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 a category-theoretic approach to quantum information, 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 through the IBM Q Experience. Reversibility is ensured by minimal complements. Such complementation is extended inductively to encompass catamorphisms on lists (vulgo folds), giving rise to the corresponding recursion scheme in reversible computation. The same idea is then applied to the setting of quantum programming, where computation is expressed by unitary transformations. This yields the notion of ‘quantamorphism’, a structural form of quantum recursion implementing cycles and folds on lists with quantum control flow. 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 and 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 naturally incurs a significant amount of errors. As quantum technology is rapidly evolving, an increase in reliability is likely in the 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, which involves companies as well as academic and research institutions 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 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 yet over: the problem with software engineering is that quality control is based on testing software artefacts 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].

Quantum programming requires revising and updating these abstractions. Despite its strong algebraic basis – cf. Hilbert spaces, linear algebra, etc. [17] – quantum mechanics is a counter-intuitive theory. Thus, programmers stand to gain even more from the structured reasoning provided by the MPC discipline. Moreover, in quantum mechanics, observations affect the state of a system. Thus, inspecting the state of a variable may interfere with the course of a quantum computation, rendering the usual method of naive step-by-step debugging impractical. More sophisticated alternatives, based for example on quantum tomography, have been proposed as ways to perform break-point debugging [18], but these seem hard to scale to long, branching computations. All the more reason to 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 software 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 [19], 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 [20], 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 in order to reason about probabilistic functions (Markov chains). On the experimental side, this requires programming over the distribution monad.

The recursive programming construction studied in [20] 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 [23]. The corresponding extension of the catamorphism concept, to be developed further in this paper, is termed “quantamorphism” – a restricted (structural) form of quantum control flow of quantum data [24].

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 [25] to recursive programs. The background of all this research is also enhanced by studies in quantum functional programming (QFP) [12], [26] and extensive research in categorical quantum physics [27], [28].

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. 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).

1. This has been referred to by the acronym LAoP (“linear algebra of programming”) [21].
2. As implemented by [22] in Haskell.
3. See e.g. [29] for a comprehensive historical overview.
Alfred Tarski (1901–83), who had a life-long struggle with quantifier notation, revived relation algebra. Together with Steven Givant he wrote a book (published posthumously) on set theory without variables [30].

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 [31]. In any case, Backus’s landmark 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 [32] 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 [32].

4 REVERSIBILITY

Standard program design relies on program refinement techniques [33], [34]. 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

\[
fx = fx' \Rightarrow x = x'
\]

holds for every \( x, x' \in X \), and it is surjective iff for all \( y \in Y \) there exists some \( x \) such that \( y = fx \).

Refinement is normally expressed in terms of a preorder \( p \leq q \) 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 \( f \), written \( g \leq f \), if

\[
fx = fx' \Rightarrow gx = gx'
\]

holds for all \( x, x' \) in the domain.

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

\[
g \subseteq f^\text{c}
\]

using the relation algebra converse operator: \( b = fa \) means the same as \( a \overset{\text{c}}{=} b \). This says that \( g \) is injective because \( g \) is smaller than the converse of a function \( f^\text{c} \), and function converses are always injective. In case \( g = f^\text{c} \) then both \( f \) and \( g \) are 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 \( A \overset{R}{\to} B \) or \( B \overset{R}{\to} A \), 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 \( b \leq a \), \( b \in s \), and so on.

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

Given relations \( B \overset{R}{\to} C \) and \( C \overset{S}{\to} A \), their composition \( B \overset{R \cdot S}{\to} A \) 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 \( id x = x \), and so on.

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 \( A \overset{R^\text{c}}{\to} B \) of a relation \( A \overset{R}{\to} B \) is such that \( a \overset{R^\text{c}}{=} b \) means the same as \( b R a \). In the case of a function \( f : A \to B \), its converse is the relation \( f^\text{c} : A \to B \) such that \( a \overset{f^\text{c}}{=} b \iff f(a) \).

A Taxonomy of Relations

A relation \( R : A \to B \) is said to be injective whenever \( R^\text{c} : R \subseteq id \) holds.\(^5\) Moreover,

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

5. For functions \((R := f)\), \( f^\text{c} \cdot f \subseteq id \) means precisely what we had before, \( f x = f x' \Rightarrow x = x' \).
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\(^6\)

\[
R \circ S \leq X \iff R \leq X \land S \leq X
\]

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

\[
(b, c) (R \circ S) a \iff b R a \land c S a
\]

specialising, in the case of functions, to

\[
f \circ g = (f, g, a).
\]

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

\[
R \leq R \circ S \text{ and } S \leq R \circ S.
\]

Facts (7) are jointly equivalent to \(\ker (R \circ S) \subseteq (\ker R) \cap (\ker S)\), which in fact is an equality

\[
\ker (R \circ S) = (\ker R) \cap (\ker S).
\]

This is a corollary of the following more general law.\(^7\)

\[
(R \circ S) \circ (Q \circ P) = (R \circ Q) \cap (S \circ P).
\]

Injectivity *shunting laws* also arise by standard relational algebra calculation\(^8\), for instance

\[
R \cdot g \leq S \iff R \leq S \cdot g^\circ
\]

Restricted to functions, the preorder \(\leq\) is universally bounded by

\[
! \leq f \leq id
\]

where \(\overset{!}{-}\) is the unique function of its type, where symbol \(1\) denotes the singleton type. Moreover,

- A function is injective iff \(id \leq f\) holds, that is, \(\ker f = id\). Consequently, \(f \circ id\) is always injective, by (7).
- Two functions \(f\) and \(g\) are said to be complementary whenever \(id \leq f \circ g\).\(^8\)

For instance, the projections \(fst (a, b) = a\) and \(snd (a, b) = b\) are complementary since \(fst \circ snd = id\).

## 7 MINIMAL COMPLEMENTS

Given some \(f\), suppose that: (a) \(id \leq f \circ g\) for some \(g\); and (b) if \(id \leq f \circ h\) for some \(h\) such that \(h \leq g\), then \(g \leq h\). Then \(g\) is said to be a minimal complement of \(f\)\(^8\). 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) \(f \circ g\) is injective even if \(f\) is not, and (b) \(f\) is implemented by \(f \circ g\), since \(f = fst \circ (f \circ g)\).

---

\(^6\) See e.g. [37] for more details.

\(^7\) Details in [15].

\(^8\) Cf. [25]. Other terminologies are monic pair [32] or jointly monic [15].
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

\[
\mathbb{B} \times \mathbb{B} \overset{\oplus}{\longrightarrow} \mathbb{B} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}
\] (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 \( \text{id} \). The identity function itself would do this,

\[
\ker \text{id} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\]

but this would be an overkill – \( \text{id} \) complements any function! Moreover, it is not minimal in this case. To reduce
injectivity we need to start adding 1s to \( \ker \text{id} \) where \( \ker \oplus \) has 0s, e.g.,

\[
\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}
\]

(11)

However, this is not 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
relation iff it is a difunctional relation, where

a relation \( R \) is difunctional iff \( R \cdot R^\circ \cdot R \subseteq R \) \cite{footnote}.

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

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

\[
\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}
\]

What we obtain is \( \ker \text{fst} \), the kernel of the first projection

\[
\mathbb{B} \overset{\text{fst}}{\longrightarrow} \mathbb{B} \times \mathbb{B} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}.
\]

So, function \( \text{fst} (a, b) = a \) 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.,

\[
\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}
\]

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

Let us see what comes out of the \( \text{fst} \)-complementation of exclusive-or

\[
\mathbb{B} \times \mathbb{B} \overset{\text{fst} \lor \oplus}{\longrightarrow} \mathbb{B} \times \mathbb{B} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}
\]

(12)

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

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{a'} \\
\text{b'}
\end{array}
\]

Why does it bear this name? We calculate

\[
\text{cnot} = \text{fst} \lor \oplus
\]

\[
\begin{array}{c}
\text{cnot} (a, b) = (a, a \oplus b) \\
\text{cnot} (0, b) = (0, b) \\
\text{cnot} (1, b) = (1, \neg b)
\end{array}
\]

Informally: \text{controlled} bit \( b \) is negated \text{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 \( \text{fst} \) complement in copying the control bit to
the output.

**Other \( \text{fst} \)-Complementations**

As a second example, we take the classical circuit

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c}
\end{array} \quad \begin{array}{c}
\text{a'} \\
\text{z}
\end{array}
\]

Can this be made into a bijection in the same way? The function
implemented is

\[
\mathbb{B}^2 \times \mathbb{B} \overset{f = \oplus \circ (\land \lor \text{id})}{\longrightarrow} \mathbb{B} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}
\]

where

\[
(f \land g) (a, b) = (f \land g) (a, b)
\]

(14)

is the "tensor" product of two functions. Let us complement \( f \) with \( \text{fst} \) again, which in this context has type \( \mathbb{B}^2 \times \mathbb{B} \overset{\text{fst}}{\longrightarrow} \mathbb{B}^2 \).

The outcome is another bijection, known as the \( \text{ccNOT} \), or Toffoli, gate.
ccnot = \text{fst} \cdot (\oplus \cdot (\wedge \times \text{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 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\end{bmatrix}

usually depicted as follows:

As for \text{cnot}, a similar calculation will lead to the pointwise version

\begin{align*}
\text{ccnot} : \mathbb{B}^2 \times \mathbb{B} & \rightarrow \mathbb{B}^2 \times \mathbb{B} \\
\text{ccnot} ((1, 1), c) &= ((1, 1), \neg c) \\
\text{ccnot} ((a, b), c) &= ((a, b), c)
\end{align*}

As a last example of \text{fst}-complementation, let us see a famous device in quantum programming arising from the following generic evolution of the \text{cnot} gate, parametric on \(A \rightarrow B\) and such that \((B; \theta, 0)\) is a monoid satisfying \(x \cdot \theta \cdot x = 0\) for all \(x \in B\):

\[ U f : (A \rightarrow B) \rightarrow (A \times B) \rightarrow (A \times B) \]

\[ U f = \text{fst} \cdot \theta \cdot (f \times \text{id}), \]

that is,

\[
\begin{array}{c}
x \\
y
\end{array}
\rightarrow
U f
\rightarrow
(f x) \theta y
\]

Clearly, for \(\theta = \oplus:\)

\[ \text{cnot} = U \text{id} \]

\[ \text{ccnot} = U (\wedge) \]

It is easy to see that, for every \(f\), \(U f\) is a \textit{bijection} because it is its own inverse:

\[
U f \cdot U f = \text{id}
\]

\[
\Leftrightarrow \quad \{ \text{Unf} (x, y) = (x, (f x) \theta y) \}
\]

\[
\text{Unf} (x, (f x) \theta y) = (x, y)
\]

\[
\Leftrightarrow \quad \{ \text{again Unf} (x, y) = (x, (f x) \theta y) \}
\]

\[
(x, (f x) \theta ((f x) \theta y)) = (x, y)
\]

\[
\Leftrightarrow \quad \{ \theta \text{ is associative and } x \theta x = 0 \}
\]

\[
(x, 0 \theta y) = (x, y)
\]

\[
\Leftrightarrow \quad \{ 0 \theta x = x \}
\]

\[
(x, y) = (x, y)
\]

\[ U f \text{ is therefore a \textit{reversible refinement} of an arbitrary } f : A \rightarrow B \] (for a monoid \(B\) as above) in the sense that\(^{10}\)

\[ \text{snd} \cdot U f \cdot (\text{id} \times 0) = f, \]

or in pointwise notation,

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

### 8 The Dual View

Before moving on and generalising \text{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 \textit{disjoint union} of \(A\) and \(B\):

\[ A + B = \{ i_1 x \mid x \in A \} \cup \{ i_2 y \mid y \in B \} \]

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

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

\[ X = [R, S] \Leftrightarrow \{ X \cdot i_1 = R \}
\]

The \textit{direct sum} of two relations arises immediately from:

\[ R + S = [i_1 \cdot R, i_2 \cdot S] \]

The isomorphism

\[ \mathbb{B} \times A \overset{\gamma}{\leftarrow} A + A = [\text{false} \cdot \text{id}, \text{true} \cdot \text{id}] \]

holds (where \texttt{false} and \texttt{true} are the obvious constant functions) and can be re-written into

\[ \gamma = [\text{false}, \text{true}] \cdot \text{id} \]

thanks to the so-called \textit{exchange law}:

\[ [R \cdot S, T \cdot V] = [R, T] \cdot [S, V]. \]

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

\[ \mathbb{B} + \mathbb{B} \overset{(\wedge)}{\rightarrow} \mathbb{B} = [\text{false}, \text{id}], \]

disjunction

\[ \mathbb{B} + \mathbb{B} \overset{(\lor)}{\rightarrow} \mathbb{B} = [\text{true}, \neg], \]

exclusive-or

\[ \mathbb{B} + \mathbb{B} \overset{\oplus}{\rightarrow} \mathbb{B} = [\text{id}, \neg], \]

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):

\[
\oplus = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}.
\]

Applying the same transformation to \( \text{fst} \)-complemented operations yields similarly expressive denotations of Boolean gates. For instance, \( \text{cnot} = \text{fst} \cdot \oplus \) is transformed into \( \text{id} + \neg \) through \( \gamma \):

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

That is, \( \gamma \) has the relational type

\[
\text{fst} \cdot \oplus \cdot \gamma \sim \text{id} + \neg.
\]

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

\[
\text{id} + (\text{id} + \neg),
\]

again an isomorphism by construction.

The isomorphism \( \gamma \) will play an important role in implementing the quantum conditional in Section 13. The coproduct construct is also inherently present in the strategy that underlies Section 14.

9 \ GENERALISING \( \text{fst} \)-COMPLEMENTATION

As seen in Section 7, the projection \( A \times B \rightarrow \text{fst} \) \( A \) plays a role in injectivity refinements, working as minimal complement in several situations. In general, \( \text{fst} \)-complementation

\[
\text{id} \leq \text{fst} \cdot f
\]

works whenever

\[
f (a, b) = f (a, b') \Rightarrow b = b'
\]

holds, that is:\( ^\text{12} \)

\[
\exists a b = \exists a b' \Rightarrow b = b'.
\]

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

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

\[
\begin{array}{c}
A \rightarrow A \\
B \rightarrow B
\end{array}
\]

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

\[
\begin{array}{c}
x \rightarrow x \\
y_0 \rightarrow f \\
y_1 \\
\end{array}
\]

and draw each of them in a different way,

\[
\begin{array}{c}
\quad x \\
y_0 \rightarrow f \\
y_1 \\
\end{array}
\]

so that they can be chained as depicted below:

\[
\begin{array}{c}
x \\
y_0 \\
y_1 \\
y_2 \\
\end{array}
\]

Clearly, \( [x_0, x_1, x_2, \ldots] \) 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 \( x \).

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

10 \ TOWARDS (CONSTRUCTIVE) RECURSIVE COMPLEMENTATION

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

Now suppose that \( k \) is a recursive function over finite lists, for instance \( k = \text{foldr} \mathcal{F} b \) for \( f : A \times B \rightarrow B \), that is,

\[
k : [A^\ast] \rightarrow B
\]

\[
k [\cdot] = b
\]

\[
k (a : x) = f (a, k x).
\]

How do we “constructively” build the corresponding (recursive, bijective) envelope of type \( A^\ast \times B \rightarrow A^\ast \times B \). Let us define \( \langle f \rangle \) such that

\[
\langle f \rangle (x, b) = \text{foldr} \mathcal{F} b x,
\]

\( ^\text{11} \) In general, \( f \) is said to have relational type \( S \rightarrow R \) whenever \( f : R \subseteq S \cdot f \) holds.

\( ^\text{12} \) We abbreviate \textit{curry} \( f \) by \( \mathcal{F} \), that is: \( \mathcal{F} a b = \text{curry} f a b = f (a, b) \).

\( ^\text{13} \) Cf. e.g. the function \textit{mapAccumR} in the Haskell language.
that is,
\[
\langle f \rangle ([|], b) = b \\
\langle f \rangle (a : x, b) = f (a, \langle f \rangle (x, b)).
\]

We can depict \( \langle f \rangle \) in the form of a commutative diagram:

\[
\begin{array}{ccc}
A^* \times B & \xrightarrow{\alpha} & B + A \times (A^* \times B) \\
\langle f \rangle \downarrow & & \downarrow \text{id+id}\times\langle f \rangle \\
B & \xleftarrow{[\text{id}, f]} & B + A \times B
\end{array}
\]

In the diagram, the isomorphism

\[
A^* \times B \xrightarrow{\alpha} B + A \times (A^* \times B)
\]

is defined by

\[
\alpha = [\text{nil} \cdot \text{id}, (\text{cons} \times \text{id}) \cdot a]
\]

where

\[
\text{nil} = [|] \\
\text{cons} (a, x) = a : x
\]

are the components of the initial algebra \( \text{in} = [\text{nil}, \text{cons}] \) of finite lists [15], and the isomorphism

\[
\begin{align*}
a : A \times (B \times C) & \to (A \times B) \times C \\
a & = \text{id} \times \text{fst} \circ (\text{snd} \times \text{snd})
\end{align*}
\]

in (25) is the associator.

Below, we will need something more general, namely:

\[
\begin{array}{ccc}
A^* \times B & \xrightarrow{\alpha} B + A \times (A^* \times B) \\
\langle h \rangle \downarrow & & \downarrow \text{id+id}\times\langle h \rangle \\
C & \xleftarrow{h} & B + A \times C
\end{array}
\]

This is specified by the universal property

\[
k = \langle h \rangle \iff k \cdot \alpha = h \cdot F\ k
\]

where \( F\ f = \text{id}+\text{id} \times f \) captures the list-recursion pattern. This \( \langle h \rangle \) can be regarded as an extension of the well-known \textit{catamorphism} combinator.\(^{14}\) All the standard laws apply, including reflection \( \langle \alpha \rangle = \text{id} \) and the loop-intercombination law

\[
\langle f \rangle \circ \langle g \rangle = \langle (f \times g) \cdot (F\ \text{fst} \times F\ \text{snd}) \rangle,
\]

often referred to as the “\textit{banana-split}” law [15]. From (28) and (26), one also infers

\[
A^* \xrightarrow{\text{fst}} A^* \times B = \langle \text{in} \rangle
\]

by easy calculation:

\[
\begin{align*}
\text{fst} \circ \langle \text{in} \rangle & \iff \{ (28) \} \\
\text{fst} \cdot \alpha & = \text{in} \cdot (\text{id}+\text{id} \times \text{fst}) \\
& \iff \{ \text{in} = [\text{nil}, \text{cons}] ; \text{coproducts} \} \\
\text{fst} \cdot \alpha & = [\text{nil}, \text{cons} \cdot (\text{id} \times \text{fst})] \\
& \iff \{ \text{definition of } \alpha \text{ (25) and } a \text{ (26)} \}
\end{align*}
\]

\textbf{Promoting \textit{fst}-Complementation}

Suppose that a given \textit{non-injective} \( f : A \times B \to B \) is complemented by \( \text{fst} : A \times B \to A \), i.e., that \( \text{fst} \circ f \) is injective. We can place it in (27) and ask: will \( \langle \text{[id}, f \rangle \) be \textit{fst}-complemented too? We start by unfolding the term \( \text{fst} \circ \langle [id}, f \rangle \):

\[
\begin{align*}
\text{fst} \circ \langle \text{[id}, f \rangle & = \langle \Psi \ (\text{fst} \circ f) \rangle, \\
\text{by introducing } \Psi \ x & = \alpha \cdot (\text{id} \times \text{x}) \cdot (\text{id} \times \text{snd}) \cdot \text{xc}
\end{align*}
\]

which shrinks to

\[
\Psi \ x = \alpha \cdot (\text{id} \times \text{x}) \cdot (\text{id} \times \text{snd}) \cdot \text{xc}
\]

using the isomorphism \( B \times (A \times C) \xrightarrow{\text{id}} A \times (B \times C) \) instead of \(A\).

Putting everything into a diagram, we obtain:

\[
\begin{array}{ccc}
A^* \times B & \xrightarrow{\alpha} B + A \times (A^* \times B) \\
\langle h \rangle \downarrow & & \downarrow \text{id+id}\times\langle h \rangle \\
A^* \times B & \xrightarrow{\text{id}} B + A \times (A^* \times B)
\end{array}
\]

Clearly, \( \Psi \) preserves injectivity, as does \( \langle . \rangle \) – see the appendix for details, available in the online supplemental material. Therefore, \( \text{fst} \circ f \) being injective ensures \( \text{fst} \circ \langle [id}, f \rangle \) is also injective. In words:

The \textit{fst}-complementation of \( f \) in \( \text{fold} \ \mathcal{F} \ \mathcal{B} \) is promoted to the \textit{fst}-complementation of the fold itself.

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

$$\text{rfold } :: \langle \{ (a, b) \rightarrow ([a], [b]), (a, b) \rightarrow ([a], b) \rangle, f \text{fold } ([], [b]), f \text{fold } ([a, b]) = (a, [b, f]) \rangle$$

where $(y, b') = \text{rfold } f (x, b)$

We can therefore rely on the reversibility of $\text{rfold } f$ provided $f$ is complemented by $\text{fst}$.

This result can be generalised by defining, given some $f : A \times B \rightarrow C \times B$, $\langle f \rangle = \langle \Psi f \rangle$ as pictured in the following diagram:

$$A^* \times B \xleftarrow{\alpha} B + A \times (A^* \times B) \quad (33)$$

$$\langle f \rangle \downarrow$$

$$C^* \times B \xrightarrow{\Psi_f} B + A \times (C^* \times B)$$

Clearly,

$$\langle id \rangle = id \quad (34)$$

since $\Psi id = a$. If $f$ is reversible, then $\langle f \rangle$ will also be reversible, as we have seen. For instance, $\langle \text{cnot} \rangle : B^* \times B \rightarrow B^* \times B$ will be reversible, because so is $\text{cnot} : B \times B \rightarrow B \times B$.

By free theorem calculation [41], we get the following properties among others:

$$\langle f \rangle \cdot (k^* \times id) = \langle f \cdot (k \times id) \rangle \quad (35)$$

$$(k^* \times id) \cdot \langle f \rangle = \langle (k \times id) \cdot f \rangle \quad (36)$$

Converting the construction $\langle \cdot \rangle$ to Haskell notation yields

$$\langle \cdot \rangle :: \langle (a, b) \rightarrow (c, b) \rangle \rightarrow ([a], b) \rightarrow ([c], b)$$

$$\langle f \rangle \cdot ([], [b]) = ([], b)$$

$$\langle f \rangle \cdot (a, b) = (c, b') \text{ where } (y, b') = \langle f \rangle \cdot (x, b)$$

$$(c, b') = f (a, b')$$

which corresponds to the standard $\text{mapAccumR}$ function.

In the sequel, we shall refer to $\langle f \rangle$ 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

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

Fig. 3. Suggestive depiction of a qubit as a superposition of classical bits. Credits: D. Moreda (IBM Quantum), “The next frontier: Quantum computing”, 2018. Slides presented at the PTH Conference, Porto, 18 May [42].

$$\begin{bmatrix}
\text{cnot } (0, b) &=& (0, b) \\
\text{cnot } (1, b) &=& (1, \neg b)
\end{bmatrix}$$

is described by the matrix:

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

Now think of a probabilistic “evolution” of $\text{cnot}$:

$$\begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & \frac{1}{2} & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}$$

In this evolution, function $\text{cnot}$ becomes probabilistic: $\text{cnot } (0, 1)$ 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:

$$\begin{bmatrix}
\text{cnot } (0, 0) &=& \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\
\text{cnot } (0, 1) &=& 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\
\text{cnot } (1, 0) &=& 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\text{cnot } (1, 1) &=& 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}}
\end{bmatrix}$$

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

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

$$T = \begin{bmatrix}
0 & 1 \\
1 & 0
\end{bmatrix} e^{\frac{\pi}{8}}$$
and the Hadamard gate,
\[ H = \begin{pmatrix} 0 & 1 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{pmatrix}, \]  
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 under functional composition, \((f \cdot g) \circ a = f(g \circ a)\), with identity \(id_a = a\). Moreover, all relations (in the same taxonomy) form another category, under relational composition, \(b(R \cdot S) \circ a = \exists c \ni bRc \land cSa\). This includes functions as a 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 [21], in which matrices are typed and written in the same way as functions or relations, i.e., as arrows (morphisms) \(A \to B\).

This denotes a matrix \(M\) whose columns are indexed by \(A\) and rows by \(B\). Under matrix multiplication, such matrices form a category, too [43].

Although the category of (entire, total) functions and categories of matrices are rather different in many respects, 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,

\[ id \otimes H = \begin{pmatrix} (0, 0) & (0, 1) & (1, 0) & (1, 1) \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{pmatrix} \]

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 \(\langle x \rangle\), for two instances of \(B \times B \to B \times B\). In the first case, \(x = \text{cnot}\) and one gets the typed matrix \(\langle \text{cnot} \rangle\) pictured in Fig. 4. This matrix clearly shows a (fragment of a) reversible function. For \(x = B\) from (37), the matrix \(\langle B \rangle\) is depicted in Fig. 5.

However, what does \(\langle B \rangle\) mean, giving that \(B\) is a matrix and not a function? This leads us to truly quantum quamtamorphisms, 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 \(\Lambda R : A \to \mathcal{P}B\), where \(\mathcal{P}B\) denotes the powerset of \(B\), under the correspondence: \(bRa \iff b \in \Lambda R a\). Under this correspondence, relation composition of \(R : A \to B\) and \(S : B \to C\) is given by

\[ c(S \circ R) a \iff c \in \bigcup \{ AS b \mid b \in \Lambda R a \} \]

This can be written in a more generic way. Let \(f\) and \(g\) abbreviate \(\Lambda R\) and \(\Lambda S\), respectively, with types \(f : A \to \mathcal{P}B\) and \(g : B \to \mathcal{P}C\). Then composition \(S \circ R\) is represented by
\(g \cdot f\), of type \(A \to PC\), defined monadically by

\[
(g \cdot f) a = \text{do}\{ b \leftarrow f a; g b \}
\]

Where does this terminology and notation come from?

It turns out that \(P\) 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 \(g \cdot f\) above. Comparing such a definition of composition with that of arbitrary functions in the original category, namely

\[
(g \cdot f) a = \text{let} b = f a \text{ in } g b,
\]

one immediately sees how the monadic \texttt{do} notation generalises the \texttt{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 \(f' = \text{ret} \cdot f\), where \(\text{ret} : A \to PA\) is the function that yields the smallest set that contains its argument, \(\text{ret} a = \{a\}\).\(^{18}\)

**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

\[
H = \begin{bmatrix}
0 & 1 \\
\frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}}
\end{bmatrix}
\]

is represented by the function

\[
\text{had} : \mathbb{B} \to \text{Vec } \mathbb{B}
\]

\[
\text{had } 0 = \left[ \begin{array}{c} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{array} \right]
\]

\[
\text{had } 1 = \left[ \begin{array}{c} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{array} \right]
\]

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

In the field of quantum information, the “Dirac notation” \(|x\rangle\) usually replaces \(\text{ret } x\), as in

\[
|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}
\]

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

\[
\text{had} : \mathbb{B} \to \text{Vec } \mathbb{B}
\]

\[
\begin{align*}
\text{had } 0 &= |0\rangle + |1\rangle \\
\text{had } 1 &= |0\rangle - |1\rangle 
\end{align*}
\]

(40)

The inhabitants of type \(\text{Vec } \mathbb{B}\) are usually known as \textit{qubits}, generalising classical \textit{bits} (Fig. 3). Bits are therefore special cases of qubits: \(|0\rangle\) and \(|1\rangle\) are classical, while e.g. \(|\frac{i}{\sqrt{2}}\rangle\) and \(|\frac{-i}{\sqrt{2}}\rangle\) are superpositions of \(|0\rangle\) and \(|1\rangle\).

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

\[
(f \otimes g) (a, b) = \text{do}\{ x \leftarrow f a; y \leftarrow g b; \text{ret } (x, y) \}
\]

(41)

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

Let us see this representation at work by looking at the structure of a famous example in quantum information – the “Alice” part of the teleportation protocol \([17]\):

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

\[
\text{bell } (a, b) = \text{do}\{ x \leftarrow \text{had } a; \text{ret } (\text{cnot } (x, b)) \}
\]

(42)

(Details in the appendix, available in the online supplemental material.) Then the second block, marked by the other dashed square, is \(B' = (H \otimes \text{id}) \cdot \text{cnot}\), where \(X'\) is the conjugate transpose of \(X\).\(^{20}\) This, using the same encoding rules, is represented by:

\[
\text{unbell } (c, a) = \text{let}\{ -a' = \text{cnot } (c, a); \text{in do}\{ b \leftarrow h \ c; \text{ret } (b, a') \}
\]

20. Recall that \(X^\dagger\) coincides with the transpose \(X^\text{c}\) when \(X\) does not involve imaginary parts.
Then the two blocks are put together via the associator isomorphism $a$, recall (26):

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

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

$$alice(c, (a, b)) =
\begin{array}{l}
\text{do} \\
\quad (a', b') \leftarrow \text{bell} (a, b); \\
\quad (c', a'') \leftarrow \text{unbell} (c, a'); \\
\quad \text{ret} (c', (a'', b'))
\end{array} \quad (43)$$

– for $\text{had}$ the Hadamard gate, recall (40) – encodes in Haskell an analogue of Fig. 6 but using this form of quantum conditional. This piece of code implements the following monadic program:

$$\text{cond} (q, p) = \text{do} \\
q' \leftarrow \text{had} q; \\
p' \leftarrow \text{if} q' \text{ then } \text{ret} (-p) \text{ else had } p; \\
\text{ret} (q', p') \quad (43)$$

This suggests the following quantum alternation combinator which, rather than measuring the control qubit to choose which branch to execute, implements a superposition of both branches controlled by the state of the control bit:

$$(f \circ g) (x, y) = \text{do} \\
a \leftarrow x; \\
b \leftarrow y; \\
c \leftarrow \text{if } a \text{ then } f (a, b) \text{ else } g (a, b); \\
\text{ret} (a, c)$$

13 **CONDITIONAL CONTROL FLOW**

Fig. 6 shows 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 for quantum programs that does not require measuring the control qubit, and which provides a useful construct for quantum programming. The quantum programming language QML [12] was the first to support this kind of conditional. For instance, the following monadic program

$$\text{cond} (q, p) = \text{do} \\
q' \leftarrow \text{had} q; \\
p' \leftarrow \text{if} q' \text{ then } \text{ret} (-p) \text{ else had } p; \\
\text{ret} (q', p') \quad (43)$$

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

$$\text{cnot} = \text{id} \circ \oplus \quad (45)$$

The corresponding linear algebra expression is

$$f \circ g = (\text{id} \otimes [f, g]) \cdot (\text{fst} \circ \gamma^*), \quad (44)$$

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

$$\begin{array}{c}
a \\
\text{if } a \text{ then } f (a, b) \text{ else } g (a, b)
\end{array}$$

As an example using this quantum alternation operator, note that

$$\text{cnot} = \text{id} \circ \oplus \quad (45)$$

This quantum alternation leads to a quantum version of the (classical) McCarthy conditional combinator, analogous to the probabilistic one given in [21]:

$$p \rightarrow f, g = (f \circ g) \cdot (p \otimes \text{id})$$

The diagram below spells out the whole pipeline:
Back to our motivating example, the following quantum McCarthy conditional

$$H \rightarrow X, H$$

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:

$$\langle Q \rangle = \Psi \cdot Q \cdot (\text{id} \oplus \text{id} \otimes \langle Q \rangle) \cdot \alpha^\circ,$$  

(46)

cf.

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

$$\langle Q \rangle \xrightarrow{\text{id} \oplus \text{id} \otimes \langle Q \rangle} B \times A \times (C^* \times B)$$

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

$$\Psi Q = \alpha \cdot (\text{id} \oplus xL \cdot (\text{id} \otimes Q) \cdot x)$$.

(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

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

holds. Comparing this with

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

we realise that bijections are exactly the classical unitary transformations.

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

$$\lambda f = \lambda q \cdot \lambda (a, b) \cdot (\text{Vec} (c, b) \rightarrow ([a], b) \rightarrow \text{Vec} ([c], b)$$

$$\langle f \rangle \langle (a, b) \rangle = \text{return} \langle (a, b) \rangle$$

$$\langle f \rangle (h: t, b) = \text{do} \{$$

$$\langle f' \rangle (t', b') = \langle f \rangle (h, b');$$

$$\text{return} \langle h', t', b' \rangle \}$$

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 the quantum operation $\text{bell}$ from (42) as the parameter of the quantamorphism. We can check what comes out for a particular computational basis input using GHci (here we are only showing the non-zero amplitudes):

$$x = \langle \text{bell} \rangle ([1, 0, 0], 1) =$$

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

Quantamorphisms implement a structural recursion on the type $A'$ of lists of $A$. In the example above, we calculated the result of $\langle \text{bell} \rangle$ when applied to a state in the computational basis where classical lists are encoded, corresponding to the list $[1, 0, 0]$. So, in a sense, what is implemented is a classical recursion performing quantum operations. To see the truly quantum nature of the control flow, note that it is possible to pass any quantum state as the first argument to the program $\langle \text{bell} \rangle$, which may be a superposition of lists (when regarded in terms of the ‘computational basis’).

This can be achieved, for example, by feeding the result of the program above into the same quantamorphism $\langle \text{bell} \rangle$:

$$y = \text{do} \{ i \leftarrow x; \langle \text{bell} \rangle i \}$$

Here $x$ is the state calculated above (48). The outcome will be:

$$y = \begin{bmatrix}
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0
\end{bmatrix}$$

The next step is – instead of simulating – to “compile” quantamorphisms such as $\langle \text{bell} \rangle$ 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 Fig. 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

![Fig. 7. Tool-chain describing the quantamorphism compilation work-flow.](image-url)
that describe the intended (recursive) quantum computations:

- **Quipper** [26] – this tool generates the quantum circuit from the unitary matrix;
- **PyZX** [48], [49] – this tool (based on the ZX-calculus) is used to optimise the quantum circuit issued by Quipper;
- **Qiskit** [50] – 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 $16 \times 16$ 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 exact\_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. 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.

We selected ibmq\_boeblingen due to its relatively high average coherence times ($77.888 \mu s / 99.935 \mu s$) and relatively low rate of CX errors (0.0118), at the time of the experiment. These are widely used metrics for comparing quantum devices [17], [51].

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., $Z \cdot Z = id$, $X \cdot X = id$, $H \cdot Z \cdot H = X$, $S \cdot H \cdot S \cdot H \cdot S = H$, 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 [49]. 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 [48].

21. The generation of the full unitary matrix represents an efficiency bottleneck in the current tool-chain. Please refer to Section 18 for a discussion of this issue and of how it can be bypassed, based on ongoing work in [47].

22. This tool can translate every standard gate from Quipper to Qiskit syntax and is available in [47].

23. $Z$ denotes the Pauli-Z gate and $S$ its square root $\sqrt{Z}$, usually called the $S$ or phase gate, which performs a phase shift by $\frac{\pi}{4}$ radians [17].
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; ‘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 transistor 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.

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

To which extent classical methodologies generalise to quantum programming is a research question that still needs to be answered. Ying [52] reviews the literature on attempts to extend the imperative and functional paradigms to quantum computing, including semantic models and verification techniques. In this setting, extending classical fixpoint recursion theory to quantum programming has proved to be a complex topic that has raised mixed feelings about the possibility of ever realising full recursion with quantum control flow; see e.g. [53].

This paper contributes to the field of quantum recursion in a different way: rather than focusing on verification techniques enabled by a general semantic theory, it proposes a constructive and compositional approach whereby more complex quantum programs are built out of smaller components using particular, semantics-rich combinators expressed at a level higher than the usual circuit model.

For this it introduces the quantamorphism recursion pattern as a reversible (unitary) extension of a classical recursion combinator known as fold or catamorphism. These implement structural recursion, having the property of always terminating, while encompassing a wide class of algorithms over lists. Thinking in terms of such generic, structural recursion patterns is a powerful programming technique supported by a deductive theory for reasoning and optimising programs by calculation. It also raises the level of discourse above that of circuit-level design, which still is the primary paradigm for quantum programming.

By restricting to such structural recursion patterns for unitary computation, we arrive from rather different considerations to a recursion scheme similar to that considered in [54]:

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, this is achieved in a substantially different way, in various respects:

- 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 (Sections 9 and 10).
- Quantamorphisms: generalisation of reversible folds to unitary folds, enabling recursive quantum programming with structural quantum control flow (Sections 11, 12, 13, and 14).
- Implementation on IBM Q: proof-of-concept implementation of quantamorphisms on IBM Q Experience devices (Sections 15 and 16).

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 [56]. Some of the circuits tested in [56] 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 recursion is still in its infancy. An open question suggested by the present work is whether one can find an extension of quantamorphisms to inductive types other than natural numbers and finite lists. Such a generalisation

24. Iterative loops, sometimes called for-loops, are also in this class, see e.g. [55].
25. For a detailed account of this experimentation see the webpage [47] or the master’s dissertation [56] for older versions.
is a challenge for future work, possibly inspired by so-called traversable structures [57].

On the experimental side, the tool-chain used in our lab setup uses GHCi and Quipper. We run the monadic quantamorphisms in GHCi to generate the unitary matrices out of which Quipper generates the quantum circuits (Fig. 7). Although this is enough for a proof-of-concept prototype demonstrating small examples, the approach of building a single large unitary in the compilation process is not scalable. Calculating this unitary is sufficient to strongly simulate the quantum computation, thus only being practical in small toy examples which can be classically simulated. This is a clear bottleneck in the toolchain presented, and shortcircuiting this step is an important focus for future research.26 Achieving this will require a thorough analysis of Quipper’s recursive circuit implementation [26].

Another challenge in the implementation process is to handle quantum errors. Circuit optimisation was able to curb the problem slightly by reducing circuit size. More elaborate, targetted compiling strategies able to further mitigate it (within the limitations of current NISQ hardware) are a topic of current research [58].

While the IBM Q Experience offers three levels of optimisation, other academic researchers have developed tools like PyZX. Future work should explore further, more elaborate compilation strategies. For example, the $|\psi\rangle$ compiler, which handles the routing problem aiming for hardware compatibility with minimal additional gate overhead, promises good results when tested against Qiskit optimisation level 1 and PyZX [59].

ACKNOWLEDGMENTS

The Authors would like to acknowledge the use of IBM Quantum services for this work. All experiments on IBM Q devices have been carried out through the IBM Q Hub at Minho. The views expressed are those of the authors, and do not reflect the official policy or position of IBM or the IBM Quantum team. The present work was done in part while Rui Soares Barbosa was with the Department of Computer Science, University of Oxford, United Kingdom.

REFERENCES

[1] J. P. Dowling and G. J. Milburn, “Quantum technology: The second quantum revolution,” Philos. Trans. Roy. Soc. Ser. A, vol. 361, no. 1809, pp. 1655–1674, 2003.
[2] A. de Touzalin, C. Marcus, F. Heijman, I. Cirac, R. Murray, and T. Calarco, “Quantum Manifesto: A new era of technology,” 2016. [Online]. Available: http://qurope.eu/manifesto
[3] A. M. Turing, “On computable numbers, with an application to the Entscheidungsproblem,” Proc. London Math. Soc., vol. 2, no. 42, pp. 230–265, 1936.
[4] A. Church, “An unsolvable problem of elementary number theory,” Amer. J. Math., vol. 58, no. 2, pp. 345–363, 1936.

26. Preliminary research reported in [47] shows how to synthesise circuits compositionally from quantamorphisms. However, this research has so far only covered the simpler case of quantum for-loops [56], and not yet quantamorphisms over lists as discussed in the current paper.

[5] F. Nebecker, Dawn of the Electronic Age: Electrical Technologies in the Shaping of the Modern World, 1914 to 1945. Hoboken, NJ, USA: Wiley–IEEE Press, 2009.
[6] S. K. Rouray, “History of electronics,” in Proc. IEEE Conf. History Electron., 2004. [Online]. Available: https://ethw.org/2004_IEEE_Conference_on_the_History_of_Electronics
[7] D. Laws, “Who invented the transistor? – Computer history museum.” Accessed: Feb. 04, 2019. [Online]. Available: http://www.computerhistory.org/atchw-who-invented-the-transistor/
[8] D. Deutsch, “Quantum theory, the Church–Turing principle and the universal quantum computer,” Proc. Roy. Soc. London Ser. A, vol. 400, no. 1818, pp. 97–117, 1985.
[9] J. Sanders and P. Zuliani, “Quantum programming,” in Proc. 5th Int. Conf. Math. Program Construction, 2000, pp. 80–99.
[10] P. Selinger, “Towards a quantum programming language,” Math. Struct. Comput. Sci., vol. 14, no. 4, pp. 527–586, 2004.
[11] M. Ying and Y. Feng, “A flowchart language for quantum programming,” IEEE Trans. Softw. Eng., vol. 37, no. 4, pp. 466–485, Jul./Aug. 2011.
[12] T. Allenkirch and J. Grattage, “A functional quantum programming language,” in Proc. 20th Ann. IEEE Symp. Logic Comput. Sci., 2005, pp. 249–258.
[13] J. Kramer, “Is abstraction the key to computing?,” Commun. ACM, vol. 50, no. 4, pp. 37–42, Apr. 2007.
[14] R. Backhouse, Mathematics of Program Construction. Nottingham, U.K.: Univ. Nottingham, 2004.
[15] R. Bird and O. de Moor, Algebra of Programming. Hoboken, NJ, USA: Prentice-Hall, 1997.
[16] S. MacLane, Categories for the Working Mathematician. New York, NY, USA: Springer, 1971.
[17] M. Nielsen and I. Chuang, Quantum Computation and Quantum Information, 10th ed. New York, NY, USA: Cambridge Univ. Press, 2011.
[18] E. Hoag, M. Zhu, and S. Decker, “qdb: Inserted tomography for breakpoint debugging in pyQuil,” Stanford Univ., Stanford, CA, USA, Student project. CS269Q: Quantum Computer Programming, 2019. [Online]. Available: https://cs269q.stanford.edu/projects2019/qdb.pdf
[19] N. Yanošky and M. Mannucci, Quantum Computing for Computer Scientists. Cambridge, U.K.: Cambridge Univ. Press, 2008.
[20] D. Murta and J. Oliveira, “A study of risk-aware program transformation,” Sci. Comput. Program., vol. 110, pp. 51–77, 2015.
[21] J. N. Oliveira, “Towards a linear algebra of programming,” Formal Aspects Comput., vol. 24, no. 4–6, pp. 433–458, 2012.
[22] M. Erwig and S. Kollmannsberger, “Functional pearls: Probabilistic functional programming in Haskell,” J. Functional Program., vol. 16, pp. 21–34, Jan. 2006.
[23] B. Coecke and E. O. Paquette, “Categories for the practising physicist,” in New Structures for Physics (Lecture Notes in Physics Series). Berlin, Germany: Springer, 2011, vol. 831, pp. 173–286.
[24] A. Neri, R. S. Barbosa, and J. N. Oliveira, “Compiling quantamorphisms for the IBM Q Experience,” Talk at 77th Meeting IFIP Work. Grp. 2.1, Jul. 2018. [Online]. Available: https://ifipwg21wiki.cs.kuleuven.be/IFIP21/Brandenburg
[25] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi, “Bidirectionalization transformation based on automatic derivation of view complement functions,” in Proc. 12th ACM SIGPLAN Int. Conf. Functional Program., 2007, pp. 47–58.
[26] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, “An introduction to quantum programming in Quipper,” in Proc. 5th Int. Conf. Reversible Comput., 2013, pp. 110–124.
[27] B. Coecke, Ed., New Structures for Physics. Berlin, Germany: Springer, 2011.
[28] I. Hasuo and N. Hoshino, “Semantics of higher-order quantum computation via geometry of interaction,” Ann. Pure Appl. Logic, vol. 168, no. 2, pp. 404–469, 2017.
[29] 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.
[30] A. Tarski and S. Givant, A Formalization of Set Theory without Variables. Providence, RI, USA: American Mathematical Society, 1987.
[31] J. Backus, “Can programming be liberated from the von Neumann style? A functional style and its algebra of programs,” Commun. ACM, vol. 21, no. 8, pp. 613–639, 1978.
[32] P. Freyd and A. Scedrov, Categories, Allegories (Mathematical Library Series). Amsterdam, The Netherlands: Elsevier Sci., 1990.
[33] C. Morgan, Programming from Specification, (Series in Computer Science). Hoboken, NJ, USA: Prentice-Hall, 1990.

[34] J. Oliveira and M. Ferreira, “Alloy meets the algebra of programming: A case study,” IEEE Trans. Softw. Eng., vol. 39, no. 3, pp. 305–326, Mar. 2013.

[35] J. Oliveira, “Programming from metamorphisms,” J. Logical Algebric Methods Program., vol. 94, pp. 15–44, 2018.

[36] G. Schmidt, Relational Mathematics (Encyclopedia of Mathematics and its Applications Series). Cambridge, U.K.: Cambridge Univ. Press, 2010.

[37] J. Oliveira, “A relation-algebraic approach to the “Hoare logic” of functional dependencies,” J. Logic Algebric Methods Program., vol. 83, no. 2, pp. 240–262, 2014.

[38] F. Banchelon and N. S Cali, “Update semantics of relational views,” ACM Trans. Database Syst., vol. 6, no. 4, pp. 557–575, 1981.

[39] C. Oalh, “Neural networks, types, and functional programming,” colah’s blog. Accessed: Aug. 27, 2021. [Online]. Available: http://colah.github.io/posts/2015-09-NN-Types-FP/

[40] R. Hinze, “Adjoint folds and unfolds – An extended study,” Sci. Comput. Program., vol. 78, no. 11, pp. 2108–2190, 2013.

[41] P. Wadler, “Theorems for free!,” in Proc. 4th Int. Symp. Functional Program. Lang. Comput. Archit., 1989, pp. 347–359.

[42] D. Moreda (IBM Quantum), “The next frontier: Quantum computing,” Talk at Porto Tech Hub Conf. Porto, May 2018. [Online]. Available: https://portotechhub.com/conference-2018

[43] H. Macedo and J. Oliveira, “Typing linear algebra: A biproduct-oriented approach,” Sci. Comput. Program., vol. 78, no. 11, pp. 2160–2191, 2013.

[44] J. Gibbons and R. Hinze, “Just do it: Simple monadic equational reasoning,” in Proc. 16th ACM SIGPLAN Int. Conf. Functional Prog.gram., 2011, pp. 2–14.

[45] A. Santos, “Haskell library: Laop-0.1.0.7,” 2020. [Online]. Available: 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 Comput. Sci., vol. 3, no. 4, pp. 1–36, 2007.

[47] A. Neri, J. N. Oliveira, and R. S. Barbosa, “Quantamorphisms guide,” Accessed: Aug. 23, 2021. [Online]. Available: https://quantamorphisms.netlify.com

[48] A. Kissinger and J. van de Wetering, “PyZX: Large scale automated diagrammatic reasoning,” in Proc. 16th Int. Conf. Quantum Phys., 2020, pp. 229–241.

[49] B. Coecke and R. Duncan, “Interacting quantum observables,” in Proc. 35th Int. Colloquium Automata, Lang. Program., 2008, pp. 5126, pp. 298–310.

[50] G. Aleksandrowicz et al., “Qiskit: An open-source framework for quantum computing (0.7.2),” Zenodo. Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2562111.

[51] J. Gambetta and S. Sheldon, “Cramming more power into a quantum device,” IBM Research Blog. Accessed: Aug. 27, 2021. [Online]. Available: https://www.ibm.com/blogs/research/2019/03/power-quantum-device

[52] M. Ying, Foundations of Quantum Programming. Cambridge, MA, USA: Elsevier, 2016.

[53] C. Bădescu and P. Panangaden, “Quantum alternation: Prospects and problems,” in Proc. 12th Int. Workshop Quantum Phys. Logic, 2015, pp. 33–42.

[54] A. Sabry, B. Valiron, and J. K. Vizzotto, “From symmetric pattern-matching to quantum control,” in Proc. 21st Int. Conf. Found. Softw. Sci. Comput. Struct., 2018, pp. 348–364.

[55] J. Oliveira, “Towards quantamorphisms – Some thoughts on (constructive) reversibility,” ICT Col Action 1405 Meeting, Cyprus, 19–21 Mar. 2018.

[56] A. Neri, “Towards quantum program calculation,” M.Sc. Thesis, Departamento de Informática, Univ. Minho, Braga, Portugal, Oct. 2018.

[57] M. Jaskelioff and O. Rypacek, “An investigation of the laws of traversals,” in Proc. 4th Workshop Math. Struct. Functional Program., 2012, vol. 76, pp. 40–49.

[58] P. Murali, J. M. Baker, A. J. Abhari, F. T. Chong, and M. Martonosi, “Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers,” in Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst., 2019, pp. 1015–1029.

[59] A. Cowtan, S. Dilkes, R. Duncan, W. Simmons, and S. Sivarajah, “Phase gadget synthesis for shallow circuits,” in Proc. 16th Int. Conf. Quantum Phys. Logic, 2020, vol. 318, pp. 213–228.

Rui Soares Barbosa received a BSc degree in computer science from Universidade do Minho in 2018, and the MSc degree in mathematics and foundations of computer science from the University of Oxford in 2010, and a DPhil degree in computer science also from Oxford in 2015, under the supervision of Samson Abramsky and Andreas Döring. He was a postdoctoral researcher at the University of Oxford from 2015 to 2019 and at the University of Edinburgh from 2019 to 2020, and he held a Simons–Berkeley research fellowship at the Simons Institute, UC Berkeley in Autumn 2017. He is currently a staff researcher in the Quantum and Linear-Optical Computation group at the International Iberian Nanotechnology Laboratory (INL) in Braga, Portugal. His interdisciplinary research addresses interrelated questions on quantum foundations, quantum computer science, and the mathematics of quantum theory, with an emphasis on logical, structural, and compositional aspects.

Ana Neri received a master’s degree in physical engineering from the University of Minho, Portugal, in 2018. She is currently working on her doctoral thesis with the University of Minho. Her affiliations include the International Iberian Nanotechnology Laboratory (INL), High Assurance Software Laboratory (HASLab), INESC TEC, and Center for Research & Development in Mathematics and Applications (CIDMA). She received the PhD scholarship on Quantum Science and Technology from the Quantum Portugal Initiative, a collaboration between Fundação para a Ciência e a Tecnologia (FCT) and International Iberian Nanotechnology Laboratory (INL).

José N. Oliveira received a degree in electrical engineering in 1978 from the University of Porto, Portugal, and the MSc and PhD degrees in computer science from the University of Manchester, U.K., in 1980 and 1984, respectively. Since 2014, he has been a full professor with the Computer Science Department, University of Minho, Portugal, a member of the High Assurance Software Laboratory (HASLab), INESC TEC/University of Minho, and a member of IFIP WG2.1 (Algorithmic Languages and Calculi) and of the Formal Methods Europe (FME) association.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.