Quantum Programming Languages And Resource-Analysis

Thomas Vinet

Mocqua Days 2026

1. Motivations

Types of quantum algorithms

- Classical control: Shor

Types of quantum algorithms

- Classical control: Shor
- Quantum control: Quantum Switch

Types of quantum algorithms

- Classical control: Shor
- Quantum control: Quantum Switch
- Seems natural to combine both flow: *hybrid language*

Resource Analysis

Resource Analysis

- On quantum circuits: properties on size, depth, width...
- Languages that guarantee polynomial size of a circuit [HPS23]; no hybrid language with resource analysis in mind.

Resource Analysis

- On quantum circuits: properties on size, depth, width...
- Languages that guarantee polynomial size of a circuit [HPS23]; no hybrid language with resource analysis in mind.
- Resource analysis on programming languages is hard.

Resource Analysis

- On quantum circuits: properties on size, depth, width...
- Languages that guarantee polynomial size of a circuit [HPS23]; no hybrid language with resource analysis in mind.
- Resource analysis on programming languages is hard.

However: term rewrite systems (TRS) have been well studied for termination and runtime complexity

Steps of the thesis

- Build a hybrid language, with general recursion and both control flows

Steps of the thesis

- Build a hybrid language, with general recursion and both control flows
- Translate the language into a TRS

Steps of the thesis

- Build a hybrid language, with general recursion and both control flows
- Translate the language into a TRS
- Adapt existing termination techniques to provide criterion for complexity

Steps of the thesis

- Build a hybrid language, with general recursion and both control flows
- Translate the language into a TRS
- Adapt existing termination techniques to provide criterion for complexity
- End goal: be able to compile the initial term to a quantum circuit of bounded size !

2. `Hyrql`: a hybrid language

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$
- Very generic approach: let the user build its own data types

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$
- Very generic approach: let the user build its own data types
- Pairs: (a, b)

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$
- Very generic approach: let the user build its own data types
- Pairs: (a, b)
- Natural numbers: $0, S(n)$

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$
- Very generic approach: let the user build its own data types
- Pairs: (a, b)
- Natural numbers: $0, S(n)$
- Lists: $[\,], h :: t$

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$
- Very generic approach: let the user build its own data types
- Pairs: (a, b)
- Natural numbers: $0, S(n)$
- Lists: $[\,], h :: t$
- Can superpose any quantum term (with some restrictions) as $\sum_{i=1}^{n} \alpha_i \cdot t_i$

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$
- Very generic approach: let the user build its own data types
- Pairs: (a, b)
- Natural numbers: $0, S(n)$
- Lists: $[\,], h :: t$
- Can superpose any quantum term (with some restrictions) as $\sum_{i=1}^{n} \alpha_i \cdot t_i$

$$|+\rangle \triangleq \frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle$$

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$
- Very generic approach: let the user build its own data types
- Pairs: (a, b)
- Natural numbers: $0, S(n)$
- Lists: $[\,], h :: t$
- Can superpose any quantum term (with some restrictions) as $\sum_{i=1}^{n} \alpha_i \cdot t_i$

$$|+\rangle \triangleq \frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle$$

$$(|0\rangle :: |+\rangle :: [\,], S(S(0))$$

Syntax choices: what type of data ?

- Quantum data represented by $|0\rangle, |1\rangle$
- Very generic approach: let the user build its own data types
- Pairs: (a, b)
- Natural numbers: $0, S(n)$
- Lists: $[\,], h :: t$
- Can superpose any quantum term (with some restrictions) as $\sum_{i=1}^{n} \alpha_i \cdot t_i$

$$|+\rangle \triangleq \frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle$$

$$(|0\rangle :: |+\rangle :: [\,], S(S(0))$$

$$\frac{1}{\sqrt{2}} \cdot (|0\rangle, S(0)) + \frac{1}{\sqrt{2}} \cdot (|+\rangle, S(0))$$

Syntax choices: functional languages

- Choice of **functional programming languages** rather than imperative programming languages, as it better fits TRS formalism and inspired from [SVV18].

$$(\lambda x.x)\,|0\rangle$$

Syntax choices: functional languages

- Choice of **functional programming languages** rather than imperative programming languages, as it better fits TRS formalism and inspired from [SVV18].

$$(\lambda x.x)\,|0\rangle$$

- To provide a relevant resource analysis, we need general recursion: $\texttt{letrec}\, f\, x = t$

Syntax choices: Linearity

- No-cloning theorem: no transformation $U(x) = x \otimes x$
- However, classical data behaves freely

Syntax choices: Linearity

- No-cloning theorem: no transformation $U(x) = x \otimes x$
- However, classical data behaves freely
- Typed language: $\Gamma; \Delta \vdash t : T$, Γ non-linear, Δ linear

Syntax choices: Linearity

- No-cloning theorem: no transformation $U(x) = x \otimes x$
- However, classical data behaves freely
- Typed language: $\Gamma; \Delta \vdash t : T$, Γ non-linear, Δ linear

$$n : \mathtt{nat}; q : \mathrm{Qbit} \vdash (q, n :: n :: [\,]) : \mathrm{Qbit} \times \mathtt{list}(\mathtt{nat})$$

Syntax choices: Linearity

- No-cloning theorem: no transformation $U(x) = x \otimes x$
- However, classical data behaves freely
- Typed language: $\Gamma; \Delta \vdash t : T$, Γ non-linear, Δ linear

$$n : \texttt{nat}; q : \mathrm{Qbit} \vdash (q, n :: n :: [\,]) : \mathrm{Qbit} \times \texttt{list}(\texttt{nat})$$

- Problem: length of a qubit list

Syntax choices: Linearity

- No-cloning theorem: no transformation $U(x) = x \otimes x$
- However, classical data behaves freely
- Typed language: $\Gamma; \Delta \vdash t : T$, Γ non-linear, Δ linear

$$n : \mathtt{nat}; q : \mathrm{Qbit} \vdash (q, n :: n :: [\,]) : \mathrm{Qbit} \times \mathtt{list}(\mathtt{nat})$$

- Problem: length of a qubit list
- `shape`: construct extracting the classical structure

$$\mathtt{shape}((|+\rangle, S(0)) \sim ((), S(0))$$

Syntax choices: Control flows

- No measurement in the language: probabilistic behaviour, can be moved
- Control flow done through *pattern-matching constructs*

Syntax choices: Control flows

- No measurement in the language: probabilistic behaviour, can be moved
- Control flow done through *pattern-matching constructs*

Classical control

$$\texttt{match}\, t \begin{Bmatrix} c_1(\overrightarrow{x_1}) \to t_1 \\ \vdots \qquad \vdots \\ c_n(\overrightarrow{x_n}) \to t_n \end{Bmatrix}$$

Quantum control

$$\texttt{qcase}\, q \begin{Bmatrix} |0\rangle \to t_0 \\ |1\rangle \to t_1 \end{Bmatrix}$$

Syntax choices: Control flows

- No measurement in the language: probabilistic behaviour, can be moved
- Control flow done through *pattern-matching constructs*

Classical control

$$\texttt{match}\, t \begin{Bmatrix} c_1(\overrightarrow{x_1}) \to t_1 \\ \vdots \qquad \vdots \\ c_n(\overrightarrow{x_n}) \to t_n \end{Bmatrix}$$

Quantum control

$$\texttt{qcase}\, q \begin{Bmatrix} |0\rangle \to t_0 \\ |1\rangle \to t_1 \end{Bmatrix}$$

- Classical control: outputs one branch, while quantum control outputs possibly a superposition

Some examples (1/3)

Standard gate: the NOT gate

$$\texttt{NOT} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Some examples (1/3)

Standard gate: the NOT gate

$$\texttt{NOT} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

In `Hyrql`:

$$\texttt{NOT} = \lambda x.\texttt{qcase}\, x \begin{Bmatrix} |0\rangle \to |1\rangle \\ |1\rangle \to |0\rangle \end{Bmatrix}$$

Abstraction

Some examples (1/3)

Standard gate: the NOT gate

$$\texttt{NOT} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

In `Hyrql`:

$$\texttt{NOT} = \lambda x.\texttt{qcase}\, x \begin{Bmatrix} |0\rangle \to |1\rangle \\ |1\rangle \to |0\rangle \end{Bmatrix}$$

Pattern-matching with two branches

Some examples (1/3)

Standard gate: the NOT gate

$$\texttt{NOT} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

In `Hyrql`:

$$\texttt{NOT} = \lambda x.\texttt{qcase}\, x \begin{Bmatrix} |0\rangle \to |1\rangle \\ |1\rangle \to |0\rangle \end{Bmatrix}$$

Output branches

Some examples (2/3)

$$\texttt{cnot} \triangleq \lambda c.\lambda t.\texttt{qcase}\, c \left\{ |0\rangle \to |0\rangle \otimes t\,, |1\rangle \to |1\rangle \otimes \texttt{not}\, t \right\}$$

Some examples (2/3)

$$\texttt{cnot} \triangleq \lambda c.\lambda t.\texttt{qcase}\, c \left\{ |0\rangle \to |0\rangle \otimes t \,, |1\rangle \to |1\rangle \otimes \texttt{not}\, t \right\}$$

$$\texttt{QS} \triangleq \lambda f.\lambda g.\lambda c.\lambda t.\texttt{qcase}\, c \begin{Bmatrix} |0\rangle \to |0\rangle \otimes f(g\ t) \\ |1\rangle \to |1\rangle \otimes g(f\ t) \end{Bmatrix}$$

Some examples (2/3)

$$\texttt{cnot} \triangleq \lambda c.\lambda t.\texttt{qcase}\, c \left\{ |0\rangle \to |0\rangle \otimes t\,, |1\rangle \to |1\rangle \otimes \texttt{not}\, t \right\}$$

$$\texttt{QS} \triangleq \lambda f.\lambda g.\lambda c.\lambda t.\texttt{qcase}\, c \begin{Bmatrix} |0\rangle \to |0\rangle \otimes f(g\ t) \\ |1\rangle \to |1\rangle \otimes g(f\ t) \end{Bmatrix}$$

$$\texttt{len} \triangleq \texttt{letrec}\, f\ l = \texttt{match}\, l \left\{ [\,] \to 0, h :: t \to S(f\ t) \right\}$$

Some examples (2/3)

$$\texttt{cnot} \triangleq \lambda c.\lambda t.\texttt{qcase}\, c \left\{ |0\rangle \to |0\rangle \otimes t\,, |1\rangle \to |1\rangle \otimes \texttt{not}\, t \right\}$$

$$\texttt{QS} \triangleq \lambda f.\lambda g.\lambda c.\lambda t.\texttt{qcase}\, c \begin{Bmatrix} |0\rangle \to |0\rangle \otimes f(g\ t) \\ |1\rangle \to |1\rangle \otimes g(f\ t) \end{Bmatrix}$$

$$\texttt{len} \triangleq \texttt{letrec}\, f\ l = \texttt{match}\, l \left\{ [\,] \to 0, h :: t \to S(f\ t) \right\}$$

Length has a non-linear behaviour, but can still be used with qubit lists:

$$\lambda x.(x, \texttt{len}(\texttt{shape}(x)))$$

Some examples (3/3)

- Bounded quantum walk

$$\texttt{letrec}\, f\ q = \lambda n.\texttt{qcase}\, q \left\{ \begin{array}{l} |0\rangle \to |0\rangle :: [\,] \\ |1\rangle \to \texttt{match}\, n \left\{ \begin{array}{c} 0 \to |1\rangle :: [\,] \\ S(m) \to |1\rangle :: (f\ |+\rangle\ m) \end{array} \right\} \end{array} \right\}$$

- Recursion on both classical and quantum data

Operational semantics

- Follows a *Call-by-value* strategy

Operational semantics

- Follows a *Call-by-value* strategy
- $(\lambda x.t)v \rightsquigarrow t\{v/x\}$

Operational semantics

- Follows a *Call-by-value* strategy
- $(\lambda x.t)v \rightsquigarrow t\{v/x\}$
- $\texttt{qcase}\ |0\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow t_0$

Operational semantics

- Follows a *Call-by-value* strategy
- $(\lambda x.t)v \rightsquigarrow t\{v/x\}$
- $\texttt{qcase}\ |0\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow t_0$
- $\texttt{qcase}\ |1\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow t_1$

Operational semantics

- Follows a *Call-by-value* strategy
- $(\lambda x.t)v \rightsquigarrow t\{v/x\}$
- $\texttt{qcase}\ |0\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow t_0$
- $\texttt{qcase}\ |1\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow t_1$

$$\texttt{qcase}\ \alpha \cdot |0\rangle + \beta \cdot |1\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow \alpha \cdot t_0 + \beta \cdot t_1$$

Operational semantics

- Follows a *Call-by-value* strategy
- $(\lambda x.t)v \rightsquigarrow t\{v/x\}$
- $\texttt{qcase}\ |0\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow t_0$
- $\texttt{qcase}\ |1\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow t_1$

$$\texttt{qcase}\ \alpha \cdot |0\rangle + \beta \cdot |1\rangle \left\{ |0\rangle \to t_0\,, |1\rangle \to t_1 \right\} \rightsquigarrow \alpha \cdot t_0 + \beta \cdot t_1$$

- Superposition: parallel reduction

Orthogonality and Norm

- Quantum states must be normalized, and quantum transformations must preserve the norm
- Tackled by introducing an orthogonality predicate $s \perp t$
- Ensures that they share the same classical structure
- For superpositions:

$$\frac{\Gamma; \Delta \vdash t_i : Q \qquad \sum_{i=1}^{n} |\alpha_i|^2 = 1 \qquad \forall i \neq j,\ t_i \perp t_j}{\Gamma; \Delta \vdash \sum_{i=1}^{n} \alpha_i \cdot t_i : Q}$$

- Quantum control $\mathtt{qcase}\, q \left\{ |0\rangle \to t_0 \,, |1\rangle \to t_1 \right\}$ requires $t_0 \perp t_1$.

Properties (1/2)

- Confluence and progress are verified
- Subject reduction holds only for terminating terms or classical terms
- Orthogonality is Π_2^0-complete, but can be decided polynomially in most cases

Properties (2/2)

- We can compile a generic subset of terms of `Hyrql` into families of quantum circuits, given that they terminate.
- Size of the circuit: runtime of the term with polynomial overhead

Properties (2/2)

- We can compile a generic subset of terms of `Hyrql` into families of quantum circuits, given that they terminate.
- Size of the circuit: runtime of the term with polynomial overhead
- When they terminate in polynomial time, it gives a characterization of `FBQP`.

Properties (2/2)

- We can compile a generic subset of terms of `Hyrql` into families of quantum circuits, given that they terminate.
- Size of the circuit: runtime of the term with polynomial overhead
- When they terminate in polynomial time, it gives a characterization of `FBQP`.
- Termination and complexity certificate implies certificate on the circuit

3. Term rewrite systems and resource analysis

Term rewrite systems

- Represent a program as a set of rules $\mathcal{R}$

$$\texttt{Not}\ |0\rangle \to |1\rangle \qquad \texttt{Not}\ |1\rangle \to |0\rangle$$

- Active field for both termination and complexity analysis: different existing techniques
- Compile `Hyrql` into TRS

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

$$\lambda x.\lambda y.\mathtt{qcase}\, x \left\{ |0\rangle \to |0\rangle \otimes y \,,\, |1\rangle \to |1\rangle \otimes \mathtt{Not}\, y \right\}$$

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

$$\lambda x.\lambda y.\mathtt{qcase}\, x \left\{ |0\rangle \to |0\rangle \otimes y \,,\, |1\rangle \to |1\rangle \otimes \mathtt{Not}\, y \right\}$$

$$\mathcal{R} = \{\bot \to |0\rangle \otimes y\}$$

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

$$\lambda x.\lambda y.\mathtt{qcase}\, x \left\{ |0\rangle \to |0\rangle \otimes y \,,\, |1\rangle \to |1\rangle \otimes \mathtt{Not}\, y \right\}$$

$$\mathcal{R} = \{\bot \to |1\rangle \otimes \mathtt{Not}\, y\}$$

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

$$\lambda x.\lambda y.\mathtt{qcase}\, x \left\{ |0\rangle \to |0\rangle \otimes y \,, |1\rangle \to |1\rangle \otimes \mathtt{Not}\, y \right\}$$

$$\mathcal{R} = \{\bot \to |0\rangle \otimes y : |0\rangle / x, \bot \to |1\rangle \otimes \mathtt{Not}\, y : |1\rangle / x\}$$

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

$$\lambda x.\lambda y.\mathtt{qcase}\, x \left\{ |0\rangle \to |0\rangle \otimes y \,, |1\rangle \to |1\rangle \otimes \mathtt{Not}\, y \right\}$$

$$\mathcal{R} = \{ y \to |0\rangle \otimes y : |0\rangle / x, y \to |1\rangle \otimes \mathtt{Not}\, y : |1\rangle / x \}$$

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

$$\lambda x.\lambda y.\mathtt{qcase}\, x \left\{ |0\rangle \to |0\rangle \otimes y \,, |1\rangle \to |1\rangle \otimes \mathtt{Not}\, y \right\}$$

$$\mathcal{R} = \{ |0\rangle\ y \to |0\rangle \otimes y, |1\rangle\ y \to |1\rangle \otimes \mathtt{Not}\, y \}$$

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

$$\lambda x.\lambda y.\mathtt{qcase}\, x \left\{ |0\rangle \to |0\rangle \otimes y \,, |1\rangle \to |1\rangle \otimes \mathtt{Not}\, y \right\}$$

$$\mathcal{R} = \{\mathtt{CNot}\ |0\rangle\ y \to |0\rangle \otimes y, \mathtt{CNot}\ |1\rangle\ y \to |1\rangle \otimes \mathtt{Not}\, y\}$$

Translation idea

- Produce $\mathcal{R}$ inductively, producing rules for each branch of `qcase` / `match`
- Keep track of what branch was taken
- When t is closed and higher-order: associate $\mathtt{f}_t$

$$\lambda x.\lambda y.\mathtt{qcase}\, x \left\{ |0\rangle \to |0\rangle \otimes y \,, |1\rangle \to |1\rangle \otimes \mathtt{Not}\, y \right\}$$

$$\mathcal{R} = \{\mathtt{CNot}\ |0\rangle\ y \to |0\rangle \otimes y, \mathtt{CNot}\ |1\rangle\ y \to |1\rangle \otimes \mathtt{Not}\, y\}$$

- Restrict the syntax, but for size: $\mathcal{O}(n) \to \mathcal{O}(n^2)$

Translation properties

Given $\mathcal{R} = \texttt{Translate}(t)$:

- The translation is done in polynomial time;
- If t is well-typed, $\mathcal{R}$ is well-defined;
- $t, \mathcal{R}$ terminate on the same inputs, to the same outputs;
- When they terminate in k, l steps, $k = \Theta(l)$.

Actually: bound is more complex for higher-order terms, but still faithful.

Coming back on our example...

$$\texttt{letrec}\, f\ q = \lambda n.\texttt{qcase}\, q \left\{ \begin{array}{l} |0\rangle \to |0\rangle :: [\,] \\ \\ |1\rangle \to \texttt{match}\, n \left\{ \begin{array}{c} 0 \to |1\rangle :: [\,] \\ S(m) \to |1\rangle :: (f\ |+\rangle\ m) \end{array} \right\} \end{array} \right\}$$

Coming back on our example...

$$\texttt{letrec}\, f\ q = \lambda n.\texttt{qcase}\, q \left\{ \begin{array}{l} |0\rangle \to |0\rangle :: [\,] \\ \\ |1\rangle \to \texttt{match}\, n \left\{ \begin{array}{c} 0 \to |1\rangle :: [\,] \\ S(m) \to |1\rangle :: (f\ |+\rangle\ m) \end{array} \right\} \end{array} \right\}$$

`Translate(bqwalk)` yields the following system:

$$\left\{ \begin{array}{ll} \texttt{BQWalk}\ |1\rangle\ 0 \to |1\rangle :: [\,] & \texttt{BQWalk}\ |0\rangle\ n \to |0\rangle :: [\,] \\ \multicolumn{2}{c}{\texttt{BQWalk}\ |1\rangle\ S(n) \to |1\rangle :: \texttt{BQWalk}\,(\texttt{Had}\ |1\rangle)\ n} \end{array} \right\}$$

Coming back on our example...

$$\texttt{letrec}\, f\ q = \lambda n.\texttt{qcase}\, q \left\{ \begin{array}{l} |0\rangle \to |0\rangle :: [\,] \\ \\ |1\rangle \to \texttt{match}\, n \left\{ \begin{array}{c} 0 \to |1\rangle :: [\,] \\ S(m) \to |1\rangle :: (f\ |+\rangle\ m) \end{array} \right\} \end{array} \right\}$$

`Translate(bqwalk)` yields the following system:

$$\left\{ \begin{array}{ll} \texttt{BQWalk}\ |1\rangle\ 0 \to |1\rangle :: [\,] & \texttt{BQWalk}\ |0\rangle\ n \to |0\rangle :: [\,] \\ \multicolumn{2}{c}{\texttt{BQWalk}\ |1\rangle\ S(n) \to |1\rangle :: \texttt{BQWalk}\,(\texttt{Had}\ |1\rangle)\ n} \end{array} \right\}$$

- `BQWalk` terminates in size $\mathcal{O}(n)$ for inputs of size $\mathcal{O}(n)$.

Coming back on our example...

$$\texttt{letrec}\, f\, q = \lambda n.\texttt{qcase}\, q \left\{ \begin{array}{l} |0\rangle \to |0\rangle :: [\,] \\ |1\rangle \to \texttt{match}\, n \left\{ \begin{array}{c} 0 \to |1\rangle :: [\,] \\ S(m) \to |1\rangle :: (f\, |+\rangle\, m) \end{array} \right\} \end{array} \right\}$$

`Translate(bqwalk)` yields the following system:

$$\left\{ \begin{array}{ll} \texttt{BQWalk}\ |1\rangle\ 0 \to |1\rangle :: [\,] & \texttt{BQWalk}\ |0\rangle\ n \to |0\rangle :: [\,] \\ \multicolumn{2}{c}{\texttt{BQWalk}\ |1\rangle\ S(n) \to |1\rangle :: \texttt{BQWalk}\,(\texttt{Had}\ |1\rangle)\, n} \end{array} \right\}$$

- `BQWalk` terminates in size $\mathcal{O}(n)$ for inputs of size $\mathcal{O}(n)$.
- Quantum circuit is of polynomial size

Coming back on our example...

$$\texttt{letrec}\, f\, q = \lambda n.\texttt{qcase}\, q \left\{ \begin{array}{l} |0\rangle \to |0\rangle :: [\,] \\ |1\rangle \to \texttt{match}\, n \left\{ \begin{array}{c} 0 \to |1\rangle :: [\,] \\ S(m) \to |1\rangle :: (f\, |+\rangle\, m) \end{array} \right\} \end{array} \right\}$$

`Translate(bqwalk)` yields the following system:

$$\left\{ \begin{array}{ll} \texttt{BQWalk}\ |1\rangle\ 0 \to |1\rangle :: [\,] & \texttt{BQWalk}\ |0\rangle\ n \to |0\rangle :: [\,] \\ \multicolumn{2}{c}{\texttt{BQWalk}\ |1\rangle\ S(n) \to |1\rangle :: \texttt{BQWalk}\,(\texttt{Had}\ |1\rangle)\, n} \end{array} \right\}$$

- `BQWalk` terminates in size $\mathcal{O}(n)$ for inputs of size $\mathcal{O}(n)$.
- Quantum circuit is of polynomial size

4. Conclusion

Conclusion and Future Steps

- Build a hybrid language, with general recursion and both control flows

Conclusion and Future Steps

- Build a hybrid language, with general recursion and both control flows ✓

Conclusion and Future Steps

- Build a hybrid language, with general recursion and both control flows ✓
- Translate the language into a TRS

Conclusion and Future Steps

- Build a hybrid language, with general recursion and both control flows ✓
- Translate the language into a TRS ✓

Conclusion and Future Steps

- Build a hybrid language, with general recursion and both control flows ✓
- Translate the language into a TRS ✓
- Adapt existing termination techniques to provide criterion for complexity

Conclusion and Future Steps

- Build a hybrid language, with general recursion and both control flows ✓
- Translate the language into a TRS ✓
- Adapt existing termination techniques to provide criterion for complexity current work...

Conclusion and Future Steps

- Build a hybrid language, with general recursion and both control flows ✓
- Translate the language into a TRS ✓
- Adapt existing termination techniques to provide criterion for complexity current work...

Future steps:

- Study of quantum TRS
- Characterize complexity classes (both for space and time)
- Provide an actual compilation algorithm

Thanks for your attention !