms. Note that, for example, we have $\mathrm{Hom}_{\mathbf{C}}(1, 2) = \mathrm{Hom}_{\mathbf{C}}(2, 3)$. Hence, when constructing

$$\mathrm{Mor}_{\mathbf{D}} \coloneqq \bigsqcup_{a,b \in \mathbb{Z}} \mathrm{Hom}_{\mathbf{C}}(a, b),$$

we have to make the sets of morphisms disjoint by attaching source and target explicitly, for example by considering triples $(a, \star, b)$ instead of just $\star$ in $\mathrm{Hom}_{\mathbf{C}}(a, b)$. However, having to specify source and target of morphisms explicitly is what we wanted to avoid in the first place when modeling categories with single sets of morphisms.

This example shows that considering a category with a single sets of morphisms instead of a category with a family of sets of morphisms only gives an advantage if $f \in \mathrm{Hom}(A, B)$ already determines A and B uniquely and the **disjoint** union can therefore be modeled by a regular union.

A.2 The technical equality of functions in CAP

In Section 1.4.3, we have omitted discussing the technical equality of functions in CAP because this equality does not appear in the thesis. We now add this missing piece.

For functions we use **extensional** equality, that is, pointwise equality in the codomain. To see why we choose to do so, consider the following functions:

- the identity $\mathrm{id}_{\mathbb{Z}}$ on the integers,
- the function f which takes an integer n and simply returns n, and
- the function g which takes an integer m and simply returns m.

In general, we can ask: Should $\mathrm{id}_{\mathbb{Z}}$, f and g be pairwise equal functions? Moreover, consider the function h which takes an integer n and returns $n + 0$. Should f and h be equal?

All possible answers have different consequences. Viewing all of the above functions as equal would be a kind of **extensional** equality ([Uni13, Section 2.9]). This means we would only consider the external properties of a function, that is, the values at all points. Conversely, viewing two of the above functions as unequal would be a kind of **intensional** equality, that is, we would not only consider the external properties of the functions but also their internal definitions.

In the context of `CompilerForCAP`, the most natural choice is to use an extensional equality, that is, to view all of the above functions as equal. For example, rewriting $n + 0$ as in the definition of h to n as in the definition of f would be a typical simplification rule applied by `CompilerForCAP` for performance reasons. `CompilerForCAP` also renames local variables, that is, it could transform f to g. Finally, `CompilerForCAP` can sometimes replace functions by library functions. For example, f and g could be replaced by the library function `IdFunc`, which is implemented in the `GAP` kernel written in C and hence faster than calling f or g. If we want those simplifications to be compatible with our notion of equality, we have to view two functions as equal if and only if they are pointwise equal.

This of course requires that functions have a domain. `GAP` has no native mechanism for specifying domains of functions, so the domains have to be specified in the documentation instead. For example, the domain of an implementation of `PreCompose` in a category $\mathbf{C}$ is given by pairs of composable morphisms in $\mathbf{C}$, as specified in the documentation of `PreCompose`.

Moreover, we lose decidability of the equality for functions with infinite domains. However, if we can prove on paper that two functions are equal, we can at least use `CompilerForCAP` as a proof assistant (see Chapter 7) to verify this proof.

Appendix B

Additional background of applications

B.1 Exterior algebras

Definition B.1.1 (Exterior algebras)**.** Let K be a field and let V be a K-vectorspace with basis $e_1, \dots, e_n$. We define the **exterior algebra** $\bigwedge V$ as the vector space with formal basis

$$\{e_{i_1} \wedge \dots \wedge e_{i_a} \mid 1 \leq i_1 < i_2 < \dots < i_a \leq n\}$$

with a multiplication $\wedge$ defined on the basis as follows:

$$(e_{i_1} \wedge \dots \wedge e_{i_a}) \wedge (e_{j_1} \wedge \dots \wedge e_{j_b}) \coloneqq \begin{cases} 0, & \text{if } \{i_1, \dots, i_a\} \cap \{j_1, \dots, j_b\} \neq \varnothing \\ \pm e_{k_1} \wedge \dots \wedge e_{k_{a+b}}, & \text{else} \end{cases}$$

where the indices k_ℓ run through $i_1, \dots, i_a, j_1, \dots, j_b$ in increasing order and the sign is obtained by flipping the sign each time two adjacent elements of $i_1, \dots, i_a, j_1, \dots, j_b$ have to be swapped to obtain an increasing order. The multiplicative identity 1 is given by the empty product. The multiplication $\wedge$ is **anticommutative**, that is, for $v, w \in V$ we have

$$v \wedge w = -(w \wedge v),$$

as can be seen by computing

$$0 = (v + w) \wedge (v + w) = v \wedge v + v \wedge w + w \wedge v + w \wedge w = v \wedge w + w \wedge v.$$

Remark B.1.2 (The center of an exterior algebra)**.** Let K be a field and let V be a K-vectorspace with basis $e_1, \dots, e_n$. The center of $\bigwedge V$ is

$$C \coloneqq \langle e_{i_1} \wedge \dots \wedge e_{i_a} \mid 1 \leq i_1 < i_2 < \dots < i_a \leq n \textbf{ with } \boldsymbol{a} \textbf{ even} \rangle_K.$$

$\bigwedge V$ is a C-algebra which as a C-module is generated by

$$1, e_1, \dots, e_n.$$

However, in general $\bigwedge V$ is not a **free** C-module. For example, we have

$$(e_1 \wedge e_2) \cdot e_1 = (e_1 \wedge e_2) \wedge e_1 = 0$$

and

$$(e_1 \wedge e_2) \cdot e_3 = e_1 \wedge e_2 \wedge e_3 = (e_2 \wedge e_3) \cdot e_1.$$

Remark B.1.3. Let K be a field and let V be a K-vectorspace with basis $e_1, \dots, e_n$. We consider the exterior algebra $E \coloneqq \bigwedge V$. In Example 4.1.4, we have seen that $\mathbf{Mat}_E$ has a $\mathbf{Mat}_K$-homomorphism structure. We have also seen that the matrices quickly become large, especially if the dimension n of V grows, because E has exponential dimension 2^n. One possible solution for this problem is to view E as a C-algebra instead of a K-algebra, where C is the center of E as in Remark B.1.2. Since E is not a free C-algebra, we have to take the relations into account and consequently get a ${}_C\mathbf{FPres}$-homomorphism structure[1] for $\mathbf{Mat}_E$. With this, the matrices indeed get much smaller because the generating system of E over C only has size $n+1$ (instead of 2^n when working over K). However, this approach comes with some additional challenges when doing computations on a computer:

- If we simply model C as a K-algebra again, we have not improved the overall situation. Instead, we need a special implementation of the center C including a fast Gröbner basis algorithm.
- Every computation on morphisms in ${}_C\mathbf{FPres}$ coming from $\mathbf{Mat}_E$ will involve the relations of E over C both in the source and in the target. These relations have to be taken into account in the computation as early as possible. For example, if a computation in ${}_C\mathbf{FPres}$ is implemented by performing a computation in $\mathbf{Mat}_C$ and only reducing by the relations afterwards, the performance will suffer greatly. Specifically, if the relations are not known to the Gröbner basis algorithm, intermediate results get unnecessarily huge. This is particularly bad if the Gröbner basis implementation fully computes syzygies even in cases where they are not needed, for example in the context of computing lifts.[2]
- The performance of Gröbner basis algorithms highly depends on the chosen monomial order, especially if the matrices are not generic but come with some structure preferring one order over the other. An example for a situation involving highly structured matrices appear when using the homomorphism structure of $\mathbf{Mat}_E$ for computing lifts in ${}_E\mathbf{FPres}$. In this case, the matrices will have the structure seen at the end of the proof of Proposition 2.6.2, where H is of the form as in Construction 4.1.3. That is, the matrices will contain many zeros and many Kronecker products with identity matrices, which essentially introduce many copies of the same matrix. In this context, using a different monomial order (for example, a position-over-term order instead of a term-over-position order, or preferring the first over the last position, or the other way round) can make a huge difference in performance. However, choosing an order for this specific application is a very fragile optimization.

[1] See Definition 4.2.2 for the definition of ${}_C\mathbf{FPres}$.

[2] For an example, see `https://github.com/Singular/Singular/pull/1038` (retrieved: January 17, 2024).

This is why viewing E as a C-algebra instead of a K-algebra has not turned out to be useful.

B.2 Introduction to quantum computing

We give a short introduction to classical and quantum computations and circuits representing such computations. More details can, for example, be found in [JAA$^+$22] and [NC10].

B.2.1 Introduction to classical and quantum computations

We first introduce a possible model of classical computation. Its properties might seem trivial at first, but will become important when compared to the model of quantum computation below.

Definition B.2.1 (A model of classical computation)**.** A computation on a classical computer can be represented by operations on binary digits, or **bits** for short, with the following properties:

1. The state of a classical bit is an element of the set $\{0, 1\}$.
2. The state of a system of n classical bits is an element of $\{0, 1\}^n$, the n-fold product of $\{0, 1\}$, that is, every such state is of the form $(x_1, \ldots, x_n)$ with $x_i \in \{0, 1\}$.
3. For given n and m, every function $\{0, 1\}^n \to \{0, 1\}^m$ is realizable on a classical computer.
4. The result of a computation on a classical computer is a state $(x_1, \ldots, x_m)$ of m classical bits, which we can read out bit by bit.
5. A classical computer can repeat a computation conditionally based on the outcome of a previous computation.

For an example for the last point, consider a computation of the factorial of n: A classical computer can multiply n with the factorial of $n - 1$ and conditionally stop when it reaches the terminating condition $n = 1$.

We now describe a corresponding model of quantum computation, adapted from [NC10, Section 4.6]. This model is quite restrictive compared to the general theory of quantum physics. In Remark B.2.3 discuss some of these restrictions.

Definition B.2.2 (A model of quantum computation)**.** A computation on a quantum computer can be represented by operations on quantum binary digits, or **qubits** for short, with the following properties:

1. The state of a qubit is an element of $\mathbb{C}^2$ of the form
$$\alpha \cdot e_0 + \beta \cdot e_1 \quad \text{for} \quad \alpha, \beta \in \mathbb{C} \quad \text{with} \quad |\alpha|^2 + |\beta|^2 = 1,$$
where e_0 and e_1 are the standard basis vectors of $\mathbb{C}^2$. If both coefficients α and β are non-zero, we say that the qubit is in **superposition**. The standard basis vectors e_0 and e_1 are usually written $|0\rangle$ and $|1\rangle$. Warning: One must not confuse $|0\rangle$ with the zero vector $0_{\mathbb{C}^2}$, and in particular $0 \cdot |0\rangle$ is not equal to $|0\rangle$!

2. The state of a system of n qubits is a normalized element of $(\mathbb{C}^2)^{\otimes n}$, the n-fold tensor product of $\mathbb{C}^2$. A pure tensor[3] of standard basis vectors $|x_1\rangle \otimes \cdots \otimes |x_n\rangle$ with $x_i \in \{0,1\}$ is abbreviated $|x_1 \ldots x_n\rangle$. With this, a state can be written as

$$\sum_{(x_1,\ldots,x_n)\in\{0,1\}^n} \lambda_{(x_1,\ldots,x_n)} \cdot |x_1 \ldots x_n\rangle \quad \text{with} \quad \sum_{x\in\{0,1\}^n} |\lambda_x|^2 = 1.$$

   If two (or more) coefficients in this representation are non-zero, the state is said to be in **superposition**. If a state cannot be written as a pure tensor, it is called **entangled**. A typical example of an entangled state is the **Greenberger–Horne–Zeilinger (GHZ) state** $\frac{1}{\sqrt{2}}(|0\ldots0\rangle + |1\ldots1\rangle)$ on n qubits.

3. For given n, every **unitary** linear map $(\mathbb{C}^2)^{\otimes n} \to (\mathbb{C}^2)^{\otimes n}$ is realizable on a quantum computer. In particular, such maps are invertible.

4. Reading out the result of a computation on a quantum computer is called a **measurement**. Measurements output classical bits and are probabilistic: When measuring a single qubit in state $\alpha \cdot |0\rangle + \beta \cdot |1\rangle$, one obtains the classical outcome 0 with probability $|\alpha|^2$ and the classical outcome 1 with probability $|\beta|^2$. More generally, when measuring a system of n qubits in state

$$\sum_{(x_1,\ldots,x_n)\in\{0,1\}^n} \lambda_{(x_1,\ldots,x_n)} \cdot |x_1 \ldots x_n\rangle$$

   one obtains the classical outcome $(x_1, \ldots, x_n)$ with probability $|\lambda_{(x_1,\ldots,x_n)}|^2$. A measurement destroys the superposition and the entanglement, that is, after measuring $(x_1, \ldots, x_n)$ the system will always be in the state $|x_1 \ldots x_n\rangle$ and any repeated measurement will return $(x_1, \ldots, x_n)$ again.

5. Just like a classical computer, a quantum computer can repeat a computation conditionally based on the measurement of a previous computation. Since the measurement produces a classical result, repetitions are typically controlled by a classical control unit.

Remark B.2.3. Definition B.2.2 is quite restrictive compared to the general theory of quantum physics. For example, it restricts to the bases consisting of tensor products of $|0\rangle$ and $|1\rangle$, which are called **computational bases**. In general, one can also use other bases, for example the bases given by tensor products of $|+\rangle := |0\rangle + |1\rangle$ and $|-\rangle := |0\rangle - |1\rangle$. Moreover, one can in principle perform measurements with regard to any **orthonormal** basis.

Another restriction is that we only describe **pure states**, that is, states representable by a vector. More generally, quantum systems can be described by **mixed states** which are represented using **density matrices**, see [NC10, Section 2.4].

Despite these restrictions, the model of quantum computation introduced in Definition B.2.2 is equivalent to many other models of quantum computation. See [NC10, Section 4.6] for a more detailed discussion concerning the restrictions and possible extensions.

[3] Here, the term **pure tensor** is used as in algebra. In quantum physics, this is called a **separable state**. There is also the notion of a **pure state** in quantum physics, but this notion is not related to pure tensors and not relevant at this point.

Example B.2.4 (A simple quantum computation)**.** We give a simple toy example of a quantum computation on 2 qubits. The vectors $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$ form a basis of the state space $(\mathbb{C}^2)^{\otimes 2} \cong \mathbb{C}^4$. We can interpret those vectors as the binary representations of the numbers 0, 1, 2, and 3, just as we would with classical bit strings. Now, we consider the function s computing the successor in $\mathbb{Z}/4\mathbb{Z}$ on canonical representatives:

n	$s(n)$
0	1
1	2
2	3
3	0

This corresponds to a permutation of the basis vectors:

n	$s(n)$
$\lvert 00\rangle$	$\lvert 01\rangle$
$\lvert 01\rangle$	$\lvert 10\rangle$
$\lvert 10\rangle$	$\lvert 11\rangle$
$\lvert 11\rangle$	$\lvert 00\rangle$

This permutation extends to a linear map $S : (\mathbb{C}^2)^{\otimes 2} \to (\mathbb{C}^2)^{\otimes 2}$ given by a permutation matrix. Permutation matrices are unitary matrices, so S is a valid quantum function.

Now we can look at the state

$$\psi := \frac{1}{\sqrt{2}}\big(|00\rangle + |01\rangle\big),$$

which is in superposition. A possible interpretation of this state would be that the system is either 0 or 1, each with probability $\frac{1}{2}$. Now we can apply U and compute

$$U(\psi) = \frac{1}{\sqrt{2}}\big(U(|00\rangle) + U(|01\rangle)\big) = \frac{1}{\sqrt{2}}\big(|01\rangle + |10\rangle\big).$$

So the result is either 1 or 2, each with probability $\frac{1}{2}$. Hence, we have computed the successors of 0 and 1 “at the same time”.

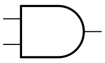
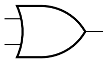
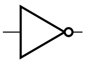
Remark B.2.5 (Computational power)**.** Classical computers and quantum computers have the same computational power, that is, one can simulate quantum computations on a classical computer and vice versa. However, quantum computers might be able to solve certain tasks much more efficiently than classical computers, for example using **Shor’s algorithm** for factoring integers. For a detailed discussion of this topic, see [NC10, Section 1.4.5].

B.2.2 Boolean circuits and quantum circuits

Finite computations on a (classical or quantum) computer can be visualized using **circuits**, that is, finite graphs whose vertices are labeled with functions applied during a computation.

Boolean circuits

Finite computations on classical bits can be described by **boolean** circuits having boolean functions like AND (“$\wedge$”), OR (“$\vee$”), and NOT (“$\neg$”) as vertices. These vertices or functions are called **gates** and are, for example, drawn as follows:

AND gate:  OR gate:  NOT gate: 

Here, the edges on the left represent the inputs and the edges on the right represent the outputs of the boolean functions.

Example B.2.6. As an example, we construct a boolean circuit computing the addition in $\mathbb{Z}/2\mathbb{Z}$ on canonical representatives:

$+$	0	1
0	0	1
1	1	0

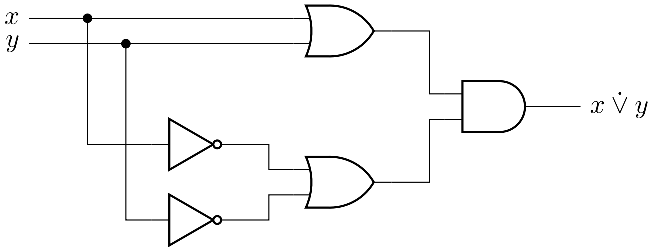
We usually identify 0 and 1 with the boolean values False and True, respectively. Hence, we can view $+$ on $\mathbb{Z}/2\mathbb{Z}$ as a boolean function:

x	y	$x+y$
False	False	False
False	True	True
True	False	True
True	True	False

This boolean function is called XOR (“$\dot\vee$”) and can be expressed using AND, OR, and NOT as follows:

$$\begin{aligned} x \mathbin{\dot\vee} y &= (x \vee y) \wedge (\neg(x \wedge y)) \\ &= (x \vee y) \wedge (\neg x \vee \neg y) \end{aligned}$$

where the second equality uses one of De Morgan's laws. We can represent this as a boolean circuit as follows:



The inputs on the top left are called x and y, corresponding to the variables in the formula $x \mathbin{\dot\vee} y$. Both inputs are copied at the junctions marked by the black dots to form inputs for both the upper OR gate and the two NOT gates. The results of the two NOT gates are combined using another OR gate. Finally, the outputs of the two OR gates are combined using an AND gate to form the output $x \mathbin{\dot\vee} y$.

Remark B.2.7 (Universal boolean gate sets)**.** We have introduced three gates above: the NOT gate, the AND gate, and the OR gate. One can show[4] that this set of gates is **universal**, that is, that any boolean function $\{0,1\}^n \to \{0,1\}^m$ for $n > 0$ can be expressed using only those gates. Here, we exclude the case $n = 0$ without inputs, because we cannot express maps $\{\} = \{0,1\}^0 \to \{0,1\}^m$ by gates which require inputs. This degenerate case could be included by adding a constant $0 : \{\} \to \{0,1\}$ to the gate set.

A question which naturally arises in this context is whether such a set is **minimal**. The set $\{\vee, \wedge, \neg\}$ is not minimal because we can use De Morgan's laws to represent AND using OR, or vice versa, together with NOT:

$$x \wedge y = \neg\big((\neg x) \vee (\neg y)\big) \qquad \text{and} \qquad x \vee y = \neg\big((\neg x) \wedge (\neg y)\big).$$

One can show that the sets $\{\neg, \wedge\}$ and $\{\neg, \vee\}$ are minimal with regard to inclusion. However, they are not minimal with regard to cardinality, as there exist universal sets with only a single element:

One can combine an AND gate followed by a NOT gate into a single gate called NAND. Given a NAND gate, we can recover NOT and AND as follows: A value can be negated by using it for both inputs of the NAND gate simultaneously. Then AND can be represented by NAND followed by negation. Hence, the NAND gate forms a single element universal set.

Quantum circuits

Computations on a quantum computer can be described by **quantum** circuits having **quantum gates** as vertices. Examples for quantum gates include:

- the **Hadamard gate** H representing the following map on a single qubit:

$$\begin{aligned} H : \mathbb{C}^2 &\to \mathbb{C}^2, \\ |0\rangle &\mapsto \frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big), \\ |1\rangle &\mapsto \frac{1}{\sqrt{2}}\big(|0\rangle - |1\rangle\big). \end{aligned}$$

- the **phase shift gates** Z_α with a **phase** $\alpha \in [0, 2\pi)$ representing the following map on a single qubit:

$$\begin{aligned} Z_\alpha : \mathbb{C}^2 &\to \mathbb{C}^2, \\ |0\rangle &\mapsto |0\rangle\,, \\ |1\rangle &\mapsto e^{i\alpha}\,|1\rangle\,. \end{aligned}$$

- the **NOT gate**, also known as an **X gate**, which flips a single qubit as follows:

$$\begin{aligned} X : \mathbb{C}^2 &\to \mathbb{C}^2, \\ |0\rangle &\mapsto |1\rangle\,, \\ |1\rangle &\mapsto |0\rangle\,. \end{aligned}$$

[4] See, for example, [NC10, Section 3.1.2].

- the **CNOT gate** (“controlled not”), also known as an **CX gate**, which flips a target qubit t if and only if a controlling qubit c is $|1\rangle$:

$\|ct\rangle$	$\|c't'\rangle := \text{CNOT}(\|ct\rangle)$
$\|00\rangle$	$\|00\rangle$
$\|01\rangle$	$\|01\rangle$
$\|10\rangle$	$\|11\rangle$
$\|11\rangle$	$\|10\rangle$

Here, the Hadamard gate and phase shift gates actually introduce quantum effects, while the NOT gate and the CNOT gate are generalizations of classical gates:

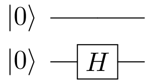
As in the classical case, we associate $|0\rangle$ and $|1\rangle$ with the boolean values False and True, respectively. Hence, a NOT gate maps True to False and vice versa as expected.

When considering the output of the CNOT gate, we see that the first qubit c' of the result is just given by c and the second qubit t' actually arises as $c \mathbin{\dot\vee} t$. Hence, CNOT can be interpreted as the quantum version of XOR, making XOR invertible by passing through one of the bits.

Example B.2.8. We can visualize the computation in Example B.2.4 as a quantum circuit on 2 qubits. We initialize the qubits with $|0\rangle$, that is, we consider the initial state $|00\rangle$. To prepare the state

$$\psi := \frac{1}{\sqrt{2}}\big(|00\rangle + |01\rangle\big),$$

we can apply the map $\text{id}_{\mathbb{C}^2} \otimes H$ to $|00\rangle$. We can visualize this as a quantum circuit as follows:

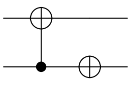


The upper horizontal line represents the first qubit, the lower horizontal line represents the second qubit. The reading direction is from left to right. Both qubits are initialized in the state $|0\rangle$, and we apply a Hadamard gate to the second qubit.

Furthermore, in Example B.2.4 we represented the successor function on $\mathbb{Z}/4\mathbb{Z}$ using the following permutation of basis vectors:

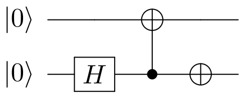
n	$s(n)$
$\|00\rangle$	$\|01\rangle$
$\|01\rangle$	$\|10\rangle$
$\|10\rangle$	$\|11\rangle$
$\|11\rangle$	$\|00\rangle$

Note that the first qubit of the result arises by flipping the first qubit of the input if and only if the second qubit is $|1\rangle$. Hence, we can compute the first qubit of the result by applying a CNOT gate to the input with the second qubit being the controlling qubit. The second qubit of the result arises by negation of the second qubit of the input. The corresponding quantum circuit looks as follows:



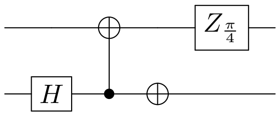
The symbol "$\oplus$" stands for NOT. If a NOT gate is controlled by another qubit, forming a CNOT gate, the controlling qubit is marked by a "$\bullet$" connected to the NOT gate. So this circuit exactly describes what we did above: Flip the first qubit depending on the second qubit, then flip the second qubit.

Applying the successor function to the state ψ defined above corresponds to concatenating the two quantum circuits. Hence, the computation in Example B.2.4 can in total be represented by the following quantum circuit:

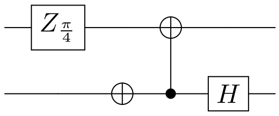


Remark B.2.9 (Universal quantum gate sets)**.** Just as in the classical case in Remark B.2.7, we can ask whether there exist **universal** sets of quantum gates. Indeed, one can show that the Hadamard gate, the phase shift gates, and the CNOT gate are sufficient for representing all unitary maps $(\mathbb{C}^2)^{\otimes n} \to (\mathbb{C}^2)^{\otimes n}$ for arbitrary n. However, recall that the phase shift gates are parametrized by a phase $\alpha \in [0, 2\pi)$, so this set is not finite. In fact, we cannot achieve true universality with a finite set of gates: The set of unitary maps $(\mathbb{C}^2)^{\otimes n} \to (\mathbb{C}^2)^{\otimes n}$ is uncountable, but starting with a finite set of gates we can only create a countable number of finite sequences. Nevertheless, one can show that the Hadamard gate, the phase shift gate $Z_{\frac{\pi}{4}}$, and the CNOT gate form an **approximatively universal set**, that is, all unitary maps $(\mathbb{C}^2)^{\otimes n} \to (\mathbb{C}^2)^{\otimes n}$ for arbitrary n can be **approximated to arbitrary accuracy** using gates from this set. For more details, see [NC10, Section 4.5] and [DKPvdW20, following Remark 2.5].

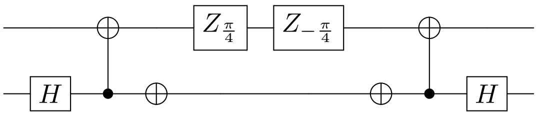
Remark B.2.10 (Inverting quantum circuits)**.** Every quantum function is invertible, so a natural question is: Given a quantum circuit corresponding to some quantum function, how can we construct a circuit corresponding to the inverse function? The answer is simple: To invert a quantum circuit built from Hadamard gates, phase shift gates, and CNOT gates, we simply reverse the circuit and invert all gates: The inverse of a phase shift gate Z_α is $Z_{-\alpha}$, and Hadamard gates and CNOT gates are self-inverse. For example, the inverse of the quantum circuit



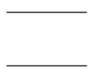
is given by



Indeed, if we compose the two circuits and form



the gates cancel successively until we obtain the empty circuit

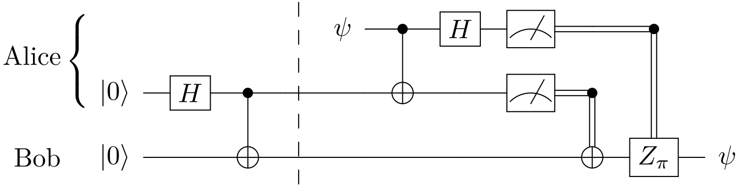


which represents the identity.

As a special case, note that for $\alpha = 0$ or $\alpha = \pi$, the phase shift gate Z_α is self-inverse. Hence, if only the phases $\alpha = 0$ and $\alpha = \pi$ appear in a quantum circuit, simply reversing the circuit already gives its inverse.

As a final, more intricate example of a quantum circuit, we consider the phenomenon of **quantum teleportation**.

Example B.2.11 (Quantum teleportation, [NC10, Section 1.3.7])**.** The phenomenon of **quantum teleportation** can be explained by the following quantum circuit:



This circuit introduces some new notation which we explain in context below.

Quantum teleportation teleports a quantum state ψ from one party, called Alice, to another party, called Bob. There are two phases, as indicated by the vertical dashed line.

First, Alice and Bob prepare an entangled state on two qubits as stated left of the vertical dashed line. The two qubits can then be physically separated over an arbitrary distance, one taken by Alice and one taken by Bob.

In the second phase, Alice wants to send Bob an unknown quantum state ψ without the two physically meeting. Alice cannot simply measure ψ to determine the state because she would only get a classical bit as an output and the state would be destroyed afterwards. Instead, she entangles the quantum state ψ with her part of the state shared in the first phase. She does this by applying a CNOT gate. Afterwards, she applies a Hadamard gate and measures both qubits, as indicated by the meter symbol. The results are classical bits, as indicated by the double-struck wires. She transmits the results to Bob via a classical communication channel. Depending on the results, Bob now applies a NOT gate and/or a Z_π gate to his part of the state shared in the first phase. Now, one can show that Bob's qubit is in state ψ. We will give a proof for this using **ZX-diagrams** in Example 5.1.5.

Summing up, thanks to the entangled quantum state shared in the first phase, Alice was able to teleport a quantum state to Bob without the two physically meeting.

Appendix C

Additional background for benchmarks

All benchmarks in this thesis were performed on a system with the following specifications:

System type	consumer notebook, on AC power
CPU	Intel(R) Core(TM) i5-7500T CPU 4 cores, 4 threads processor base frequency: 2.70GHz max turbo frequency: 3.30GHz locked to 2.70GHz
CPU scaling driver	`intel_pstate`
CPU scaling governor	performance
Memory	2 × 8GB DDR4-2400, dual channel mode
Operation system	Arch Linux (rolling release)
Linux kernel version	6.6.11
`GAP` [GAP22]	version 4.12.2
`Julia` [BEKS17]	version 1.10.0
`Singular` [DGPS23]	version 4.3.2.p10
`CAP_project` [CAP24]	git commit 7bb28f599094
`CategoricalTowers` [Cat24]	git commit e937a781a6ed
`FinSetsForCAP` [BMZ24]	git commit daeb79bcc982
`ZXCalculusForCAP` [Zic24j]	git commit 7909f045a812

Appendix D

Listings

D.1 Code used in Chapter 7

Listing D.1.1:

```
LoadPackage( "CAP", false );
monoid_as_category_ZZ := CreateCapCategoryWithDataTypes(
    "MonoidAsCategory( ZZ )", IsCapCategory,
    IsCapCategoryObject, IsCapCategoryMorphism, IsCapCategoryTwoCell,
    IsUnicodeCharacter, IsInt, fail
);
```

Back to the generated paragraph on page 148

Listing D.1.2:

```
star := UChar( 8902 ); # ’⋆’ has code point 8902 in decimal notation
unique_object := AsCapCategoryObject( monoid_as_category_ZZ, star );
```

Back to the generated paragraph on page 149

Listing D.1.3:

```
AddIsWellDefinedForMorphisms( monoid_as_category_ZZ,
  { cat, mor } -> AsInteger( mor ) in Integers
);
AddIsCongruentForMorphisms( monoid_as_category_ZZ,
  { cat, mor1, mor2 } -> AsInteger( mor1 ) = AsInteger( mor2 )
);
AddIsEqualForObjects( monoid_as_category_ZZ,
  { cat, obj1, obj2 } -> true
);
AddPreCompose( monoid_as_category_ZZ,
  { cat, mor1, mor2 } -> AsCapCategoryMorphism( cat,
    Source( mor1 ), AsInteger( mor1 ) * AsInteger( mor2 ), Target( mor2 )
  )
);
AddIdentityMorphism( monoid_as_category_ZZ,
  { cat, obj } -> AsCapCategoryMorphism( cat,
    obj, One( Integers ), obj
  )
);
```

Back to the generated paragraph on page 149

Listing D.1.4:

```
Finalize( monoid_as_category_ZZ );;
Integers!.LaTeXSymbol := "\\mathbb{Z}";;
monoid_as_category_ZZ!.LaTeXSymbol
                            := "\\boldsymbol{\\mathcal{C}}(\\mathbb{Z})";;
monoid_as_category_ZZ!.VariableLaTeXSymbols := rec(
    alpha := "m",
    beta := "n",
    gamma := "l"
);;
```

Back to the generated paragraph on page 149

Listing D.1.5:

```
LoadPackage( "CompilerForCAP", false );
CapJitEnableProofAssistantMode( );
```

Back to the generated paragraph on page 153

Listing D.1.6:

```
statement := function ( cat, A, B, C, D, m, n, l )
  local m_n, left, n_l, right;

    m_n := PreCompose( m, n );
    left := PreCompose( m_n, l );

    n_l := PreCompose( n, l );
    right := PreCompose( m, n_l );

    return IsCongruentForMorphisms( left, right );
end;;
```

Back to the generated paragraph on page 153

Listing D.1.7:

```
StateLemma(
    "composition is associative",
    statement,
    monoid_as_category_ZZ,
    [ "category", "object", "object", "object", "object",
      "morphism", "morphism", "morphism" ],
    [
        rec( src_template := "Source( m )", dst_template := "A" ),
        rec( src_template := "Target( m )", dst_template := "B" ),
        rec( src_template := "Source( n )", dst_template := "B" ),
        rec( src_template := "Target( n )", dst_template := "C" ),
        rec( src_template := "Source( l )", dst_template := "C" ),
        rec( src_template := "Target( l )", dst_template := "D" ),
    ]
);
```

Back to the generated paragraph on page 153

Listing D.1.8:

```
# see previous listing
```

Back to the generated paragraph on page 154

Listing D.1.9:

```
PrintLemma( );
```

Back to the generated paragraph on page 154

Listing D.1.10:

```
# see previous listing
```

Back to the generated paragraph on page 154

Listing D.1.11:

```
ApplyLogicTemplate( rec(
    variable_names := [ "a", "b", "c" ],
    variable_filters := [ IsInt, IsInt, IsInt ],
    src_template := "a * (b * c)",
    dst_template := "(a * b) * c",
) );
```

Back to the generated paragraph on page 155

Listing D.1.12:

```
# see previous listing
```

Back to the generated paragraph on page 155

Listing D.1.13:

```
PrintLemma( );
```

Back to the generated paragraph on page 155

Listing D.1.14:

```
# see previous listing
```

Back to the generated paragraph on page 155

Listing D.1.15:

```
AssertLemma( );
```

Back to the generated paragraph on page 155

Listing D.1.16:

```
# see previous listing
```

Back to the generated paragraph on page 155

Listing D.1.17:

```
StateProposition( monoid_as_category_ZZ, "is_category" );
```

Back to the generated paragraph on page 156

Listing D.1.18:

```
# see previous listing
```

Back to the generated paragraph on page 156

Listing D.1.19:

```
StateNextLemma( );
```

Back to the generated paragraph on page 156

Listing D.1.20:

```
# see previous listing
```

Back to the generated paragraph on page 156

Listing D.1.21:

```
PrintLemma( );
```

Back to the generated paragraph on page 156

Listing D.1.22:

```
AttestValidInputs( );
```

Back to the generated paragraph on page 156

Listing D.1.23:

```
# see previous listing
```

Back to the generated paragraph on page 156

Listing D.1.24:

```
AssertLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.25:

```
StateNextLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.26:

```
PrintLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.27:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "a", "b", "c" ],
        variable_filters := [ IsInt, IsInt, IsInt ],
        src_template := "a * (b * c)",
        dst_template := "(a * b) * c",
    )
);
```

Back to the generated paragraph on page 157

Listing D.1.28:

```
AssertLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.29:

```
StateNextLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.30:

```
PrintLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.31:

```
AttestValidInputs( );
```

Back to the generated paragraph on page 157

Listing D.1.32:

```
AssertLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.33:

```
StateNextLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.34:

```
PrintLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.35:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "a" ],
        variable_filters := [ IsInt ],
        src_template := "One( Integers ) * a",
        dst_template := "a",
    )
);
```

Back to the generated paragraph on page 157

Listing D.1.36:

```
AssertLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.37:

```
StateNextLemma( );
```

Back to the generated paragraph on page 157

Listing D.1.38:

```
PrintLemma( );
```

Back to the generated paragraph on page 158

Listing D.1.39:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "a" ],
        variable_filters := [ IsInt ],
        src_template := "a * One( Integers )",
        dst_template := "a",
    )
);
```

Back to the generated paragraph on page 158

Listing D.1.40:

```
AssertLemma( );
```

Back to the generated paragraph on page 158

Listing D.1.41:

```
AssertProposition( );
```

Back to the generated paragraph on page 158

Listing D.1.42:

```
LoadPackage( "FreydCategoriesForCAP", false );
dummy := DummyCategory( rec(
    name := "a preadditive category",
    list_of_operations_to_install := [
        "IsEqualForObjects",
        "IsWellDefinedForObjects",
        "IsWellDefinedForMorphismsWithGivenSourceAndRange",
        "IsCongruentForMorphisms",
        "PreCompose",
        "IdentityMorphism",
        "SumOfMorphisms",
        "ZeroMorphism",
        "AdditiveInverseForMorphisms",
    ],
    properties := [
        "IsAbCategory", # another name for a preadditive category
    ],
) );;
dummy!.LaTeXSymbol := "\\mathbf{C}";;
additive_closure := AdditiveClosure( dummy );
additive_closure!.LaTeXSymbol := "\\mathbf{C}^\\oplus";;
LoadPackage( "CompilerForCAP", false );
CapJitEnableProofAssistantMode( );
```

Back to the generated paragraph on page 160

Listing D.1.43:

```
StopCompilationAtPrimitivelyInstalledOperationsOfCategory( dummy );
```

Back to the generated paragraph on page 160

Listing D.1.44:

```
CapJitAddLogicTemplate( rec(
    variable_names := [ "alpha", "i", "j" ],
    variable_filters := [ IsAdditiveClosureMorphism, IsInt, IsInt ],
    src_template := "Source( MorphismMatrix( alpha )[i][j] )",
    dst_template := "ObjectList( Source( alpha ) )[i]",
) );
CapJitAddLogicTemplate( rec(
    variable_names := [ "alpha", "i", "j" ],
    variable_filters := [ IsAdditiveClosureMorphism, IsInt, IsInt ],
    src_template := "Target( MorphismMatrix( alpha )[i][j] )",
    dst_template := "ObjectList( Target( alpha ) )[j]",
) );
```

Back to the generated paragraph on page 160

Listing D.1.45:

```
StateProposition( additive_closure, "is_category" );
```

Back to the generated paragraph on page 161

Listing D.1.46:

```
StateNextLemma( );
```

Back to the generated paragraph on page 161

Listing D.1.47:

```
PrintLemma( );
```

Back to the generated paragraph on page 161

Listing D.1.48:

```
AttestValidInputs( );
```

Back to the generated paragraph on page 162

Listing D.1.49:

```
AssertLemma( );
```

Back to the generated paragraph on page 162

Listing D.1.50:

```
StateNextLemma( );
```

Back to the generated paragraph on page 162

Listing D.1.51:

```
PrintLemma( );
```

Back to the generated paragraph on page 162

Listing D.1.52:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "phi__X__Y", "W", "X", "l", "psi_k" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "
            category", "morphism", "object", "object", "integer", "morphism" ],
             dummy ),
        src_template := "PreCompose( cat, SumOfMorphisms( cat, W, List( [ 1 ..
            l ], k -> psi_k ), X ), phi__X__Y )",
        dst_template := "SumOfMorphisms( cat, W, List( [ 1 .. l ], k ->
            PreCompose( cat, psi_k, phi__X__Y ) ), Target( phi__X__Y ) )",
    )
);
```

Back to the generated paragraph on page 162

Listing D.1.53:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "phi__X__Y", "Y", "Z", "l", "chi_k" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "
            category", "morphism", "object", "object", "integer", "morphism" ],
             dummy ),
        src_template := "PreCompose( cat, phi__X__Y, SumOfMorphisms( cat, Y,
            List( [ 1 .. l ], k -> chi_k ), Z ) )",
        dst_template := "SumOfMorphisms( cat, Source( phi__X__Y ), List( [ 1 ..
             l ], k -> PreCompose( cat, phi__X__Y, chi_k ) ), Z )",
    )
);
```

Back to the generated paragraph on page 162

Listing D.1.54:

```
PrintLemma( );
```

Back to the generated paragraph on page 162

Listing D.1.55:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "phi", "psi", "chi" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "
            category", "morphism", "morphism", "morphism" ], dummy ),
        src_template := "PreCompose( cat, phi, PreCompose( cat, psi, chi ) )",
        dst_template := "PreCompose( cat, PreCompose( cat, phi, psi ), chi )",
    )
);
```

Back to the generated paragraph on page 162

Listing D.1.56:

```
PrintLemma( );
```

Back to the generated paragraph on page 163

Listing D.1.57:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "m", "n", "phi_ij", "X", "Y" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "
            category", "integer", "integer", "morphism", "object", "object" ],
            dummy ),
        src_template := "SumOfMorphisms( cat, X, List( [ 1 .. m ], i -> SumOfMorphisms( cat, X, List( [ 1 .. n ], j -> phi_ij ), Y ) ), Y )",
        dst_template := "SumOfMorphisms( cat, X, List( [ 1 .. n ], j -> SumOfMorphisms( cat, X, List( [ 1 .. m ], i -> phi_ij ), Y ) ), Y )",
    )
);
```

Back to the generated paragraph on page 163

Listing D.1.58:

```
AssertLemma( );
```

Back to the generated paragraph on page 163

Listing D.1.59:

```
StateNextLemma( );
```

Back to the generated paragraph on page 163

Listing D.1.60:

```
PrintLemma( );
```

Back to the generated paragraph on page 163

Listing D.1.61:

```
AttestValidInputs( );
```

Back to the generated paragraph on page 163

Listing D.1.62:

```
AssertLemma( );
```

Back to the generated paragraph on page 163

Listing D.1.63:

```
StateNextLemma( );
```

Back to the generated paragraph on page 163

Listing D.1.64:

```
PrintLemma( );
```

Back to the generated paragraph on page 163

Listing D.1.65:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "psi", "P", "phi_1", "phi_2" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "category", "morphism", "bool", "morphism", "morphism" ], dummy ),
        src_template := "PreCompose( cat, CAP_JIT_INTERNAL_EXPR_CASE( P, phi_1, true, phi_2 ), psi )",
        dst_template := "CAP_JIT_INTERNAL_EXPR_CASE( P, PreCompose( cat, phi_1, psi ), true, PreCompose( cat, phi_2, psi ) )",
    )
);
```

Back to the generated paragraph on page 164

Listing D.1.66:

```
PrintLemma( );
```

Back to the generated paragraph on page 164

Listing D.1.67:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "phi__X__Y", "X" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "category", "morphism", "object" ], dummy ),
        src_template := "PreCompose( cat, IdentityMorphism( cat, X ), phi__X__Y )",
        dst_template := "phi__X__Y",
    )
);
```

Back to the generated paragraph on page 164

Listing D.1.68:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "phi__X__Y", "W", "X" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "category", "morphism", "object", "object" ], dummy ),
        src_template := "PreCompose( cat, ZeroMorphism( cat, W, X ), phi__X__Y )",
        dst_template := "ZeroMorphism( cat, W, Target( phi__X__Y ) )",
    )
);
```

Back to the generated paragraph on page 164

Listing D.1.69:

```
PrintLemma( );
```

Back to the generated paragraph on page 164

Listing D.1.70:

```
# CONDITION: the case ‘i = k‘ actually occurs
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "l", "i", "phi_k", "X", "Y" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "category", "integer", "integer", "morphism", "object", "object" ], dummy ),
        src_template := "SumOfMorphisms( cat, X, List( [ 1 .. l ], k -> CAP_JIT_INTERNAL_EXPR_CASE( i = k, phi_k, true, ZeroMorphism( cat, X, Y ) ) ), Y )",
        dst_template := "(k -> phi_k)(i)",
    )
);
```

Back to the generated paragraph on page 164

Listing D.1.71:

```
AssertLemma( );
```

Back to the generated paragraph on page 164

Listing D.1.72:

```
StateNextLemma( );
```

Back to the generated paragraph on page 164

Listing D.1.73:

```
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "psi", "P", "phi_1", "phi_2" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "category", "morphism", "bool", "morphism", "morphism" ], dummy ),
        src_template := "PreCompose( cat, psi, CAP_JIT_INTERNAL_EXPR_CASE( P, phi_1, true, phi_2 ) )",
        dst_template := "CAP_JIT_INTERNAL_EXPR_CASE( P, PreCompose( cat, psi, phi_1 ), true, PreCompose( cat, psi, phi_2 ) )",
    )
);

ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "phi__X__Y", "Y" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "category", "morphism", "object" ], dummy ),
        src_template := "PreCompose( cat, phi__X__Y, IdentityMorphism( cat, Y ) )",
        dst_template := "phi__X__Y",
    )
);

ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "phi__X__Y", "Y", "Z" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "category", "morphism", "object", "object" ], dummy ),
        src_template := "PreCompose( cat, phi__X__Y, ZeroMorphism( cat, Y, Z ) )",
        dst_template := "ZeroMorphism( cat, Source( phi__X__Y ), Z )",
    )
);

# CONDITION: the case ‘k = i‘ actually occurs
ApplyLogicTemplate(
    rec(
        variable_names := [ "cat", "l", "i", "phi_k", "X", "Y" ],
        variable_filters := CAP_INTERNAL_GET_DATA_TYPES_FROM_STRINGS( [ "category", "integer", "integer", "morphism", "object", "object" ], dummy ),
        src_template := "SumOfMorphisms( cat, X, List( [ 1 .. l ], k -> CAP_JIT_INTERNAL_EXPR_CASE( k = i, phi_k, true, ZeroMorphism( cat, X, Y ) ) ) ), Y )",
        dst_template := "(k -> phi_k)(i)",
    )
);

AssertLemma( );
```

Back to the generated paragraph on page 164

Listing D.1.74:

```
AssertProposition( );
```

Back to the generated paragraph on page 164

Listing D.1.75:

```
StateProposition( additive_closure, "has_zero_object" );
```

Back to the generated paragraph on page 165

Listing D.1.76:

```
StateNextLemma( );
```

Back to the generated paragraph on page 165

Listing D.1.77:

```
PrintLemma( );
```

Back to the generated paragraph on page 165

Listing D.1.78:

```
AttestValidInputs( );
```

Back to the generated paragraph on page 165

Listing D.1.79:

```
AssertLemma( );
```

Back to the generated paragraph on page 165

Listing D.1.80:

```
StateNextLemma( );
```

Back to the generated paragraph on page 165

Listing D.1.81:

```
PrintLemma( );
```

Back to the generated paragraph on page 165

Listing D.1.82:

```
AttestValidInputs( );
```

Back to the generated paragraph on page 165

Listing D.1.83:

```
AssertLemma( );
```

Back to the generated paragraph on page 165

Listing D.1.84:

```
StateNextLemma( );
```

Back to the generated paragraph on page 165

Listing D.1.85:

```
StateNextLemma( );
```

Back to the generated paragraph on page 166

Listing D.1.86:

```
PrintLemma( );
```

Back to the generated paragraph on page 166

Listing D.1.87:

```
AttestValidInputs( );
```

Back to the generated paragraph on page 166

Listing D.1.88:

```
AssertLemma( );
```

Back to the generated paragraph on page 166

Listing D.1.89:

```
StateNextLemma( );
```

Back to the generated paragraph on page 166

Listing D.1.90:

```
AssertProposition( );
```

Back to the generated paragraph on page 166

Index