Compiling quantamorphisms for the IBM Q-Experience

IFIP WG2.1 meeting #77

Brandenburg, Germany, July 2018

J.N. Oliveira
(joint work with A. Neri and R.S. Barbosa)

INESC TEC & University of Minho
Context

- Bridging **U.Minho** / **INESC TEC** / **INL** (Braga, Portugal)
- Academic partner of the **IBM** Quantum Network
Is History going to repeat itself?

1944 (Colossus)  

2018 (IBM Q Experience)
Quantum computing

Phantasy — ?

“Threat” — ?

Opportunity — ✔

FP folk apt for it, as I will try to show 😊
Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is

40 years of the algebra of programs
Age of **MapReduce** predicted by John Backus

**MapReduce** in Backus’ notation

\[(/g) \cdot (\alpha f)\]

**MapReduce** in “modern” notation

\[(|g|) \cdot (\textbf{fmap} \ f)\]

However,

\textit{we are spending too much energy in all this...}

**Thermodynamics** — **Landauer’s principle** ("logically irreversible manipulation of information leads to an increase in entropy").
Reversibility was not a concern in 1978.

Program design by source-to-source transformation (1980s) sought efficiency only.

Function $g$ in

$$f \cdot g = id$$

is injective because it has a left inverse (which is surjective). Put in another way, via the algebra of relations:

$$g \subseteq f^\circ$$

— converse of functional ($f^\circ$) is injective (and smaller than injective is injective).
Refine for injectivity

New concern — refine programs towards **injective** solutions.

Need for an **injectivity** (pre)order, e.g.

Since we need to compute **non-injective** operations anyway, these have to run inside injective "envelopes" delaying their **observation** as much as possible.

**Complementation** is one such possible envelope, behaving nicely wrt the required preorder.
Comparing functions / relations for injectivity

Given a function \( f : A \rightarrow B \), define its **converse** as the relation \( f^\circ : A \leftarrow B \) such that \( a f^\circ b \Leftrightarrow b = f a \). Then

\[
f \text{ injective} \iff f x = f x' \Rightarrow x = x'
\]

abbreviates to:

\[
f^\circ \cdot f \subseteq \text{id}
\]

Moreover, \( g \) **less injective** than \( f \)

\[
g \leq f \iff f x = f x' \Rightarrow g x = g x'
\]

simplifies to:

\[
g \leq f \iff f^\circ \cdot f \subseteq g^\circ \cdot g
\]
The whole picture (relation 'bestiary')

binary relation

injective  →  entire  →  simple  →  surjective

representation  →  function  →  abstraction

injection  →  surjection

bijection

where

\[ R \text{ injective} \iff R^\circ \cdot R \subseteq \text{id} \]
\[ R \text{ entire} \iff \text{id} \subseteq R \cdot R^\circ \]
\[ R \text{ simple} \iff R^\circ \text{ injective} \]
\[ R \text{ surjective} \iff R^\circ \text{ entire} \]
Relations as matrices

It helps if we depict relations using (Boolean) matrices, for instance negation (a bijection) $\neg = \begin{array}{c|cc}
0 & 1 \\
0 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}$

exclusive-or (surjective but not injective): $(\vee) = \begin{array}{c|cccc}
0 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
\end{array}$

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.
Going (more) injective

We are interested in exploiting the **injectivity** preorder,

\[ R \leq S \iff \ker S \subseteq \ker R \]

as a **refinement ordering** guiding us towards more and more **injective** computations.

This ordering is rich in properties, for instance it is upper-bounded

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

(1)

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

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

In the case of functions:

\[(f \uplus g) a = (f a, g a)\]

(2)
Going (more) injective

Cancellation via (1) means that **pairing** always **increases injectivity**:

\[ R \leq R \upwedge S \quad \text{and} \quad S \leq R \upwedge S. \]  

(3)

(3) unfolds to \( \ker (R \upwedge S) \subseteq (\ker R) \cap (\ker S) \), which is in fact an equality

\[ \ker (R \upwedge S) = (\ker R) \cap (\ker S) \]  

(4)

itself a corollary of the more general:

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

(5)

Injectivity **shunting laws** also arise as Galois connections, e.g.

\[ R \cdot g \leq S \quad \iff \quad R \leq S \cdot g^\circ \]
Ordering functions by injectivity

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

$$1 \leq f \leq id$$

where $1 \leftarrow A$ is the unique function of its type.

- A function is injective iff $id \leq f$. Thus $f \nabla id$ is always injective (3).
- Two functions $f$ e $g$ are said to be complementary wherever $id \leq (f \nabla g)$.\(^1\)

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

\(^1\)Cf. (Matsuda et al., 2007). Other terminologies are monic pair (Freyd and Scedrov, 1990) or jointly monic (Bird and de Moor, 1997).
Minimal complements

**Minimal complements** — *Given* $f$, *suppose* (a) $\text{id} \leq f \triangledown g$; (b) if $\text{id} \leq f \triangledown h$ and $h \leq g$ then $g \leq h$.

*Then* $g$ *is said to be a minimal complement of* $f$ *(Bancilhon and Spyros, 1981).*

Minimal complements (not unique in general) characterize *“what is missing”* from the original function for *injectivity* to hold.

**Example:** Non-injective $\begin{array}{c}
 2 \leftarrow \uparrow \\
 2 \times 2
\end{array} = \begin{bmatrix}
 1 & 0 & 0 & 1 \\
 0 & 1 & 1 & 0
\end{bmatrix}$ has minimal complement $\begin{array}{c}
 2 \leftarrow \text{fst} \\
 2 \times 2
\end{array} = \begin{bmatrix}
 1 & 1 & 0 & 0 \\
 0 & 0 & 1 & 1
\end{bmatrix}$. 
Complementing $\vee$

As is well-known, by complementing $\vee$ with $\text{fst}$

$$2 \times 2 \xleftarrow{\text{fst} \downarrow (\vee)} 2 \times 2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

we get a bijection — the classical CNOT quantum gate:

$$\begin{cases} \text{cnot} (0, b) = (0, b) \\ \text{cnot} (1, b) = (1, \neg b) \end{cases}$$
Generic \( \text{fst} \)-complementation

Generalize \( \vee \) to monoid \((A; \theta, 0)\) such that:\(^2\)

\[
x \theta x = 0
\]

(6)

Then

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

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

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

is reversible for \textbf{any} \( f : A \rightarrow B \).

\(^2\)There should be a name for this but I can’t remember it now.
Generic \textit{fst}-complementation

\begin{center}
\begin{tikzpicture}
\node (A) at (0,0) {$U\ f$};
\node (B) at (1,0) {$(f\ x)\ \theta\ y$};
\node (C) at (0,-1) {$x$};
\node (D) at (1,-1) {$y$};
\node (E) at (1,0) {$x$};
\draw[->] (A) -- (C);
\draw[->] (A) -- (D);
\draw[->] (B) -- (E);
\end{tikzpicture}
\end{center}

\textbf{bijective} because it is its self inverse:

\[(U\ f) \cdot (U\ f) = id\]
\[
\Leftrightarrow \quad \{\ U\ f\ (x,\ y) = (x,\ (f\ x)\ \theta\ y) \ \}\]
\[
U\ f\ (x,\ (f\ x)\ \theta\ y) = (x,\ y)
\]
\[
\Leftrightarrow \quad \{\ \text{again}\ U\ f\ (x,\ y) = (x,\ (f\ x)\ \theta\ y) \ \}\]
\[
(x,\ (f\ x)\ \theta\ ((f\ x)\ \theta\ y)) = (x,\ y)
\]
\[
\Leftrightarrow \quad \{\ \theta\ \text{is associative and } x\ \theta\ x = 0 \ \}\]
\[
(x,\ 0\ \theta\ y) = (x,\ y)
\]
\[
\Leftrightarrow \quad \{\ 0\ \theta\ x = x \ \}\]
\[
(x,\ y) = (x,\ y)
\]
Chaining $\text{fst}$-complemented computations

Drawing $y_0 \rightarrow f \rightarrow y_1$ instead of $x_1 \rightarrow f \rightarrow y_1$

one may think of **chaining** such computations,

$y_0 \rightarrow f \rightarrow f \rightarrow y_2$, $y_0 \rightarrow f \rightarrow f \rightarrow f \rightarrow y_3$, ...
Similar construction in neural networks

(RNN = accumulating maps\(^3\))

Yes — \texttt{mapAccumR} in Haskell

How is \textit{injectivity} ensured?

\(^3\)Source: \textit{Neural Networks, Types, and Functional Programming} by C. Olah, 2015.
The role of $A \xleftarrow{\text{fst}} A \times B$

$\text{fst}$-complementation,

$id \leq \text{fst} \circ f$

means

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

i.e. it means $f$ injective on the second argument once the first is fixed.

Moreover, $A \times B \xrightarrow{\text{fst}} A$ paired with a function of type $A \times B \rightarrow B$ makes room (type-wise) for a bijection of type $A \times B \rightarrow A \times B$.

Can $(\text{fst} \circ _{-})$ be extended recursively?
Towards (constructive) recursive complementation

Suppose we want to offer arbitrary $k : A \rightarrow B$ in a bijective “envelope” (injectivity alone does not work for e.g. quantum computing, as we shall see).

The “smallest” (generic) type for such an enveloped function is $A \times B \rightarrow A \times B$.

Now suppose $k$ is a recursive function, e.g. $k = \text{foldr } \bar{f} \ b$, for $f : A \times B \rightarrow B$, that is

\[
\begin{align*}
   k : A^* & \rightarrow B \\
   k [] & = b \\
   k (a : x) & = f (a, k x)
\end{align*}
\]

How do we “constructively” build the corresponding (recursive, bijective) envelope of type $A^* \times B \rightarrow A^* \times B$?
Going general (folds)

Let us define $\langle f \rangle$ such that $k = \text{foldr} \; \bar{f} \; b \times = \langle f \rangle \; (x, b)$, that is:

$$\langle f \rangle \; ([], b) = b$$
$$\langle f \rangle \; (a : x, b) = f \; (a, \langle f \rangle \; (x, b))$$

Thus

$$A^* \times B \leftarrow^\alpha B + A \times (A^* \times B)$$

$\langle f \rangle$

$$\downarrow$$

$$B \leftarrow [id, f] B + A \times B$$

As usual,

$$X + Y = \{ i_1 \; x \mid x \in X \} \cup \{ i_2 \; y \mid y \in Y \}$$

is disjoint union of $X$ and $Y$ — assuming $i_1 \cdot i_2^\circ = \bot$ — and $[R, S]$ is the unique relation $X$ such that $X \cdot i_1 = R$ and $X \cdot i_2 = S$. 
Towards reversible folds

**NB:**

\[ A^* \times B \leftarrow^\alpha B + A \times (A^* \times B) \]

is the isomorphism

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

(7)

where

\[ (A \times B) \times C \leftarrow^a A \times (B \times C) = (\text{id} \times \text{fst}) \downarrow (\text{snd} \cdot \text{snd}) \]  

(8)

Functions

\[ \text{nil} = [] \]

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

are the components of the initial algebra of lists \( \text{in} = [\text{nil}, \text{cons}] \).
Universal property

We actually need something more general:

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

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

Universal property

\[
k = \langle h \rangle \iff k \cdot \alpha = h \cdot F k
\]  
(9)

where \( F f = id + id \times f \).

From (9,8) one infers

\[
A^* \xleftarrow{\text{fst}} A^* \times B = \langle \text{in} \rangle
\]  
(10)
Suppose non-injective $f : A \times B \rightarrow B$ is complemented by $\text{fst} : A \times B \rightarrow A$. The following diagram shows how to use injective $\text{fst} \, \triangledown \, f$ to build an envelope for $\text{foldr} \, \bar{f} \, b$:

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

\[
\Downarrow \langle h \rangle
\]

\[
A^* \times B \xleftarrow{\alpha} h \downarrow \phi (\text{fst} \, \triangledown \, f)
\]

\[
B + A \times (A^* \times B)
\]

\[
\Downarrow \text{id} + \text{id} \times \langle h \rangle
\]

\[
A^* \times B \xleftarrow{\alpha} \phi (\text{fst} \, \triangledown \, f)
\]

where

\[
\Phi \, x = \text{id} + (x \text{l} \cdot (\text{id} \times x) \cdot x \text{l})
\]

resorting to isomorphism $A \times (B \times C) \xrightarrow{x \text{l}} B \times (A \times C)$. 

Promoting complementation
Promoting complementation

Note that \( \text{fst} \uplus \sqbrack{\text{id}, f} \) also has type \( A^* \times B \rightarrow A^* \times B \), recall

\[
A^* \times B \xleftarrow{\alpha} B + A \times (A^* \times B) \\
\downarrow \quad \downarrow \quad \downarrow \quad \downarrow \quad \downarrow \\
B \xleftarrow{[\text{id}, f]} B + A \times B
\]

How do

\[
\sqbrack{\alpha \cdot \Phi (\text{fst} \uplus f)} \quad \text{and} \quad \text{fst} \uplus \sqbrack{\text{id}, f}
\]

compare to each other?

Knowing by (10) that \( \text{fst} = \sqbrack{\text{in}} \) we appeal to the popular loop-intercombination law known as “banana-split”:

\[
\sqbrack{f} \uplus \sqbrack{g} = \sqbrack{(f \times g) \cdot (F \text{fst} \uplus F \text{snd})}
\] (11)
Promoting complementation

We reason:

\[ \text{fst} \; \downarrow \; \llbracket \text{id} , \; f \rrbracket \]

\[ = \quad \{ \text{banana-split} \} \]

\[ \llbracket (\text{in} \times [\text{id} , \; f]) \cdot (F \; \text{fst} \; \downarrow \; F \; \text{snd}) \rrbracket \]

\[ = \quad \{ \text{pairing laws (products)} \} \]

\[ \llbracket [\text{nil} , \; \text{cons} \cdot (\text{id} \times \text{fst})] \; \downarrow \; [\text{id} , \; f \cdot (\text{id} \times \text{snd})] \rrbracket \]

\[ = \quad \{ \text{exchange law} \} \]

\[ \llbracket [\text{nil} \; \downarrow \; \text{id} , \; (\text{cons} \cdot (\text{id} \times \text{fst})) \; \downarrow \; (f \cdot (\text{id} \times \text{snd}))] \rrbracket \]

\[ = \quad \{ \text{products} ; \; a \cdot a^\circ = \text{id} \} \]

\[ \llbracket \alpha \cdot a^\circ \cdot (\text{id} \times \text{fst}) \; \downarrow \; (f \cdot (\text{id} \times \text{snd})) \rrbracket \]

\[ \Phi (\text{fst} \; \downarrow \; f) \]
Promoting complementation

Thus

\[
fst \downarrow \Downarrow [id, f] = \Downarrow \alpha \cdot (\Phi (fst \downarrow f))\]

(13)

Clearly, \( \Phi \) preserves injectivity, as does \( \Downarrow \) (details in the appendix).

Summary:

\[
fst \downarrow f \text{ injective } \Rightarrow \Downarrow \alpha \cdot (\Phi (fst \downarrow f)) \text{ injective}
\]

That is, \( fst \)-complementation of \( f \) in \( k = \text{foldr } \bar{f} \ b \) is promoted to the \( fst \)-complementation of the fold itself.

\( fst \)-complement \textbf{propagated} inductively.
In standard Haskell

In standard Haskell, we can rely on the reversibility of

\[
\text{rfold} :: (a \rightarrow b \rightarrow b) \rightarrow ([a], b) \rightarrow ([a], b)
\]

\[
\text{rfold} f ([], b) = ([], b)
\]

\[
\text{rfold} f (a : x, b) = (a : x, f a b)
\]

provided \( f \) is complemented by \( \text{fst} \):

\[
\text{rfold} \quad \text{foldr } \overline{f} \quad b \quad x
\]
Going quantum

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

\[
\begin{align*}
\text{\text{cnot}} (0, b) &= (0, b) \\
\text{\text{cnot}} (1, b) &= (1, \neg b)
\end{align*}
\]

\[
\begin{array}{c|cccc}
\text{Input} & 0 & 1 & 0 & 1 \\
\hline
0, 0 & 1 & 0 & 0 & 0 \\
0, 1 & 0 & 1 & 0 & 0 \\
1, 0 & 0 & 0 & 0 & 1 \\
1, 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]

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

\[
\begin{array}{c|cccc}
\text{Input} & 0 & 1 & 0 & 1 \\
\hline
0, 0 & 1 & 0 & 0 & 0 \\
0, 1 & 0 & \frac{1}{2} & 0 & 0 \\
1, 0 & 0 & \frac{1}{2} & 0 & 1 \\
1, 1 & 0 & 0 & 1 & 0 \\
\end{array}
\]
Moving further to **quantum** corresponds to generalizing probabilities to **amplitudes**, for instance

\[
\begin{array}{c|cccc}
 & \bigcirc & \bigcirc & \bigcirc & \bigcirc \\
(0,0) & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} & 0 \\
(0,1) & 0 & \frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \\
(1,0) & 0 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\
(1,1) & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} & 0 \\
\end{array}
\]

**bell** =

Amplitudes are **complex** numbers indicating the **superposition** of information at quantum information level.
Quantum information

Credits: IBM Research AI & Q
Quantum programs (QP) are made of elementary units called quantum gates, for instance the so-called Hadamard gate,

\[
\text{had} = \begin{array}{c|cc}
0 & 1 & 1 \\
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & -
\end{array}
\]

which is a component of the previous example.

The approach is compositional, using two main combinators — composition (\(\cdot\)) and (tensor) product (\(\otimes\)).

Functional programmers (FP) familiar with pointfree (or monadic) notation are particularly well-positioned to understand QP.
Quantum abstraction

Bird’s-eye view of the **structure** of a famous example (the “Alice” part of the teleportation protocol):

(Cf. *entangled* photon pairs)
Quantum abstraction

$|\psi\rangle$

$A$

$B$

$H$

$H$

Quantum abstraction
Quantum abstraction
Quantum abstraction

$|\psi\rangle$

$A$

$B$

$H$

Quantamorphisms
Quantum abstraction

|ψ⟩

A

B

h

C

h

C
Quantum abstraction

$$|\psi\rangle$$

So

$$alice = (\text{unbell} \otimes \text{id}) \cdot a \cdot (\text{id} \otimes \text{bell})$$  \hspace{1cm} (14)

where $$\text{bell} = \text{cnot} \cdot (\text{had} \otimes \text{id})$$. 
Quantum abstraction (monadic)

It turns out that

$$alice = (unbell \otimes id) \cdot a \cdot (id \otimes bell)$$

can also be written

$$alice \ (c, \ (a, \ b)) =$$

$$\textbf{do} \ \{$$
$$\quad (a', \ b') \leftarrow \textit{bell} \ (a, \ b);$$
$$\quad (c', \ a'') \leftarrow \textit{unbell} \ (c, \ a');$$
$$\quad \textbf{return} \ (c', \ (a'', \ b'))$$
$$\}$$

— just standard \textbf{monadic} programming

Where is the \textbf{quantum} part gone? Details next.
Monads for quantum programming

Back to the **Hadamard** gate,

\[
\text{had} = \begin{array}{cc}
0 & 1 \\
\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\
\frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}}
\end{array}
\]

note that it can be written *pointwise* as

\[
\text{had} :: 2 \rightarrow \text{Vec} 2
\]

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

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

or even as

\[
\text{had} 0 = \frac{|0\rangle + |1\rangle}{\sqrt{2}}
\]

\[
\text{had} 1 = \frac{|0\rangle - |1\rangle}{\sqrt{2}}
\]

defining

\[
|0\rangle = \left[\begin{array}{c}1 \\
0\end{array}\right] \quad |1\rangle = \left[\begin{array}{c}0 \\
1\end{array}\right]
\]

— the two possible states of a bit (Dirac’s notation).
Physics (again) making it happen...

![Diagram](image)

but this time it sounds far more challenging — particle spins, ion traps, ...

"(...) the implementation of quantum computing machines represents a formidable challenge to the communities of engineers and applied physicists." (Yanofsky and Mannucci, 2008)

IBM, Google, Microsoft are all investing a lot on such (quantum) physics!
Monads for quantum programming

*Vec A* represents the datatype of all complex-valued vectors with base *A*.

Thus *A → Vec B* is a function representing a **matrix** of type *A → B*.

In *QP* there is a restriction, thought: *f : A → Vec B* must represent a **unitary transformation**.

---

*A ℂ-valued matrix *U* is unitary iff *U · U† = U† · U = id*, where *U† is the **conjugate** transpose of *U*.

---

Compare with

\[ f · f° = f° · f = id \]

— **isomorphisms** are exactly the **classical** unitary transformations.
Quantamorphisms

*Vec A* is a monad whose Kleisli arrows are the matrices that we have seen before.

Everything goes smoothly when we interpret the diagrams before in the Kleisli, extending *biections* to *unitary transformations*.

We can encode the categorial operations monadically, as we know, namely the *tensor* product

\[ \otimes : (A \to \text{Vec } X) \to (B \to \text{Vec } Y) \to (A \times B) \to \text{Vec } (X \times Y) \]

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

Note that \( \text{return } a = |a\rangle \).
So we can encode “quantamorphisms” as monadic programs, for instance

\[
\alpha \circ \beta :: ((a, b) \to \text{Vec}(c, b)) \to ([a], b) \to \text{Vec}([c], b)
\]
\[
\alpha f \circ ([], b) = \text{return} ([], b)
\]
\[
\alpha f \circ (h : t, b) = \text{do} \begin{cases}
(t', b') & \leftarrow \alpha f \circ (t, b); \\
(h'', b'') & \leftarrow f(h, b'); \\
\text{return} (h'': t', b'')
\end{cases}
\]

It controls qubit \( b \) according to a list of classical bits using the quantum operator \( f \) (unitary). The outcome is unitary.
Suppose we use *bell* to control the input qubit (much superposition expected!). We may check what comes out, for instance, in *GHCi*:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>([0, 0, 0, 0], 0)</td>
<td>0.24999997</td>
</tr>
<tr>
<td>([1, 0, 0, 0], 0)</td>
<td>−0.24999997</td>
</tr>
<tr>
<td>([0, 1, 0, 0], 0)</td>
<td>−0.24999997</td>
</tr>
<tr>
<td>([1, 1, 0, 0], 0)</td>
<td>0.24999997</td>
</tr>
<tr>
<td>([0, 0, 1, 0], 0)</td>
<td>−0.24999997</td>
</tr>
<tr>
<td>([1, 0, 1, 0], 0)</td>
<td>0.24999997</td>
</tr>
<tr>
<td>([0, 1, 1, 0], 0)</td>
<td>0.24999997</td>
</tr>
<tr>
<td>([1, 1, 1, 0], 0)</td>
<td>−0.24999997</td>
</tr>
<tr>
<td>([0, 0, 0, 1], 0)</td>
<td>0.24999997</td>
</tr>
<tr>
<td>([1, 0, 0, 1], 0)</td>
<td>−0.24999997</td>
</tr>
<tr>
<td>([0, 1, 0, 1], 0)</td>
<td>−0.24999997</td>
</tr>
<tr>
<td>([1, 1, 0, 1], 0)</td>
<td>0.24999997</td>
</tr>
<tr>
<td>([0, 0, 1, 1], 0)</td>
<td>−0.24999997</td>
</tr>
<tr>
<td>([1, 0, 1, 1], 0)</td>
<td>0.24999997</td>
</tr>
<tr>
<td>([0, 1, 1, 1], 0)</td>
<td>0.24999997</td>
</tr>
<tr>
<td>([1, 1, 1, 1], 0)</td>
<td>−0.24999997</td>
</tr>
</tbody>
</table>

Instead of simulating, how does one “compile” $\langle \langle \text{bell} \rangle \rangle$ towards a quantum device?
How does it compile?

Tool-chain:

- **GHCi** — depending on the resources (number of qubits available), we select a range of values of the input that can be represented in such resources, generate the corresponding unitary matrix
- **Quipper** (Green et al., 2013) — generates the quantum circuit from such a matrix
- **QISKit** — Python interface to the hardware, adding error-correction extra circuitry
- **IBM-Q** — the actual hardware where QISKit runs its code.
IBM Q-experience devices

IBM Q 20 Tokyo [ibmq_20_tokyo]

IBM Q 20 Austin [QS1_1]

IBM Q 16 Rueschlikon [ibmqx5]

IBM Q 5 Tenerife [ibmqx4]

IBM Q 5 Yorktown [ibmqx2]
### Compiling for 5 qubits

Matrix sent to Quipper for `bell`:

<table>
<thead>
<tr>
<th></th>
<th>(0, 0)</th>
<th>(0, 1)</th>
<th>(1, 0)</th>
<th>(1, 1)</th>
<th>(0, 0, 0)</th>
<th>(0, 0, 1)</th>
<th>(0, 1, 0)</th>
<th>(0, 1, 1)</th>
<th>(1, 0, 0)</th>
<th>(1, 0, 1)</th>
<th>(1, 1, 0)</th>
<th>(1, 1, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>([1], 0)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>([1], 1)</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(0], 0)</td>
<td>0</td>
<td>0</td>
<td>1/\sqrt{2}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(0], 1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1/\sqrt{2}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>([0], 0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>([0], 1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>([1], 0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>([1], 1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Quantum circuit

First part of quantum circuit generated from the given program:

Simpler example — \( \langle \text{had} \otimes \text{id} \rangle \) compiles to a much simpler circuit:

(With thanks to: Ana Neri, Afonso Rodrigues, Rui S. Barbosa)
Running the circuits on IBM-Q

2 em máquina real:

```
In [34]: backend = 'ibmqx2'  # Backend where you execute your program
 circuits = ['Circuit']   # Group of circuits to execute
 shots = 1024             # Number of shots to run the program
 max_credits = 3          # Maximum number of credits to spend

qp.set_api(Qconfig.APItoken, Qconfig.config['url'])  # set up the API

result_real = qp.execute(circuits, backend, shots=shots, max_credits)

In [35]: result_real.get_counts('Circuit')


"00110" 32.0%
"00010" 27.8%
"00011" 9.0%
"00100" 7.8%
"00000" 7.7%
"00111" 6.9%
"00001" 4.6%
"00101" 4.1%
```

Each job performs 1000 runs for the given input provided and returns the outcome of the measurements, see aside.

Relatively high percentage of errors, still.
Wrapping up

Quantamorphisms — recursive quantum programming strategies dispensing with measurements.

Simpler semantics.

Emphasis on structural control. But the concept is still very experimental.

Towards correct by construction reversible/quantum programs.

Source: IBM Q Experience website
Wrapping up

Current MSc work by Ana Neri — “proof of concept”.

Experimental — needs a lot of work on both the theory and practical sides.

Carries further previous WG2.1 work in this field, recall e.g. *Quantum functional programming* by Mu and Bird (2001).

Many open questions, eg.

*How far can we go without measuring quantum states?*

Cf. *if _ then _ else ’s*...
Conditionals: to measure or not to measure...

Compare

\[
\begin{align*}
&\text{fig7.4 } h(p, q) = \text{do } \{ \\
&\quad q' \leftarrow \text{had } q; \\
&\quad p' \leftarrow \text{if } q' \\
&\quad \quad \text{then return } (\neg p) \\
&\quad \text{else had } p; \\
&\quad \text{return } (p', q') \\
&\}\end{align*}
\]

with

Conditional on the left does not interfere with the quantum effect — but, it is the same thing as measuring the state and taking decisions?

---

\footnote{Fig.7.4 of (Yanofsky and Mannucci, 2008), page 236.}
Annex
Proof of (10)

\[ \text{fst} = \langle \text{in} \rangle \]
\[ \Leftrightarrow \{ \text{(9)} \} \]
\[ \text{fst} \cdot \alpha = \text{in} \cdot (\text{id} + \text{id} \times \text{fst}) \]
\[ \Leftrightarrow \{ \text{in; coproducts} \} \]
\[ \text{fst} \cdot \alpha = [\text{nil}, \text{cons} \cdot (\text{id} \times \text{fst})] \]
\[ \Leftrightarrow \{ \text{definition of } \alpha \text{ and } a \} \]
\[ \text{true} \]
\[ \square \]
Annex — \( \langle \_ \rangle \) preserves injectivity

Let \( k = \langle f \rangle \). By the UP (9), \( k = f \cdot (F k) \cdot \alpha^o \). We calculate \( K = \ker k \) assuming \( \ker f = id \):

\[
K = k^o \cdot k
\]

\[\Leftrightarrow\] \{ unfold \( f \cdot F k \cdot \alpha^o \) \}

\[
K = \alpha \cdot F k^o \cdot f^o \cdot f \cdot F k \cdot \alpha^o
\]

\[\Leftrightarrow\] \{ assumption: \( f^o \cdot f = id \) \}

\[
K = \alpha \cdot F k^o \cdot F k \cdot \alpha^o
\]

\[\Leftrightarrow\] \{ \( F (R \cdot S) = (F R) \cdot (F S) \) and \( F R^o = (F R)^o \) \}

\[
K = \alpha \cdot F (k^o \cdot k) \cdot \alpha^o
\]

\[\Leftrightarrow\] \{ \( K = k^o \cdot k; \) UP (for relations) \}

\[
K = \langle \alpha \rangle
\]

\[\Leftrightarrow\] \{ Reflexion: \( \langle \alpha \rangle = id \) \}

\[
K = id
\]
Checking $g$ (12)

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

$$a^\circ ((id \times \text{fst}) \land (f \cdot (id \times \text{snd}))) (a, (x, b))$$

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

$$a^\circ ((a, x), f(a, b))$$

$$= \{ \text{associate to the righ isomorphism } a^\circ \}$$

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

□
Proof of (10)

\[ \text{fst} \cdot \alpha = [\text{nil}, \text{cons} \cdot \text{fst} \cdot \text{a}] \]

\[ \Leftrightarrow \{ (8) \} \]

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

\[ \Leftrightarrow \{ \text{+-absorption} \} \]

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

\[ \Leftrightarrow \{ \text{\textbf{in}} = [\text{nil}, \text{cons}]; \text{universal property (9)} \} \]

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

\[ \square \]
References


