Semantics of Quantum Loops and Recursion

Louis Lemonnier

University of Edinburgh

THE UNIVERSITY *of* EDINBURGH
informatics

QSL seminar. 17th October 2024

Programming languages

```
def fact(n):
        if n == 0:
                return(1)
        else:
                return(n*fact(n-1))
```

Programming languages

```
let rec fact (n:int) =
        if n=0 then 1
        else n * fact (n-1)
```

Programming languages

```
fact:
        movq   $0, %rax
        je     .case_zero
        movq   %rdi, %rax
        pushq  %rax
        subq   $1, %rdi
        call   fact
        popq   %rbx
        multq  %rbx, %rax
        jmp    .else
.case_zero:
        movq   $1, %rax
.else:
        ret
```

Programming languages

$$\lambda m.\lambda g.(\lambda x.g(xx))(\lambda x.g(xx))(\lambda p.\lambda a.\lambda b.pab)$$
$$(\lambda n.n(\lambda x.(\lambda a.\lambda b.b))(\lambda a.\lambda b.a))(\lambda f.\lambda x.fx)$$
$$(m(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda x.x)))$$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: formal program execution $t \rightarrow t'$.

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: formal program execution $t \to t'$.

For example, $(\lambda x.x + 2)3 \to$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: formal program execution $t \to t'$.

For example, $(\lambda x.x + 2)3 \to 3 + 2 \to$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: formal program execution $t \to t'$.

$$\text{For example, } (\lambda x.x + 2)3 \to 3 + 2 \to 5.$$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: formal program execution $t \to t'$.

$$\text{For example, } (\lambda x.x + 2)3 \to 3 + 2 \to 5.$$

Denotational semantics: function $[\![t]\!]$, usual in mathematical terms.

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: formal program execution $t \to t'$.

$$\text{For example, } (\lambda x.x + 2)3 \to 3 + 2 \to 5.$$

Denotational semantics: function $[\![t]\!]$, usual in mathematical terms.

Preserves composition: $\quad [\![t; t']\!] = [\![t']\!] \circ [\![t]\!]$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: formal program execution $t \to t'$.

$$\text{For example, } (\lambda x.x + 2)3 \to 3 + 2 \to 5.$$

Denotational semantics: function $[\![t]\!]$, usual in mathematical terms.

Preserves composition: $[\![t; t']\!] = [\![t']\!] \circ [\![t]\!]$

Strong enough model if relation between operational and denotational:

$$t \simeq t' \text{ iff } [\![t]\!] = [\![t']\!] .$$

Semantics

Comes from Ancient Greek, *to provide a meaning*.

Operational semantics: formal program execution $t \to t'$.

For example, $(\lambda x.x + 2)3 \to 3 + 2 \to 5$.

Denotational semantics: function $[\![t]\!]$, usual in mathematical terms.

Preserves composition: $[\![t; t']\!] = [\![t']\!] \circ [\![t]\!]$

Strong enough model if relation between operational and denotational:

$$t \simeq t' \text{ iff } [\![t]\!] = [\![t']\!] .$$

- Prove that the language does what it is supposed to do,
- Compile the language and perform optimisations safely,
- Inform on which features can be added to the language.

The point of semantics

It is hard to prove properties through the syntax.

The point of semantics

It is hard to prove properties through the syntax.

With semantics, one can prove properties without running the code.

The point of semantics

It is hard to prove properties through the syntax.

With semantics, one can prove properties without running the code.

In some cases, semantics can also help add features to the language.

The point of semantics

It is hard to prove properties through the syntax.

With semantics, one can prove properties without running the code.

In some cases, semantics can also help add features to the language.

This is what happens in the recent papers of Chris and Robin (and co-authors).

The point of semantics

It is hard to prove properties through the syntax.

With semantics, one can prove properties without running the code.

In some cases, semantics can also help add features to the language.

This is what happens in the recent papers of Chris and Robin (and co-authors).

But we are not telling this story.

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
        return(n+1)
```

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
        return(n+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & n+1 \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
        return(n+1)
```

```
def f(n):
        return(g(n)+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & n+1 \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
        return(n+1)
```

```
def f(n):
        return(g(n)+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & n+1 \end{cases}$$

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & [\![g]\!](n)+1 \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
    return(n+1)
```

```
def f(n):
    return(g(n)+1)
```

```
def f(n):
    while True:
        pass
    return(n+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & n+1 \end{cases}$$

$$[\![f]\!] = \begin{cases} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & [\![g]\!](n)+1 \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
    return(n+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & n+1 \end{cases}$$

```
def f(n):
    return(g(n)+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & [\![g]\!](n)+1 \end{cases}$$

```
def f(n):
    while True:
        pass
    return(n+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & ??? \end{cases}$$

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
        return(n+1)
```

```
def f(n):
        return(g(n)+1)
```

```
def f(n):
        while True:
                pass
        return(n+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & n+1 \end{cases}$$

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & [\![g]\!](n)+1 \end{cases}$$

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & ??? \end{cases}$$

We need to account for non termination!

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
        return(n+1)
```

```
def f(n):
        return(g(n)+1)
```

```
def f(n):
        while True:
                pass
        return(n+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & n+1 \end{cases}$$

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & [\![g]\!](n)+1 \end{cases}$$

$$[\![f]\!] = \begin{cases} \mathbb{N} & \rightarrow & \mathbb{N} \\ n & \mapsto & ??? \end{cases}$$

We need to account for non termination!

$\perp$ means undefined.

A gentle walk through classical programming

There needs to be a corresponding semantics to all programs.

```
def f(n):
        return(n+1)
```

```
def f(n):
        return(g(n)+1)
```

```
def f(n):
        while True:
                pass
        return(n+1)
```

$$[\![f]\!] = \begin{cases} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & n+1 \end{cases}$$

$$[\![f]\!] = \begin{cases} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & [\![g]\!](n)+1 \end{cases}$$

$$[\![f]\!] = \begin{cases} \mathbb{N} & \to & \mathbb{N} \\ n & \mapsto & ??? \end{cases}$$

We need to account for non termination!

$\perp$ means undefined.

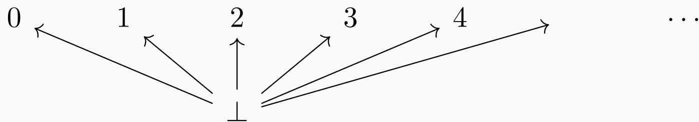
$$[\![f]\!] = \begin{cases} \mathbb{N} \cup \{\perp\} & \to & \mathbb{N} \cup \{\perp\} \\ n & \mapsto & \perp \end{cases}$$

Domain theory (example)

$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with nice properties.

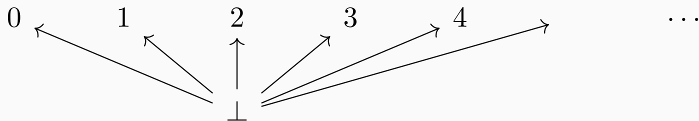
Domain theory (example)

$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with nice properties.



Domain theory (example)

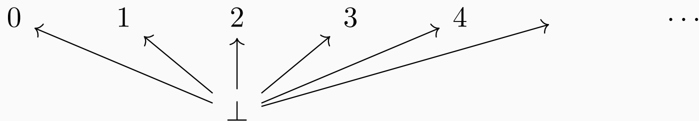
$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with nice properties.



The order generalises to functions: $f \leq g$ if for all x, we have $f(x) \leq g(x)$.

Domain theory (example)

$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with nice properties.

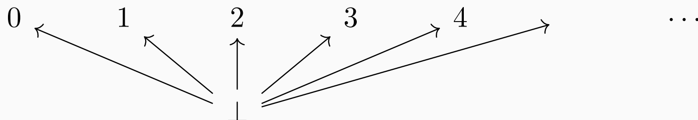


The order generalises to functions: $f \leq g$ if for all x, we have $f(x) \leq g(x)$.

The set $[\mathbb{N}_\perp \to \mathbb{N}_\perp]$ of monotone functions also has nice properties.

Domain theory (example)

$\mathbb{N}_\perp \stackrel{\text{def}}{=} \mathbb{N} \cup \{\perp\}$ is a partial-ordered set with nice properties.



The order generalises to functions: $f \leq g$ if for all x, we have $f(x) \leq g(x)$.

The set $[\mathbb{N}_\perp \to \mathbb{N}_\perp]$ of monotone functions also has nice properties.

How do we use it?

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ the constant function with output $\perp$.

$\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$\perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \cdots$$

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ the constant function with output $\perp$.

$\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$\perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \cdots$$

$F(\perp) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \cdots$$

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ the constant function with output $\perp$.

$F(\perp) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \cdots$$

$F(F(\perp)) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad \perp \quad \perp \quad \perp \quad \perp \quad \perp \quad \cdots$$

Interpretation of recursion

Define $F : [[\mathbb{N}_\bot \to \mathbb{N}_\bot] \to [\mathbb{N}_\bot \to \mathbb{N}_\bot]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\bot : \mathbb{N}_\bot \to \mathbb{N}_\bot$ the constant function with output $\bot$.

$F(F(\bot)) : \mathbb{N}_\bot \to \mathbb{N}_\bot$ is:

$$1 \quad 1 \quad \bot \quad \bot \quad \bot \quad \bot \quad \bot \quad \cdots$$

$F(F(F(\bot))) : \mathbb{N}_\bot \to \mathbb{N}_\bot$ is:

$$1 \quad 1 \quad 2 \quad \bot \quad \bot \quad \bot \quad \bot \quad \cdots$$

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ the constant function with output $\perp$.

$F(F(F(\perp))) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad \perp \quad \perp \quad \perp \quad \perp \quad \cdots$$

$F(F(F(F(\perp)))) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad \perp \quad \perp \quad \perp \quad \cdots$$

Interpretation of recursion

Define $F\colon [[\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp\colon \mathbb{N}_\perp \to \mathbb{N}_\perp$ the constant function with output $\perp$.

$F(F(F(F(\perp))))\colon \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad \perp \quad \perp \quad \perp \quad \cdots$$

$F(F(F(F(F(\perp)))))\colon \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad 24 \quad \perp \quad \perp \quad \cdots$$

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ the constant function with output $\perp$.

$F(F(F(F(F(\perp))))) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad 24 \quad \perp \quad \perp \quad \cdots$$

$F(F(F(F(F(F(\perp)))))) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad 24 \quad 120 \quad \perp \quad \cdots$$

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ the constant function with output $\perp$.

$F(F(F(F(F(\perp))))) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad 24 \quad \perp \quad \perp \quad \cdots$$

$F(F(F(F(F(F(\perp)))))) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad 24 \quad 120 \quad \perp \quad \cdots$$

The interpretation of the factorial is then the limit of $(F^n(\perp))_n$.

Interpretation of recursion

Define $F : [[\mathbb{N}_\perp \to \mathbb{N}_\perp] \to [\mathbb{N}_\perp \to \mathbb{N}_\perp]]$ as:

$$F(g)(n) = \begin{cases} 1 & \text{if } n = 0, \\ n \times g(n-1) & \text{else.} \end{cases}$$

Write $\perp : \mathbb{N}_\perp \to \mathbb{N}_\perp$ the constant function with output $\perp$.

$F(F(F(F(F(\perp))))) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad 24 \quad \perp \quad \perp \quad \cdots$$

$F(F(F(F(F(F(\perp)))))) : \mathbb{N}_\perp \to \mathbb{N}_\perp$ is:

$$1 \quad 1 \quad 2 \quad 6 \quad 24 \quad 120 \quad \perp \quad \cdots$$

The interpretation of the factorial is then the limit of $(F^n(\perp))_n$.

A notion of limit linked to the order is necessary.

What about Quantum Programming?

Quantum channels: e.g. CPTP maps $\mathcal{L}(H_A) \to \mathcal{L}(H_B)$.

What about Quantum Programming?

Quantum channels: e.g. CPTP maps $\mathcal{L}(H_A) \to \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ if $g - f$ is positive.

What about Quantum Programming?

Quantum channels: e.g. CPTP maps $\mathcal{L}(H_A) \to \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ if $g - f$ is positive.

Admits a very nice structure in the Heisenberg picture.

What about Quantum Programming?

Quantum channels: e.g. CPTP maps $\mathcal{L}(H_A) \to \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ if $g - f$ is positive.

Admits a very nice structure in the Heisenberg picture.

What about Quantum Programming?

Quantum channels: e.g. CPTP maps $\mathcal{L}(H_A) \to \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ if $g - f$ is positive.

Admits a very nice structure in the Heisenberg picture.

- ⋆ (almost) everything done similar to classical computing.

What about Quantum Programming?

Quantum channels: e.g. CPTP maps $\mathcal{L}(H_A) \to \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ if $g - f$ is positive.

Admits a very nice structure in the Heisenberg picture.

- ⋆ (almost) everything done similar to classical computing.
- ⋆ with probabilies (because of measurement).

What about Quantum Programming?

Quantum channels: e.g. CPTP maps $\mathcal{L}(H_A) \to \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ if $g - f$ is positive.

Admits a very nice structure in the Heisenberg picture.

- ⋆ (almost) everything done similar to classical computing.
- ⋆ with probabilies (because of measurement).
- ⋆ and a non duplication of terms because of no-cloning.

What about Quantum Programming?

Quantum channels: e.g. CPTP maps $\mathcal{L}(H_A) \to \mathcal{L}(H_B)$.

Löwner order: $f \leq g$ if $g - f$ is positive.

Admits a very nice structure in the Heisenberg picture.

- ⋆ (almost) everything done similar to classical computing.
- ⋆ with probabilies (because of measurement).
- ⋆ and a non duplication of terms because of no-cloning.

The undefined function is interpreted as the constant map 0 (no need to add a $\perp$).

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a programming flavour.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a programming flavour.

Its classical counterpart is interpreted with partial injective functions.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a programming flavour.

Its classical counterpart is interpreted with partial injective functions.

- ⋆ a full partial injection is a bijection.

The Reversible Case

[SabryValironVizzotto2018] Language for expressing (quantum) reversible programs.

The language is very expressive and has a programming flavour.

Its classical counterpart is interpreted with partial injective functions.

- $\star$ a full partial injection is a bijection.

Its quantum counterpart is interpreted as contractive linear maps.

The Reversible Case

`[SabryValironVizzotto2018]` Language for expressing (quantum) reversible programs.

The language is very expressive and has a programming flavour.

Its classical counterpart is interpreted with partial injective functions.

- ⋆ a full partial injection is a bijection.

Its quantum counterpart is interpreted as contractive linear maps.

- ⋆ a full contractive map is a unitary.

The Reversible Case

`[SabryValironVizzotto2018]` Language for expressing (quantum) reversible programs.

The language is very expressive and has a programming flavour.

Its classical counterpart is interpreted with partial injective functions.

- $\star$ a full partial injection is a bijection.

Its quantum counterpart is interpreted as contractive linear maps.

- $\star$ a full contractive map is a unitary.

Is it possible to interpret loop or recursion in this setting?

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

It is Turing-complete! [ChardonnetLemonnierValiron2024]

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

It is Turing-complete! [ChardonnetLemonnierValiron2024]

Given two sets A and B, the partial injections $[A \rightharpoonup B]$ have nice properties for fixpoints.

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

It is Turing-complete! [ChardonnetLemonnierValiron2024]

Given two sets A and B, the partial injections $[A \rightharpoonup B]$ have nice properties for fixpoints.

(The category of partial injective functions between sets is enriched in pointed dcpos.)

Quick stop: Recursion in Classical Reversibility

Suppose we have a syntax with:

- Only reversible functions,
- Inductive types,
- A fixpoint operator,
- A CbV operational semantics.

It is Turing-complete! [ChardonnetLemonnierValiron2024]

Given two sets A and B, the partial injections $[A \rightharpoonup B]$ have nice properties for fixpoints.

(The category of partial injective functions between sets is enriched in pointed dcpos.)

What about its quantum counterpart?

The Quantum Case

Reversible quantum operations are subunitaries (partial isometries) such as:

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

$$|+\rangle\langle 0| + |-\rangle\langle 1| = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \qquad |+\rangle\langle 0| = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} \qquad |-\rangle\langle 1| = \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

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

The Quantum Case

Reversible quantum operations are subunitaries (partial isometries) such as:

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

$$|+\rangle\langle 0| + |-\rangle\langle 1| = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \qquad |+\rangle\langle 0| = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} \qquad |-\rangle\langle 1| = \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

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

Arguably, undefined ($\perp$) in a (2-dim.) Hilbert space is the zero vector $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

The Quantum Case

Reversible quantum operations are subunitaries (partial isometries) such as:

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

$$|+\rangle\langle 0| + |-\rangle\langle 1| = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \qquad |+\rangle\langle 0| = \begin{bmatrix} \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 \end{bmatrix} \qquad |-\rangle\langle 1| = \begin{bmatrix} 0 & \frac{1}{\sqrt{2}} \\ 0 & -\frac{1}{\sqrt{2}} \end{bmatrix}$$

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

Arguably, undefined ($\perp$) in a (2-dim.) Hilbert space is the zero vector $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$.

$\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ is less defined than $\begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} = |0\rangle\langle 0|$ which is less defined than $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$.

A disappointment

However, this order does not interact well with composition:

$$
\begin{aligned}
& & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} &\leq \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \\
\text{thus} & & \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \circ \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \circ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} &\leq \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \circ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, \\
\text{i.e.} & & 1/2 &\leq 0.
\end{aligned}
$$

A disappointment

However, this order does not interact well with composition:

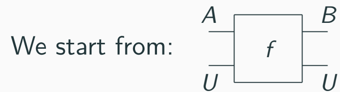
$$
\begin{array}{lrcl}
 & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} & \leq & \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \\
\text{thus} & \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \circ \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \circ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix} & \leq & \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{bmatrix} \circ \begin{bmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{bmatrix}, \\
\text{i.e.} & 1/2 & \leq & 0.
\end{array}
$$

Theorem (No go theorem)

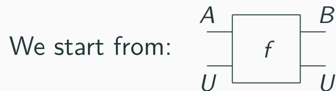
There is no such order on subunitary maps between Hilbert spaces.

Recursion through a Trace

We start from:



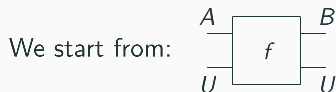
Recursion through a Trace

We start from: 

The recurring example: $g\colon \begin{cases} \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) & \to & \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) \\ \texttt{left}\ n & \mapsto & \texttt{right}\ (n, 1) \\ \texttt{right}\ (n+1, m) & \mapsto & \texttt{right}\ (n, n \times m) \\ \texttt{right}\ (0, m) & \mapsto & \texttt{left}\ m \end{cases}$

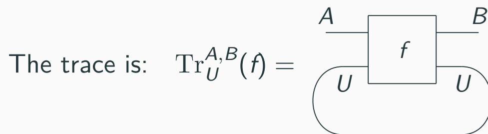
Recursion through a Trace

We start from:

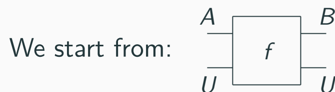


The recurring example: $g\colon \begin{cases} \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) & \to & \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) \\ \texttt{left}\ n & \mapsto & \texttt{right}\ (n, 1) \\ \texttt{right}\ (n+1, m) & \mapsto & \texttt{right}\ (n, n \times m) \\ \texttt{right}\ (0, m) & \mapsto & \texttt{left}\ m \end{cases}$

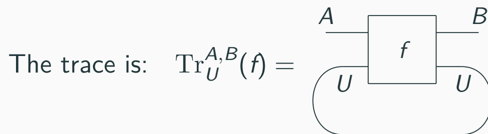
The trace is: $\mathrm{Tr}_U^{A,B}(f) =$



Recursion through a Trace

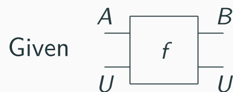
We start from: 

The recurring example: $g \colon \begin{cases} \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) & \to & \mathbb{N} \uplus (\mathbb{N} \times \mathbb{N}) \\ \texttt{left}\ n & \mapsto & \texttt{right}\ (n, 1) \\ \texttt{right}\ (n+1, m) & \mapsto & \texttt{right}\ (n, n \times m) \\ \texttt{right}\ (0, m) & \mapsto & \texttt{left}\ m \end{cases}$

The trace is: $\mathrm{Tr}_U^{A,B}(f) =$ 

And $\mathrm{Tr}_{\mathbb{N} \uplus \mathbb{N}}^{\mathbb{N},\mathbb{N}}(g)(n) = n!$.

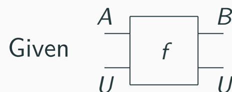
The Quantum Case



The canonical trace formula is:

$$\mathrm{Tr}(f) = f_{AB} + \sum_{n} f_{UB} f_{UU}^{n} f_{AU}$$

The Quantum Case



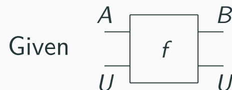
The canonical trace formula is:

$$\mathrm{Tr}(f) = f_{AB} + \sum_{n} f_{UB} f_{UU}^{n} f_{AU}$$

This is a trace for unitaries between finite dimension Hilbert space.

However:

The Quantum Case

Given 

The canonical trace formula is:

$$\mathrm{Tr}(f) = f_{AB} + \sum_{n} f_{UB} f_{UU}^{n} f_{AU}$$

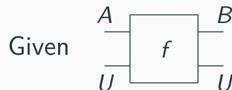
This is a trace for unitaries between finite dimension Hilbert space.

However:

Lemma (Towards no go)

The canonical trace on infinite dimension Hilbert spaces is not a trace.

The Quantum Case

Given 

The canonical trace formula is:

$$\mathrm{Tr}(f) = f_{AB} + \sum_{n} f_{UB} f_{UU}^{n} f_{AU}$$

This is a trace for unitaries between finite dimension Hilbert space.

However:

Lemma (Towards no go)

The canonical trace on infinite dimension Hilbert spaces is not a trace.

My take:

Conjecture: No go

There is no suitable trace for (sub)unitaries between infinite dimension Hilbert spaces.

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows non termination.

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows non termination.
- *"Non-terminating quantum-controlled program do not make sense."*

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows non termination.
- *"Non-terminating quantum-controlled program do not make sense."*
- But no mathematical account of this point yet.

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows non termination.
- *"Non-terminating quantum-controlled program do not make sense."*
- But no mathematical account of this point yet.

Terminaison needs to be ensured!

Until we meet again

Is quantum recursion not possible?

If you ask people in the community:

- Classical recursion allows non termination.
- *"Non-terminating quantum-controlled program do not make sense."*
- But no mathematical account of this point yet.

Terminaison needs to be ensured!

A technique called guarded recursion.

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \cdots$$

where H_n is the lists of qubits up to size n.

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \cdots$$

where H_n is the lists of qubits up to size n.

Then, a function is described as a family of functions $\{f_i \colon H_i \to H_i\}$.

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \cdots$$

where H_n is the lists of qubits up to size n.

Then, a function is described as a family of functions $\{f_i \colon H_i \to H_i\}$.

In this setting, admissible fixed points [Lemonnier2024].

Guarded Quantum Recursion

Let H be the Hilbert space for lists of qubits.

Decompose H as the following sequence:

$$H_0 \longleftarrow H_1 \longleftarrow H_2 \longleftarrow H_3 \longleftarrow \cdots$$

where H_n is the lists of qubits up to size n.

Then, a function is described as a family of functions $\{f_i \colon H_i \to H_i\}$.

In this setting, admissible fixed points [Lemonnier2024].

However, the expressivity is limited.

Conclusion

Conclusion

- Reversibility does not prevent expressivity.

Conclusion

- Reversibility does not prevent expressivity.
- Does not naturally generalise to the reversible quantum case.

Conclusion

- Reversibility does not prevent expressivity.
- Does not naturally generalise to the reversible quantum case.
- Do you think quantum recursion makes sense?

Conclusion

- Reversibility does not prevent expressivity.
- Does not naturally generalise to the reversible quantum case.
- Do you think quantum recursion makes sense?

Thank you!