# 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)

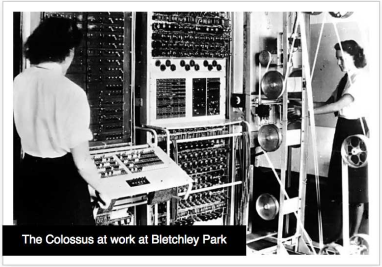
INESC TEC & University of Minho

# Context

QUANTA ⟨L|A|B⟩

- 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"** — **?**

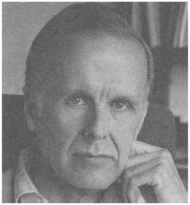
**Opportunity** —

**FP** folk apt for it, as I will try to show

# 1978–2018

## 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

# 1978–2018

Age of **MapReduce** predicted by John Backus

**MapReduce** in Backus' notation

$(/g) \cdot (\alpha\ f)$

**MapReduce** in "modern" notation

$(\!|g|\!) \cdot (\mathbf{fmap}\ f)$

However,

*we are spending too much* **energy** *in all this...*

**Thermodynamics** — LANDAUER'S PRINCIPLE ("logically **irreversible** manipulation of information leads to an increase in entropy").

# Reversible functions

**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** :

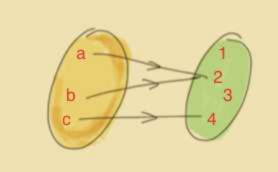
$$g \subseteq f^{\circ}$$

— converse of **functional** ($f^{\circ}$) is **injective** (and smaller than injective is injective).

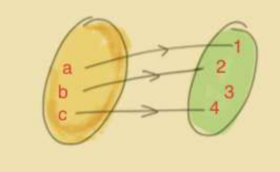
# Refine for injectivity

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

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




$\leqslant$




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 $f : A \to B$, define its **converse** as the relation $f^\circ : A \leftarrow B$ such that $a\ f^\circ\ b \Leftrightarrow b = f\ a$. Then

$$f \ \textbf{injective} \quad \Leftrightarrow \quad f\ x = f\ x' \Rightarrow x = x'$$

abbreviates to:

$$f^\circ \cdot f \ \subseteq \ id$$

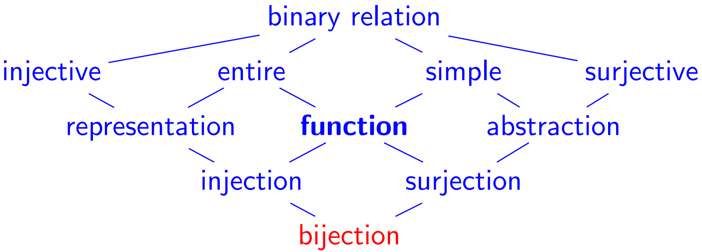
Moreover, $g$ **less injective** than $f$

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

simplifies to:

$$g \leqslant f \quad \Leftrightarrow \quad f^\circ \cdot f \ \subseteq \ g^\circ \cdot g$$

# The whole picture (relation 'bestiary')




where

$$R \textbf{ injective} \Leftrightarrow \underbrace{R^\circ \cdot R}_{\ker R} \subseteq id \qquad R \textbf{ simple} \Leftrightarrow R^\circ \textbf{ injective}$$

$$R \textbf{ entire} \Leftrightarrow id \subseteq \underbrace{R \cdot R^\circ}_{\mathrm{img}\ R} \qquad R \textbf{ surjective} \Leftrightarrow R^\circ \textbf{ entire}$$

# Relations as matrices

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

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

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,

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

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

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

$$R \triangledown S \leqslant X \quad\Leftrightarrow\quad R \leqslant X \wedge S \leqslant X \tag{1}$$

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

$$(b, c)\;(R \triangledown S)\;a \quad\Leftrightarrow\quad b\;R\;a \wedge c\;S\;a$$

In the case of functions:

$$(f \triangledown g)\;a = (f\;a, g\;a) \tag{2}$$

# Going (more) injective

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

$$R \leqslant R \vartriangledown S \quad \text{and} \quad S \leqslant R \vartriangledown S. \tag{3}$$

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

$$\ker\ (R \vartriangledown S) = (\ker\ R) \cap (\ker\ S) \tag{4}$$

itself a corollary of the more general:

$$(R \vartriangledown S)^\circ \cdot (Q \vartriangledown P) = (R^\circ \cdot Q) \cap (S^\circ \cdot P) \tag{5}$$

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

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

# Ordering functions by injectivity

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

$$! \leqslant f \leqslant id$$

where $1 \xleftarrow{\;!\;} A$ is the unique function of its type.

- A function is **injective** iff $id \leqslant f$. Thus $f \triangledown id$ is always **injective** (3).
- Two functions $f$ e $g$ are said to be **complementary** wherever $id \leqslant (f \triangledown g)$.[1]

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

[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)* $id \leqslant f \triangledown g$ ; *(b) if* $id \leqslant f \triangledown h$ *and* $h \leqslant g$ *then* $g \leqslant h$.

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

---

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

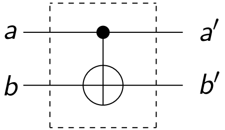
EXAMPLE: Non-injective $\mathbf{2} \xleftarrow{\dot{\vee}} \mathbf{2} \times \mathbf{2} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$ has minimal complement $\mathbf{2} \xleftarrow{fst} \mathbf{2} \times \mathbf{2} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$.

# Complementing $(\dot{\vee})$

As is well-known, by complementing $(\dot{\vee})$ with *fst*

$$\mathbf{2} \times \mathbf{2} \xleftarrow{fst \triangledown (\dot{\vee})} \mathbf{2} \times \mathbf{2} \ = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

we get a **bijection** — the classical **CNOT** quantum gate:



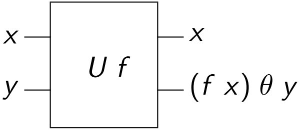

$$\begin{cases} cnot\ (0, b) = (0, b) \\ cnot\ (1, b) = (1, \neg\ b) \end{cases}$$

# Generic *fst*-complementation

Generalize $\dot{\vee}$ to monoid $(A; \theta, 0)$ such that:[2]

$$x \; \theta \; x = 0 \tag{6}$$

Then

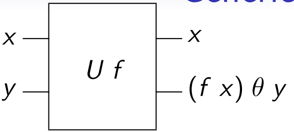



$$U\ f : (A \to B) \to (A \times B) \to (A \times B)$$
$$U\ f = fst \triangledown (\theta \cdot (f \times id))$$

is reversible for **any** $f : A \to B$.

[2] There should be a name for this but I can't remember it now.

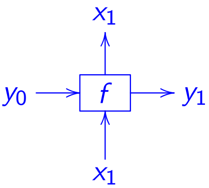
# Generic *fst*-complementation

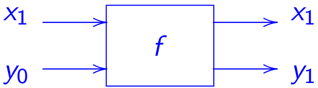



**bijective** because it is its self inverse:

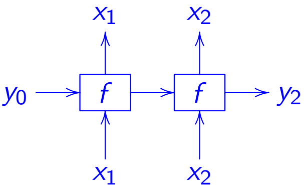
$$
\begin{array}{ll}
 & (U\ f) \cdot (U\ f) = id \\
\Leftrightarrow & \quad \{\ U\ f\ (x, y) = (x, (f\ x)\ \theta\ y)\ \} \\
 & U\ f\ (x, (f\ x)\ \theta\ y) = (x, y) \\
\Leftrightarrow & \quad \{\ \text{again } U\ f\ (x, y) = (x, (f\ x)\ \theta\ y)\ \} \\
 & (x, (f\ x)\ \theta\ ((f\ x)\ \theta\ y)) = (x, y) \\
\Leftrightarrow & \quad \{\ \theta \text{ is associative and } x\ \theta\ x = 0\ \} \\
 & (x, 0\ \theta\ y) = (x, y) \\
\Leftrightarrow & \quad \{\ 0\ \theta\ x = x\ \} \\
 & (x, y) = (x, y)
\end{array}
$$

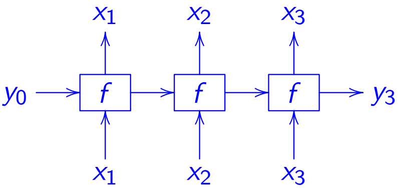
# Chaining *fst*-complemented computations

Drawing 

instead of 

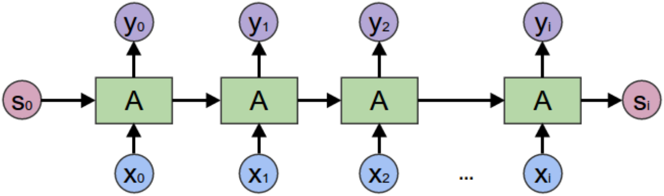

one may think of **chaining** such computations,



, 

, ...

# Similar construction in neural networks




(RNN = accumulating maps[3])

Yes — mapAccumR in Haskell

How is **injectivity** ensured?

[3]Source: **Neural Networks, Types, and Functional Programming** by C. Olah, 2015.

# The role of $A \xleftarrow{fst} A \times B$

*fst*-complementation,

$$id \leqslant fst \triangledown f$$

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

Can $(fst \triangledown \_)$ be extended **recursively**?

# Towards (constructive) recursive complementation

Suppose we want to offer **arbitrary** $k : A \to B$ 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 $A \times B \to A \times B$.

Now suppose $k$ is a **recursive** function, e.g. $k = \textbf{foldr}\ \overline{f}\ b$, for $f : A \times B \to B$, that is

$$k : A^* \to B$$
$$k\ [\,] = b$$
$$k\ (a : x) = f\ (a, k\ x)$$

How do we "constructively" build the corresponding (**recursive**, **bijective**) envelope of type $A^* \times B \to A^* \times B$?

# Going general (folds)

Let us define $(\!|f|\!)$ such that $k = \textbf{foldr}\ \overline{f}\ b\ x = (\!|f|\!)\ (x, b)$, that is:

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

Thus

$$
\begin{tikzcd}
A^* \times B \arrow[d, "(\!|f|\!)"'] & B + A \times (A^* \times B) \arrow[l, "\alpha"'] \arrow[d, "id + id \times (\!|f|\!)"] \\
B & B + A \times B \arrow[l, "{[id, f]}"]
\end{tikzcd}
$$

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**:

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

is the isomorphism

$$\alpha = [nil \triangledown id, (cons \times id) \cdot \mathbf{a}] \tag{7}$$

where

$$(A \times B) \times C \xleftarrow{\mathbf{a}} A \times (B \times C) \; = (id \times fst) \triangledown (snd \cdot snd) \tag{8}$$

Functions

$$nil\ \_ = [\,]$$
$$cons\ (a, x) = a : x$$

are the components of the initial algebra of lists $\mathbf{in} = [nil, cons]$.

# Universal property

We actually need something more general:

$$
\begin{tikzcd}
A^* \times B \arrow[d, "(\!|h|\!)"'] & B + A \times (A^* \times B) \arrow[l, "\alpha"'] \arrow[d, "id + id \times (\!|h|\!)"] \\
C & B + A \times C \arrow[l, "h"]
\end{tikzcd}
$$

**Universal** property

$$k = (\!|h|\!) \quad \Leftrightarrow \quad k \cdot \alpha = h \cdot \mathbf{F}\ k \tag{9}$$

where $\mathbf{F}\ f = id + id \times f$.

From (9,8) one infers

$$A^* \xleftarrow{fst} A^* \times B \ = (\!|\mathbf{in}|\!) \tag{10}$$

# Promoting complementation

Suppose **non-injective** $f : A \times B \rightarrow B$ is complemented by $fst : A \times B \rightarrow A$. The following diagram shows how to use **injective** $fst \triangledown f$ to build an envelope for **foldr** $\overline{f}\ b$:

$$
\begin{tikzcd}
A^* \times B \arrow[d, "\llparenthesis h \rrparenthesis"'] & & B + A \times (A^* \times B) \arrow[ll, "\alpha"'] \arrow[d, "id + id \times \llparenthesis h \rrparenthesis"] \\
A^* \times B & & B + A \times (A^* \times B) \arrow[ll, "h"] \arrow[dl, "\Phi\ (fst \triangledown f)"] \\
& B + A \times (A^* \times B) \arrow[ul, "\alpha"] &
\end{tikzcd}
$$

where

$$\Phi\ x = id + (\mathbf{xl} \cdot (id \times x) \cdot \mathbf{xl})$$

resorting to isomorphism $A \times (B \times C) \xrightarrow{\mathbf{xl}} B \times (A \times C)$ .

# Promoting complementation

Note that $fst \triangledown (\!|[id, f]|\!)$ also has type $A^* \times B \to A^* \times B$, recall

$$\begin{tikzcd} A^* \times B \arrow[d, "k"'] & B + A \times (A^* \times B) \arrow[l, "\alpha"'] \arrow[d, "id + id \times k = \mathbf{F}\ k"] \\ B & B + A \times B \arrow[l, "{[id\, ,f]}"] \end{tikzcd} \qquad k = (\!|[id, f]|\!)$$

How do

$$(\!|\alpha \cdot \Phi\ (fst \triangledown f)|\!) \quad \text{and} \quad fst \triangledown (\!|[id, f]|\!)$$

compare to each other?

Knowing by (10) that $fst = (\!|\mathbf{in}|\!)$ we appeal to the popular loop-intercombination law known as **"banana-split"**:

$$(\!|f|\!) \triangledown (\!|g|\!) = (\!|(f \times g) \cdot (\mathbf{F}\ fst \triangledown \mathbf{F}\ snd)|\!) \tag{11}$$

# Promoting complementation

We reason:

$$
\begin{array}{ll}
& fst \triangledown (\![ [id, f] ]\!) \\
= & \qquad \{ \text{ banana-split } \} \\
& (\![ (\mathbf{in} \times [id, f]) \cdot (\mathbf{F}\ fst \triangledown \mathbf{F}\ snd) ]\!) \\
= & \qquad \{ \text{ pairing laws (products) } \} \\
& (\![ [nil, cons \cdot (id \times fst)] \triangledown [id, f \cdot (id \times snd)] ]\!) \\
= & \qquad \{ \text{ exchange law } \} \\
& (\![ [nil \triangledown id, (cons \cdot (id \times fst)) \triangledown (f \cdot (id \times snd))] ]\!) \\
= & \qquad \{ \text{ products ; } \mathbf{a} \cdot \mathbf{a}^{\circ} = id \ \} \\
& (\![ \alpha \cdot \underbrace{\mathbf{a}^{\circ} \cdot (id \times fst) \triangledown (f \cdot (id \times snd))}_{\Phi\ (fst \triangledown f)} ]\!)
\end{array}
\qquad (12)
$$

# Promoting complementation

Thus

$$fst \triangledown (\![id, f]\!) = (\!\alpha \cdot (\Phi\ (fst \triangledown f))\!) \tag{13}$$

Clearly, $\Phi$ preserves injectivity, as does $(\!\_\!)$ (details in the appendix).

Summary:

$$fst \triangledown f \ \textbf{injective} \Rightarrow (\!\alpha \cdot (\Phi\ (fst \triangledown f))\!) \ \textbf{injective}$$

That is, $fst$-complementation of $f$ in $k = \textbf{foldr}\ \overline{f}\ b$ is promoted to the $fst$-complementation of the fold itself.

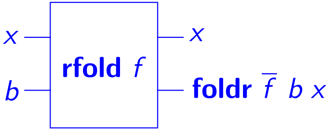
$fst$-complement **propagated** inductively.

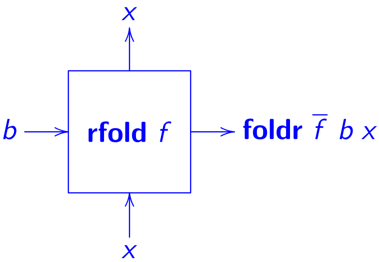
# In standard Haskell

In standard Haskell, we can rely on the reversibility of

$$\begin{aligned}
&\mathbf{rfold} \;:: (a \to b \to b) \to ([a], b) \to ([a], b)\\
&\mathbf{rfold}\; f\; ([\,], b) = ([\,], b)\\
&\mathbf{rfold}\; f\; (a : x, b) = (a : x, f\; a\; b)
\end{aligned}$$

provided $f$ is complemented by $fst$:



# Going quantum

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

$$\left\{ \begin{array}{l} \mathit{cnot}\ (0,b) = (0,b) \\ \mathit{cnot}\ (1,b) = (1,\neg\ b) \end{array} \right. \quad = \quad \begin{array}{c|cccc} & \rotatebox{90}{(0,0)} & \rotatebox{90}{(0,1)} & \rotatebox{90}{(1,0)} & \rotatebox{90}{(1,1)} \\ \hline (0,0) & 1 & 0 & 0 & 0 \\ (0,1) & 0 & 1 & 0 & 0 \\ (1,0) & 0 & 0 & 0 & 1 \\ (1,1) & 0 & 0 & 1 & 0 \end{array}$$

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

$$\begin{array}{c|cccc} & \rotatebox{90}{(0,0)} & \rotatebox{90}{(0,1)} & \rotatebox{90}{(1,0)} & \rotatebox{90}{(1,1)} \\ \hline (0,0) & 1 & 0 & 0 & 0 \\ (0,1) & 0 & \frac{1}{2} & 0 & 0 \\ (1,0) & 0 & \frac{1}{2} & 0 & 1 \\ (1,1) & 0 & 0 & 1 & 0 \end{array}$$

# Going quantum

Moving further to **quantum** corresponds to generalizing probabilities to **amplitudes**, for instance

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

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

# Quantum information


Credits: IBM Research AI & Q

# Going quantum

Quantum programs (**QP**) are made of elementary units called quantum **gates**, for instance the so-called **Hadamard** gate,

$$
\mathit{had} = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 1 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{array}
$$

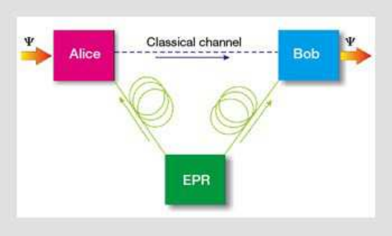
which is a component of the previous example.

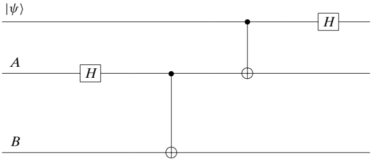
The approach is compositional, using two main combinators — **composition** ($\cdot$) and (tensor) **product** ( $\otimes$ ).

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

# Quantum abstraction

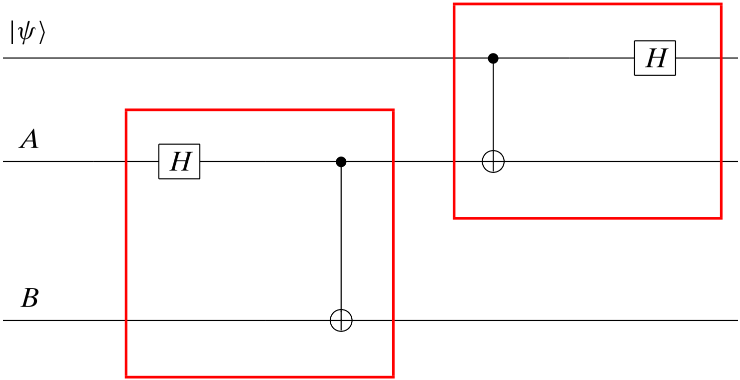
Bird's-eye view of the **structure** of a famous example (the "Alice" part of the teleportation protocol):



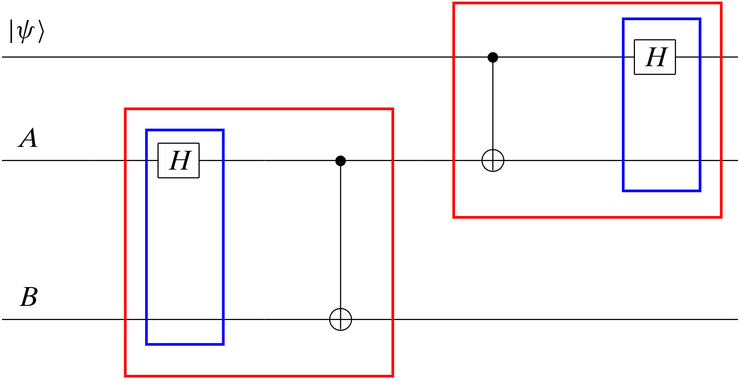




(Cf. **entangled** photon pairs)

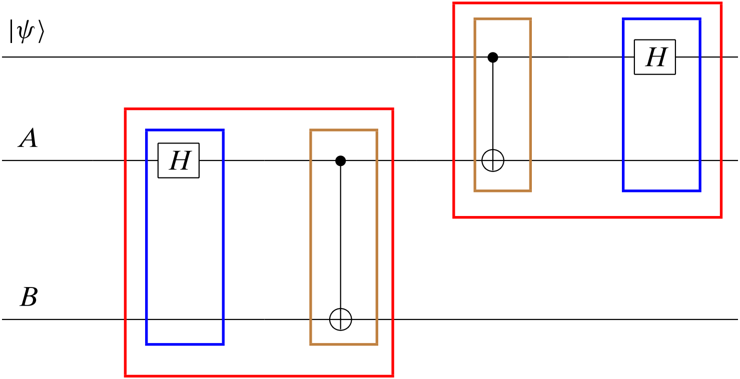
# Quantum abstraction



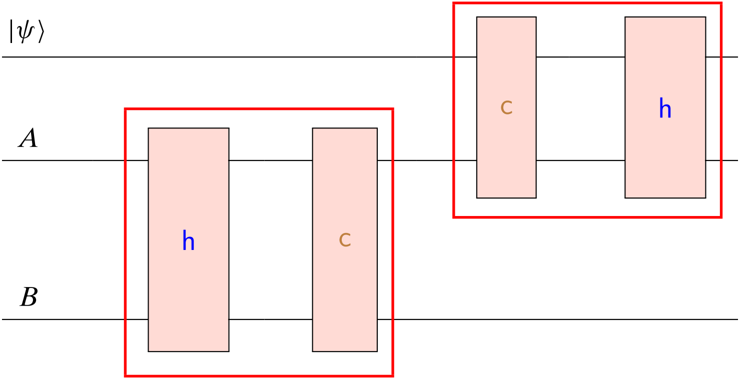
# Quantum abstraction



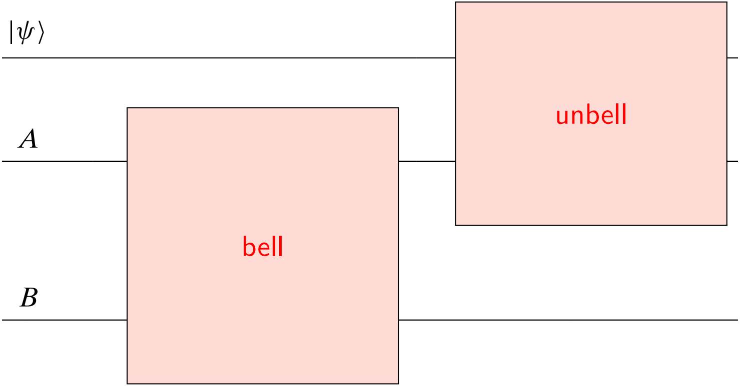
# Quantum abstraction



# Quantum abstraction



# Quantum abstraction




So

$$alice = (unbell \otimes id) \cdot \mathbf{a} \cdot (id \otimes bell) \tag{14}$$

where $bell = cnot \cdot (had \otimes id)$.

# Quantum abstraction (monadic)

It turns out that

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

can also be written

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

— just standard **monadic** programming

Where is the **quantum** part gone? Details next.

# Monads for quantum programming

Back to the **Hadamard** gate,

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

note that it can be written *pointwise* as

$$\begin{aligned} & had :: \mathbf{2} \rightarrow Vec\ \mathbf{2} \\ & had\ 0 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix} \\ & had\ 1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} \end{aligned}$$

or even as

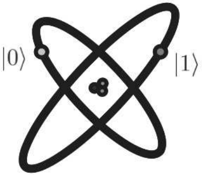
$$\begin{aligned} & had :: \mathbf{2} \rightarrow Vec\ \mathbf{2} \\ & had\ 0 = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \\ & had\ 1 = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \end{aligned}$$

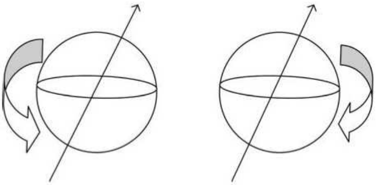
defining

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

— the two possible states of a bit (Dirac's notation).

# Physics (again) making it happen...



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 $A \rightarrow Vec\ B$ is a function representing a **matrix** of type $A \rightarrow B$.

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

---

*A $\mathbb{C}$-valued matrix $U$ is unitary iff $U \cdot U^{\dagger} = U^{\dagger} \cdot U = id$, where $U^{\dagger}$ is the* **conjugate** *transpose of $U$.*

---

Compare with

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

— **isomorphisms** are exactly the **classical** unitrary 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 **bijections** to **unitary transformations**.

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

$$
\begin{array}{l}
\otimes : (A \to Vec\ X) \to (B \to Vec\ Y) \to (A \times B) \to Vec\ (X \times Y) \\
(f \otimes g)\ (a, b) = \mathbf{do}\ \{ \\
\quad x \leftarrow f\ a; \\
\quad y \leftarrow g\ b; \\
\quad \mathbf{return}\ (x, y) \}
\end{array}
$$

Note that $\mathbf{return}\ a = |a\rangle$.

# Quantamorphisms

So we can encode *"quantamorphisms"* as monadic programs, for instance

$$
\begin{array}{l}
(\!|\cdot|\!) :: ((a,b) \rightarrow \mathit{Vec}\,(c,b)) \rightarrow ([a],b) \rightarrow \mathit{Vec}\,([c],b) \\
(\!|f|\!)\,([\,],b) = \textbf{return}\,([\,],b) \\
(\!|f|\!)\,(h:t,b) = \textbf{do}\,\{ \\
\quad (t',b') \leftarrow (\!|f|\!)\,(t,b); \\
\quad (h'',b'') \leftarrow f\,(h,b'); \\
\quad \textbf{return}\,(h'':t',b'') \\
\quad \}
\end{array}
$$

It controls **qubit** $b$ according to a **list** of classical bits using the **quantum** operator $f$ (unitary). The outcome is unitary.

# Quantamorphisms

Suppose we use *bell* to control the input qubit (much superposition expected!). We may check what comes out, for instance, in **GHCi**:

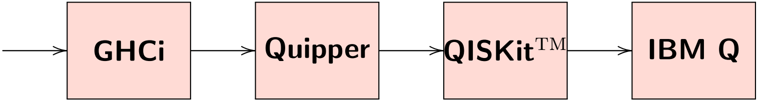
$$\llparenthesis\, \mathit{bell}\, \rrparenthesis\, ([0,1,1,1], 0) =$$

| | |
|---|---|
| ([0, 0, 0, 0], 0) | 0.24999997 |
| ([1, 0, 0, 0], 0) | −0.24999997 |
| ([0, 1, 0, 0], 0) | −0.24999997 |
| ([1, 1, 0, 0], 0) | 0.24999997 |
| ([0, 0, 1, 0], 0) | −0.24999997 |
| ([1, 0, 1, 0], 0) | 0.24999997 |
| ([0, 1, 1, 0], 0) | 0.24999997 |
| ([1, 1, 1, 0], 0) | −0.24999997 |
| ([0, 0, 0, 1], 0) | 0.24999997 |
| ([1, 0, 0, 1], 0) | −0.24999997 |
| ([0, 1, 0, 1], 0) | −0.24999997 |
| ([1, 1, 0, 1], 0) | 0.24999997 |
| ([0, 0, 1, 1], 0) | −0.24999997 |
| ([1, 0, 1, 1], 0) | 0.24999997 |
| ([0, 1, 1, 1], 0) | 0.24999997 |
| ([1, 1, 1, 1], 0) | −0.24999997 |

Instead of simulating, how does one "compile" $\llparenthesis\, \mathit{bell}\, \rrparenthesis$ towards a quantum device?

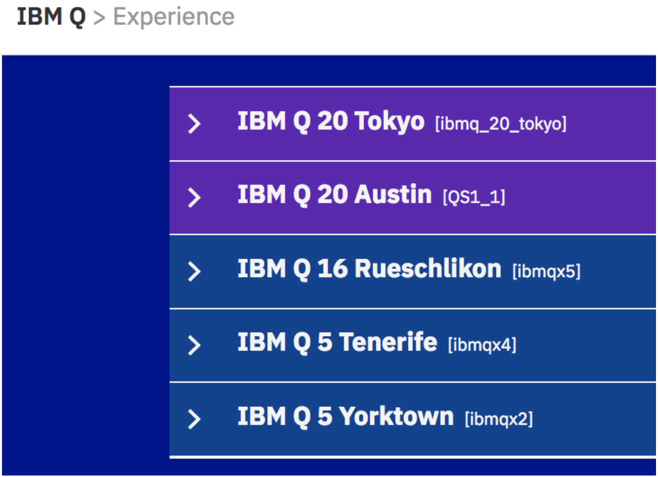
# How does it compile?

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



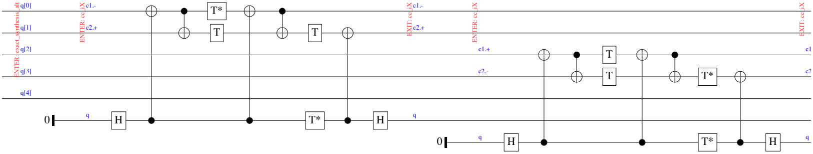
# Compiling for 5 qubits

Matrix sent to Quipper for $(\!|\ \mathit{bell}\ |\!)$:

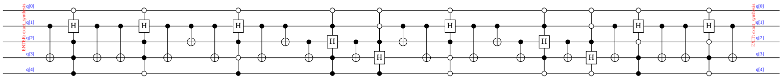
| | $([\,],0)$ | $([\,],1)$ | $([0],0)$ | $([0],1)$ | $([0,0],0)$ | $([0,0],1)$ | $([1,0],0)$ | $([1,0],1)$ | $([1],0)$ | $([1],1)$ | $([0,1],0)$ | $([0,1],1)$ | $([1,1],0)$ | $([1,1],1)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $([\,],0)$ | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0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

# Quantum circuit

First part of quantum circuit generated from the given program:




Simpler example — ⦇ *had* ⊗ *id* ⦈ compiles to a much simpler circuit:




*(With thanks to: Ana Neri, Afonso Rodrigues, Rui S. Barbosa)*

# Running the circuits on IBM-Q

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

Relatively high percentage of errors, still.

## 2 em máquina real:

```
In [34]: backend = 'ibmqx2'    # Backend where you execute your progr
         circuits = ['Circuit']    # Group of circuits to execute
         shots = 1024            # Number of shots to run the program
         max_credits = 3            # Maximum number of credits to spe
         qp.set_api(Qconfig.APItoken, Qconfig.config['url']) # set t

         result_real = qp.execute(circuits, backend, shots=shots, ma

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

Out[35]: {'00000': 79,
          '00001': 47,
          '00010': 285,
          '00011': 92,
          '00100': 80,
          '00101': 42,
          '00110': 328,
          '00111': 71}

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

# Wrapping up

**Quantamorphisms** — recursive quantum programming strategies dispensing with **measurements**.

Simpler semantics.

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


Source: IBM Q Experience website

Towards **correct by construction** reversible/quantum programs.

# 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).

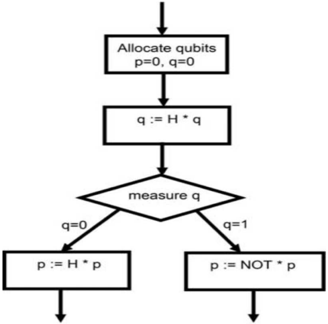
Many open questions, eg.

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

Cf. **if _ then _ else** 's...

# Conditionals: to measure or not to measure...

Compare[4]




with

$$
\begin{array}{l}
\mathit{fig7\_4}\ h\ (p,q) = \textbf{do}\ \{ \\
\quad q' \leftarrow \mathit{had}\ q; \\
\quad p' \leftarrow \textbf{if}\ q' \\
\qquad \textbf{then return}\ (\neg\ p) \\
\qquad \textbf{else}\ \mathit{had}\ p; \\
\quad \textbf{return}\ (p', q') \\
\quad \}
\end{array}
$$

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

[4]Fig.7.4 of (Yanofsky and Mannucci, 2008), page 236.

# Annex

# Proof of (10)

$$
\begin{array}{ll}
 & fst = (\!|\mathbf{in}|\!) \\
\Leftrightarrow & \qquad \{\ (9)\ \} \\
 & fst \cdot \alpha = \mathbf{in} \cdot (id + id \times fst) \\
\Leftrightarrow & \qquad \{\ \mathbf{in}\text{; coproducts}\ \} \\
 & fst \cdot \alpha = [nil, cons \cdot (id \times fst)] \\
\Leftrightarrow & \qquad \{\ \text{definition of } \alpha \text{ and } \mathbf{a}\ \} \\
 & true
\end{array}
$$

$\Box$

# Annex — $(\!|\_|\!)$ preserves injectivity

Let $k = (\!|f|\!)$. By the UP (9), $k = f \cdot (\mathbf{F}\ k) \cdot \alpha^\circ$. We calculate $K = \mathrm{ker}\ k$ assuming $\mathrm{ker}\ f = id$:

$$
\begin{array}{ll}
 & K = k^\circ \cdot k \\
\Leftrightarrow & \quad \{\ \text{unfold } f \cdot \mathbf{F}\ k \cdot \alpha^\circ\ \} \\
 & K = \alpha \cdot \mathbf{F}\ k^\circ \cdot f^\circ \cdot f \cdot \mathbf{F}\ k \cdot \alpha^\circ \\
\Leftrightarrow & \quad \{\ \text{assumption: } f^\circ \cdot f = id\ \} \\
 & K = \alpha \cdot \mathbf{F}\ k^\circ \cdot \mathbf{F}\ k \cdot \alpha^\circ \\
\Leftrightarrow & \quad \{\ \mathbf{F}\ (R \cdot S) = (\mathbf{F}\ R) \cdot (\mathbf{F}\ S) \text{ and } \mathbf{F}\ R^\circ = (\mathbf{F}\ R)^\circ\ \} \\
 & K = \alpha \cdot \mathbf{F}\ (k^\circ \cdot k) \cdot \alpha^\circ \\
\Leftrightarrow & \quad \{\ K = k^\circ \cdot k;\ \text{UP (for relations)}\ \} \\
 & K = (\!|\alpha|\!) \\
\Leftrightarrow & \quad \{\ \text{Reflexion: } (\!|\alpha|\!) = id\ \} \\
 & K = id
\end{array}
$$

# Checking $g$ (12)

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

$$
\begin{aligned}
& \mathbf{a}^{\circ}\ ((id \times fst) \triangledown (f \cdot (id \times snd))\ (a,(x,b)) \\
= & \quad \{\ \text{composition; } fst \text{ and } snd \text{ projections}\ \} \\
& \mathbf{a}^{\circ}\ ((a,x), f\ (a,b)) \\
= & \quad \{\ \text{associate to the righ isomorphism } \mathbf{a}^{\circ}\ \} \\
& (a,(x,f\ (a,b)))
\end{aligned}
$$

$\square$

# Proof of (10)

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

$$\Leftrightarrow \qquad \{ \ (8) \ \}$$

$$fst \cdot \alpha = [nil, cons \cdot (id \times fst)]$$

$$\Leftrightarrow \qquad \{ \ +\text{-absorption} \ \}$$

$$fst \cdot \alpha = [nil, cons] \cdot (id + id \times fst))$$

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

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

$\square$

# References

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM TDS*, 6(4):557–575, December 1981.

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

P.J. Freyd and A. Scedrov. *Categories, Allegories*, volume 39 of *Mathematical Library*. North-Holland, 1990.

A.S. Green, P.L. Lumsdaine, N.J. Ross, P. Selinger, and B. Valiron. An introduction to quantum programming in Quipper. *CoRR*, cs.PL(arXiv:1304.5485v1), 2013.

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

S.C. Mu and R. Bird. Quantum functional programming, 2001. 2nd Asian Workshop on Programming Languages and Systems, KAIST, Dajeaon, Korea, December 17-18, 2001.

N.S. Yanofsky and M.A. Mannucci. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008. doi: 10.1017/CBO9780511813887.