Compiling quantamorphisms
for the IBM Q Experience

Ana Neri, Rui Soares Barbosa, and José N. Oliveira

Abstract—Based on the connection between the categorical derivation of classical programs from specifications and the category-theoretic approach to quantum physics, this paper contributes to extending the laws of classical program algebra to quantum programming. This aims at building *correct-by-construction* quantum circuits to be deployed on quantum devices such as those available at the IBM Q Experience.

Quantum circuit reversibility is ensured by minimal complements, extended recursively. Measurements are postponed to the end of such recursive computations, termed *"quantamorphisms"*, thus maximising the quantum effect.

Quantamorphisms are classical catamorphisms which, extended to ensure quantum reversibility, implement quantum cycles (vulg. for-loops) and quantum folds on lists. By Kleisli correspondence, quantamorphisms can be written as monadic functional programs with quantum parameters. This enables the use of Haskell, a monadic functional programming language, to perform the experimental work. Such calculated quantum programs prepared in Haskell are pushed through Quipper to the Qiskit interface to IBM Q quantum devices. The generated quantum circuits – often quite large – exhibit the predicted behaviour. However, running them on real quantum devices incurs into a significant amount of errors. As quantum devices are constantly evolving, an increase in reliability is likely in the near future, allowing for our programs to run more accurately.

Index Terms—Quantum computing, algebra of programming, reversibility, IBM Q experience

◆

1 INTRODUCTION

As is well known, there are highly complex problems that cannot be efficiently solved by classical computers. On the other hand, classical system design is under pressure to decrease the size of circuits as much as possible. In this context, *quantum technologies* appear as prime candidates to support a new computing era — the *quantum computing* age [1], [2].

This prospect is attracting both industry and academia, the former primarily interested in understanding the potential advantages of quantum computing for their business and the latter interested in pushing quantum science and technology even further. Companies at the vanguard of quantum technology (namely IBM, Google, and Microsoft) are already exploring it, leading to new consortia between industry and academia. An example of this is the IBM Quantum Network (IBM Q for short), which involves a number of companies (e.g. Mitsubishi Chemical) as well as academic institutions (e.g. the University of Oxford) and is aimed primarily at sharing know-how.

The question arises: how much of the *classical* way of programming can evolve and contribute to quantum programming? The current paper proposes one such evolution, termed *quantamorphism*, that enables the construction of quantum (recursive) programs in a structured way. To motivate this concept, it is worth looking back to the past and

(briefly) reviewing how similar strategies arose in classical programming. Indeed, as history goes, there are striking similarities between the evolution of quantum computing and that of its classical forerunner.

Classical computing is rooted on mathematical abstractions that led in particular to the *Turing machine* [3] – which is still regarded as the canonical abstract notion of a programmable computer – and to the λ-calculus [4] – a mathematical system that provided the basis for functional programming.

A step from abstraction to reality was made possible by advances in physics, such as the invention of triodes (1912) and then of transistors (1948), leading to the integrated circuits that are the basis of the *in silico* technology of today [5], [6], [7].

Once such devices were first employed to store information in realistic situations, it became clear that further abstraction was required. This led to the explicit adoption of formal logic, a very important abstraction still in use today. As the aphorism says, *"logic is the language of computing"*.

Analogously to classical computing, but several decades later, quantum computing was also born out of mathematical abstractions, this time with the description of the first universal quantum computer by Deutsch [8]. And the parallel goes on: nowadays, quantum physicists and engineers are testing strategies to implement such abstract concepts, linking theory to reality once again.

Soon ideas for quantum programming arose [9], not only at the flowchart level [10], [11] but also in the functional programming style [12]. And so, in a similar fashion to what happened for classical computation, *software* started finding its way into quantum computation's history.

The birth of software as an independent technology took place in the 1950s. But it soon was faced with a crisis

• *A. Neri and J. N. Oliveira are with HASLab/INESC TEC, Universidade do Minho, Braga, Portugal.*
  E-mail: ana.neri@quantalab.uminho.pt, jno@di.uminho.pt
• *R. S. Barbosa is with INL – International Iberian Nanotechnology Laboratory, Braga, Portugal. This work was done in part while RSB was with the Department of Computer Science, University of Oxford, United Kingdom.*
  E-mail: rui.soaresbarbosa@inl.int

because an effective discipline of programming was lacking. The term *software engineering* appeared in the 1960s and was the subject of a conference supported by NATO that took place in Garmisch, Germany in 1968. People at this conference expressed concerns and called for theoretical foundations. This resulted in the birth of the principles of *structured programming* that became popular in the 1970s. But, in a sense, the 1968 crisis is not over yet: the problem with software engineering is that quality control is based on *testing* software artifacts *after* they have been built, and not on ensuring quality in a stepwise manner, as advocated by academia since the 1970s.

Some believe that the problem is lack of mathematical abstraction once again [13]. Stepping back to the original computational abstractions of the 1930s, the λ-calculus was developed with the aim of creating a model of computation based exclusively on *function* abstraction and application. This led to a mathematically robust style of programming known as functional programming (FP), which has become a reliable paradigm for software production. The *correct-by-construction* programming techniques proposed in this field have had a significant impact on software theory. Such techniques promise a significant reduction in development costs by avoiding dependence on testing and debugging.

Correct-by-construction design techniques advocate the calculation of programs from problem specifications. This is the main aim of the so-called "Mathematics of Program Construction" (MPC) discipline [14], a branch of mathematics applied to program calculation based on logic and relational algebra.

In the functional setting, such a discipline has led to the so-called *"Algebra of Programming"* (AoP) which is the subject of textbook [15]. The branch of mathematics that supports the AoP abstractions is category theory [16].

Despite its strong algebraic basis – cf. Hilbert spaces, linear algebra, etc. [17] – quantum mechanics is still a counter-intuitive theory and one that will require further abstractions for programmers. In quantum mechanics, every observation implies the destruction of superposed states, spoiling the quantum advantage altogether. This renders current step-by-step *debugging* strategies obsolete and nearly impossible: one needs to *get it right* from the very beginning!

2 RESEARCH QUESTIONS

Because testing and debugging cannot apply to quantum programming, at least in current standards, the traditional life-cycle based on *edit-compile-run* is not an option. This further increases the need for correct-by-construction methods, leading us into the main research questions addressed by the current paper:

1) Is it possible to extend the MPC culture, principles and constructions – which have been so effective in disciplining the whole field of (classical) recursive functional programming and data structuring – to quantum programming?
2) Is it viable to apply such constructions to derive programs down to the level of actually *running* them on the experimental quantum devices of today?

An important requirement to take into account when scaling classical paradigms to the quantum level is *reversibility*, because quantum programs are limited to *unitary transformations* [18], which are special cases of reversible operations. Therefore, this research largely intersects with that on *reversible* computing.

A similar extension of the MPC paradigm to *probabilistic programming* has been shown to be viable in practice [19], although in a very different context: that of reasoning about program reliability in the presence of faulty hardware. The laws of that approach require *typed* linear algebra rather than just the algebra of *relations*[1] in order to reason about probabilistic functions (Markov chains). On the experimental side, this requires programming over the *distribution monad*.[2]

The recursive programming construction studied in [19] is the so-called (probabilistic) *catamorphism* [15]. The idea in the current paper is to generalise from such catamorphisms to unitary transformations over a vector space monad implementing finite-dimensional Hilbert spaces [22]. The corresponding extension of the catamorphism concept, to be developed further in this paper, is termed "quantamorphism" – a restricted form of *recursive* quantum control of *quantum* data [23].

A half-way concept between classical functions and unitary transformations is that of a reversible function, also known as *isomorphism* or *bijection*. The paper will contribute to the current investment in reversible computing by extending a technique known as *complementation* [24] to recursive programs. The background of all this research is also enhanced by studies in quantum functional programming (QFP) [12], [25] and extensive research in categorical quantum physics [22], [26], [27].

3 ALGEBRA OF PROGRAMMING

The standard algebra of programming [15] is an evolution of the binary relation algebra pioneered by Augustus de Morgan (1806–71). Later, Peirce (1839–1914) invented *quantifier* notation to explain de Morgan's algebra of relations.[3] De Morgan's pioneering work was ill-fated: the language invented to explain his calculus of relations became eventually more popular than the calculus itself – it is nowadays known as first-order logic (FOL).

Alfred Tarski (1901–83), who had a life-long struggle with quantifier notation, revived relation algebra. Together with Steve Givant he wrote a book (published posthumously) on *set theory without variables* [29].

Meanwhile, category theory [16] was born, stressing the description of mathematical concepts in terms of abstract *arrows* (morphisms) and *diagrams*, unveiling a compositional, abstract language of universal *combinators* that is inherently *generic* and *pointfree*.

The category of sets and functions immediately provided a basis for pointfree functional reasoning, but this was by and large ignored by John Backus (1924–2007) in his FP algebra of programs [30]. In any case, Backus's landmark

---

1. This has been referred to by the acronym LAoP ("linear algebra of programming") [20].
2. As implemented by [21] in Haskell.
3. See e.g. [28] for a comprehensive historical overview.

FP paper was the first to show how relevant this reasoning style is to programming. This happened four decades ago.

A bridge between the two pointfree schools – the relational and the categorical – was eventually established by Freyd and Scedrov [31] in their proposal of the concept of an *allegory*, which instantiates to *typed* relation algebra. The pointfree algebra of programming (AoP) as it is understood today [15] stems directly from [31].

4 REVERSIBILITY

Standard program design relies on program *refinement* techniques [32], [33]. A program (or specification, or model) is refined wherever it leads to a more *defined* version of it, in a double sense: more deterministic and more responsive. The limit of a refinement process is always a *function*: a totally defined and fully deterministic computational process.

Quantum programming brings with it a new concern in programming, that of *reversibility*. This concern is relatively new in the traditional algebra of programming. In fact, classical *program* design, e.g. by source-to-source *transformation*, primarily seeks time and space *efficiency* but not *reversibility*.

Reversible computations are *functions* that are injective and surjective – that is, *bijective*. Recall that a function $f : X \to Y$ is injective iff

$$f\ x = f\ x' \Rightarrow x = x'$$

holds for every $x, x' \in X$, and *surjective* iff, for all $y \in Y$, there exists some x such that $y = f\ x$.

Refinement is normally expressed in terms of a *preorder* $p \leqslant q$ meaning that program q is more refined than program p, that is, q is closer to an implementation of p.[4] Refinement towards reversibility calls for an *injectivity* pre-order – one that will enable us to order the functions in the picture below in the way shown:
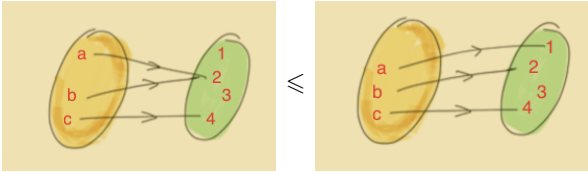
The intuition is that injective functions *discriminate* more: in the picture, a and b are mapped onto the same output (number 2) by the less injective function on the left-hand side, while on the right-hand side the fact that a and b are different is preserved at image level ($1 \neq 2$).

In general, g is said to be *less injective* than f if

$$g \leqslant f \quad \Leftrightarrow \quad f\ x = f\ x' \Rightarrow g\ x = g\ x'$$

holds.

A problem with the definitions just given, which are standard in mathematics, is that they are declarative but not calculational. Moreover, there are simpler ways of saying the same things. For instance, function g in

$$f \cdot g = id$$

is *injective* because it has a left inverse f (which is *surjective*). An equivalent way of writing $f \cdot g = id$ is

---

4. This ensures that a program always refines itself (reflexivity) and that a refinement of a refinement is also a refinement (transitivity).

$$g \subseteq f^\circ$$

using the relation algebra *converse* operator: $b = f\ a$ means the same as $a\ f^\circ\ b$. This says that g injective because g is smaller than the *converse* of a function (f°), and function converses are always injective. In case $g = f^\circ$ then *both* f and g will be injective and surjective, i.e. bijections.

Instead of relying directly on first-order logic, this style of argumentation relies on *relation algebra* [15], as detailed next.

5 FUNCTIONS AND RELATIONS

In the same way that we declare a function $f : A \to B$ by specifying its input type A and output type B, and write $f : B \leftarrow A$ to mean exactly the same thing, so we write $R : B \leftarrow A$ or $R : A \to B$, or even $A \xrightarrow{R} B$ or $B \xleftarrow{R} A$, to declare the type of a relation R. Moreover, we use infix notation $b\ R\ a$ to denote $(b, a) \in R$, in the tradition of $b \leqslant a$, $b \in s$, and so on.

Functions are special cases of relations. We use lowercase letters (e.g. f, g, ...) to denote functions and uppercase letters (e.g. R, S, ...) to denote relations. The singularity of functions as relations is captured by $b\ f\ a \Leftrightarrow b = f\ (a)$.

Given relations $B \xleftarrow{R} C$ and $C \xleftarrow{S} A$, their composition $C \xleftarrow{R \cdot S} A$ is defined by $b\ (R \cdot S)\ a$ iff, for some $c \in C$, $b\ R\ c$ and $c\ S\ a$ hold. In the case of functions, this specialises to the familiar composition of functions: $b\ (f \cdot g)\ a$ means $b = f\ (g\ a)$. The unit of composition is the identity function $id\ x = x$, that is, $R \cdot id = R = id \cdot R$.

Relations of the same type are ordered by entailment, i.e. inclusion. This is denoted by $R \subseteq S$ meaning that $b\ R\ a$ logically implies $b\ S\ a$ for all b, a.

The converse $A \xleftarrow{R^\circ} B$ of a relation $A \xrightarrow{R} B$ is such that $a\ R^\circ\ b$ means the same as $b\ R\ a$. In the case of a function $f : A \to B$, its *converse* is the relation $f^\circ : A \leftarrow B$ such that $a\ f^\circ\ b \Leftrightarrow b = f\ a$.

A taxonomy of relations

A relation $R : A \to B$ is said to be *injective* whenever $R^\circ \cdot R \subseteq id$ holds.[5] Moreover:

$$R\ injective \Leftrightarrow \underbrace{R^\circ \cdot R}_{\ker R} \subseteq id$$
$$R\ simple \Leftrightarrow R^\circ\ injective$$
$$R\ entire \Leftrightarrow id \subseteq \underbrace{R \cdot R^\circ}_{img\ R}$$
$$R\ surjective \Leftrightarrow R^\circ\ entire$$

leading to the taxonomy of Figure 1. Below we shall use the definition of *kernel* of a relation:

$$\ker R \stackrel{\text{def}}{=} R^\circ \cdot R$$

In the case of functions, $a'\ (\ker f)\ a$ means $f\ a' = f\ a$, that is, a' and a have the same image under f. For any f, $\ker f$ is always an *equivalence* relation [34]. If f is injective, this equivalence is the identity. Moreover,

$$f \text{ is bijective} \quad \Leftrightarrow \quad \ker f = id \wedge \text{img}\ f = id \qquad (1)$$

that is, $f^\circ \cdot f = id$ and $f \cdot f^\circ = id$.

---

5. For functions ($R := f$), $f^\circ \cdot f \subseteq id$ means precisely what we had before, $f\ x = f\ x' \Rightarrow x = x'$.
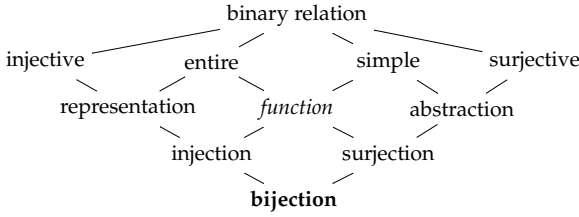
Fig. 1. Binary relation taxonomy (relation 'bestiary').

Relations as matrices

Following [35], it is helpful to depict relations using (Boolean) *matrices*, for instance Boolean *negation*

$$\mathbb{B} \xleftarrow{\neg} \mathbb{B} = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array} \tag{2}$$

(a *bijection*, also known as the X-gate), *exclusive-or*

$$\mathbb{B} \xleftarrow{\oplus} \mathbb{B} \times \mathbb{B} = \begin{array}{c|cccc} & 0 & 0 & 1 & 1 \\ & 0 & 1 & 0 & 1 \\ \hline 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{array} \tag{3}$$

(surjective but not injective), and so on. Clearly:

- *Function* matrices have *exactly one* 1 in every column.
- *Bijections* are square matrices with exactly one 1 in every *column* and in every *row*.

6 INJECTIVITY PREORDER

The injectivity preorder [36] is defined by

$$R \leqslant S \quad \Leftrightarrow \quad \ker S \subseteq \ker R$$

As an example, take two list functions: *elems* computing the set of all items of a list and *bagify* keeping the bag of such elements. The former loses more information (order and multiplicity) than the latter (which forgets the order only). Thus $elems \leqslant bagify$.

Below, we shall explore this preorder as a *refinement ordering* guiding us towards more and more *injective* computations, heading to *reversibility*.

This injectivity preorder is rich in properties. For instance, it is upper-bounded[6]

$$R \mathbin{\triangledown} S \leqslant X \quad \Leftrightarrow \quad R \leqslant X \wedge S \leqslant X \tag{4}$$

by relation *pairing*, which is defined in the expected way

$$(b, c)\,(R \mathbin{\triangledown} S)\,a \quad \Leftrightarrow \quad b\,R\,a \wedge c\,S\,a \tag{5}$$

specialising, in the case of functions, to

$$(f \mathbin{\triangledown} g)\,a = (f\,a, g\,a). \tag{6}$$

Cancellation over (4) means that *pairing* always *increases injectivity*:

$$R \leqslant R \mathbin{\triangledown} S \quad \text{and} \quad S \leqslant R \mathbin{\triangledown} S. \tag{7}$$

Facts (7) are jointly equivalent to $\ker (R \mathbin{\triangledown} S) \subseteq (\ker R) \cap (\ker S)$, which in fact is an equality:

$$\ker (R \mathbin{\triangledown} S) = (\ker R) \cap (\ker S). \tag{8}$$

6. See e.g. [36] for more details.

This is a corollary of the following more general law:[7]

$$(R \mathbin{\triangledown} S)^{\circ} \cdot (Q \mathbin{\triangledown} P) = (R^{\circ} \cdot Q) \cap (S^{\circ} \cdot P). \tag{9}$$

Injectivity *shunting laws* also arise by standard relational algebra calculation [36], for instance:

$$R \cdot g \leqslant S \quad \Leftrightarrow \quad R \leqslant S \cdot g^{\circ}$$

Restricted to *functions*, the preorder $(\leqslant)$ is *universally bounded* by

$$! \leqslant f \leqslant id$$

where $1 \xleftarrow{!} A$ is the unique function of its type, where symbol 1 denotes the singleton type. Moreover,

- A function is *injective* iff $id \leqslant f$ holds, that is, $\ker f = id$. Consequently, $f \mathbin{\triangledown} id$ is always *injective*, by (7).
- Two functions f and g are said to be *complementary* whenever $id \leqslant f \mathbin{\triangledown} g$.[8]

For instance, the *projections* $fst\,(a, b) = a$ and $snd\,(a, b) = b$ are complementary since $fst \mathbin{\triangledown} snd = id$.

7 MINIMAL COMPLEMENTS

Given some f, suppose that: (a) $id \leqslant f \mathbin{\triangledown} g$ for some g; (b) if $id \leqslant f \mathbin{\triangledown} h$ for some h such that $h \leqslant g$, then $g \leqslant h$ holds. Then g is said to be a *minimal complement* of f [37]. Minimal complements (not unique in general) capture *"what is missing"* from the original function for *injectivity* to hold.

Calculating a minimal complement g for a given function f can be regarded as a *correct-by-construction* strategy for implementing f in a reversible way, in the sense that: (a) $f \mathbin{\triangledown} g$ is injective even if f is not, and (b) f is implemented by $f \mathbin{\triangledown} g$, since $f = fst \cdot (f \mathbin{\triangledown} g)$.

For Boolean functions, minimal complements are easy to calculate using matrices. In the following example we wish to calculate a minimal complement for the (non-injective) exclusive-or Boolean operator:

$$\mathbb{B} \xleftarrow{\oplus} \mathbb{B} \times \mathbb{B} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \tag{10}$$

We start from the kernel of $\oplus$:

$$\ker \oplus = \ker \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

By (8), the kernel of a complement g has to cancel all 1s in $\ker \oplus$ that fall outside the diagonal id. The identity function itself would do this,

$$\ker id = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

but this would be an overkill – id complements any function! Moreover, it is not minimal in this case. To reduce

7. Details in [15].
8. Cf. [24]. Other terminologies are *monic pair* [31] or *jointly monic* [15].

injectivity we need to start adding 1s to ker id where ker $\oplus$ has 0s, e.g.

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \tag{11}$$

However, this isn't a kernel anymore. Why? Because it is not an equivalence relation: it is *reflexive* (cf. diagonal) and *symmetric*, but not *transitive*.

To handle transitivity we resort to a basic result in relation algebra: *a symmetric and reflexive relation is an* equivalence *iff it is a difunctional relation*, where

a relation R is *difunctional* iff $R \cdot R^\circ \cdot R \subseteq R$ [34].

One can construct finite *difunctional* relations easily, by inspection: just make sure that columns either do not intersect or are the same. Clearly, (11) is not difunctional.

To make it difunctional, we have to surgically cancel zeros symmetrically, outside the diagonal:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

What we obtain is ker fst, the kernel of the first projection $\mathbb{B} \xleftarrow{fst} \mathbb{B} \times \mathbb{B} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$. So, function $fst\ (a, b) = a$ is a minimal complement of $\oplus$.

We said that minimal complements are not unique in general and this is one such case. Indeed, we might have decided to perform alternative cancellations, e.g.

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

ending up this time with ker snd, the kernel of the other projection $snd\ (a, b) = b$. So, both fst and snd are *minimal complements* of $\oplus$.

Let us see what comes out of the fst-complementation of exclusive-or:

$$\mathbb{B} \times \mathbb{B} \xleftarrow{fst^\triangledown \oplus} \mathbb{B} \times \mathbb{B} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{12}$$

This is a well-known *bijection*, in fact a familiar gate known as CX, or CNOT (for "*controlled not*"), usually depicted as follows:
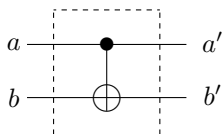

Why does it bear this name? We calculate:

$$cnot = fst^\triangledown \oplus$$
$$\Leftrightarrow \qquad \{ \text{ go pointwise } \}$$
$$cnot\ (a, b) = (a, a \oplus b))$$
$$\Leftrightarrow \qquad \{ \text{ since } 0 \oplus b = b \text{ and } 1 \oplus b = \neg\ b \}$$
$$\begin{cases} cnot\ (0, b) = (0, b) \\ cnot\ (1, b) = (1, \neg\ b) \end{cases} \tag{13}$$

Informally: *controlled* bit b is negated *iff* the *control* bit a is set; otherwise, the gate does nothing.

Thus we have designed this gate following a *constructive* approach – we built it by *minimal* complementation. Note the role of the fst complement in copying the control bit to the output.

Other fst-complementations

As a second example, we take the classical circuit
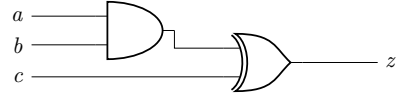
Can this be made into a bijection in the same way? The function implemented is

$$\mathbb{B}^2 \times \mathbb{B} \xrightarrow{f = \oplus \cdot ((\wedge) \times id)} \mathbb{B} \quad = \quad \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$
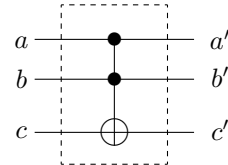
where

$$(f \times g)\ (a, b) = (f\ a, f\ b) \tag{14}$$

is the "tensor" product of two functions. Let us complement f with fst again, which in this context has type $\mathbb{B}^2 \times \mathbb{B} \xrightarrow{fst} \mathbb{B}^2$. The outcome is another *bijection*, known as the CCNOT, or Toffoli, gate

$$ccnot = fst^\triangledown (\oplus \cdot ((\wedge) \times id)) = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

usually depicted as follows:

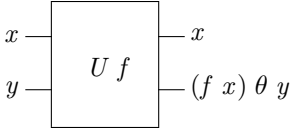As for $cnot$, a similar calculation will lead to the pointwise version:

$$ccnot : \mathbb{B}^2 \times \mathbb{B} \to \mathbb{B}^2 \times \mathbb{B}$$
$$ccnot\ ((1, 1), c) = ((1, 1), \neg\ c)$$
$$ccnot\ ((a, b), c) = ((a, b), c)$$

As a last example of fst-complementation, let us see a famous device in quantum programming arising from the following generic evolution of the CNOT gate, parametric

on $A \xrightarrow{f} B$ and such that $(B; \theta, 0)$ is a monoid satisfying $x \; \theta \; x = 0$ for all $x \in B$:[9]

$$U \; f : (A \to B) \to (A \times B) \to (A \times B)$$
$$U \; f = fst \triangledown \theta \cdot (f \times id) \, ,$$

that is,

Clearly, for $\theta = \oplus$:

$$cnot = U \; id$$
$$ccnot = U(\wedge)$$

It is easy to see that, for every f, $U f$ is a *bijection* because it is its own inverse:

$$U f \cdot U f = id$$
$$\Leftrightarrow \qquad \{ \; U f \; (x, y) = (x, (f \; x) \; \theta \; y) \; \}$$
$$U f \; (x, (f \; x) \; \theta \; y) = (x, y)$$
$$\Leftrightarrow \qquad \{ \; \text{again } U f \; (x, y) = (x, (f \; x) \; \theta \; y) \; \}$$
$$(x, (f \; x) \; \theta \; ((f \; x) \; \theta \; y)) = (x, y)$$
$$\Leftrightarrow \qquad \{ \; \theta \text{ is associative and } x \; \theta \; x = 0 \; \}$$
$$(x, 0 \; \theta \; y) = (x, y)$$
$$\Leftrightarrow \qquad \{ \; 0 \; \theta \; x = x \; \}$$
$$(x, y) = (x, y)$$
$$\square$$

$U f$ is therefore a *reversible refinement* of an arbitrary $f : A \to B$ (for a monoid B as above) in the sense that[10]

$$snd \cdot U f \cdot (id \times \underline{0}) = f \, ,$$

or in pointwise notation,

$$f \; x = b \; \textbf{where} \; (\_, b) = U f \; (x, 0).$$

8 THE DUAL VIEW

Before moving on and generalising *fst*-complementation to more interesting programming constructs, we present a non-standard perspective of Boolean gates which is based on coproducts $(A + B)$ rather than products $(A \times B)$. By $A + B$ we mean the *disjoint union* of A and B:

$$A + B = \{ i_1 \; x \mid x \in A \} \cup \{ i_2 \; y \mid y \in B \} \tag{15}$$

where i_1 and i_2 are injective. (Disjointness relies on assuming $i_1^{\circ} \cdot i_2 = \bot$, that is, for all x and y, $i_1 \; x \neq i_2 \; y$.)

Given any two relations $A \xrightarrow{R} C$ and $B \xrightarrow{S} C$, there exists a unique relation $A + B \xrightarrow{X} C$ such that $X \cdot i_1 = R$ and $X \cdot i_2 = S$. We denote that relation by $[R, S]$:

$$X = [R, S] \quad \Leftrightarrow \quad \left\{ \begin{array}{l} X \cdot i_1 = R \\ X \cdot i_2 = S \end{array} \right. \tag{16}$$

---

9. Our convention is that $M \cdot N$ takes precedence over all other binary combinators, so $fst \triangledown \theta \cdot (f \times id)$ means $fst \triangledown (\theta \cdot (f \times id))$. For economy of notation, we overload the θ symbol to denote both the uncurried and curried versions of the operator.

10. $\underline{0}$ is the everywhere 0-constant function. In general, $\underline{k} \; x = k$ for all suitably typed x and k.

The *direct sum* of two relations arises immediately from:

$$R + S = [i_1 \cdot R, i_2 \cdot S] \tag{17}$$

The isomorphism

$$\mathbb{B} \times A \xleftarrow{\gamma} A + A \; = [false \triangledown id, true \triangledown id]) \tag{18}$$

holds (where *false* and *true* are the obvious constant functions) and can be re-written into

$$\gamma = [false, true] \triangledown [id, id] \tag{19}$$

thanks to the so-called *exchange law*:

$$[R \triangledown S, T \triangledown V] \;\; = \;\; [R, T] \triangledown [S, V]. \tag{20}$$

So, we have that $\mathbb{B} \times \mathbb{B}$ is isomorphic to $\mathbb{B} + \mathbb{B}$ through γ (18). This provides us with an alternative (dual) view of logic gates, for instance: conjunction

$$\mathbb{B} + \mathbb{B} \xrightarrow{(\wedge) \cdot \gamma} \mathbb{B} \; = [false, id], \tag{21}$$

disjunction

$$\mathbb{B} + \mathbb{B} \xrightarrow{(\vee) \cdot \gamma} \mathbb{B} \; = [true, \neg], \tag{22}$$

exclusive-or

$$\mathbb{B} + \mathbb{B} \xrightarrow{\oplus \cdot \gamma} \mathbb{B} \; = [id, \neg], \tag{23}$$

and so on.

Note how $\mathbb{B} + \mathbb{B}$ captures the second bit of a Boolean gate once the first is set to false (on the left of the sum) or to true (on the right of the sum). So (23) immediately tells that exclusive-or behaves as the identity in the first case and as negation in the second. In matrix notation – cf. (10):

$$\oplus = \left[\begin{array}{c} \left[\begin{array}{cc} 1 & 0 \\ 0 & 1 \end{array} \right] \end{array} \middle| \begin{array}{c} \left[\begin{array}{cc} 0 & 1 \\ 1 & 0 \end{array} \right] \end{array} \right].$$

Applying the same transformation to *fst*-complemented operations yields similarly expressive denotations of Boolean gates. For instance, $cnot = fst \triangledown \oplus$ is transformed into $id + \neg$ through γ:

$$cnot \cdot \gamma$$
$$= \qquad \{ \; cnot = fst \triangledown \oplus; \text{ pairing; (23)} \; \}$$
$$(fst \cdot \gamma) \triangledown [id, \neg]$$
$$= \qquad \{ \; \text{(19); pairing; exchange law} \; \}$$
$$[false \triangledown id, true \triangledown \neg]$$
$$= \qquad \{ \; true \cdot f = f; \text{(18); pairing laws} \; \}$$
$$\gamma \cdot (id + \neg)$$

That is, γ has the relational type[11]

$$fst \triangledown \oplus \xleftarrow{\gamma} id + \neg \, .$$

Written as $id + \neg$, $cnot$ is immediately seen to be an isomorphism, because id and $\neg$ are so. In the same setting, the Toffoli gate $ccnot$ will be expressed as

$$id + (id + \neg),$$

again an isomorphism by construction.

---

11. In general, f is said to have relational type $S \leftarrow R$ whenever $f \cdot R \subseteq S \cdot f$ holds.

Isomorphism γ will play an important role in implementing a form of conditional quantum control in section 13. The coproduct construct is also inherently present in the strategy that underlies section 14.

9 GENERALISING fst-COMPLEMENTATION

As seen in section 7, the projection $A \times B \xrightarrow{fst} A$ plays a role in injectivity refinements, working as minimal complement in several situations. In general, fst-complementation

$$id \leqslant fst \circ f$$

works whenever

$$f\,(a, b) = f\,(a, b') \Rightarrow b = b'$$

holds, that is[12]:

$$\overline{f}\;a\;b = \overline{f}\;a\;b' \Rightarrow b = b'.$$

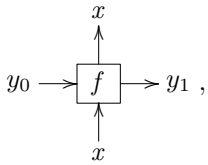In other words, f is left-cancellative: it is *injective* on the *second* argument once the *first* is fixed.

Wherever $A \times B \xrightarrow{fst} A$ complements a function of type $A \times B \longrightarrow B$, it makes room (type-wise) for a *bijection* of type $A \times B \longrightarrow A \times B$. Can $fst \circ (\_)$ be extended to more elaborate computations, e.g. *recursively*? Note that such $A \times B \to A \times B$ computations, of shape

$$
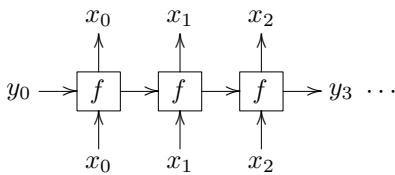\begin{array}{ccc}
A \longrightarrow & \boxed{} & \longrightarrow A \\
B \longrightarrow & & \longrightarrow B
\end{array}
$$

can be chained together. Take, for instance, n copies of

$$
\begin{array}{ccc}
x \longrightarrow & \boxed{f} & \longrightarrow x \\
y_0 \longrightarrow & & \longrightarrow y_1
\end{array}
$$
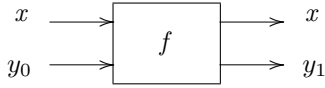
and draw each of them in a different way,

$$
\begin{array}{c}
x \\
\uparrow \\
y_0 \longrightarrow \boxed{f} \longrightarrow y_1 \;, \\
\uparrow \\
x
\end{array}
$$

so that they can be chained as depicted below:

$$
\begin{array}{cccc}
x_0 & x_1 & x_2 & \\
\uparrow & \uparrow & \uparrow & \\
y_0 \longrightarrow \boxed{f} \longrightarrow \boxed{f} \longrightarrow \boxed{f} \longrightarrow y_3 \; \cdots \\
\uparrow & \uparrow & \uparrow & \\
x_0 & x_1 & x_2 &
\end{array}
$$

Clearly, $[x_0, x_1, x_2, ...]$ can be regarded as a control-sequence, which is passed along to the output. Meanwhile, the input y_0 is subject to an accumulation of transformations performed by f, one for each x_i.

Interestingly, this chain can be regarded as an instance of a functional programming pattern known as the *accumulating map*.[13] This pattern turns up in various contexts (namely in neural networks, see Figure 2). In the sequel, it will be shown to be an instance of a construct that we shall introduce shortly and term *quantamorphism*, as it will generalise to quantum computing later.

12. We abbreviate $curry\ f$ by $\overline{f}$, that is: $\overline{f}\;a\;b = curry\;f\;a\;b = f\,(a, b)$.
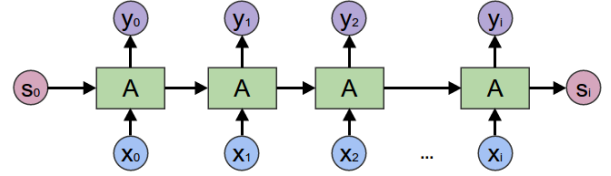13. Cf. e.g. the function $mapAccumR$ in the Haskell language.

Fig. 2. Recurrent neural network (RNN) depicted in [38] as an instance of an accumulating map.

10 TOWARDS (CONSTRUCTIVE) RECURSIVE COMPLEMENTATION

Suppose one wants to offer an *arbitrary* function $k : A \to B$ wrapped in a *bijective* "envelope", as happened above in the derivation of $cnot$ and other gates. The "smallest" (*generic*) type for such an enveloped function is $A \times B \to A \times B$.

Now suppose that k is a *recursive* function over finite lists, for instance $k = \text{foldr}\ \overline{f}\ b$ for $f : A \times B \to B$, that is,

$$
\begin{aligned}
&k : A^* \to B \\
&k\,[\,] = b \\
&k\,(a : x) = f\,(a, k\ x)\,.
\end{aligned}
$$

How do we "constructively" build the corresponding (*recursive*, *bijective*) envelope of type $A^* \times B \to A^* \times B$? Let us define $(\!|f|\!)$ such that

$$(\!|f|\!)\,(x, b) = \text{foldr}\ \overline{f}\ b\ x, \tag{24}$$

that is,

$$
\begin{aligned}
&(\!|f|\!)\,([\,], b) = b \\
&(\!|f|\!)\,(a : x, b) = f\,(a, (\!|f|\!)\,(x, b)).
\end{aligned}
$$

We can depict $(\!|f|\!)$ in the form of a commutative diagram:

$$
\begin{array}{ccc}
A^* \times B & \xleftarrow{\ \alpha\ } & B + A \times (A^* \times B) \\
{\scriptstyle (\!|f|\!)}\downarrow & & \downarrow{\scriptstyle id + id \times (\!|f|\!)} \\
B & \xleftarrow[{[id,\,f]}]{} & B + A \times B
\end{array}
$$

In the diagram, the isomorphism

$$A^* \times B \xleftarrow{\ \alpha\ } B + A \times (A^* \times B)$$

is defined by

$$\alpha = [nil \circ id, (cons \times id) \cdot \mathsf{a}] \tag{25}$$

where

$$
\begin{aligned}
&nil\ \_ = [\,] \\
&cons\,(a, x) = a : x
\end{aligned}
$$

are the components of the initial algebra $in = [nil, cons]$ of finite lists [15], and the isomorphism

$$
\begin{aligned}
&\mathsf{a} : A \times (B \times C) \to (A \times B) \times C \\
&\mathsf{a} = (id \times fst) \circ (snd \cdot snd)
\end{aligned} \tag{26}
$$

in (25) is the associator.

Below, we will need something more general, namely:

$$
\begin{array}{ccc}
A^* \times B & \xleftarrow{\ \alpha\ } & B + A \times (A^* \times B) \\
{\scriptstyle (\!|h|\!)}\downarrow & & \downarrow{\scriptstyle id + id \times (\!|h|\!)} \\
C & \xleftarrow[{h}]{} & B + A \times C
\end{array} \tag{27}
$$

This is specified by the *universal* property

$$k = (\!(h)\!) \quad \Leftrightarrow \quad k \cdot \alpha = h \cdot \mathsf{F}\, k \tag{28}$$

where $\mathsf{F}\, f = id + id \times f$ captures the list-recursion pattern. This $(\!(h)\!)$ can be regarded as an extension of the well-known *catamorphism* combinator.[14] All the standard laws apply, including reflexion $(\!(\alpha)\!) = id$ and the loop-intercombination law

$$(\!(f)\!) \triangledown (\!(g)\!) = (\!((f \times g) \cdot (\mathsf{F}\, fst \triangledown \mathsf{F}\, snd))\!), \tag{29}$$

often referred to as the *"banana-split"* law [15]. From (28) and (26), one also infers

$$A^* \xleftarrow{\ fst\ } A^* \times B \ = (\!(in)\!) \tag{30}$$

by easy calculation:

$$fst = (\!(in)\!)$$
$$\Leftrightarrow \qquad \{\ (28)\ \}$$
$$fst \cdot \alpha = in \cdot (id + id \times fst)$$
$$\Leftrightarrow \qquad \{\ in = [nil, cons]; \text{ coproducts } \}$$
$$fst \cdot \alpha = [nil, cons \cdot (id \times fst)]$$
$$\Leftrightarrow \qquad \{\ \text{definition of } \alpha \text{ (25) and a (26) } \}$$
$$true$$
$$\square$$

Promoting fst-complementation

Suppose that a given *non-injective* $f : A \times B \to B$ is complemented by $fst : A \times B \to A$, i.e. that $fst \triangledown f$ is injective. We can place it in (27) and ask: will $(\!([id, f])\!)$ be fst-complemented too? We start by unfolding the term $fst \triangledown (\!([id, f])\!)$:

$$fst \triangledown (\!([id, f])\!)$$
$$= \qquad \{\ (30) \text{ followed by banana-split (29) } \}$$
$$(\!((in \times [id, f]) \cdot (\mathsf{F}\, fst \triangledown \mathsf{F}\, snd))\!)$$
$$= \qquad \{\ in = [nil, cons]]; \text{ pairing laws (products) } \}$$
$$(\!([nil, cons \cdot (id \times fst)] \triangledown [id, f \cdot (id \times snd)])\!)$$
$$= \qquad \{\ \text{exchange law (20) } \}$$
$$(\!([nil \triangledown id, (cons \cdot (id \times fst)) \triangledown (f \cdot (id \times snd))])\!)$$
$$= \qquad \{\ \text{products}; \ a \cdot a^\circ = id \ \}$$
$$(\!(\alpha \cdot (id + a^\circ \cdot ((id \times fst) \triangledown f \cdot (id \times snd))))\!) \tag{31}$$

Thus,

$$fst \triangledown (\!([id, f])\!) \quad = \quad (\!(\Psi\, (fst \triangledown f))\!),$$

by introducing

$$\Psi\, x = \alpha \cdot (id + a^\circ \cdot ((id \times fst) \triangledown snd \cdot x \cdot (id \times snd))),$$

which shrinks to

$$\Psi\, x = \alpha \cdot (id + \mathsf{xl} \cdot (id \times x) \cdot \mathsf{xl}) \tag{32}$$

14. See e.g. [39] for a thorough study of such kind of extensions to the standard theory [15].

using the isomorphism $B \times (A \times C) \xleftarrow{\ \mathsf{xl}\ } A \times (B \times C)$ instead of a.[15] Putting everything into a diagram, we obtain:


Clearly, Ψ preserves injectivity, as does $(\!(\_)\!)$ – see the appendix for details. Therefore, $fst \triangledown f$ being injective ensures $fst \triangledown (\!([id, f])\!)$ is also injective. In words:

> *The fst-complementation of f in* foldr $\overline{f}$ b *is promoted to the fst-complementation of the fold itself.*

That is to say, fst-complementation is *propagated* inductively across lists and we get the construction of a *reversible fold*, defined by rfold $f = (\!(\Psi\, (fst \triangledown f))\!)$. Unfolding the definition and adding variables, we get, in standard Haskell notation:

```
rfold :: ((a, b) → b) → ([a], b) → ([a], b)
rfold f ([], b) = ([], b)
rfold f (a : x, b) = (a : y, f (a, b'))
    where (y, b') = rfold f (x, b)
```

We can therefore rely on the reversibility of rfold f
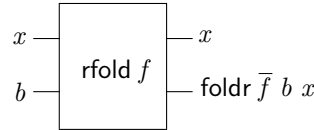
provided f is complemented by fst.

This result can be generalised by defining, given some $f : A \times B \to C \times B$, $(\!| f |\!) = (\!(\Psi\, f)\!)$ as pictured in the following diagram:

Clearly,

$$(\!| id |\!) = id \tag{34}$$

since $\Psi\, id = \alpha$. If f is reversible, then $(\!| f |\!)$ will also be reversible, as we have seen. For instance, $(\!| cnot |\!) : \mathbb{B}^* \times \mathbb{B} \to \mathbb{B}^* \times \mathbb{B}$ will be reversible, because so is $cnot : \mathbb{B} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}$.

By *free theorem* calculation [40], we get the following properties among others:[16]

$$(\!| f |\!) \cdot (k^* \times id) = (\!| f \cdot (k \times id) |\!) \tag{35}$$
$$(k^* \times id) \cdot (\!| f |\!) = (\!| (k \times id) \cdot f |\!) \tag{36}$$

Converting the construction $(\!| \cdot |\!)$ to Haskell notation yields

```
(|·|) :: ((a, b) → (c, b)) → ([a], b) → ([c], b)
(|f|) ([], b) = ([], b)
(|f|) (a : x, b) = (c : y, b'') where
    (y, b') = (|f|) (x, b)
    (c, b'') = f (a, b')
```

which corresponds to the standard $mapAccumR$ function.

15. Calculations can be found in the appendix.
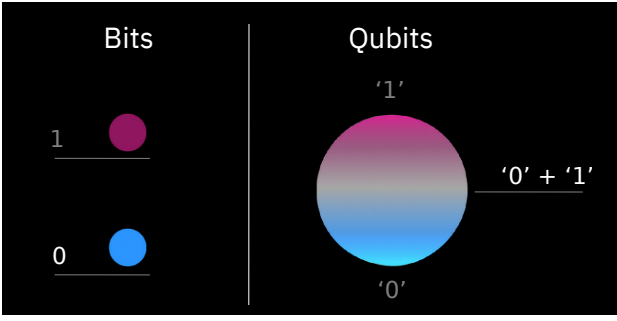16. k^* denotes the usual $map\ k$ operation on sequences.

Fig. 3. Suggestive depiction of a *qubit* as a *superposition* of classical bits. Credits: [41].

In the sequel, we shall refer to $\langle\!\langle f \rangle\!\rangle$ as being a *quantamorphism*. The reason for this is that this generalises nicely to quantum programming (for f reversible) as we shall see next.

11 GOING QUANTUM

Recall that *functions* can be represented by matrices, e.g. the *controlled-not*

$$\begin{cases} cnot\ (0,b) = (0,b) \\ cnot\ (1,b) = (1,\neg\ b) \end{cases}$$

is described by the matrix:

	(0,0)	(0,1)	(1,0)	(1,1)
(0,0)	1	0	0	0
(0,1)	0	1	0	0
(1,0)	0	0	0	1
(1,1)	0	0	1	0

Now think of a *probabilistic* "evolution" of *cnot*:

	(0,0)	(0,1)	(1,0)	(1,1)
(0,0)	1	0	0	0
(0,1)	0	$\frac{1}{2}$	0	0
(1,0)	0	$\frac{1}{2}$	0	1
(1,1)	0	0	1	0

In this evolution, function *cnot* becomes *probabilistic*: *cnot* $(0,1)$ will evaluate to either $(0,1)$ or $(1,0)$ with equal probability (50%).

Moving further to *quantum computing* corresponds to generalising probabilities to *amplitudes*, for instance:

$$B = \begin{array}{c|cccc} & (0,0) & (0,1) & (1,0) & (1,1) \\ \hline (0,0) & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\ (0,1) & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\ (1,0) & 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ (1,1) & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \end{array} \qquad (37)$$

Amplitudes are *complex* numbers indicating the *superposition* of information at quantum information level (Figure 3).

Quantum programs (QP) are made of elementary units called quantum *gates*, for instance the T-gate,[17]

$$T = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1 & 0 \\ 1 & 0 & e^{i\frac{\pi}{4}} \end{array},$$

17. Where $e^{i\ x} = cos\ x + i\ (sin\ x)$ – recall Euler's formula.

and the *Hadamard* gate,

$$H = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{array}. \qquad (38)$$

which is a "component" of (37) in a sense to be made precise soon.

The question is: how do we combine the functional approach of the previous sections with such a dramatic quantum extension? Reviewing some background material is required at this point.

All *functions* in the taxonomy of Fig. 1 form a *category* [42] under functional composition, $(f \cdot g)\ a = f\ (g\ a)$, with identity $id\ a = a$. Moreover, all relations in the same taxonomy form another category, under relational composition, $b\ (R \cdot S)\ a \Leftrightarrow \langle \exists\ c\ ::\ b\ R\ c \wedge c\ S\ a \rangle$. This includes functions as special case, as already seen.

Both functions and relations can be regarded as $\{0,1\}$-matrices, provided 0 and 1 are regarded as Boolean values. Interestingly, such matrices can be extended to arbitrary (typed) matrices where 0 and 1 are, respectively, the unit of addition and of multiplication of a semiring structure. One such semiring is the field of complex numbers, allowing us to include matrices such as H and B above.

This leads us to a *linear algebra of programming* [20], in which matrices are typed and written in the same way as functions or relations, i.e. as arrows (morphisms) $B \xleftarrow{M} A$. This denotes a matrix M whose columns are indexed by A and rows by B. Under matrix multiplication, such matrices will form a category, too [43].

Although the category of (entire, total) functions and categories of matrices don't exactly have the same properties, they share a common ground of useful constructs. Coproducts correspond to direct sums of matrices, denoted by $M \oplus N$, and there is a *tensor* (14) given by the Kronecker product, written $M \otimes N$. For instance,

$$id \otimes H = \begin{array}{c|cccc} & (0,0) & (0,1) & (1,0) & (1,1) \\ \hline (0,0) & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & 0 \\ (0,1) & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & 0 \\ (1,0) & 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ (1,1) & 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{array}$$

Finally, one can interpret diagrams such as (33) in a category of matrices, meaning that such categories have *catamorphisms*. Let us consider two examples of $\langle\!\langle x \rangle\!\rangle$, for two instances of $\mathbb{B} \times \mathbb{B} \xrightarrow{x} \mathbb{B} \times \mathbb{B}$. In the first case, $x = cnot$ and one gets the typed matrix $\langle\!\langle cnot \rangle\!\rangle$ pictured in Fig. 4. This matrix clearly shows a (fragment of a) reversible function. For $x = B$ from (37), the matrix $\langle\!\langle B \rangle\!\rangle$ is depicted in Fig. 5.

However, what does $\langle\!\langle B \rangle\!\rangle$ mean, giving that B is a matrix and not a function? This leads us to truly quantum *quantamorphisms*, the main topic of the sections that follow.

12 QUANTUM ABSTRACTION

It is well known that every relation $R : A \to B$ can be represented faithfully as a set-valued function $\Lambda R : A \to \mathsf{P}\ B$, where $\mathsf{P}\ B$ denotes the powerset of B, under the correspondence: $b\ R\ a \Leftrightarrow b \in \Lambda R\ a$. Under this correspondence,

	$(\,[\,],0)$	$(\,[\,],1)$	$([0],0)$	$([0],1)$	$([0,0],0)$	$([0,0],1)$	$([1,0],0)$	$([1,0],1)$	$([1],0)$	$([1],1)$	$([0,1],0)$	$([0,1],1)$	$([1,1],0)$	$([1,1],1)$
$(\,[\,],0)$	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$(\,[\,],1)$	0	1	0	0	0	0	0	0	0	0	0	0	0	0
$([0],0)$	0	0	1	0	0	0	0	0	0	0	0	0	0	0
$([0],1)$	0	0	0	1	0	0	0	0	0	0	0	0	0	0
$([0,0],0)$	0	0	0	0	1	0	0	0	0	0	0	0	0	0
$([0,0],1)$	0	0	0	0	0	1	0	0	0	0	0	0	0	0
$([1,0],0)$	0	0	0	0	0	0	0	1	0	0	0	0	0	0
$([1,0],1)$	0	0	0	0	0	0	1	0	0	0	0	0	0	0
$([1],0)$	0	0	0	0	0	0	0	0	0	1	0	0	0	0
$([1],1)$	0	0	0	0	0	0	0	0	1	0	0	0	0	0
$([0,1],0)$	0	0	0	0	0	0	0	0	0	0	0	1	0	0
$([0,1],1)$	0	0	0	0	0	0	0	0	0	0	1	0	0	0
$([1,1],0)$	0	0	0	0	0	0	0	0	0	0	0	0	1	0
$([1,1],1)$	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Fig. 4. Matrix for $\langle\!\langle cnot \rangle\!\rangle$ truncated to input (output) lists of maximum length 2 for visualisation purposes.

relation composition of $R : A \to B$ and $S : B \to C$ is given by

$$c\,(S \cdot R)\,a \Leftrightarrow c \in \bigcup\{\Lambda S\;b \mid b \in \Lambda R\;a\,\}.$$

This can be written in a more generic way. Let f and g abbreviate ΛR and ΛS, respectively, with types $f : A \to \mathsf{P}\;B$ and $g : B \to \mathsf{P}\;C$. Then composition $S \cdot R$ is represented by $g \bullet f$, of type $A \to \mathsf{P}\;C$, defined *monadically* by

$$(g \bullet f)\,a = \mathbf{do}\;\{\,b \leftarrow f\;a;\,g\;b\,\}$$

Where does this terminology and notation come from?

It turns out that $\mathsf{P}\;X$ is a *monad* [44] in the category of sets, and monads induce particular categories known as *Kleisli* categories. The category of sets and relations "is" the Kleisli category induced by the monad P in the original category of sets and functions, where it is represented by P-valued functions composed as $g \bullet f$ above. Comparing such a definition of composition with that of arbitrary functions in the original category, namely

$$(g \cdot f)\,a = \mathbf{let}\;b = f\;a\;\mathbf{in}\;g\;b,$$

one immediately sees how the monadic **do** notation generalises the **let** notation used in ordinary mathematics and programming languages.

On the other hand, any function $f : A \to B$ can be represented in the Kleisli category by $f' = \mathbf{ret} \cdot f$, where $\mathbf{ret} : A \to \mathsf{P}\;A$ is the function that yields the smallest set that contains its argument, $\mathbf{ret}\;a = \{\,a\,\}$.[18]

Matrices and the vector space monad

In the same way that a relation can be faithfully represented by a set-valued function, any matrix can be represented by a vector-valued function. Each such vector corresponds to a column of the original matrix. For instance, the *Hadamard* gate

$$H = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{array} \tag{39}$$

is represented by the function

$$had :: \mathbb{B} \to Vec\;\mathbb{B}$$
$$had\;\mathsf{0} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$$
$$had\;\mathsf{1} = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}$$

That is to say, the[19] category of matrices is the Kleisli category induced by the monad Vec, where $Vec\;X$ denotes the type of all vectors with basis X. This means that for each $x \in X$ there is a column vector $\mathbf{ret}\;x$ such that $\mathbf{ret} : X \to Vec\;X$ represents the identity matrix $id : X \to X$.

In the quantum field, the "Dirac notation" $|x\rangle$ usually replaces $\mathbf{ret}\;x$, as in

$$|\mathsf{0}\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \qquad |\mathsf{1}\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

for $X = \mathbb{B}$. Using this notation, the Hadamard gate can be redefined as follows:

$$had :: \mathbb{B} \to Vec\;\mathbb{B}$$
$$had\;\mathsf{0} = \frac{|\mathsf{0}\rangle + |\mathsf{1}\rangle}{\sqrt{2}} \tag{40}$$
$$had\;\mathsf{1} = \frac{|\mathsf{0}\rangle - |\mathsf{1}\rangle}{\sqrt{2}}$$
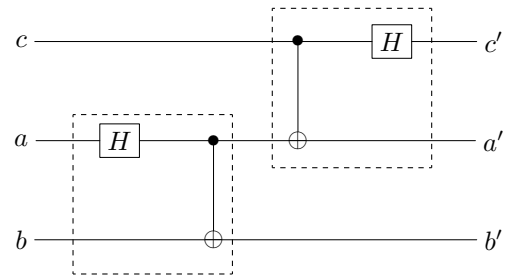
The inhabitants of type $Vec\;\mathbb{B}$ are usually known as *qubits*, generalising classical *bits* (Fig. 3). Bits are therefore special cases of qubits: $|\mathsf{0}\rangle$ and $|\mathsf{1}\rangle$ are classical, while e.g. $\frac{|\mathsf{0}\rangle + |\mathsf{1}\rangle}{\sqrt{2}}$ and $\frac{|\mathsf{0}\rangle - |\mathsf{1}\rangle}{\sqrt{2}}$ are *superpositions* of $|\mathsf{0}\rangle$ and $|\mathsf{1}\rangle$.

By Kleisli correspondence, all matrix-categorical operations can be encoded monadically, as for instance in the following definition of the Kronecker (tensor) product

$$(f \otimes g)\,(a, b) = \mathbf{do}\;\{$$
$$x \leftarrow f\;a;$$
$$y \leftarrow g\;b; \tag{41}$$
$$\mathbf{ret}\;(x, y)\,\}$$

where, for $f : A \to Vec\;X$ and $g : B \to Vec\;Y$, the function $f \otimes g$ is of type $(A \times B) \to Vec\;(X \times Y)$.

Let us see this representation at work by looking at the *structure* of a famous example in quantum computing – the "Alice" part of the teleportation protocol [17]:

---

18. For economy of space in the mathematical layout, **ret** abbreviates the more usual return keyword.

19. The use of definite article "the" arises from a simplification in the current paper: all matrices are regarded as complex number valued. In general, the abstract notion of a vector space (or, more generally, of a semimodule) is doubly parametric: both its basis and its underlying semiring can change [45].

Another important assumption is that every vector $v \in Vec\;X$ is supposed to have *finite* support, that is, the number of nonzero entries in v is finite.

	$([],0)$	$([],1)$	$([0],0)$	$([0],1)$	$([0,0],0)$	$([0,0],1)$	$([1,0],0)$	$([1,0],1)$	$([1],0)$	$([1],1)$	$([0,1],0)$	$([0,1],1)$	$([1,1],0)$	$([1,1],1)$
$([],0)$	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$([],1)$	0	1	0	0	0	0	0	0	0	0	0	0	0	0
$([0],0)$	0	0	$\frac{1}{\sqrt{2}}$	0	0	0	0	0	$\frac{1}{\sqrt{2}}$	0	0	0	0	0
$([0],1)$	0	0	0	$\frac{1}{\sqrt{2}}$	0	0	0	0	0	$\frac{1}{\sqrt{2}}$	0	0	0	0
$([0,0],0)$	0	0	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0
$([0,0],1)$	0	0	0	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$
$([1,0],0)$	0	0	0	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$
$([1,0],1)$	0	0	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0
$([1],0)$	0	0	0	$\frac{1}{\sqrt{2}}$	0	0	0	0	0	$-\frac{1}{\sqrt{2}}$	0	0	0	0
$([1],1)$	0	0	$\frac{1}{\sqrt{2}}$	0	0	0	0	0	$-\frac{1}{\sqrt{2}}$	0	0	0	0	0
$([0,1],0)$	0	0	0	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$
$([0,1],1)$	0	0	0	0	$\frac{1}{2}$	0	$\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	0	$-\frac{1}{2}$	0
$([1,1],0)$	0	0	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	0	$\frac{1}{2}$	0
$([1,1],1)$	0	0	0	0	0	$\frac{1}{2}$	0	$-\frac{1}{2}$	0	0	0	$-\frac{1}{2}$	0	$\frac{1}{2}$

Fig. 5. Matrix for $\langle\!| B |\!\rangle$ in the same range as Fig. 4.

The first block, marked by the dashed square covering inputs a and b, is the matrix $B = cnot \cdot (H \otimes id)$ – recall (37) – which creates a so-called Bell state. Let $bell = \Lambda B$. As $cnot$ is classical, we have to use $\mathbf{ret} \cdot cnot$ in the monadic encoding of B:

$$bell\,(a,b) = \mathbf{do}\,\{x \leftarrow had\ a; \mathbf{ret}\,(cnot\,(x,b))\} \qquad (42)$$

(Details in the appendix.) Then the second block, marked by the other dashed square, is $B^\dagger = (H \otimes id) \cdot cnot$, where $X^\dagger$ is the *conjugate* transpose of X.[20] This, using the same encoding rules, is represented by:

$$unbell\,(c,a) = \mathbf{let}\,(\_,a') = cnot\,(c,a)$$
$$\mathbf{in}\,\mathbf{do}\,\{b \leftarrow h\ c; \mathbf{ret}\,(b,a')\}$$

Then the two blocks are put together via the associator isomorphism a, recall (26):

$$A = (unbell \otimes id) \cdot \mathsf{a} \cdot (id \otimes bell)$$

Finally, $alice = \Lambda A$ becomes the monadic function:

$$alice\,(c,(a,b)) =$$
$$\mathbf{do}\,\{$$
$$(a',b') \leftarrow bell\,(a,b);$$
$$(c',a'') \leftarrow unbell\,(c,a');$$
$$\mathbf{ret}\,(c',(a'',b'))$$
$$\}$$

13 CONDITIONAL CONTROL

Figure 6 quotes a conditional flowchart expressed in the functional quantum programming language QFC [10]. Note how the conditional control involves a measurement, and thus happens at the *classical* level, with the subsequent branch being chosen depending on the (classical) outcome of said measurement.

It turns out that there is a different kind of conditional quantum control which does not require measuring the control qubit, and which provides a useful construct for

20. Recall that $X^\dagger$ coincides with X° wherever X does not involve imaginary parts.
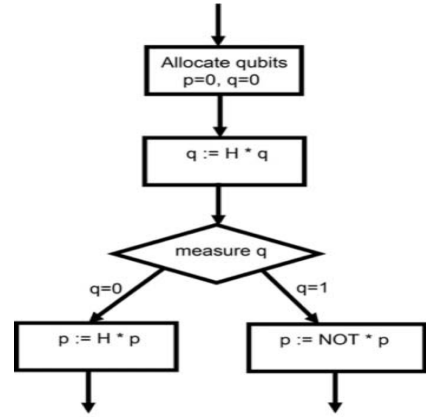
Fig. 6. Conditional flowchart in QFC, taken from Fig.7.4 of [18], page 236. H is the Hadamard gate.

quantum programming. The quantum programming language QML [12] was the first to support this kind of control. For instance, the following monadic program

$$cond\,(q,p) = \mathbf{do}\,\{$$
$$q' \leftarrow had\ q;$$
$$p' \leftarrow \mathbf{if}\ q'\ \mathbf{then}\ \mathbf{ret}\,(\neg\ p)\ \mathbf{else}\ had\ p; \qquad (43)$$
$$\mathbf{ret}\,(q',p')\}$$

– for had the Hadamard gate, recall (40) – encodes in Haskell an analogue of Figure 6 but using such a form of quantum conditional control. This piece of code implements the following unitary matrix:

	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$(0,0)$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{\sqrt{2}}{2}$	$-\frac{\sqrt{2}}{2}$
$(0,1)$	0	0	$\frac{\sqrt{2}}{2}$	$-\frac{\sqrt{2}}{2}$
$(1,0)$	$\frac{1}{2}$	$\frac{1}{2}$	$-\frac{1}{2}$	$-\frac{1}{2}$
$(1,1)$	$\frac{\sqrt{2}}{2}$	$-\frac{\sqrt{2}}{2}$	0	0

This suggests the following quantum conditional combinator which, rather than measuring the control bit, implements a superposition of conditionals:
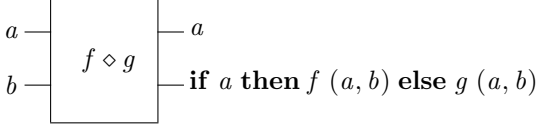
$$(f \diamond g)\,(x, y) = \mathbf{do}\ \{$$
$$\quad a \leftarrow x;$$
$$\quad b \leftarrow y;$$
$$\quad c \leftarrow \mathbf{if}\ a\ \mathbf{then}\ f\,(a, b)\ \mathbf{else}\ g\,(a, b);$$
$$\quad \mathbf{ret}\,(a, c)\,\}$$

The corresponding linear algebra expression is

$$f \diamond g = (id \otimes [f, g]) \cdot (fst\ ^{\triangledown} \gamma^{\circ}), \tag{44}$$

where the isomorphism γ from (18) plays a central role. In the form of a gate, this combinator looks like:

As an example of this quantum "choice" operator, note that

$$cnot = id \diamond \oplus \tag{45}$$

is the expected conditional version of case-based definition (13). The calculation of (45) is almost immediate:

$$cnot = id \diamond \oplus$$
$$\Leftrightarrow \qquad \{\ (12)\ \text{and}\ (44)\ ;\ \text{pairing absorption}\ \}$$
$$fst\ ^{\triangledown} \oplus = fst\ ^{\triangledown} ([id, \neg] \cdot \gamma^{\circ})$$
$$\Leftarrow \qquad \{\ \text{Leibniz}\ \}$$
$$\oplus = [id, \neg] \cdot \gamma^{\circ}$$
$$\Leftrightarrow \qquad \{\ \text{isomorphism}\ \gamma\ \}$$
$$\oplus \cdot \gamma = [id, \neg]$$
$$\Leftrightarrow \qquad \{\ (23)\ \}$$
$$true$$
$$\square$$

Quantum "choice" leads to a quantum extension of the (classical) McCarthy conditional functional combinator similar to the probabilistic one given in [20]:

$$p \to f, g = (f \diamond g) \cdot (p \otimes id)$$

The diagram below spells out the whole pipeline:

Back to our motivating example, the following quantum McCarthy conditional

$$H \to X, H$$

expresses in rather compact matrix notation the function *cond* given above, recall (43,2,39).

14 QUANTAMORPHISMS

We are now in a position to interpret diagram (33) in the category of matrices. This makes sense because initial algebras in the category of sets and functions lift to Kleisli

categories over it [46]. We obtain the following (recursive) definition of quantamorphisms as matrices:

$$\langle\!| Q |\!\rangle = \Psi\ Q \cdot (id \oplus id \otimes \langle\!| Q |\!\rangle) \cdot \alpha^{\circ}, \tag{46}$$

cf.

where the parameter matrix Q is of type $A \times B \to A \times B$ and

$$\Psi\ Q = \alpha \cdot (id \oplus \mathsf{xl} \cdot (id \otimes Q) \cdot \mathsf{xl}). \tag{47}$$

For this to be a quantum program there is a restriction, however: Q must be a *unitary transformation*. A $\mathbb{C}$-valued matrix M is unitary iff

$$M \cdot M^{\dagger} = M^{\dagger} \cdot M = id$$

holds. Comparing this with

$$f \cdot f^{\circ} = f^{\circ} \cdot f = id$$

we realise that *isomorphisms* are exactly the *classical* unitary transformations.

Recall that *Vec A* is the data type of all $\mathbb{C}$-valued vectors with base A and that $A \to Vec\ B$ is a function representing a *matrix* of type $A \to B$. So all linear algebra expressions can be encoded as *Vec*-valued functions and the *quantamorphism* diagram above becomes the following *Vec*-monadic program when rendered in the concrete syntax of Haskell:

```
⟨|·|⟩ :: ((a, b) → Vec (c, b)) → ([a], b) → Vec ([c], b)
⟨|f|⟩ ([], b) = ret ([], b)
⟨|f|⟩ (h : t, b) = do {
    (t', b') ← ⟨|f|⟩ (t, b);
    (h'', b'') ← f (h, b');
    ret (h'' : t', b'')
    }
```

The parameter $f = \Lambda Q$ must represent some unitary Q. Then $\langle\!| f |\!\rangle$ controls *qubit* b according to the *list* of bits passed as first parameter and the *quantum* operator Q. The outcome is unitary.

We can use the above monadic program to simulate quantum folds. For instance, suppose we use *bell* from (42) to control the input qubit. We can check what comes out using *GHCi* (here we are only showing the non-zero amplitudes):

$$x = \langle\!| bell |\!\rangle\,([0, 1, 1, 1], 0) =$$

$\mathbb{B}^* \times \mathbb{B}$	
$([0, 0, 0], 1)$	$\frac{1}{2\sqrt{2}}$
$([1, 0, 0], 0)$	$-\frac{1}{2\sqrt{2}}$
$([0, 1, 0], 0)$	$\frac{1}{2\sqrt{2}}$
$([1, 1, 0], 1)$	$-\frac{1}{2\sqrt{2}}$
$([0, 0, 1], 0)$	$\frac{1}{2\sqrt{2}}$
$([1, 0, 1], 1)$	$-\frac{1}{2\sqrt{2}}$
$([0, 1, 1], 1)$	$\frac{1}{2\sqrt{2}}$
$([1, 1, 1], 0)$	$-\frac{1}{2\sqrt{2}}$

$$\tag{48}$$

We can also superpose a quantamorphism with itself by passing quantum information to the control part itself, scaling up what happens at elementary gate level:

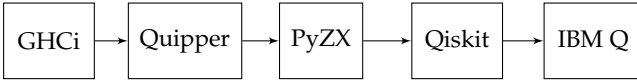$$y = \mathbf{do}\ \{\ i \leftarrow x; \langle\!| bell |\!\rangle\ i\ \}$$

Fig. 7. Tool-chain describing the quantamorphism compilation work-flow.

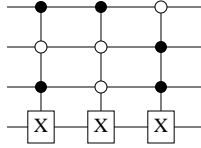Fig. 8. Circuit for $\langle\!\langle cnot \rangle\!\rangle$ restricted to three control qubits.

Fig. 9. Manual decomposition of the $\langle\!\langle cnot \rangle\!\rangle$ circuit of Fig. 8 [17].
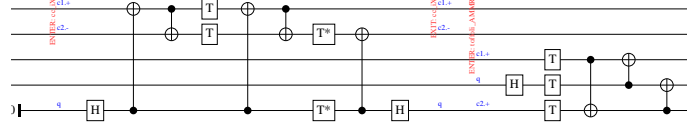
Fig. 10. First part of decomposed circuit for $\langle\!\langle cnot \rangle\!\rangle$ (Fig. 8).

Here x is the highly superposed stated calculated above (48). The outcome will be:

$$y = \begin{array}{c|c} \mathbb{B}^* \times \mathbb{B} & \\ \hline ([1,0,0],0) & \frac{1}{2} \\ ([1,0,0],1) & \frac{1}{2} \\ ([0,1,1],0) & -\frac{1}{2} \\ ([0,1,1],1) & \frac{1}{2} \end{array}$$

The next step is – instead of simulating – to "compile" quantamorphisms such as $\langle\!\langle bell \rangle\!\rangle$ so that they can run on a real quantum device. The process of compiling and running such quantum programs is described below.

15 IMPLEMENTATION

Recall that our main goal is to generate real (non-trivial) quantum programs and to run them on quantum hardware, namely on IBM Q Experience devices.

The current strategy consists in using the tool-chain depicted in Figure 7, which has five main steps:

- *GHCi* – depending on the resources (i.e. the number of qubits available), the monadic quantamorphisms are used to generate the finite, unitary matrices that describe the intended (recursive) quantum computations;
- *Quipper* [25] – this tool generates the *quantum circuit* from the unitary matrix;
- *PyZX* [47], [48] – this tool (based on the theory of the ZX-calculus) is used to optimise the quantum circuit issued by Quipper;
- *Qiskit [49]* – the quantum circuit generated by the previous steps is passed to this Python interface, which optimises circuits for the restrictions of a particular physical quantum processor and manages the executions of experiments on remote-access backends;
- *IBM Q* – this is the actual hardware where *Qiskit* runs the final code.

GHCi and Quipper

The practical implementation of a quantamorphism starts with the generation of the corresponding unitary matrix. The case study presented in this section is the quantamorphism over the control-not quantum gate accepting lists of
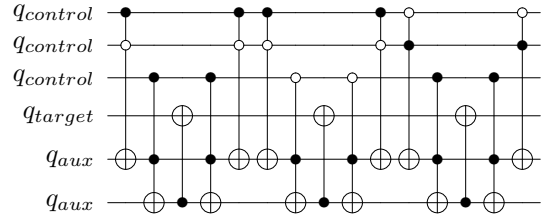
bits restricted by the number of control qubits (3 qubits). The corresponding 16x16 matrix given in Fig. 11 is the outcome of running the quantamorphism of section 14 in GHCi under such size restrictions.

Then another functional programming language, Quipper, is used to generate the corresponding quantum circuit, shown in Fig. 8, using Quipper's `exat_synthesis` functionality to synthesise the circuit from the matrix.

Although this circuit looks small and feasible, IBM Q Experience devices are unprepared to handle this kind of multi-qubit gates. As Quipper is not bound to any particular hardware, it allows the production of circuits like this, which require a decomposition stage, as explained next.

The manual decomposition of this circuit is easy, see Fig. 9. However, the outcome demands two ancillary qubits and Toffoli gates, which entail further decomposition. This is circumvented by using another Quipper function, `decompose_generic`, which generates a suitable (but longer) decomposition – see Fig. 10.

Qiskit, PyZX and IBM Q Experience

Qiskit is an open-source software for quantum computation. Using Qiskit is one of two ways to interact with the IBM Q Experience, which in turn is a cloud platform providing interaction with real quantum devices.

Since the manual translation of Quipper circuits into Qiskit syntax is error-prone, a tool – QuippertoQiskit – was developed to assist in this phase of the pipeline.[21]

The experiments were performed with version 0.14.1 of Qiskit and run in the ibmq_boeblingen device, version 1.0.6. Although a considerable number of IBM Q devices are available to the public, this specific system is exclusive to the IBM Q Network.

The selection of ibmq_boeblingen derives from its relatively high average decoherence times (77.888 μs/99.935μs) and a relatively low average of CX errors (0.0118). It is important to use devices with high decoherence times because this is the lifetime of the qubit state. After such time there are

---

21. This tool can translate every standard gate from Quipper to Qiskit syntax and is available in [50].

	$([\,],0)$	$([\,],1)$	$([0],0)$	$([0],1)$	$([0,0],0)$	$([0,0],1)$	$([1,0],0)$	$([1,0],1)$	$([1],0)$	$([1],1)$	$([0,1],0)$	$([0,1],1)$	$([1,1],0)$	$([1,1],1)$	$([0,0,0],0)$	$([0,0,0],1)$
$([\,],0)$	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$([\,],1)$	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$([0],0)$	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
$([0],1)$	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
$([0,0],0)$	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
$([0,0],1)$	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
$([1,0],0)$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
$([1,0],1)$	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
$([1],0)$	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
$([1],1)$	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
$([0,1],0)$	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
$([0,1],1)$	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
$([1,1],0)$	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
$([1,1],1)$	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
$([0,0,0],0)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
$([0,0,0],1)$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Fig. 11. Unitary matrix describing the semantics of quantamorphism $\langle\!\langle cnot \rangle\!\rangle$ restricted to the finite lists representable by 3 control qubits ($2^3 = 8$ lists, ranging from the empty list $[\,]$ to $[0,0,0]$).

little to no guarantees the state is the theoretically expected one. On the other hand, CX gates are the gates that create entangled states, where the probability of errors is higher.

Recall that the circuit of Fig. 8 required decomposition (Fig.10). Implementing this decomposition in Qiskit leads to 118 operations (gates) with depth 81. The size of the program is an issue because the longer a quantum program takes to run, the greater the chances are of qubits losing their state (quantum decoherence). Moreover, 51 of the gates are control gates, which cause an increase in error rate.

The size obstacle is inflated yet again when the circuit is compiled to the actual quantum device. Such a compilation results in a circuit with size 172 (with 125 control gates) and depth 132.

Luckily, there are some ways to optimise the transpiler process. The following step saw the circuit go through the four types of transpiler optimisation supplied by Qiskit and also be rewritten with PyZX.

The most straightforward optimisation of a circuit is its transformation locally with some known equations, e.g.[22] $Z \cdot Z = id, X \cdot X = id, H \cdot Z \cdot H = X, S \cdot H \cdot S \cdot H \cdot S = H$, etc. PyZX follows a different approach, which avoids having to deal with a large number of equations. In the first step, it converts the circuits into smaller sections, named *spiders*. The spiders compose a ZX-diagram, which is internally just a graph that can be optimised using the equations of the ZX calculus [48]. The outcome of this optimisation is not a circuit, but PyZX generates a new one from it. In other words, PyZX is a tool implementing the theory of ZX-calculus for automated rewriting of large-scale quantum circuits [47].

The circuit resulting from PyZX underwent Qiskit optimisations at levels 2 and 3 (in other words, optimisations that consider the errors of the selected backend). A summary of the results can be found in the following table:

	init	backend	opt2	opt3	zxB	zxo2	zxo3
Size	*118*	172	174	208	*46*	86	103
CX	*51*	125	125	**109**	*17*	60	**58**
Depth	*81*	132	**122**	139	*31*	**55**	64

The table shows the total number of gates, number of CX gates, and depth of each quantum circuit. Column 'init' refers to the initial circuit implemented in Qiskit;

22. Z denotes the Pauli-Z gate and S the Swap gate [17].

'backend' is the circuit that actually runs on the backend ibmq_boeblingen without optimisations; 'opt2' and 'opt3' are the circuits after the optimised transpiler levels 2 and 3, respectively; 'zxB' corresponds to the circuit that went through PyZX and the transpiler with no optimisations; finally the last two ('zxo2' and 'zxo3') went through PyZX and optimisation levels 2 and 3, respectively. The best results are highlighted in bold.

16 RESULTS ON THE IBM Q BOEBLINGEN

Qiskit comprises four modules: *Terra*, *Aer*, *Aqua*, and *Ignis*. *Terra* serves to create circuits, *Aer* allows various types of simulations, *Aqua* handles the quantum algorithms, and finally, the main function of *Ignis* is to study and mitigate quantum errors.

Since quantum errors are a serious problem of large quantum circuits, the *Ignis* module is essential, making it possible to find the average measurement fidelity of the qubits (0.796) and set a filter to mitigate errors in the results.

As expected, the results of the simulation agreed with the unitary matrix of Fig. 11. In particular, simulations of the initial and the PyZX circuits with all classical outcomes support the feasibility of compiling quantum programs with this method – see the table in Fig. 12.

The extensive depth of the circuits could already lead the reader to the conclusion that the results of execution in the real device are not as pleasing.

Figures 13 and 14 plot the outcome of experimenting with inputs $|0000\rangle$ and $|1011\rangle$, respectively, in the real device. In the first case, the expected outcome is $|0000\rangle$, since the target should not change. In the second case, the expected result is $|1010\rangle$. We focus our attention on the result of measuring the target qubit. Obtaining the expected results in the real device happens roughly in 50% of the measurements. However, manipulating the circuit to decrease its size, optimising considering errors of the device, and filtering the outcomes reveals a tendency towards the theoretical results.

17 CONCLUSIONS

Quantum programming is a new, promising paradigm for computing, and as such one that is receiving much attention and investment.

Initial and PyZX circuit		Typed input
input	output	
0000	0000	$([\,], 0)$
0001	0001	$([\,], 1)$
0010	0010	$([0], 0)$
0011	0011	$([0], 1)$
0100	0100	$([0,0], 0)$
0101	0101	$([0,0], 1)$
0110	**0111**	$([1,0], 0)$
0111	**0110**	$([1,0], 1)$
1000	**1001**	$([1], 0)$
1001	**1000**	$([1], 1)$
1010	**1011**	$([0,1], 0)$
1011	**1010**	$([0,1], 1)$
1100	1100	$([1,1], 0)$
1101	1101	$([1,1], 1)$
1110	1110	$([0,0,0], 0)$
1111	1111	$([0,0,0], 1)$

Fig. 12. Comparison between input and outputs of the initial and the PyZX circuits for simulations in Qiskit Aer. This aims to show that both circuits have the same behavior.
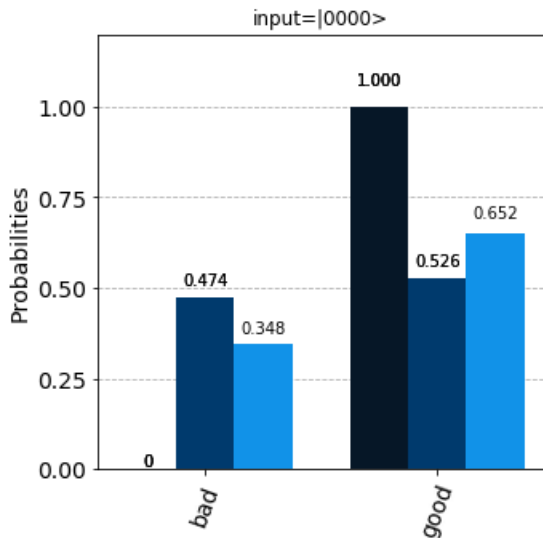

Fig. 14. Plot similar to that of Fig. 13 for input $([0,1], 1)$, i.e. $|1011\rangle$ at bit level.

Fig. 13. Qiskit plot of the probability of bad/good results issued by $\langle\!\langle cnot \rangle\!\rangle$ subject to input $([\,], 0)$, i.e. $|0000\rangle$ at bit level, running on ibmq_boeblingen. Good results are the ones where the measurement of the target qubit is zero, and bad results are the measurements where the target qubit is one. The darkest blue bars correspond to the simulation, i.e. the ideal 100% of obtaining the right output (on the good side), 0% on the bad side. The middle bars represent the results of running the circuit in the real device with no optimisations. Finally, the light blue bars display the performance using the PyZX compiler, optimisation 3 and mitigation.

QRAM[23] is a widely accepted model for quantum computing systems, consisting of a classical computer playing the role of the master and a quantum device accessed by the master on request [18]. This model regards measurements as an intrinsic part of a quantum computation, since this is the only way master and slave can communicate. Moreover, measurements are indeed part of well-known quantum strategies, for instance teleportation [17].

However, measurements add complexity to the mathematical models of quantum computing, arising from the
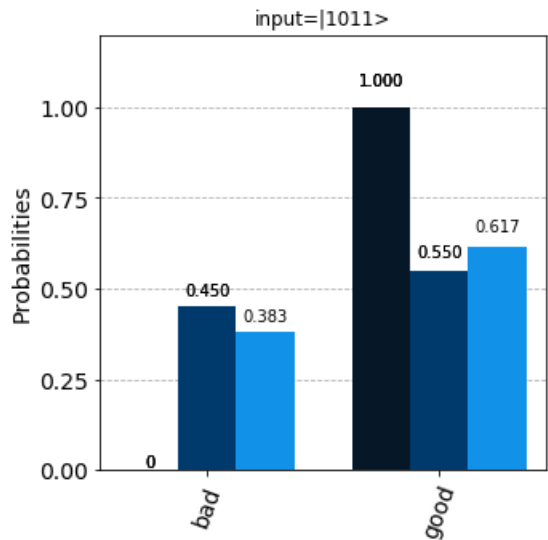
need to combine effects of two kinds: *quantum* effects explained by quantum physics and the *probabilistic* effect of reading quantum data.

It turns out that the classic master/quantum slave interplay is often thought of too atomically, perhaps influenced by the imperative programming principle that access to data always implies reading it from memory. While classical reading does not harm the data, reading quantum data spoils the quantum effect. From this perspective, measuring as little as possible is a good idea. This paper investigates measurement-free quantum computations.

Another level of complexity arises from the aim to extend arbitrary classical iteration and recursion to quantum programming. This has led to an extension of classical fixpoint theory to the new setting which ends up with mixed feelings about the possibility of ever truly realising quantum control [51].

The theory of classical recursion is well-structured around a taxonomy of recursive patterns, termed *morphisms* due to their inspiration in category theory [52]. One member of this *zoo*, called the *catamorphism*, has the property of always terminating, while encompassing a wide class of algorithms over inductive data structures such as lists, trees, and so on. Recursion in the current paper is bound to the catamorphism pattern.

By restricting itself to structural recursion without measurements, this paper achieves quantum control as in reference [53]:

> The novel aspect of quantum control that we are able to capture here is a notion of quantum loops. These loops were believed to be hard, if not impossible. What makes it work in our approach is the fact that we are firmly within a closed quantum system, without measurements. (...) As we restrict fixpoints to structural recursion, valid isos are regular enough to capture unitarity.

However, we achieve this aim in a substantially different way, in various respects:

23. QRAM stands for Quantum Random Access Memory Model [18].

- Reversibility: by generic calculation of reversible envelopes for non-reversible operations, based on minimal complements (Section 7). Classical gates such as controlled-not and the Toffoli gate arise in this way.
- Recursion: by generalising complementation to a class of catamorphisms called *folds*, which are recursive functions over finite lists[24] (Sections 9 and 10).
- Quantamorphisms: generalisation of reversible folds to unitary folds, enabling recursive quantum computing under structural quantum control (Sections 11 to 14).
- Implementation on IBM Q: proof-of-concept implementation of quantamorphisms on IBM Q Experience devices (Sections 15 to 16).[25]

Concerning the last step, running the generated circuits in real devices shows evidence of decoherence problems, albeit tending to the correct behaviour.

Real quantum devices are still in an initial stage and significant enhancements to the systems took place while doing the research reported in this paper. Some important functions in Qiskit were also altered, bugs were removed, and the whole system had a significant update. A comprehensive account of all the tests carried out on IBM Q devices can be found in reference [55]. Some of the circuits tested in [55] have been re-tested showing reduced error rates.

Such fast advances in a relatively short time increase confidence with respect to a follow-up to this work. Better results are expected by re-testing the work already reported, albeit possibly encountering other unforeseen limitations of quantum devices.

18 FUTURE WORK

Quantum control is still in its infancy. Tuned to the current paper, this observation calls for an extension of quantamorphisms to inductive types other than natural numbers and finite lists. Such a generalisation is a challenge for future work, possibly inspired by so-called *traversable* structures [56].

On the experimental side, the tool-chain used in our lab setup uses GHCi and Quipper. As both run Haskell programs, merging these two first blocks of the tool-chain seems viable and interesting to explore. Achieving this will require a thorough analyse of Quipper's recursive circuit implementation [57].

It is likely that implementing circuits with these methods will result in an initial circuit larger than the circuit implemented through a matrix. Similarly to classical reversible programs, quantum programs (reversible by definition) tend to add a substantial amount of garbage.

A clear challenge in the implementation process is to handle quantum errors. Compiling strategies able to curb this obstacle are under open debate. While IBM Q Experience developed three different kinds of optimisation levels, other academic researchers have developed tools like PyZX. Despite progress, both strategies were not enough to

achieve clean results. Therefore, future work should explore further, more elaborate compilation strategies. For example, the $t|ket\rangle$ compiler, which controls the routing problem aiming for hardware compatibility with minimal additional gate overhead [58], seems to show impressive results when tested against Qiskit optimisation level 1 and PyZX [59].

ACKNOWLEDGEMENTS

REFERENCES

[1] J. P. Dowling and G. J. Milburn, "Quantum technology: the second quantum revolution," *Philosophical Transactions of the Royal Society A*, vol. 361, no. 1809, 2003.

[2] A. de Touzalin, C. Marcus, F. Heijman, I. Cirac, R. Murray, and T. Calarco, "Quantum Manifesto: A new era of technology," 2016.

[3] A. M. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 2, no. 42, pp. 230–265, 1936. [Online]. Available: http://www.cs.helsinki.fi/u/gionis/cc05/OnComputableNumbers.pdf

[4] A. Church, "An unsolvable problem of elementary number theory," *American Journal of Mathematics*, vol. 58, no. 2, p. 345, 1936. [Online]. Available: http://www.jstor.org/stable/2371045?origin=crossref

[5] F. Nebeker, *Dawn of the Electronic Age: Electrical Technologies in the Shaping of the Modern World, 1914 to 1945*. Wiley-IEEE Press, 2009.

[6] S. K. Routray, "History of Electronics," *The 2004 IEEE Conference on the History of Electronics (CHE2004)*, 2004. [Online]. Available: http://www.ieeeghn.org/wiki/images/8/87/Routray.pdf

[7] D. Laws, "Who invented the transistor? — computer history museum," http://www.computerhistory.org/atchm/who-invented-the-transistor/, December 2013, (Accessed on 04/02/2019).

[8] D. Deutsch, "Quantum theory, the church–turing principle and the universal quantum computer," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 400, no. 1818, pp. 97–117, 1985. [Online]. Available: http://rspa.royalsocietypublishing.org/content/400/1818/97

[9] J. Sanders and P. Zuliani, "Quantum programming," in *MPC 2000*, ser. LNCS, R. Backhouse and J. Oliveira, Eds., vol. 1837. Springer, 2000, pp. 80–99. [Online]. Available: https://doi.org/10.1007/10722010_6

[10] P. Selinger, "Towards a quantum programming language," *Mathematical. Structures in Comp. Sci.*, vol. 14, no. 4, pp. 527–586, Aug. 2004. [Online]. Available: https://doi.org/10.1017/S0960129504004256

[11] M. Ying and Y. Feng, "A flowchart language for quantum programming," *IEEE Trans. Soft. Eng.*, vol. 37, no. 4, pp. 466–485, 2011.

[12] T. Altenkirch and J. Grattage, "A functional quantum programming language," in *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, 2005, pp. 249–258. [Online]. Available: https://doi.org/10.1109/LICS.2005.1

[13] J. Kramer, "Is abstraction the key to computing?" *Commun. ACM*, vol. 50, no. 4, pp. 37–42, April 2007.

[14] R. Backhouse, *Mathematics of Program Construction*. Univ. of Nottingham, 2004, draft of book in preparation. 608 pages.

[15] R. Bird and O. de Moor, *Algebra of Programming*, ser. Series in Computer Science. Prentice-Hall, 1997.

[16] S. MacLane, *Categories for the Working Mathematician*. Springer-Verlag, 1971.

---

24. Iterative *loops*, sometimes called *for*-loops, are also in this class, see e.g. [54].

25. For a detailed account of this experimentation see the webpage [50] or the master's dissertation [55] for older versions.

[17] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information: 10th Anniversary Edition*, 10th ed. New York, NY, USA: Cambridge University Press, 2011.

[18] N. Yanofsky and M. Mannucci, *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008.

[19] D. Murta and J. Oliveira, "A study of risk-aware program transformation," *SCP*, vol. 110, pp. 51–77, 2015.

[20] J. N. Oliveira, "Towards a linear algebra of programming," *Formal Asp. Comput.*, vol. 24, no. 4-6, pp. 433–458, 2012. [Online]. Available: https://doi.org/10.1007/s00165-012-0240-9

[21] M. Erwig and S. Kollmannsberger, "Functional pearls: Probabilistic functional programming in Haskell," *J. Funct. Program.*, vol. 16, pp. 21–34, January 2006.

[22] B. Coecke, Ed., *New Structures for Physics*, ser. Lecture Notes in Physics. Springer-Verlag, 2011, no. 831.

[23] J. Oliveira, "Compiling quantamorphisms for the IBM Q-experience," July 2018, talk at the IFIP WG 2.1 #77 Meeting, Brandenburg (Germany). Joint work with A. Neri and R.S. Barbosa.

[24] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi, "Bidirectionalization transformation based on automatic derivation of view complement functions," 2007, 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007), Freiburg, Germany, October 1-3.

[25] A. Green, P. Lumsdaine, N. Ross, P. Selinger, and B. Valiron, "An introduction to quantum programming in Quipper," *CoRR*, vol. cs.PL, no. arXiv:1304.5485v1, 2013.

[26] I. Hasuo and N. Hoshino, "Semantics of higher-order quantum computation via geometry of interaction," *CoRR*, vol. abs/1605.05079, 2016. [Online]. Available: http://arxiv.org/abs/1605.05079

[27] S. Abramsky, R. S. Barbosa, M. Karvonen, and S. Mansfield, "A comonadic view of simulation and quantum resources," in *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*. IEEE, 2019, pp. 1–12. [Online]. Available: https://doi.org/10.1109/LICS.2019.8785677

[28] R. Maddux, "The origin of relation algebras in the development and axiomatization of the calculus of relations," *Studia Logica*, vol. 50, pp. 421–455, 1991.

[29] A. Tarski and S. Givant, *A Formalization of Set Theory without Variables*. American Mathematical Society, 1987, aMS Colloquium Publications, volume 41, Providence, Rhode Island.

[30] J. Backus, "Can programming be liberated from the von Neumann style? a functional style and its algebra of programs," *CACM*, vol. 21, no. 8, pp. 613–639, August 1978.

[31] P. Freyd and A. Scedrov, *Categories, Allegories*, ser. Mathematical Library. North-Holland, 1990, vol. 39.

[32] C. Morgan, *Programming from Specification*, ser. Series in Computer Science. Prentice-Hall International, 1990, c.A.R. Hoare, series editor.

[33] J. Oliveira and M. Ferreira, "Alloy meets the algebra of programming: A case study," *IEEE Trans. Soft. Eng.*, vol. 39, no. 3, pp. 305–326, 2013.

[34] J. Oliveira, "Programming from metaphorisms," *JLAMP*, vol. 94, pp. 15–44, January 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2352220817301645

[35] G. Schmidt, *Relational Mathematics*, ser. Encyclopedia of Mathematics and its Applications. Cambridge University Press, November 2010, no. 132.

[36] J. Oliveira, "A relation-algebraic approach to the "Hoare logic" of functional dependencies," *JLAP*, vol. 83, no. 2, pp. 249–262, 2014.

[37] F. Bancilhon and N. Spyratos, "Update semantics of relational views," *ACM TDS*, vol. 6, no. 4, pp. 557–575, Dec. 1981.

[38] C. Olah, "Neural networks, types, and functional programming," 2015, blog: http://colah.github.io/posts/2015-09-NN-Types-FP/, last read: October 22, 2020.

[39] R. Hinze, "Adjoint folds and unfolds — an extended study," *Science of Computer Programming*, vol. 78, no. 11, pp. 2108–2159, 2013.

[40] P. Wadler, "Theorems for free!" in *4th International Symposium on Functional Programming Languages and Computer Architecture*. London: ACM, Sep. 1989, pp. 347–359.

[41] D. Moreda, "The next frontier: Quantum computing," 2018, slides presented at the PTH Conference, Porto, 18 May.

[42] B. Coecke and É. Paquette, "Categories for the practising physicist," in *New Structures for Physics*. Springer-Verlag, 2011, pp. 173–283, chapter nr. 3 of [22].

[43] H. Macedo and J. Oliveira, "Typing linear algebra: A biproduct-oriented approach," *SCP*, vol. 78, no. 11, pp. 2160–2191, 2013.

[44] J. Gibbons and R. Hinze, "Just do it: simple monadic equational reasoning," in *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ser. ICFP'11. New York, NY, USA: ACM, 2011, pp. 2–14.

[45] A. Santos, "Haskell library: laop-0.1.0.7," 2020, available from https://hackage.haskell.org/package/laop-0.1.0.7/docs/LAoP-Dist.html.

[46] I. Hasuo, B. Jacobs, and A. Sokolova, "Generic trace semantics via coinduction," *Logical Methods in Computer Science*, vol. 3, no. 4, pp. 1–36, 2007.

[47] A. Kissinger and J. van de Wetering, "Pyzx: Large scale automated diagrammatic reasoning," *arXiv preprint arXiv:1904.04735*, 2019.

[48] B. Coecke and R. Duncan, "Interacting quantum observables," in *ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings*, 2008, pp. 298–310. [Online]. Available: https://doi.org/10.1007/978-3-540-70583-3_25

[49] G. Aleksandrowicz, T. Alexander, P. Barkoutsos, L. Bello, Y. Ben-Haim, D. Bucher, F. J. Cabrera-Hernández, J. Carballo-Franquis, A. Chen, C.-F. Chen, J. M. Chow, A. D. Córcoles-Gonzales, A. J. Cross, A. Cross, J. Cruz-Benito, C. Culver, S. D. L. P. González, E. D. L. Torre, D. Ding, E. Dumitrescu, I. Duran, P. Eendebak, M. Everitt, I. F. Sertage, A. Frisch, A. Fuhrer, J. Gambetta, B. G. Gago, J. Gomez-Mosquera, D. Greenberg, I. Hamamura, V. Havlicek, J. Hellmers, Ł. Herok, H. Horii, S. Hu, T. Imamichi, T. Itoko, A. Javadi-Abhari, N. Kanazawa, A. Karazeev, K. Krsulich, P. Liu, Y. Luh, Y. Maeng, M. Marques, F. J. Martín-Fernández, D. T. McClure, D. McKay, S. Meesala, A. Mezzacapo, N. Moll, D. M. Rodríguez, G. Nannicini, P. Nation, P. Ollitrault, L. J. O'Riordan, H. Paik, J. Pérez, A. Phan, M. Pistoia, V. Prutyanov, M. Reuter, J. Rice, A. R. Davila, R. H. P. Rudy, M. Ryu, N. Sathaye, C. Schnabel, E. Schoute, K. Setia, Y. Shi, A. Silva, Y. Siraichi, S. Sivarajah, J. A. Smolin, M. Soeken, H. Takahashi, I. Tavernelli, C. Taylor, P. Taylour, K. Trabing, M. Treinish, W. Turner, D. Vogt-Lee, C. Vuillot, J. A. Wildstrom, J. Wilson, E. Winston, C. Wood, S. Wood, S. Wörner, I. Y. Akhalwaya, and C. Zoufal, "Qiskit: An open-source framework for quantum computing," 2019.

[50] A. Neri, J. N. Oliveira, and R. S. Barbosa, "Quantamorphisms guide," Mar 2020, [Online; accessed 9. Mar. 2020]. [Online]. Available: https://quantamorphisms.netlify.com

[51] C. Bădescu and P. Panangaden, "Quantum alternation: Prospects and problems," *Electronic Proceedings in Theoretical Computer Science*, vol. 195, p. 33–42, Nov 2015. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.195.3

[52] R. Hinze, N. Wu, and J. Gibbons, "Conjugate hylomorphisms – or: The mother of all structured recursion schemes," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15. New York, NY, USA: ACM, 2015, pp. 527–538. [Online]. Available: http://doi.acm.org/10.1145/2676726.2676989

[53] A. Sabry, B. Valiron, and J. K. Vizzotto, "From symmetric pattern-matching to quantum control (extended version)," *CoRR*, vol. abs/1804.00952, 2018. [Online]. Available: http://arxiv.org/abs/1804.00952

[54] J. Oliveira, "Towards quantamorphisms — some thoughts on (constructive) reversibility," 2018, iCT Cost Action 1405 Meeting, Cyprus, 19-21 March, 2018.

[55] A. Neri, "Towards quantum program calculation," Master's thesis, University of Minho, Informatics Department, October 2018.

[56] M. Jaskelioff and O. Rypacek, "An Investigation of the Laws of Traversals," *Electronic Proceedings in Theoretical Computer Science*, vol. 76, no. Msfp, pp. 40–49, 2012.

[57] R. Eisenberg, A. S. Green, P. L. Lumsdaine, K. K. (ACS), S. M. (ACS), B. Mohan, W. N. (ACS), J. Ravelomanantsoa-Ratsimihah, N. J. Ross, A. S. (ACS), P. Selinger, B. Valiron, A. V. (ACS), and S. A. Zdancewic, "The quipper language," https://www.mathstat.dal.ca/~selinger/quipper/, 2016, (Accessed on 08/09/2018).

[58] A. Cowtan, S. Dilkes, R. Duncan, A. Krajenbrink, W. Simmons, and S. Sivarajah, "On the qubit routing problem," *arXiv*, Feb 2019.

[59] A. Cowtan, S. Dilkes, R. Duncan, W. Simmons, and S. Sivarajah, "Phase Gadget Synthesis for Shallow Circuits," *arXiv*, Jun 2019. [Online]. Available: https://arxiv.org/abs/1906.01734

APPENDIX

Checking g (31)

Recall $g\ (a,(x,b)) = (a,(x,f\ (a,b)))$ in:

$\quad$ $\mathsf{a}^\circ\ ((id \times fst)\ {}_\triangledown\ (f \cdot (id \times snd))\ (a,(x,b))$

$=\quad$ { composition; fst and snd projections }

$\quad$ $\mathsf{a}^\circ\ ((a,x),f\ (a,b))$

$=\quad$ { associate to the righ isomorphism a° }

$\quad$ $(a,(x,f\ (a,b)))$

$\quad\square$

Calculation of (32)

Let $\beta = (id \times fst)\ {}_\triangledown\ (id \times snd)$. Then $\mathsf{xl} = (snd \times id) \cdot \beta$ and

$$\mathsf{a}^\circ \cdot (id \times snd) = \mathsf{xl} \cdot (snd \times id) \qquad (49)$$

holds. Then:

$\quad$ $\Psi\ x$

$=\quad$ { starting definion of $\Psi\ x$ }

$\quad$ $\mathsf{a}^\circ \cdot ((id \times fst)\ {}_\triangledown\ snd \cdot x \cdot (id \times snd))$

$=\quad$ { factor β to the right }

$\quad$ $\mathsf{a}^\circ \cdot (id \times snd \cdot x) \cdot \beta$

$=\quad$ { (49) ; product functor }

$\quad$ $\mathsf{xl} \cdot (id \times x) \cdot (snd \times id) \cdot \beta$

$=\quad$ { $\mathsf{xl} = (id \times x) \cdot \beta$ }

$\quad$ $\mathsf{xl} \cdot (id \times x) \cdot \mathsf{xl}$

$\quad\square$

Ψ (32) *preserves injectivity*

For injective x the kernel of $\mathsf{xl} \cdot (id \times x) \cdot \mathsf{xl}$ is $\mathsf{xl}^\circ \cdot (id \times \ker x) \cdot \mathsf{xl} = \mathsf{xl}^\circ \cdot \mathsf{xl} = id$ since kernel distributes by products. Then $\ker\ (\Psi\ x) = id$ since kernels also distributes by coproducts.

⦅_⦆ *preserves injectivity*

Let $k = ⦅f⦆$. By the UP (28), $k = f \cdot (\mathsf{F}\ k) \cdot \alpha^\circ$. We calculate $K = \ker k$ assuming $\ker f = id$:

$\quad$ $K = k^\circ \cdot k$

$\Leftrightarrow\quad$ { unfold $f \cdot \mathsf{F}\ k \cdot \alpha^\circ$ }

$\quad$ $K = \alpha \cdot \mathsf{F}\ k^\circ \cdot f^\circ \cdot f \cdot \mathsf{F}\ k \cdot \alpha^\circ$

$\Leftrightarrow\quad$ { assumption: $f^\circ \cdot f = id$ }

$\quad$ $K = \alpha \cdot \mathsf{F}\ k^\circ \cdot \mathsf{F}\ k \cdot \alpha^\circ$

$\Leftrightarrow\quad$ { $\mathsf{F}\ (R \cdot S) = (\mathsf{F}\ R) \cdot (\mathsf{F}\ S)$ and $\mathsf{F}\ R^\circ = (\mathsf{F}\ R)^\circ$ }

$\quad$ $K = \alpha \cdot \mathsf{F}\ (k^\circ \cdot k) \cdot \alpha^\circ$

$\Leftrightarrow\quad$ { $K = k^\circ \cdot k$; UP (for relations) }

$\quad$ $K = ⦅\alpha⦆$

$\Leftrightarrow\quad$ { Reflexion: $⦅\alpha⦆ = id$ }

$\quad$ $K = id$

Do-notation calculus

$$\begin{aligned}\mathbf{do}\ \{x \leftarrow \mathbf{ret}\ a; f\ x\} &= \\ \mathbf{do}\ \{x \leftarrow f\ a; \mathbf{ret}\ x\} &= \quad f\ a\end{aligned} \qquad (50)$$

$$\begin{aligned}\mathbf{do}\ \{x \leftarrow f\ a; \mathbf{do}\ \{y \leftarrow g\ x; h\ y\}\} &= \\ \mathbf{do}\ \{x \leftarrow f\ a; y \leftarrow g\ x; h\ y\}\end{aligned} \qquad (51)$$

$$\begin{aligned}\mathbf{do}\ \{y \leftarrow \mathbf{do}\ \{x \leftarrow f\ a; g\ x\}; h\ y\} &= \\ \mathbf{do}\ \{x \leftarrow f\ a; y \leftarrow g\ x; h\ y\}\end{aligned} \qquad (52)$$

(For a comprehensive account of the **do**-notation calculus please see [44].)

Details of the calculation of (42)

First, the term $H \otimes id$:

$\quad$ $f = \Lambda(H \otimes id)$

$\Leftrightarrow\quad$ { Kleisli correspondence }

$\quad$ $f = had \otimes \mathbf{ret}$

$\Leftrightarrow\quad$ { (41) }

$\quad$ $f\ (a,b) = \mathbf{do}\ \{x \leftarrow had\ a; y \leftarrow \mathbf{ret}\ b; \mathbf{ret}\ (x,y)\}$

$\Leftrightarrow\quad$ { (50) }

$\quad$ $f\ (a,b) = \mathbf{do}\ \{x \leftarrow had\ a; \mathbf{ret}\ (x,b)\}$

$\quad\square$

Then

$\quad$ $g = \Lambda(cnot \cdot (H \otimes id)) = (\mathbf{ret} \cdot cnot) \bullet f$,

that is:

$\quad$ $g\ (a,b) = \mathbf{do}\ \{(y,z) \leftarrow f\ (a,b); (\mathbf{ret} \cdot cnot)\ (y,z)\}$

$\Leftrightarrow\quad$ { inline f calculated above }

$\quad$ $g\ (a,b) = \mathbf{do}\ \{$
$\quad\quad (y,z) \leftarrow \mathbf{do}\ \{x \leftarrow had\ a; \mathbf{ret}\ (x,b)\};$
$\quad\quad \mathbf{ret}\ (cnot\ (y,z))$
$\quad\quad \}$

$\Leftrightarrow\quad$ { (52) }

$\quad$ $g\ (a,b) = \mathbf{do}\ \{$
$\quad\quad x \leftarrow had\ a; (y,z) \leftarrow \mathbf{ret}\ (x,b);$
$\quad\quad \mathbf{ret}\ (cnot\ (y,z))\}$

$\Leftrightarrow\quad$ { (50,51,52) }

$\quad$ $g\ (a,b) = \mathbf{do}\ \{x \leftarrow had\ a; \mathbf{ret}\ (cnot\ (x,b))\}$

$\quad\square$