Towards quantamorphisms — some thoughts on (constructive) reversibility

NII, Tokyo

23rd of January, 2018

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

INESC TEC & University of Minho
Thanks to NII!

Trip planned long ago... 😊

Long friendship — Zhenjiang, can you remember GTTSE’05?

We had met before — cf. (Mu et al., 2004), which relates to this talk!

Summer School on
Generative and Transformational Techniques
in Software Engineering
4 - 8 July, 2005, Braga, Portugal
http://www.di.uminho.pt/GTTSE2005
Computing versus energy

Thermodynamics view of computing.

Green computing calling for less energy consumption.

Landauer’s principle: irreversible computation accounted for energy consumption (entropy).
Information is physical

**Physics of information** — a branch of science.

**Quantum computing** — a quantum mechanics view of computation (	extit{bijective} transformations $\rightarrow$ \textit{unitary} transformations).

**Bidirectional** programming (BX)

**AIM** — achieve reversible / quantum programming constructively.

Inspiration from **functional** programming.

Ut facient opus signa

("Let symbols do the work")

[...] by the aid of symbolism, we can make transitions in reasoning almost **mechanically** by the eye

[...] Civilisation advances by extending the number of important operations which can be performed **without thinking** about them.”

(Alfred Whitehead, 1911)
Start from BX (total, functional)

**GetPut:**

\[ \text{get} \cdot \text{put} = \text{fst} \]  \hspace{1cm} (1)

**PutGet:**

\[ \text{put} \cdot (\text{get} \downarrow \text{id}) = \text{id} \]  \hspace{1cm} (2)

**Composition** combinator:

\[ (f \cdot g) x = f (g x) \]

**Identity:**

\[ \text{id} \, x = x \]

**Pairing** combinator:

\[ (f \, \downarrow \, g) \, x = (f \, x, \, g \, x) \]

**Projections:**

\[ \text{fst} \, (a, \, b) = a \quad \text{snd} \, (a, \, b) = b \]
Calculating properties of $get + put$

**PutGet** ensures that $put$ is **surjective**,

$$\forall s :: (\exists v, s' :: s = put(v, s'))$$

since $\underbrace{f \cdot g}_{\text{surjective}} = id$ in general.

Moreover, $get$ is also **surjective** and **uniquely** determined by $put$. Why and how?

To answer these questions we have to do our first generalization:

“(…) like the move from real numbers to complex ones, the move [from functions] to relations increases our powers of expression” (Bird and de Moor, 1997)
Calculating properties of \(\text{get} \perp \text{put}\)

We generalize \(y = f \cdot x\) to \(y \circ R \cdot x\), and use the same arrows to denote both, e.g. \(\xymatrix{X \ar[r]^f & Y} \quad \text{and} \quad \xymatrix{X \ar[r]^R & Y}\).

Some people like writing \(y \circ R \cdot x \iff (y, x) \in R\), but we simply read \(y \circ R \cdot x\) as “it is true that \(y\) is related to \(x\) by \(R\)”; or simply, “\(y \circ R \cdot x\) holds”.

\textit{John Loves Mary}. \(2 < 3\). As simple as that.

To say that \textit{Mary} is loved by \textit{John} simply write \textit{Mary Loves} \(\circ\) \textit{John}.

In general: \(y \circ R \cdot x \iff x \circ R^\circ \cdot y\) — this is the \textit{converse} operation, or \textit{passive voice}:

\[
(R \cdot S)^\circ = S^\circ \cdot R^\circ \\
id^\circ = id
\]

\textbf{Composition} generalizes to \(y \circ (R \cdot S) \cdot x \iff (\exists z :: y \circ R \cdot z \land z \circ S \cdot x)\).
Calculating properties of \( \text{get} + \text{put} \)

The other ingredient of the generalization is that relations are ordered by a partial order, \( R \subseteq S \iff (\forall y, x :: y R x \Rightarrow y S x) \).

Functions are the only relations \( f, g \) such that the following hold:

\[
\begin{align*}
    f \cdot R \subseteq S & \iff R \subseteq f^\circ \cdot S \\
    f \subseteq g & \iff f = g \iff g \subseteq f
\end{align*}
\]  

Conventions: functions in lowercase, general relations in uppercase.

Taking converses,

\[
R \cdot f^\circ \subseteq S \iff R \subseteq S \cdot f
\]  

also holds. Why do functions enjoy such nice shunting rules?
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) $\neg = \begin{array}{c|cc} 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 0 \end{array}$

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

and so on.

**Functions** have exactly one 1 in every column.

**Bijections** have exactly one 1 in every column and in every row.
Functions, bijections, etc

Thus

\[ f \text{ function} \iff \text{img } f \subseteq \text{id} \land \text{id} \subseteq \text{ker } f \]

\[ f \text{ bijection} \iff f^\circ \text{ function} \iff \text{img } f = \text{id} \land \text{id} = \text{ker } f \]

These are the properties ensure the rules given earlier for functions.
By such rules, \texttt{GETPUT} re-writes to

\[
\text{get} \cdot \text{put} = \text{fst} \iff \begin{cases} 
\text{put} \subseteq \text{get}^\circ \cdot \text{fst} \\
\text{fst} \cdot \text{put}^\circ \subseteq \text{get}
\end{cases}
\]

and \texttt{PUTGET} to

\[
\text{put} \cdot (g \uplus \text{id}) = \text{id} \iff g \uplus \text{id} \subseteq \text{put}^\circ
\]

From this we infer:

- \textit{get} is \textbf{surjective} — because \text{put}^\circ and \text{fst} are so, and thumb rule: \textit{larger than surjective is surjective}.

- \textit{put} \textbf{determines get} — if some other \textit{get}' exists, \textit{get} = \textit{get}' — next slide.
**put determines get**

\[
\text{true} \\
\Leftrightarrow \quad \{ \text{PUTGET of new } get' \} \\
\text{put} \subseteq get' \circ \operatorname{fst} \\
\Rightarrow \quad \{ \text{monotonicity} \} \\
\text{put} \cdot (get \downarrow id) \subseteq get' \circ \operatorname{fst} \cdot (get \downarrow id) \\
\Leftrightarrow \quad \{ \text{PUTGET of first } get \} \\
id \subseteq get' \circ \operatorname{fst} \cdot (get \downarrow id) \\
\Leftrightarrow \quad \{ \text{shunting, } \operatorname{fst} \cdot (f \downarrow g) = f \} \\
get' \subseteq get \\
\Leftrightarrow \quad \{ \text{function equality} \} \\nget' = get \\
\square
\]
Bad *puts*...

However, some *puts* have no *get*. Why?

Recall **GetPut** in version

\[ \text{fst} \cdot \text{put}^\circ \subseteq \text{get} \]

As *get* is **simple**, and *smaller than simple is simple*, \( \text{fst} \cdot \text{put}^\circ \) has to be simple too:

\[ \text{fst} \cdot \text{put}^\circ \text{ simple} \]

\[ \iff \{ \text{R simple} \iff R \cdot R^\circ \subseteq \text{id} \} \]

\[ \text{fst} \cdot \text{put}^\circ \cdot \text{put} \cdot \text{fst}^\circ \subseteq \text{id} \]

\[ \iff \{ \text{shunting rules} \} \]

\[ \text{put}^\circ \cdot \text{put} \subseteq \text{fst}^\circ \cdot \text{fst} \]

\[ \iff \{ \text{injectivity preorder: } R \leq S \iff \ker S \subseteq \ker R \} \]

\[ \text{fst} \leq \text{put} \]
put more injective than fst (sorry!)

Counter-example:

Is exclusive-or

\[(\vee) : 2 \times 2 \rightarrow 2\]

\[(\vee) = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}\]

a good put? No! — just compute

\[fst \cdot (\vee^\circ) = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \top\]

and observe that it is not simple.\(^1\)

\(^1\)We denote by \(B \leftarrow^\top A\) the largest relation of type \(B \leftarrow A\).
put more injective than \textit{fst} (sorry!)

The same counter-example using the injectivity preorder:

\[ \text{fst} \preceq (\dot{\lor}) \]

\[
\iff \quad \{ \quad R \preceq S \iff \ker S \subseteq \ker R \quad \}
\]

\[ \ker (\dot{\lor}) \subseteq \ker \text{fst} \]

\[
\iff \quad \{ \quad \text{kernel matrices} \quad \}
\]

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

\[
\iff \quad \{ \quad \text{pointwise inclusion} \quad \}
\]

\textit{false}
How to design a (good) put?

To obtain a good \( \text{put} : V \times S \rightarrow S \),

- **refine** \( \text{fst} : V \times S \rightarrow V \) according to the **injectivity preorder** — i.e. find \( \text{put} \) s.t. \( \text{fst} \leq \text{put} \).

- Then obtain \( \text{get} : S \rightarrow V \) by computing \( \text{fst} \cdot \text{put}^\circ \).

**Example:** starting point for a good \( 2 \times 3 \xrightarrow{\text{put}} 3 \) is

\[
\ker \left( 2 \xleftarrow{\text{fst}} 2 \times 3 \right) = \begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
\end{bmatrix}
\]
Designing a good *put*

Note that \( \ker \; \text{put} \) must have 3 equivalence classes (\( \#S = 3 \)) because \( \text{put} \) is surjective.

Since \( \ker \; \text{fst} \) has 2 equivalence classes (\( \text{fst} \) surjective, \( \#V = 2 \)), the best we can do is to split one of these in two, eg.

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

that is:

\[
\text{put} = \begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\end{bmatrix}
\]
Designing a good \textit{put}

As this is a \textbf{good put by construction}, its \textit{get} is immediately calculated:\footnote{Note that \textit{fst} \cdot \textit{put}$^\circ$ is always \textbf{entire} because \textit{put} is \textbf{surjective}.}

\[
\text{get} = \text{fst} \cdot \text{put}^\circ = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
\]

That is:

\begin{align*}
\text{put} (a, 1) &= 1 \\
\text{put} (a, 2) &= \text{put} (a, 3) = 2 \\
\text{put} (b, \_ ) &= 3
\end{align*}

\begin{align*}
\text{get} 1 &= \text{get} 2 = a \\
\text{get} 3 &= b
\end{align*}

(We make \( V = \{a, b\} \) just for visualizing \( V \) and \( S \) differently.)
Designing a good *put*

**Exercise:** How many good *puts* there are of type $3 \times 4 \rightarrow 4$? And what is the corresponding *get*? Start from

\[
ker \left( 3 \xleftarrow{\text{fst}} 3 \times 3 \right) = \begin{bmatrix}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1
\end{bmatrix}
\]

and refine.
Going partial

As in (Ko and Hu, 2018), BX become more general once we drop totality (entireness).

Thus *put* and *get* become just simple relations (\(= \) partial functions) \(P\) and \(G\) with \(\text{GetPut}+\text{PutGet}\)

\[
P \subseteq G^\circ \cdot \text{fst}
\]

\[
G \upharpoonright \text{id} \subseteq P^\circ
\]

by immediate generalization of what we had before:

\[
\text{put} \subseteq \text{get}^\circ \cdot \text{fst}
\]

\[
\text{get} \upharpoonright \text{id} \subseteq \text{put}^\circ
\]

Here is how \(\text{GetPut}+\text{PutGet}\) (6,7) read with variables:

\[
s' P (v, s) \Rightarrow v G s'
\]

\[
v G s \Rightarrow s P (v, s)
\]
Going (more) injective

As we did with $\text{fst} \leq \text{put}$, we are now interested in further 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 — the way to reversibility.

This ordering is rich in properties, for instance it is upper-bounded\(^3\)

$$R \updownarrow S \leq X \iff R \leq X \land S \leq X$$

\(^3\)Details in (Oliveira, 2014). \textbf{NB}: pairing generalizes to relations in the expected way: $(b, c) (R \updownarrow S) a \iff b R a \land c S a$. 
Going (more) injective

Therefore, by cancellation of (8), we have that **pairing** always increases injectivity:

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

(9)

The inclusion \( \ker (R \triangleleft S) \subseteq (\ker R) \cap (\ker S) \) is in fact an equality

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

itself a corollary of the more general:

\[ (R \triangleleft S)^\circ \cdot (Q \triangleleft P) = (R^\circ \cdot Q) \cap (S^\circ \cdot P) \]  

(10)

Injectivity **shunting laws** also exist, e.g.

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

Restricted to functions, \((\leq)\) is \textit{universally} bounded by

\[ ! \leq f \leq id \]

where \(1 \leftarrow ! A\) is the unique function of its type. (1 is the singleton type.) Moreover,

- A function is \textit{injective} iff \(id \leq f\)

Thus \(f \downarrow id\) is always \textit{injective} (9).

- Two functions \(f \; e \; g\) are said to be \textit{complementary} wherever \(id \leq (f \downarrow g)\).

For instance, \textit{fst} and \textit{snd} are complementary since \(fst \downarrow snd = id\).

\[\text{4Cf. (Matsuda et al., 2007). Other terminologies are monic pair (Freyd and Scedrov, 1990) or jointly monic (Bird and de Moor, 1997).}\]
Minimal complements — Suppose (a) \( id \leq f \triangleright g \); (b) if \( id \leq f \triangleright h \) and \( h \leq g \) then \( g \leq h \).

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

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

**Example:** Non-injective \( 2 \leftarrow \triangleright 2 \times 2 = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \) has minimal complement \( 2 \leftarrow \text{fst} \ 2 \times 2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \).

How can we be sure it is minimal?
Minimal complements

We start from

$$\ker (\dot{\lor}) = \ker \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$$

Clearly, \( \ker g \) has to cancel all 1’s that fall outside the diagonal,

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

but this is an overkill — \( g = id \) in this case!

We can add 1’s where \( \ker (\dot{\lor}) \) has 0’s, e.g.

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

but this isn’t a kernel anymore — why?
Minimal complements

Kernels of functions are *equivalence relations* — *reflexive* (cf. diagonal), *symmetric* and *transitive*.

How do we ensure this?

By ensuring that the matrix depicts a *rational*, or *difunctional* relation:

\[ R \text{ is difunctional iff } R \cdot R^\circ \cdot R \subseteq R. \]

**Fact:** a symmetric+reflexive relation is an *equivalence* iff it is difunctional.

One can construct *difunctional* relations easily: just make sure that columns either don’t intersect or are the same.
Ensuring difunctionality

Cancel zeros symmetrically, outside the diagonal:

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

Alternatively:

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

So, both \ fst and \ snd are minimal complements of \ \dot{\vee}. 
Complementing ($\dot{\vee}$)

What do we get by complementing ($\dot{\vee}$) with $fst$:

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

This is a well-known bijection, in fact a (classical) quantum gate known as CNOT (for "controlled not") and depicted as follows:

Why does it bear this name?
Complementing (\(\dot{\vee}\))

\[ cnot = \text{fst} \downarrow (\dot{\vee}) \]

\[ \Leftrightarrow \quad \{ \text{pointwise} \} \]

\[ cnot (a, b) = (a, a \dot{\vee} b)) \]

\[ \Leftrightarrow \quad \{ \text{since } 0 \dot{\vee} b = b \text{ and } 1 \dot{\vee} b = \neg b \} \]

\[ \left\{ \begin{array}{l}
    cnot (0, b) = (0, b) \\
    cnot (1, b) = (1, \neg b)
\end{array} \right. \]

Informally: **controlled** bit \(b\) is negated *iff* the **control** bit \(a\) is set.

Thus we have a **constructive** approach to designing this gate — we build it by **minimal** complementation. (Not the standard interpretation!)
Other $fst$-complementations

Take the classical circuit

\[ f = \overline{\cdot} \cdot (\wedge \otimes id) \rightarrow 2^2 = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix} \]

Can it be made into a bijection in the same way?

Let us complement it with \( 2^2 \times 2 \xrightarrow{fst} 2^2 \) again. (Next slide.)
Other \textit{fst}-complementations

We get another \textit{bijection}, known as the \textbf{CCNOT} gate 😊:

\begin{align*}
  \text{ccnot} &= \text{fst} \circ (\vee \cdot (\wedge \otimes \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}
\end{align*}

\textbf{ccnot} : \(2^2 \times 2 \rightarrow 2^2 \times 2\)

\begin{align*}
  \text{ccnot} \ (\ (1, 1), c \ ) &= \ (\ (1, 1), \neg \ c \ ) \\
  \text{ccnot} \ (\ (a, b), c \ ) &= \ (\ (a, b), c \ )
\end{align*}
Other $fst$-complementations

A famous device in quantum programming is the following evolution of the $CNOT$ gate,

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

parametric on $2 \xrightarrow{f} 2$:

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

Clearly, $cnot = U \text{id}$. 
Other \textit{fst}-complementations

\begin{equation}
(U f) \cdot (U f) = id
\end{equation}

\iffalse
\begin{align*}
(U f) \cdot (U f) &= id \\
\Rightarrow & \{ U f (x, y) = (x, f x \hat{\vee} y) \}
\end{align*}
\fi

\begin{align*}
U f (x, f x \hat{\vee} y) &= (x, y) \\
\Rightarrow & \{ \text{again } U f (x, y) = (x, f x \hat{\vee} y) \}
\end{align*}

\begin{align*}
(x, f x \hat{\vee} (f x \hat{\vee} y)) &= (x, y) \\
\Rightarrow & \{ \hat{\vee} \text{ is associative and } x \hat{\vee} x = 0 \}
\end{align*}

\begin{equation}
(x, 0 \hat{\vee} y) = (x, y)
\end{equation}

\begin{align*}
\Rightarrow & \{ 0 \hat{\vee} x = x \}
\end{align*}

\begin{equation}
(x, y) = (x, y)
\end{equation}

\square
Pause

What have we achieved thus far?

A **constructive** approach to **reversibility** — instead of accepting (e.g. quantum gates as) “inventions”, we start (**functionally**) from the functions that we want to make available, e.g.

```
a
b
---
c
  z
```

and then **refine** them into reversible programs by pairing them with minimal **complements**.

That is, the original gate is taken as **specification**, the reversible one as **implementation**.

Never forget to **program from specifications** 😊 (Morgan, 1990).
The role of $A \xleftarrow{fst} A \times B$

We have seen that $A \xleftarrow{fst} A \times B$ plays a prominent role in the calculations thus far.

The starting point for calculating $S \xleftarrow{put} V \times S$

$fst \leq put$

is a property known as the semi-injectivity of $put$ (Foster et al., 2007):

$$put(a, c) = put(a', c') \Rightarrow a = a'$$

(Just unfold $fst \leq put$ and go pointwise.)

$fst$ is often a good minimal complement — can $(fst \downarrow \_)$ be extended recursively?
Our examples have been fortunate in the sense that projection
$A \times B \xrightarrow{\text{fst}} A$ was paired with a function of type
$A \times B \rightarrow B$, making room for a bijection of type
$A \times B \rightarrow A \times B$.

Suppose we want to offer arbitrary $f : A \rightarrow B$ in a bijective
“envelope” (that’s what reversible/quantum computing is all about).

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

Now suppose $f$ is a recursive function, e.g. $f = \text{foldr } g \ b$. 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 $\langle f \rangle (x, b) = \text{foldr} \overline{f} b x$ where $\overline{f} a b = f (a, b)$:

\[
\begin{align*}
\langle f \rangle ([], b) &= b \\
\langle f \rangle (a : x, b) &= f (a, \langle f \rangle (x, b))
\end{align*}
\]

Thus

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

NB:

\[X + Y = \{ i_1 x \mid x \in X \} \cup \{ i_2 y \mid y \in Y \}\]

is disjoint union of $X$ and $Y$ — thanks to $i_1 \cdot i_2^\circ = \bot$ — and $[R, S]$ is the unique relation $X$ such that $X \cdot i_1 = R$ and $X \cdot i_2 = S$. 
Going general \((\mathbb{N}_0)\)

Let us start from a simpler fold, that over natural numbers (for \(f \ i \ n = f^n \ i\)):

\[
\begin{align*}
\text{for } f \ i \ 0 &= i \\
\text{for } f \ i \ (n + 1) &= f \ (\text{for } f \ i \ n)
\end{align*}
\]

Via the same procedure, this becomes

\[
\begin{align*}
\mathbb{N}_0 \times B &\xleftarrow{\alpha} B + \mathbb{N}_0 \times B \\
\langle f \rangle &\downarrow \quad \downarrow \text{id} + \langle f \rangle \\
C &\xleftarrow{f} B + C
\end{align*}
\]

where (constant functions are denoted by \(k \times = k\)):

\[
\alpha = [0 \downarrow \text{id}, \text{succ} \times \text{id}] = [0, \text{succ} \cdot \text{fst}] \uparrow [\text{id}, \text{snd}]
\]
Going general \((\mathbb{N}_0)\)

Universal property (UP):

\[ k = \langle f \rangle \iff k \cdot \alpha = f \cdot (id + k) \quad (11) \]

Reflexion: \(\langle \alpha \rangle = \text{id}\); Projection:

\[ \text{fst} \cdot \alpha = [0, \text{succ} \cdot \text{fst}] \]
\[ \iff \{ \text{fusion}+ \} \]
\[ \text{fst} \cdot \alpha = [0, \text{succ}] \cdot (id + \text{fst}) \]
\[ \iff \{ \text{universal property} \} \]
\[ \text{fst} = \langle [0, \text{succ}] \rangle \]

\[\Box\]

Complementation \(\mathbb{N}_0 \times B \overset{\langle [0, \text{succ}] \rangle \lor \langle [id, f] \rangle}{\leftarrow} \mathbb{N}_0 \times B\) brings "banana-split" to mind...
Banana-split

As with standard folds (catamorphisms) the “banana-split” rule states:

\[ \langle f\rangle \bowtie \langle g\rangle = \langle (f \cdot (id + \text{fst})) \bowtie (g \cdot (id + \text{snd})) \rangle \]

For any \( f : B \to B \), let us define

\[ N_0 \times B \xleftarrow{\Psi f} N_0 \times B = \text{fst} \bowtie \langle [id, f] \rangle \]

That is, \( \Psi f (n, b) = (n, f^n b) \) is a for-loop which keeps its input. We will show that it preserves injectivity.

First of all, we calculate \( \Psi f \) following the standard style. (Next slide.)
Calculating $\Psi f$

\[
\Psi f
= \begin{cases}
\Psi f = \mathit{fst} \uplus \llbracket \mathit{id}, f \rrbracket ; \ \text{reflexion} \\
\llbracket 0, \mathit{succ} \rrbracket \uplus \llbracket \mathit{id}, f \rrbracket \\
\llbracket 0, \mathit{succ} \cdot \mathit{fst} \rrbracket \uplus [\mathit{id}, f \cdot \mathit{snd}] \\
\llbracket 0 \uplus \mathit{id}, \mathit{succ} \times f \rrbracket
\end{cases}
\]

From $\Psi f = \llbracket 0 \uplus \mathit{id}, \mathit{succ} \times f \rrbracket$ we derive, by the UP:

\[
\Psi f \cdot (0 \uplus \mathit{id}) = 0 \uplus \mathit{id}
\]
\[
\Psi f \cdot (\mathit{succ} \times \mathit{id}) = (\mathit{succ} \times f) \cdot \Psi f
\]
\[ \Psi \text{ preserves injectivity} \]

First note that \([0 \uparrow id, \text{succ} \times f]\) is \textbf{injective} iff \(f\) is injective, by the following rule:

\[ [R, S] \text{ injective iff both } R, S \text{ injective and } R^\circ \cdot S \subseteq \bot. \]

(Note that \(0^\circ \cdot \text{succ} \subseteq \bot\) since there is no \(n \in \mathbb{N}_0\) such that \(\text{succ } n = 0\).)

Therefore, to show that \(\Psi f = \llbracket [0 \uparrow id, \text{succ} \times f] \rrbracket\) preserves \textbf{injectivity} it is enough to show that \(\llbracket \_ \rrbracket\) does so:

\[ f \text{ injective } \Rightarrow \llbracket f \rrbracket \text{ injective} \quad (12) \]

(Proof in the annex.)
Moving to a truly quantum setting

Quoting (Mu et al., 2004):

The motivation to study languages for reversible programs traditionally comes from the thermodynamics view of computation.

What about quantum programming (QP)?

In QP we actually rely on quantum mechanics to run our programs. How can this be?

Quantum mechanics (QM) is normally “explained” using linear algebra.

Relation algebra and linear algebra are tightly related. Moving from the former to the latter is quite smooth.
“Matrices are arrows”

In the same way we extended functional declarations \( f : A \rightarrow B \) to relational ones, \( R : A \rightarrow B \), we do the same for matrices:

\[
M : A \rightarrow B \text{ declares a matrix with } \#A\text{-many columns and } \#B\text{-many rows. Writing } M : A \rightarrow B \text{ or } M : B \leftarrow A \text{ is the same.}^{5}
\]

In QM, matrices are complex-number-valued, for instance that describing the so-called T-gate,

\[
2 \xleftarrow{T} 2 = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}
\]

where \( e^{ix} = \cos x + i \sin x \) (Euler’s formula).

---

\(^5\)Assume \( A \) and \( B \) finite, for simplicity.
"Matrices are arrows"

(Constant) functions of type $1 \rightarrow A$ expand to \textit{column-vectors} of complex numbers, for instance, $2 \xleftarrow{q} 1 = \begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{bmatrix}$.

What about arrow \textbf{composition}, recall $f \cdot g$ and $R \cdot S$?

Easy: $M \cdot N$ is \textbf{matrix multiplication}:

$$b(M \cdot N)c = \langle \sum a :: (b M a) \times (a N c) \rangle$$

\textbf{NB}: we denote \textbf{matrix cells}, e.g. $b M a$, as we did for relations. Why a different notation?
Our original relations and functions are accepted, as \{0, 1\}-valued matrices.

Functions, in particular, are the only \{0, 1\}-matrices such that \(! \cdot f = !\).

But they become “divisible”. For instance, you can take “the sqrt of negation”, since

\[
\neg = (\sqrt{\neg}) \cdot (\sqrt{\neg})
\]

where

\[
\sqrt{\neg} = \frac{1}{2} \begin{bmatrix} 1 + i & 1 - i \\ 1 - i & 1 + i \end{bmatrix}
\]

Thus one moves into the wonderland of actual quantum logic, in which classical logic operations are no longer primitive.
What kind of matrix is $\sqrt{-1}$?

It is *unitary* — a refined notion of *reversible*:

\[ A = M \begin{pmatrix} A \end{pmatrix} \] is unitary iff

\[ M^\dagger \cdot M = id = M \cdot M^\dagger \] (13)

where $M^\dagger = \overline{M}^\circ$ is the *conjugate transpose* of $M$ and:

\[ x + y i = x - y i \]

\[ \begin{bmatrix} M & N \\ P & Q \end{bmatrix} = \begin{bmatrix} \overline{M} & \overline{N} \\ \overline{P} & \overline{Q} \end{bmatrix} \]

Quantum mechanical processes governed by *unitary* matrices are the building blocks of QP.
Reversible → Unitary

To what extent does what we did for reversibility apply to QP?

The nice story is that our investment in pointfree notation pays off now.

Recall, for example,

\[
\begin{array}{c}
\Psi f \\
\hline
n \\
\hline
b \\
\hline
f^n b \\
\end{array}
\]

defined by

\[
\Psi f \cdot \alpha = [0 \triangledown id, succ \times f] \cdot (id + \Psi f)
\]

We just need to extend pairing (\(\_ \triangledown \_\)) and junction \([\_, \_]\) to arbitrary matrices.
Pairing gives rise to the Khatri-Rao product:

\[(x, y) (M \nabla N) a = (x M a) (y N a)\]

What about \(R \cup S\) and \(R \cap S\)? They become (cell-wise) addition and multiplication, respectively:

\[b (M + N) a = B M a + b N a\]
\[b (M \times N) a = (B M a) (b N a)\]

Note that, unlike \(R \cup R = R\), \(M + M = 2 M\).

Linearity is the essence it all:

\[Q \cdot (M + N) = Q \cdot M + Q \cdot N\]
\[(M + N) \cdot Q = M \cdot Q + N \cdot Q\]
Tensor product and direct sum

The Khatri-Rao product leads the so-called Kronecker (or tensor) product

\[
\begin{array}{c|c}
A & B \\
M & N \\
--- & --- \\
C & D \\
\end{array}
\quad \quad \quad
\begin{array}{c}
A \times B \\
C \times D \\
\end{array}
\]

by

\[
M \otimes N = (M \cdot \text{fst}) \triangledown (N \cdot \text{snd})
\]

— cf. relational product \( R \times S \).

Finally, \([R, S]\) corresponds to \([M|N]\) which collates matrices horizontally, for instance:

\[
[id|\neg] = \begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix} \begin{bmatrix}
0 & 1 \\
1 & 0 \\
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 & 1 \\
0 & 1 & 1 & 0 \\
\end{bmatrix}
\]
Towards “quantamorphisms”

The following property of relations

\[ [R, S] \cdot [P, Q]^\circ = R \cdot P^\circ \cup S \cdot Q^\circ \]

also holds for matrices:

\[ [M|N] \cdot [P|Q]^\circ = M \cdot P^\circ + N \cdot Q^\circ \]  

(14)

Then

\[ \Psi M = [0 \updownarrow id, (\text{succ} \otimes M) \cdot \Psi M] \cdot \alpha^\circ \]

\[ \Leftrightarrow \{ \text{unfold } \alpha \} \]

\[ \Psi M = [0 \updownarrow id, (\text{succ} \otimes M) \cdot \Psi M] \cdot [0 \updownarrow id, \text{succ} \otimes id]^\circ \]

\[ \Leftrightarrow \{ \} \]

\[ \Psi M = (0 \updownarrow id) \cdot (0 \updownarrow id)^\circ + (\text{succ} \otimes M) \cdot \Psi M \cdot (\text{succ}^\circ \otimes id) \]
Towards “quantamorphisms”

Thus we obtain a recursive matrix definition whose least fixpoint is

$$\psi \ M = \mu X \cdot (B + (\text{succ} \otimes M) \cdot X \cdot (\text{succ}^\circ \otimes \text{id}))$$

where $B = (0 \upharpoonright \text{id}) \cdot (0 \upharpoonright \text{id})^\circ$

is the “quantamorphism”

implementing the quantum for gate which iterates $M$ over the second input controlled by the first one (a natural number).
Quantamorphism $\Psi \ M$ in Matlab / Octave

```matlab
function R = quanta(n,M)

% n * b <----- alpha ------- b + n * b
%  |
%  |
%  X      id + X
%  |
%  |
%  V  V
% n * b <----- [ A B ]------- b + n * b

[b,a] = size(M);
if ~(b==a)
    error('M must be square');
else
    R0=zeros(n*b,n*b); id=eye(b);
    A=kron(const(b,n,1),id);
    alpha=kron(succ(n),id);
    B=kron(succ(n),M);
    C=[A B];
    R = fix(b,R0,C,alpha);
end
end

function R = fix(b,X,C,alpha)
    id=eye(b);
    Y= C*(oplus(id,X))*alpha;
    if (Y==X) R = X; else R = fix(b,Y,C,alpha); end
end
```
Iterating a phase-shift gate

Consider the so-called phase shift gate defined by

\[ R_{\phi} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i \phi} \end{bmatrix} \]

Recalling \( e^{i \phi} = \cos \phi + i \sin \phi \), we get, for instance,

\[ R_{\pi/6} = \begin{bmatrix} 1 & 0 \\ 0 & 0.867 + 0.5 i \end{bmatrix} \]

The finite approximation to

\[ \Psi R_{\pi/6} \]

\[ b \]

\[ R_{\pi/6}^n b \]

for \( \#B = 2 \) and control \( n \leq 4 \) is given in the next slide.
Iterating a phase-shift gate

Complex matrix $f_4$ is unitary.

Note the effect of complementation ($\text{fst} \; \wedge \; \_\_\_\_\; \text{shifts}$) shifting the corresponding iteration of gate $R_{\frac{\pi}{6}}$ along the diagonal.
**Summary**

Quantamorphisms have the advantage over other quantum strategies of dispensing with *measurements*. But the concept is still experimental.

Building upon previous work on *stochastic folds* in LAoP (Murta and Oliveira, 2015).

The (linear) algebra of *(unitary)* quantamorphisms is the topic of Ana Neri’s MSc project (grantee INESC TEC).

Towards **correct by construction** quantum programs.

Doomed to repeat history?

Classical computing — Happy blend of diverse bodies of knowledge:

- Philosophy
  - Formal Logic
- Maths
  - Automata Calculus
- Physics
  - Semiconductor Electronics
- Linguistics
  - Grammars Languages
1936

Turing (1912-1954) develops in detail an abstract notion of what we now call a programmable computer — known as the Turing machine.

1936

Church defines the \( \lambda \)-calculus, the basis of functional programming.

Church-Turing thesis: \( \lambda \)-computable \( \Leftrightarrow \) Turing-computable.
Physics made it happen...

Vacuum tubes, triodes (1912)

Credits: https://en.wikipedia.org/wiki/Triode
Physics made it happen...

Transistors (1948)
Quantum literature is vast (2000s)
Physics (again) will make 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)

Intuition far less helpful... Thus the need for a calculational approach!
Annex — proof of (12)

Let $k = \Psi f$. By the UP (11), $k = f \cdot (id + k) \cdot \alpha^\circ$. We calculate

$K = \ker k$ assuming $\ker f = id$:

\[
K = k^\circ \cdot k
\]
\[
\Leftrightarrow \quad \left\{ \text{unfold } f \cdot F k \cdot \alpha^\circ \right\}
\]
\[
K = \alpha \cdot F k^\circ \cdot f^\circ \cdot f \cdot F k \cdot \alpha^\circ
\]
\[
\Leftrightarrow \quad \left\{ \text{assumption: } f^\circ \cdot f = id \right\}
\]
\[
K = \alpha \cdot F k^\circ \cdot F k \cdot \alpha^\circ
\]
\[
\Leftrightarrow \quad \left\{ F (R \cdot S) = (F R) \cdot (F S) \text{ and } F R^\circ = (F R)^\circ \right\}
\]
\[
K = \alpha \cdot F k^\circ \cdot k \cdot \alpha^\circ
\]
\[
\Leftrightarrow \quad \left\{ K = k^\circ \cdot k; \text{ UP (for relations)} \right\}
\]
\[
K = \llangle \alpha \rrangle
\]
\[
\Leftrightarrow \quad \left\{ \text{Reflexion: } \llangle \alpha \rrangle = id \right\}
\]
\[
K = id
\]
In the **functional** case:

\[ f \cdot (R + S) \subseteq S \cdot g \Rightarrow \langle f \rangle \cdot (id \times R) \subseteq S \cdot \langle g \rangle \]  

(15)

recall

\[ \mathbb{N}_0 \times B \xleftarrow{\alpha} B + \mathbb{N}_0 \times B \]

\[ \langle f \rangle \downarrow \quad \quad \downarrow \quad \quad \quad \quad \downarrow \quad \quad \quad \quad \downarrow \quad \quad \quad \quad \downarrow \quad \quad \quad \quad \downarrow \]

\[ C \quad \xleftarrow{f} \quad B + C \]

Corollaries (**fusion** laws):

\[ \langle f \rangle \cdot (id \times r) = \langle f \cdot (r + id) \rangle \]

\[ f \cdot (id + s) = s \cdot g \Rightarrow \langle f \rangle = s \cdot \langle g \rangle \]

To do: check these properties in the **linear algebra** case.
References


