Resource-Aware Quantum Programming with General Recursion and Quantum Control

Kostia Chardonnet
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Emmanuel Hainry
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Romain Péchoux
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Thomas Vinet
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

Abstract

This paper introduces the hybrid quantum language with general recursion `Hyrql`, driven towards resource-analysis. By design, `Hyrql` does not require the specification of an initial set of quantum gates. Hence, it is well amenable towards a generic cost analysis, unlike languages that use different sets of quantum gates, which yield quantum circuits of distinct complexity.

Regarding resource-analysis, we show how to relate the runtime of an expressive fragment of `Hyrql` programs with the size of the corresponding quantum circuits. We also manage to capture the class of functions computable in quantum polynomial time, which, by Yao's Theorem, corresponds to families of circuits of polynomial size. Consequently, this result paves the way for the use of termination and runtime-analysis techniques designed for classical programs to guarantee bounds on the size of quantum circuits.

2012 ACM Subject Classification Mathematics of computing → Lambda calculus; Theory of computation → Quantum complexity theory

Keywords and phrases Hybrid Quantum Programs, Resource Analysis

Digital Object Identifier 10.4230/LIPIcs.FSCD.2026.12

Funding This work is supported by the Plan France 2030 through the PEPR integrated project EPiQ (ANR-22-PETQ-0007) and the HQI platform (ANR-22-PNCQ-0002); by the European Union through the MSCA Staff Exchanges project QCOMICAL (Grant Agreement ID: 101182520); and by the Maison du Quantique MaQuEst.

1 Introduction

Motivations. Most well-known quantum algorithms, such as Shor's algorithm [45], were historically designed on the Quantum Random Access Machine (QRAM) model [31]. In this model, a program interacts with a quantum memory through basic operations complying with the laws of quantum mechanics. These operations include a fixed set of quantum gates, chosen to be universal, as well as a probabilistic measurement of qubits. Consequently, the control flow is purely classical, i.e., depends on the (classical) outcome of a measurement. Based on this paradigm, several high-level quantum programming languages have been introduced, each with different purposes and applications, from assembly code [17, 18], to imperative languages [25], circuit description languages [27], and λ-calculi [44].

A relevant issue was to extend these models to programs with *quantum control*, also known as coherent control, enabling a "program as data" treatment for the quantum paradigm. Quantum control consists in the ability to write superpositions of programs in addition to superposition of data and increases the expressive power of quantum programming

languages. It allows the programmer to write algorithms such as the *quantum switch*, which uses fewer resources (quantum gates) than algorithms with classical control [15] and is physically implementable [1, 40]. Hence, quantum control provides a computational advantage over classical control [3, 46, 32]. In the last decades, several quantum programming languages implementing this concept have been designed, non-exhaustively [2, 41, 23, 35, 53]. To get the best of both worlds, a natural next step was the development of *hybrid languages*, i.e., languages that allow classical and quantum flow/data to be combined. The hybrid paradigm is conveying considerable practical interest: for example, it is used by quantum variational algorithms, a class of quantum algorithms leveraging both classical and quantum computing resources to find approximate solutions to optimization problems; and the properties of hybrid languages have also been deeply studied, non-exhaustively [48, 51, 20, 9].

Since hybrid languages offer interesting prospects in terms of expressiveness and optimal resource consumption, a relevant and open issue concerns the development of resource-aware (static) analyses of their programs. These static analyses can be applied to predict the low-level resources required to execute quantum algorithms, such as the depth or size of a quantum circuit.

Contributions. This work solves the above issue for the first time by introducing a HYbrid Recursive Quantum Language `Hyrql` on which a resource analysis can be performed. `Hyrql` is a hybrid extension of the functional quantum language `Spm` (*Symmetrical pattern-matching*) of [41, 35]. In `Spm`, the programmer can directly write down unitary applications, combining quantum superposition of terms with pattern-matching. Unitarity is enforced by a (decidable) linear typing discipline using an orthogonality predicate. `Spm` is, however, a purely quantum language, which makes the manipulation (and, consequently, resource analysis) of classical information awkward, as it can be neither discarded nor duplicated by linearity. The hybrid nature of `Hyrql` relies on a typing discipline which delineates a clear separation between quantum/linear and classical/non-linear data in the typing contexts. Similarly to `Spm`, `Hyrql` does not include measurement: the flow from quantum data to classical data is handled through the use of a `shape` construct, as introduced in [27], which returns the classical information on the structure of data containing quantum information, without getting any information on the value of the quantum states. For example, the `shape` of a qubit list provides classical (duplicable) information, such as the number of qubits. The decision to provide `Hyrql` with a `Spm`-architecture was motivated by two important factors. First, in contrast with most of its competitors, `Spm` does not require the inclusion of an initial set of quantum gates, making its resource analysis generic. Second, thanks to its pattern-matching design, `Spm` allows for the reuse of a wide variety of tools designed for resource analysis (e.g., termination or complexity) of term rewriting systems.

Our paper contains the following main contributions:

- The introduction of the hybrid language with general recursion `Hyrql`, as well as its operational semantics, type system, and standard properties: confluence (Theorem 13), progress (Theorem 15), and subject reduction (Theorem 16).
- A proof that the orthogonality predicate in `Hyrql` is Π_2^0-complete, hence undecidable (Theorem 18). Consequently, typing is not decidable. This is not surprising as the language encompasses general recursion. The decidability of typing can still be recovered, on expressive sub-fragments of the language, e.g., when `Hyrql` is restricted to finite types and terminating programs (Proposition 19).
- A compilation from terms (of the good type) to quantum circuits, whose size is bounded by the runtime-complexity of the initial term (Theorem 27). In particular, terms that terminate in polynomial time characterize exactly the complexity class `FBQP` (Theorem 29

Table 1 Comparison table between hybrid quantum languages.

	`Hyrql`	`FOQ` [28]	`RQC++` [52]	`DLPZ` [20]	`Silq` [11]	`Qunity` [48]
Resource-aware	✓	✓	✗	✗	✗	✗
General recursion	✓	✗	✓	✗	✗	✗
Higher-order	✓	✗	✗	✓	✗	✗

and 30). This result is not trivial, as lambda calculi do not comply with van Emde Boas' invariant thesis [47], i.e., a polynomial time reduction on a lambda term may correspond to an exponential time reduction on a Turing machine.

- An overview on how complexity results can be obtained and automatized by translating our language into term-rewrite systems, to use existing techniques for runtime analysis (e.g., interpretations [36], recursive path orderings [21, 22], dependency pairs [4], size-change principle [34]).

We illustrate the expressive power of the language through several examples: implementation of basic quantum gates (Hadamard gate, Example 1), higher-order (quantum switch, Example 2), quantum algorithms (quantum Fourier transform, Example 3), and general recursion (bounded quantum walk, Example 4).

Related works. In Table 1, we provide a non-exhaustive comparison between the main families of hybrid quantum languages which treat classical data as first-class citizen. Towards that end, we consider the three following criteria.

- *Resource-aware* highlights if the language is designed towards resource analysis, that is, allows to characterize well-known quantum complexity classes. This is the case of `FOQ` [28, 29], which characterizes, under restrictions, quantum polynomial time and quantum polylogarithmic time [26]. However, `Hyrql` has a greater expressive power than `FOQ` as it is not restricted to unitary operators. Moreover, all standard inductive datatypes can be expressed in `Hyrql`, whereas `FOQ` is restricted to lists of qubits.
- *General recursion* specifies whether the language includes unbounded recursion on classical data. The ability to handle this type of recursion is necessary for a resource analysis to be relevant (i.e., not trivial as in the case of strong normalizing languages).
- *Higher-order* denotes the ability of the language to feature general higher-order terms, i.e., functions taking functions as input. While some languages of Table 1 have syntactic constructs to do so, most of them are restricted to first-order calls.

As we are interested in resource analysis in the general setting, Table 1 only compares `Hyrql` with hybrid languages, rather than with existing languages with only quantum control, non-exhaustively [41, 23, 51, 53]. Other studies have already been carried out on resource analysis in quantum programming languages. They use techniques (e.g., type systems) developed in the field of Implicit Computational Complexity (ICC, see [39]) to characterize quantum complexity classes [19], to infer the expected cost or expected value of a quantum program [6], or to infer upper bounds on quantum resources (depth and size of circuits) [16]. However, these studies are restricted to classical control. For example, [19] studies the use of soft linear logic [33] on a quantum lambda calculus [43], [6] adapts expectation transformers on a language based on `QPL` [42], and [16] develops a dependent type system on a variant of the Quipper circuit description language [27].

Table 2 Syntax of the `Hyrql` language.

$$
\begin{array}{lll}
\text{(Values)} & v ::= x & \mid |0\rangle \mid |1\rangle \mid c(v_1,\ldots,v_n) \mid \lambda x.t \mid \texttt{letrec}\, f\, x = t \mid \sum_{i=1}^{n} \alpha_i \cdot v_i \\
\text{(Terms)} & t ::= x & \mid |0\rangle \mid |1\rangle \mid \texttt{qcase}\, t \left\{|0\rangle \to t_0\,, |1\rangle \to t_1\right\} \\
 & & \mid c(t_1,\ldots,t_n) \mid \texttt{match}\, t\, \{c_1(\overrightarrow{x_1}) \to t_1\,, \ldots, c_n(\overrightarrow{x_n}) \to t_n\} \\
 & & \mid \lambda x.t \mid \texttt{letrec}\, f\, x = t \mid t_1 t_2 \mid \sum_{i=1}^{n} \alpha_i \cdot t_i \mid \texttt{shape}(t)
\end{array}
$$

2 The `Hyrql` Programming Language

We introduce the syntax, the operational semantics, and the type system of `Hyrql` along with illustrating examples.

2.1 Syntax

The syntax of Terms and Values in `Hyrql` is provided in Table 2. Terms feature variables, denoted $x, y, f, \ldots$ and taken from an infinite countable set. We denote by $\overrightarrow{e}$ a (finite and possible empty) sequence of elements $e_1, \ldots, e_n$. Qubits are introduced through basis states $|0\rangle, |1\rangle$, allowing us to write quantum conditionals $\texttt{qcase}\, t \left\{|0\rangle \to t_0\,, |1\rangle \to t_1\right\}$, which correspond to the superposition of t_0 and t_1 controlled by qubit t. `Hyrql` also features classical constructors c^s consisting of a constructor symbol c, and a signature s indicating the type of the constructor. Signatures will be formalized in Section 2.3 and will be explicitly given only when required. Constructors can be used in constructor application $c(t_1, \ldots, t_n)$ and through pattern-matching, sometimes abbreviated as $\texttt{match}_{1\leq i\leq n}\, t\, \{c_i(\overrightarrow{x_i}) \to t_i\}$. Each constructor symbol c comes with a fixed arity and is always fully applied. We assume the existence of standard inductive constructors for unit $()$, tensor products $\otimes$, natural numbers 0 and S, and lists $[\,]$ and $::$. For convenience, constructors are sometimes used in an infix notation. For example, tensor products will be written as $x \otimes y$ and list constructors as $h :: t$.

Higher-order is featured via a standard λ-abstraction $\lambda x.t$, and a construct for general recursion $\texttt{letrec}\, f\, x = t$. Term application is denoted by $t_1 t_2$.

Given amplitudes $\alpha_i \in \mathbb{C}$, the term $\sum_{i=1}^{n} \alpha_i \cdot t_i$ represents a *superposition* of terms t_i. In the special case where $n = 2$, we just write $\alpha_1 \cdot t_1 + \alpha_2 \cdot t_2$. In a dual manner, the `shape` construct, introduced in [27], returns the classic structural information on data containing quantum information. For example, the shape of a qubit list is a copy of the list structure without the qubits (see Example 7).

In order to avoid conflicts between free and bound variables, we will always work up to α-conversion and use Barendregt's convention [8, p. 26] which consists in keeping all bound and free variable names distinct, even when this remains implicit. We also define a partial order $\lhd$ on terms, where $s \lhd t$ if s is a *strict subterm* of t. We denote $\trianglelefteq$ as the reflexive closure of $\lhd$. The *size of a term* is written $|t|$ and is defined standardly.

As we work with amplitudes and superpositions, Terms and Values have to be considered with respect to a vector space structure. Towards that end, we introduce an equivalence relation $\equiv$ in Table 3. Such definition relies on *equivalence contexts*, defined by the following grammar:

$$
\begin{array}{lll}
\text{(Equiv. contexts)} & C_{\equiv} ::= \diamond & \mid \texttt{qcase}\, C_{\equiv} \left\{|0\rangle \to t_0\,, |1\rangle \to t_1\right\} \mid c(\overrightarrow{t_1}, C_{\equiv}, \overrightarrow{t_2}) \mid t C_{\equiv} \mid C_{\equiv} t \\
 & & \mid \texttt{match}_{1\leq i\leq n}\, C_{\equiv}\, \{c_i(\overrightarrow{x_i}) \to t_i\} \mid t + C_{\equiv} \mid \texttt{shape}(C_{\equiv})
\end{array}
$$

Table 3 Equivalence relation $\equiv \subseteq$ Terms $\times$ Terms.

$$t_1 + t_2 \equiv t_2 + t_1 \qquad t_1 + (t_2 + t_3) \equiv (t_1 + t_2) + t_3 \qquad 1 \cdot t \equiv t \qquad t + 0 \cdot t' \equiv t$$
$$\alpha \cdot (\beta \cdot t) \equiv (\alpha\beta) \cdot t \qquad \alpha \cdot (t_1 + t_2) \equiv \alpha \cdot t_1 + \alpha \cdot t_2 \qquad \alpha \cdot t + \beta \cdot t \equiv (\alpha + \beta) \cdot t$$

$$\texttt{qcase}\,(\textstyle\sum_{j=1}^m \alpha_j \cdot s_j)\,\{|0\rangle \to t_0\,, |1\rangle \to t_1\} \equiv \sum_{j=1}^m \alpha_j \cdot (\texttt{qcase}\, s_j\,\{|0\rangle \to t_0\,, |1\rangle \to t_1\})$$
$$c(\overrightarrow{t_1}, \textstyle\sum_{j=1}^m \alpha_j \cdot s_j, \overrightarrow{t_2}) \equiv \sum_{j=1}^m \alpha_j \cdot c(\overrightarrow{t_1}, s_j, \overrightarrow{t_2})$$
$$\texttt{match}_{1\le i\le n}\,(\textstyle\sum_{j=1}^m \alpha_j \cdot s_j)\,\{c_i(\overrightarrow{x_i}) \to t_i\} \equiv \sum_{j=1}^m \alpha_j \cdot (\texttt{match}_{1\le i\le n}\, s_j\,\{c_i(\overrightarrow{x_i}) \to t_i\})$$
$$t(\textstyle\sum_{i=1}^m \alpha_j \cdot s_j) \equiv \sum_{j=1}^m \alpha_j \cdot t\, s_j$$
$$C_\equiv[t] \equiv C_\equiv[t'] \qquad \text{when } t \equiv t'$$

An equivalence context $C_\equiv$ is thus a term with one hole $\diamond$; let $C_\equiv[t]$ be the term obtained by filling the hole with t in $C_\equiv$. In Table 3, the rules from the two first lines make the definition of the $\sum$ construct unambiguous/sound, i.e., it corresponds exactly to repeated applications of the $+$ construct. The five last rules of Table 3 are commutation rules highlighting the linearity of quantum terms.

▶ **Example 1** (Quantum State and Hadamard Gate)**.** We can define the orthogonal basis states as $|\pm\rangle \triangleq \frac{1}{\sqrt{2}} \cdot |0\rangle \pm \frac{1}{\sqrt{2}} \cdot |1\rangle$. The Hadamard gate can be encoded in `Hyrql`:

$$\texttt{Had} \triangleq \lambda x.\texttt{qcase}\, x\,\{|0\rangle \to |+\rangle\,, |1\rangle \to |-\rangle\}$$

▶ **Example 2** (Quantum Switch)**.** The quantum switch [15] is a program which, on two unitaries U, V and a two-qubit state $|\phi\rangle$, applies UV and VU, in superposition, to the second qubit, controlled by the value of the first qubit. It can be implemented in `Hyrql` by $\texttt{QS}\ U\ V\ |\phi\rangle$, where `QS` is defined as:

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

▶ **Example 3** (Quantum Fourier Transform)**.** The well-known Quantum Fourier Transform (QFT) can be encoded in our language by the following terms:

$$\texttt{cphase} \triangleq \lambda x.\lambda n.\texttt{match}\, x\,\{c \otimes t \to \texttt{qcase}\, c\,\{|0\rangle \to |1\rangle \otimes t\,, |1\rangle \to |0\rangle \otimes \phi\, n\, t\}\}$$
$$\texttt{rot} \triangleq \texttt{letrec}\, f\, q = \lambda l.\lambda n.\texttt{match}\, l \left\{ \begin{array}{l} [\,] \to q \otimes [\,] \\ h :: t \to \texttt{let}\, h' \otimes q' = \texttt{cphase}(h \otimes q)(n)\, \texttt{in} \\ \qquad\quad \texttt{let}\, q'' \otimes t' = f(q')(t)(S(n))\, \texttt{in} \\ \qquad\quad q'' \otimes (h' :: t') \end{array} \right\}$$
$$\texttt{rec} \triangleq \texttt{letrec}\, f\, l = \texttt{match}\, l \left\{ \begin{array}{l} [\,] \to [\,] \\ h :: t \to \texttt{let}\, h' \otimes t' = \texttt{rot}(\texttt{had}\, h)(t)(2)\, \texttt{in}\, h' :: (f\, t') \end{array} \right\}$$
$$\texttt{swap} \triangleq \texttt{letrec}\, f\, l = \lambda u.\texttt{match}\, l\,\{[\,] \to u, h :: t \to \texttt{swap}(t)(h :: u)\}$$
$$\texttt{qft} \triangleq \lambda l.\texttt{swap}(\texttt{rec}(l), [\,])$$

Table 4 Reduction rules of the language.

$$\frac{}{\texttt{qcase}\ |0\rangle\ \{|0\rangle \to t_0\,, |1\rangle \to t_1\} \rightsquigarrow t_0}\ (\text{Qcase}_0) \qquad \frac{}{\texttt{qcase}\ |1\rangle\ \{|0\rangle \to t_0\,, |1\rangle \to t_1\} \rightsquigarrow t_1}\ (\text{Qcase}_1)$$

$$\frac{}{\texttt{match}_{1\le i\le n}\ c_j(\overrightarrow{v_p})\ \{c_i(\overrightarrow{x_i}) \to t_i\} \rightsquigarrow t_j\{\overrightarrow{v_p}/\overrightarrow{x_j}\}}\ (\text{Match}) \qquad \frac{}{(\lambda x.t)v_p \rightsquigarrow t\{v_p/x\}}\ (\text{Lbd})$$

$$\frac{}{(\texttt{letrec}\ f\ x = t)v_p \rightsquigarrow t\{\texttt{letrec}\ f\ x = t/f, v_p/x\}}\ (\text{Rec})$$

$$\frac{\sum_{i=1}^n \alpha_i \cdot p_i \in \texttt{CAN} \setminus \text{Value} \qquad p_i \rightsquigarrow^? t_i}{\sum_{i=1}^n \alpha_i \cdot p_i \rightsquigarrow \sum_{i=1}^n \alpha_i \cdot t_i}\ (\text{Can}) \qquad \frac{}{\texttt{shape}(|0\rangle) \rightsquigarrow ()}\ (\text{Shape}_0)$$

$$\frac{}{\texttt{shape}(|1\rangle) \rightsquigarrow ()}\ (\text{Shape}_1) \qquad \frac{}{\texttt{shape}(c^s(\overrightarrow{v_p})) \rightsquigarrow c^{\texttt{shape}(s)}(\texttt{shape}(\overrightarrow{v_p}))}\ (\text{Shape}_c)$$

$$\frac{\sum_{i=1}^n \alpha_i \cdot v_i \in \texttt{CAN}}{\texttt{shape}(\sum_{i=1}^n \alpha_i \cdot v_i) \rightsquigarrow \texttt{shape}(v_1)}\ (\text{Shape}_s) \qquad \frac{t \rightsquigarrow t'}{\texttt{shape}(t) \rightsquigarrow \texttt{shape}(t')}\ (\text{Shape}_E)$$

$$\frac{p \rightsquigarrow t}{E[p] \rightsquigarrow E[t]}\ (E) \qquad \frac{t \equiv t_1 \qquad t_1 \rightsquigarrow t_1' \qquad t_1' \equiv t'}{t \rightsquigarrow t'}\ (\text{Equiv})$$

Note that, as our natural numbers are defined recursively, it is not possible to directly build a phase $1/2^n$. We thus assume that we are given a term ϕ such that $\phi\, n\ |0\rangle = |0\rangle$ and $\phi\, n\ |1\rangle = e^{i\pi/2^{n-1}}\ |1\rangle$. To shorten terms, we also use the notations $\texttt{let}\, x \otimes y = s\ \texttt{in}\, t \triangleq \texttt{match}\, s\, \{x \otimes y \to t\}$ and $2 \triangleq S(S(0))$.

▶ **Example 4** (Bounded quantum walk)**.** `Hyrql` allows one to write recursive programs that are classically and quantumly controlled. The program `bqwalk` produces all the possible quantum walk paths of size at most n, starting from a quantum state q:

$$\texttt{repeat} \triangleq \texttt{letrec}\, g\, n = \texttt{match}\, n\, \{0 \to [\,], S(m) \to |0\rangle :: g\, m\}$$

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

In Example 3, as expected, for a given size of the list argument, the program can be compiled to a fixed quantum circuit. On the contrary, the programs from Examples 2 and 4 do not not naturally compile to a single quantum circuit when the size of the arguments is fixed. Thus, `Hyrql` is more than a circuit description language [27].

2.2 Operational Semantics

In this section, we define the call-by-value operational semantics of `Hyrql`. As we are interested in resource analysis, we want to avoid reducing non-meaningful terms, e.g., a reduction of t in $s + 0 \cdot t \ \equiv\ s$ is meaningless. Towards that end, we introduce *canonical forms* (Definition 5), which take care of removing ill-behaved terms before applying a reduction. It

relies on *pure terms*, which are defined by the following grammar:

$$\begin{aligned}\text{(Pure terms)}\quad p ::= x \mid |0\rangle \mid |1\rangle \mid \texttt{qcase}\, p\left\{|0\rangle \to t_0\,, |1\rangle \to t_1\right\} \mid c(p_1,\dots,p_n)\\ \mid \texttt{match}_{1\le i\le n}\, p\,\{c_i(\overrightarrow{x_i}) \to t_i\} \mid \lambda x.t \mid \texttt{letrec}\, f\, x = t \mid tp \mid \texttt{shape}(t)\end{aligned}$$

A *pure value*, denoted v_p, is a pure term that is a value.

▶ **Definition 5** (Canonical form). *A canonical form of a term t is any term $\sum_{i=1}^{n} \alpha_i \cdot p_i$ such that $t \equiv \sum_{i=1}^{n} \alpha_i \cdot p_i$, where p_i are pure terms, $\alpha_i \neq 0$ and $p_i \equiv p_j$ implies $i = j$. The set of canonical forms is denoted by* CAN.

The canonical form always exists and is unique for well-typed terms (Lemma 14). Note that the naming *pure terms* comes from [23] and is not related to *pure states* in quantum computing, as pure terms could semantically yield a mixed quantum state.

We also define *evaluation contexts* by the following grammar:

$$\begin{aligned}\text{(Evaluation contexts)}\quad E ::= \diamond \mid \texttt{qcase}\, E\left\{|0\rangle \to t_0\,, |1\rangle \to t_1\right\} \mid c(\overrightarrow{p}, E, \overrightarrow{v_p})\\ \mid \texttt{match}_{1\le i\le n}\, E\,\{c_i(\overrightarrow{x_i}) \to t_i\} \mid tE \mid Ev_p\end{aligned}$$

Again, let $E[t]$ be the term obtained by filling the hole $\diamond$ with t in E.

The operational semantics of Hyrql is described in Table 4 as a reduction relation $\rightsquigarrow \subseteq$ Terms $\times$ Terms, where $\{t/x\}$ denotes the standard substitution of a variable x by a term t. The reduction implements a *call-by-value* strategy.

In the rule (Can), $t \rightsquigarrow^{?} t'$ holds if either $t \rightsquigarrow t'$ or, $\neg(\exists t'',\ t \rightsquigarrow t'')$ and $t' = t$. Intuitively, it means that we apply one reduction for each element of a superposition, when possible. In the rule (Shape$_c$), $\texttt{shape}(\overrightarrow{v})$ is syntactic sugar for $\texttt{shape}(v_1),\dots,\texttt{shape}(v_n)$, and the shape of a signature $\texttt{shape}(s)$ is defined formally in Definition 9. In all rules of Table 4, except for the (Can) and (Equiv) rules, the left-hand-side term is a pure term (i.e., not a superposition). This implies that any summation must be expressed as a canonical form, which avoids reducing a subterm with a null amplitude, or more generally to reduce two identical terms that would sum up to 0. The goal is to avoid reductions that have no physical meaning, as this would invalidate any resource analysis result.

We define $\rightsquigarrow^*$ as the reflexive and transitive closure of $\rightsquigarrow$. For $k \in \mathbb{N}$, we also write $t \rightsquigarrow^{\le k} t'$, when t reduces to t' in at most k steps. A term t *terminates*, if any chain of reductions starting from t reaches a value, meaning that $t \rightsquigarrow^{\le k} v$ holds for some k.

▶ **Example 6.** Let us consider the Had program from Example 1, we can check that $\texttt{Had}\,|0\rangle$ reduces to the expected value:

$$\begin{aligned}\texttt{Had}\,|0\rangle &\rightsquigarrow \texttt{qcase}\ |0\rangle\left\{|0\rangle \to |+\rangle\,, |1\rangle \to |-\rangle\right\} && \text{via (Lbd)}\\ &\rightsquigarrow |+\rangle && \text{via (Qcase}_0\text{)}\end{aligned}$$

Since the Hadamard gate is its own inverse, $\texttt{Had}\,|+\rangle$ is expected to reduce to $|0\rangle$. Reducing $\texttt{Had}\,|+\rangle$ cannot be done using (Lbd), as $|+\rangle$ is not a pure term, but rather by developing the sum and reducing each element. Let $h_v \triangleq \texttt{qcase}\ |v\rangle\left\{|0\rangle \to |+\rangle\,, |1\rangle \to |-\rangle\right\}$ for $v \in \{0,1\}$. The following reduction holds:

$$\dfrac{\dfrac{\dfrac{}{\texttt{Had}\,|0\rangle \rightsquigarrow h_0}\ \text{(Lbd)} \qquad \dfrac{}{\texttt{Had}\,|1\rangle \rightsquigarrow h_1}\ \text{(Lbd)}}{\frac{1}{\sqrt{2}}\cdot\texttt{Had}\,|0\rangle + \frac{1}{\sqrt{2}}\cdot\texttt{Had}\,|1\rangle \rightsquigarrow \frac{1}{\sqrt{2}}\cdot h_0 + \frac{1}{\sqrt{2}}\cdot h_1}\ \text{(Can)} \qquad \frac{1}{\sqrt{2}}\cdot\texttt{Had}\,|0\rangle + \frac{1}{\sqrt{2}}\cdot\texttt{Had}\,|1\rangle \equiv \texttt{Had}\,|+\rangle}{\texttt{Had}\,|+\rangle \rightsquigarrow \frac{1}{\sqrt{2}}\cdot h_0 + \frac{1}{\sqrt{2}}\cdot h_1}\ \text{(Equiv)}$$

We have expressed $\texttt{Had}\,|+\rangle$ as its canonical form, then reduced simultaneously each element of the superposition. Similarly, one can derive the reduction $\frac{1}{\sqrt{2}} \cdot h_0 + \frac{1}{\sqrt{2}} \cdot h_1 \rightsquigarrow |0\rangle$, by rewriting the reduced term via $\equiv$. Chaining both results yields $\texttt{Had}\,|+\rangle \rightsquigarrow^2 |0\rangle$ as expected.

▶ **Example 7.** Let us define the qubit list $l \triangleq |+\rangle :: |0\rangle :: [\,]$. Its shape $\texttt{shape}(l)$ reduces as follows:

$$\begin{aligned}\texttt{shape}(|+\rangle :: |0\rangle :: [\,]) &\rightsquigarrow \texttt{shape}(|0\rangle :: |0\rangle :: [\,])\\ &\rightsquigarrow \texttt{shape}(|0\rangle) :: \texttt{shape}(|0\rangle :: [\,])\\ &\rightsquigarrow \texttt{shape}(|0\rangle) :: \texttt{shape}(|0\rangle) :: \texttt{shape}([\,])\\ &\rightsquigarrow^* () :: () :: [\,]\end{aligned}$$

As $\texttt{shape}(|+\rangle :: |0\rangle :: [\,]) \equiv \texttt{shape}(\frac{1}{\sqrt{2}} \cdot |0\rangle :: |0\rangle :: [\,] + \frac{1}{\sqrt{2}} \cdot |1\rangle :: |0\rangle :: [\,])$, which reduces to $\texttt{shape}(|0\rangle :: |0\rangle :: [\,])$ using (Shape_s), the first reduction $\texttt{shape}(|+\rangle :: |0\rangle :: [\,]) \rightsquigarrow \texttt{shape}(|0\rangle :: |0\rangle :: [\,])$ is derived using (Equiv). The remaining reductions are derived by applying either (Shape_0) or (Shape_c), and where :: and $[\,]$ are written without signatures.

Now, let us derive the length of the list l, with the help of the following term:

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

One can then derive the following reductions, starting from $\texttt{len}(\texttt{shape}(l))$:

$$\begin{aligned}\texttt{len}(\texttt{shape}(l)) &\rightsquigarrow^* \texttt{len}(() :: () :: [\,])\\ &\rightsquigarrow \texttt{match}\,() :: () :: [\,]\, \big\{[\,] \to 0, h :: t \to S(\texttt{len}\, t)\big\}\\ &\rightsquigarrow S(\texttt{len}(() :: [\,])) \rightsquigarrow^* S(S(\texttt{len}([\,]))) \rightsquigarrow^* S(S(0))\end{aligned}$$

While this produces the expected result, one may want to derive the length from the initial term $\texttt{len}(l)$ instead. However, the same process would yield $\texttt{len}(l) \rightsquigarrow^* \sqrt{2}S(S(0))$, which is a non-normalized term. Furthermore, len is a function that discards its input, thus does not coincide with an expected linear treatment of quantum data, as discussed in the next section; in particular, $\texttt{len}(l)$ is not a typable term of our language, while $\texttt{len}(\texttt{shape}(l))$ is (see Example 11).

2.3 Type System

Types and Contexts. Types in Hyrql are provided by the following grammar:

$$\text{(Types)} \qquad T ::= \text{Qbit} \mid B \mid T \multimap T \mid T \Rightarrow T$$

Qbit is the type for qubits and B is a *constructor type* from a fixed set $\mathbb{B}$ containing the unit type $\mathbb{1}$. A *basic type*, noted $B^{|\rangle}$, is either a qubit type, or a constructor type B, i.e., $B^{|\rangle} \in \{\text{Qbit}\} \cup \mathbb{B}$. A constructor symbol c comes with a signature $s = B_1^{|\rangle}, \ldots, B_n^{|\rangle} \to B$ of arity n, defining the inputs and output types of c. For simplicity, we write $s = B$ when s is of arity 0. Each constructor type is defined by its set of constructors:

$$\text{Cons}(B) \triangleq \{c^s \mid s = B_1^{|\rangle}, \ldots, B_n^{|\rangle} \to B\}$$

By construction, $\text{Cons}(B) \cap \text{Cons}(B') = \emptyset$ when $B \neq B'$. We consider the following constructor types: the unit type $\mathbb{1}$ with a unique constructor $()^{\mathbb{1}}$; tensor types $B_1^{|\rangle} \otimes B_2^{|\rangle}$ for given types $B_1^{|\rangle}, B_2^{|\rangle}$ with a unique constructor $\otimes^{B_1^{|\rangle}, B_2^{|\rangle} \to B_1^{|\rangle} \otimes B_2^{|\rangle}}$; natural numbers nat with constructors $0^{\texttt{nat}}$ and $S^{\texttt{nat} \to \texttt{nat}}$; lists $\texttt{list}(B^{|\rangle})$ for a given type $B^{|\rangle}$, with constructors $[\,]^{\texttt{list}(B^{|\rangle})}$ and $::^{B^{|\rangle}, \texttt{list}(B^{|\rangle}) \to \texttt{list}(B^{|\rangle})}$.

The language also features two distinct *higher-order* types for linear and non-linear functions, denoted respectively by $\multimap$ and $\Rightarrow$.

In the typing discipline, it will be useful to distinguish between types based on whether or not they contain quantum data. The latter must follow the laws of quantum mechanics (no-cloning), whereas classical types are more permissive. Towards that end, we define inductively the set of *quantum constructor types* $\mathbb{B}_Q$ by

$$\mathbb{B}_Q \triangleq \{B \in \mathbb{B} \mid \exists\, c^{B_1^{|\rangle},\dots,B_n^{|\rangle}\to B}, \exists\, i, B_i^{|\rangle} \in \mathbb{B}_Q \cup \{\text{Qbit}\}\}.$$

A quantum constructor type $B_Q \in \mathbb{B}_Q$ has (at least) a constructor symbol with a type Qbit or (inductively) a quantum constructor type in its signature. The set $\mathbb{B}_C$ of *classical constructor types* B_C is defined by $\mathbb{B}_C \triangleq \mathbb{B} \setminus \mathbb{B}_Q$. For example, $\mathbb{1}$, $\texttt{nat} \in \mathbb{B}_C$ whereas $\texttt{list}(\text{Qbit}) \in \mathbb{B}_Q$. This allows us to split the types between *quantum types* Q and *classical types* C:

$$Q \;::=\; \text{Qbit} \mid B_Q \qquad C \;::=\; B_C \mid T \multimap T \mid T \Rightarrow T$$

It is now assumed that the inputs of each constructor are ordered, containing first variables of classical type, then variables of quantum type, enabling a hybrid behaviour.

Typing contexts are defined as follows:

$$\Gamma, \Delta \quad ::= \quad \varnothing \mid \{\, x : T \,\} \mid \{\, [x : T] \,\} \mid \Gamma \cup \Delta,$$

where $[x : T]$ is called a *boxed variable*, indicating that x is a linear variable captured by a `shape` construct. We define the *domain* of a context Δ as the set of its variables, i.e., $\mathrm{dom}(\Delta) \triangleq \{\, x \mid \exists T, x : T \in \Delta \vee [x : T] \in \Delta \,\}$. Whenever we write Γ, Δ or $\Gamma; \Delta$, it is assumed that Γ and Δ are *compatible*, that is, $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$. We use the shorthand notation $\overrightarrow{x} : \overrightarrow{T}$ for $x_1 : T_1, \dots, x_n : T_n$; and $[\Delta]$ for $[x_1 : T_1], \dots, [x_n : T_n]$.

Typing rules. A *typing judgment* is written $\Gamma; \Delta \vdash t : T$, describing that the term t is *well-typed*, with type T, under *non-linear context* Γ and *linear context* Δ. The typing rules of the language are defined in Table 5. A well-typed term $\Gamma; \Delta \vdash t : T$ is *closed* if $\Gamma = \Delta = \emptyset$. A well-typed closed term t is said to be *total*, if for any well-typed closed and total values $v_1, \dots, v_n$ such that $tv_1 \dots v_n$ is a well-typed closed term of type $B^{|\rangle}$, $tv_1 \dots v_n$ terminates.

Typing rules rely on an external *orthogonality* predicate $\perp$ on terms. This predicate uses the notion of Θ-*context substitutions*, which, for a given context Θ, denote the substitutions σ such that for any $x : T$ with $x : T \in \Theta$ or $[x : T] \in \Theta$, $\sigma(x) = v$, where v is a well-typed closed and total value of type T. Typing also requires the definition of an an *inner product* on values with a canonical form, defined below, where $\delta_{x,y}$ is the Kronecker symbol.

$$\langle v, w \rangle \triangleq \sum_{i=1}^{n}\sum_{j=1}^{m} \alpha_i \beta_j^* \delta_{v_i, w_j}, \qquad \text{where } v \equiv \sum_{i=1}^{n} \alpha_i \cdot v_i \in \texttt{CAN} \text{ and } w \equiv \sum_{j=1}^{m} \beta_j \cdot w_j \in \texttt{CAN}$$

▶ **Definition 8** (Orthogonality). *Let* $\Gamma; \Delta \vdash t : Q$ *and* $\Gamma; \Delta \vdash t' : Q$ *be two well-typed terms.* t *and* t' *are* orthogonal*, written* $t \perp t'$*, if for any* $\Gamma \cup \Delta$*-context substitution* σ*:*

- $\sigma(\texttt{shape}(t)) \rightsquigarrow^* v$*,* $\sigma(\texttt{shape}(t')) \rightsquigarrow^* v'$ *and* $v = v'$*;*
- $\sigma(t) \rightsquigarrow^* w \in \texttt{CAN}$*,* $\sigma(t') \rightsquigarrow^* w' \in \texttt{CAN}$*, and* $\langle w, w' \rangle = 0$*.*

These conditions imply that the reduced values share the same classical structure, and that their quantum data is orthogonal. For example, orthogonal lists must be of the same size. Orthogonality is used in the typing rules (qcase) and (sup) of Table 5 to ensure the

Table 5 Typing rules of the language.

$$\frac{}{\Gamma; x : T \vdash x : T}\ (\text{ax}) \qquad \frac{}{\Gamma, x : C; \varnothing \vdash x : C}\ (\text{ax}_c) \qquad \frac{}{\Gamma; \varnothing \vdash |0\rangle : \text{Qbit}}\ (\text{ax}_0)$$

$$\frac{}{\Gamma; \varnothing \vdash |1\rangle : \text{Qbit}}\ (\text{ax}_1) \qquad \frac{\Gamma; \Delta \vdash t : \text{Qbit} \qquad \Gamma; \Delta' \vdash t_0 : Q \qquad \Gamma; \Delta' \vdash t_1 : Q \qquad t_0 \perp t_1}{\Gamma; \Delta, \Delta' \vdash \texttt{qcase}\, t \left\{ |0\rangle \to t_0 \,, |1\rangle \to t_1 \right\} : Q}\ (\text{qcase})$$

$$\frac{s = B_1^{|\rangle}, \ldots, B_n^{|\rangle} \to B \qquad \Gamma; \Delta_i \vdash t_i : B_i^{|\rangle}}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash c^s(t_1, \ldots, t_n) : B}\ (\text{cons})$$

$$\frac{\text{Cons}(B) = \{\, c_i^{s_i} \,\}_{i=1}^n \qquad s_i = \overrightarrow{C_i}, \overrightarrow{Q_i} \to B \qquad \Gamma_1; \Delta_1 \vdash t : B \qquad \Gamma_2, \overrightarrow{y_i} : \overrightarrow{C_i}; \Delta_2, \overrightarrow{z_i} : \overrightarrow{Q_i} \vdash t_i : B^{|\rangle}}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash \texttt{match}_{1 \le i \le n}\, t\, \{c_i^{s_i}(\overrightarrow{y_i}, \overrightarrow{z_i}) \to t_i\} : B^{|\rangle}}\ (\text{match})$$

$$\frac{\Gamma; \Delta, x : T \vdash t : T'}{\Gamma; \Delta \vdash \lambda x.t : T \multimap T'}\ (\text{abs}) \qquad \frac{\Gamma, x : C; \Delta \vdash t : T}{\Gamma; \Delta \vdash \lambda x.t : C \Rightarrow T}\ (\text{abs}_c) \qquad \frac{\Gamma, f : T; \varnothing \vdash \lambda x.t : T}{\Gamma; \varnothing \vdash \texttt{letrec}\, f\, x = t : T}\ (\text{rec})$$

$$\frac{\Gamma; \Delta \vdash t_1 : T \multimap T' \qquad \Gamma; \Delta' \vdash t_2 : T}{\Gamma; \Delta, \Delta' \vdash t_1 t_2 : T'}\ (\text{app}) \qquad \frac{\Gamma; \Delta \vdash t_1 : C \Rightarrow T \qquad \Gamma; \varnothing \vdash t_2 : C}{\Gamma; \Delta \vdash t_1 t_2 : T}\ (\text{app}_c)$$

$$\frac{\Gamma; \Delta \vdash t_i : Q \quad \sum_{i=1}^n |\alpha_i|^2 = 1 \quad \forall i \neq j,\ t_i \perp t_j}{\Gamma; \Delta \vdash \sum_{i=1}^n \alpha_i \cdot t_i : Q}\ (\text{sup}) \qquad \frac{\Gamma; \Delta \vdash t : B^{|\rangle}}{\Gamma, [\Delta]; \varnothing \vdash \texttt{shape}(t) : \texttt{shape}(B^{|\rangle})}\ (\text{shape})$$

$$\frac{\Gamma, [x : B^{|\rangle}]; \Delta, y : B^{|\rangle} \vdash t : T}{\Gamma; \Delta, y : B^{|\rangle} \vdash t\{y/x\} : T}\ (\text{contr}) \qquad \frac{\Gamma; \Delta \vdash t : T \qquad t \equiv t'}{\Gamma; \Delta \vdash t' : T}\ (\text{equiv})$$

feasibility of typed terms, similarly to existing languages [23, 35, 20]. Note that the rules requiring $\perp$ apply it only to well-typed subterms, thus the type system is not circular. While an alternative condition could have been given with dependent types, Definition 8 gives a uniform process that is defined for any type, including types that could be defined later by a programmer, and a simpler type system. As we will prove that values have a unique canonical form (Lemma 14), the inner-product condition is sound.

We also define the *shape of a basic type* below.

▶ **Definition 9.** *The shape of a basic type is defined inductively as* $\texttt{shape}(\text{Qbit}) \triangleq \mathbb{1}$*, and* $\texttt{shape}(B)$ *is the type defined by the constructor set*

$$\text{Cons}(\texttt{shape}(B)) \triangleq \{\, c^{\texttt{shape}(s)} \mid c^s \in \text{Cons}(B) \,\}$$

where the shape of a signature $s = B_1^{|\rangle}, \ldots, B_n^{|\rangle} \to B$ *is defined as*

$$\texttt{shape}(s) \triangleq \texttt{shape}(B_1^{|\rangle}), \ldots, \texttt{shape}(B_n^{|\rangle}) \to \texttt{shape}(B).$$

In particular, the shape of a list of qubits is a list of units, i.e., $\texttt{shape}(\texttt{list}(\text{Qbit})) = \texttt{list}(\mathbb{1})$, which is coherent with the reduction rules in Table 4. By construction, it always holds that $\texttt{shape}(B^{|\rangle}) \in \mathbb{B}_C$. The shape of a basic type is used in the typing rule (shape) of Table 5, which extracts the classical structure of a quantum term, boxes any variable in Δ, and puts

it in the non-linear context. The judgment obtained has an empty linear context and thus can be used classically (i.e., duplicated or discarded). A boxed variable x can be removed using rule (contr) of Table 5, by associating with a linear variable of same type y, allowing to read both the quantum data and the classic structural information of y (see Example 11).

Typing derivations. We write $\pi \triangleright \Gamma; \Delta \vdash t : T$ for the tree of root $\Gamma; \Delta \vdash t : T$ derived using the rules of Table 5. We also denote by $\pi \rhd (\mathrm{r})[t_1, \dots, t_n]$ the subtree of π rooted by (r) and whose premises are $t_1, \dots, t_n$.

▶ **Example 10.** The quantum switch from Example 2 is typable in `Hyrql` with the following notations: $v \triangleq |0\rangle \otimes f(g\,t)$, $w \triangleq |1\rangle \otimes g(f\,t)$, $\Delta \triangleq f : \mathrm{Qbit} \multimap \mathrm{Qbit}, g : \mathrm{Qbit} \multimap \mathrm{Qbit}$, $s \triangleq \mathtt{match}\, q \left\{(c \otimes t) \to \mathtt{qcase}\, c \left\{|0\rangle \to v\,, |1\rangle \to w\right\}\right\}$, and $Q_2 \triangleq \mathrm{Qbit} \otimes \mathrm{Qbit}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\varnothing; c : \mathrm{Qbit} \vdash c : \mathrm{Qbit} \quad \varnothing; \Delta, t : \mathrm{Qbit} \vdash v : Q_2 \quad \varnothing; \Delta, t : \mathrm{Qbit} \vdash w : Q_2 \quad v \perp w}{\varnothing; \Delta, c : \mathrm{Qbit}, t : \mathrm{Qbit} \vdash \mathtt{qcase}\, c \left\{|0\rangle \to v\,, |1\rangle \to w\right\} : Q_2} \quad \varnothing; q : Q_2 \vdash q : Q_2}{\varnothing; \Delta, q : Q_2 \vdash \mathtt{match}\, q \left\{(c \otimes t) \to \mathtt{qcase}\, c \left\{|0\rangle \to v\,, |1\rangle \to w\right\}\right\} : Q_2}}{\varnothing; \Delta \vdash \lambda q.s : Q_2 \multimap Q_2}}{\varnothing; f : \mathrm{Qbit} \multimap \mathrm{Qbit} \vdash \lambda g.\lambda q.s : (\mathrm{Qbit} \multimap \mathrm{Qbit}) \multimap (Q_2 \multimap Q_2)}}{\varnothing; \varnothing \vdash \mathtt{QS} : (\mathrm{Qbit} \multimap \mathrm{Qbit}) \multimap (\mathrm{Qbit} \multimap \mathrm{Qbit}) \multimap (Q_2 \multimap Q_2)}$$

The predicate $v \perp w$ is verified, as for any $\{f, g, t\}$-context substitution σ, both $\sigma(\mathtt{shape}(v))$ and $\sigma(\mathtt{shape}(w))$ reduce to $() \otimes ()$; $\sigma(v) \rightsquigarrow^* |0\rangle \otimes v'$ and $\sigma(w) \rightsquigarrow^* |1\rangle \otimes w'$ and their inner product is zero. Note that we substitute by total values, thus v, w terminate and we can evaluate the inner product.

▶ **Example 11.** Let us come back on Example 7, which introduced the following term that computes the length of a list:

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

Notice that we discard h, thus the list is not used linearly. Therefore, we can only type `len` as a non-linear function, i.e., $\varnothing; \varnothing \vdash \mathtt{len} : \mathtt{list}(B) \Rightarrow \mathtt{nat}$ for a given $B \in \mathbb{B}_C$. However, we can still compute the length of a qubit list with $\lambda x.(x \otimes \mathtt{len}(\mathtt{shape}(x)))$, whose typing derivation π is written below. For conciseness, we hide the signature of $\otimes$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[y : \mathtt{list}(\mathrm{Qbit})]; x : \mathtt{list}(\mathrm{Qbit}) \vdash x : \mathtt{list}(\mathrm{Qbit})}\,(\mathrm{ax}) \quad \dfrac{\dfrac{\dfrac{}{\varnothing; y : \mathtt{list}(\mathrm{Qbit}) \vdash y : \mathtt{list}(\mathrm{Qbit})}\,(\mathrm{ax})}{[y : \mathtt{list}(\mathrm{Qbit})]; \varnothing \vdash \mathtt{shape}(y) : \mathtt{list}(\mathbb{1})}\,(\mathrm{shape})}{[y : \mathtt{list}(\mathrm{Qbit})]; \varnothing \vdash \mathtt{len}(\mathtt{shape}(y)) : \mathtt{nat}}\,(\mathrm{app}_c)}{[y : \mathtt{list}(\mathrm{Qbit})]; x : \mathtt{list}(\mathrm{Qbit}) \vdash x \otimes \mathtt{len}(\mathtt{shape}(y)) : \mathtt{list}(\mathrm{Qbit}) \otimes \mathtt{nat}}\,(\mathrm{cons})}{\varnothing; x : \mathtt{list}(\mathrm{Qbit}) \vdash x \otimes \mathtt{len}(\mathtt{shape}(x)) : \mathtt{list}(\mathrm{Qbit}) \otimes \mathtt{nat}}\,(\mathrm{contr})}{\varnothing; \varnothing \vdash \lambda x.(x \otimes \mathtt{len}(\mathtt{shape}(x))) : \mathtt{list}(\mathrm{Qbit}) \multimap \mathtt{list}(\mathrm{Qbit}) \otimes \mathtt{nat}}\,(\mathrm{app})$$

As $\mathtt{shape}(y)$ is of classical constructor type, it can be used as an input of `len`, allowing us to compute the length of y, given by the subtree $\pi \rhd (\mathrm{app}_c)[\mathtt{shape}(y)]$. The rule (contr) removes the marker introduced by (shape), by substituting y with a variable of same type, x, thus yielding a closed term. Note that while x appears twice in the final term, the quantum information of x has a linear behaviour, and only the classical structure is used non-linearly.

3 Main Results

This section is devoted to checking that `Hyrql` satisfies usual properties of typed languages. Moreover, it studies the complexity of verifying the orthogonality predicate. We also prove that we can compile an expressive subset of terms of `Hyrql` into quantum circuits, with a guarantee on the size of the circuit. This gives us a characterization of the class of functions computable in quantum polynomial time, known as `FBQP` [10].

3.1 Standard Properties

By design of the type system, the non-linear context Γ can be weakened and typing still holds. Weakening does not hold for Δ, as intended, due to its linear behavior.

▶ **Proposition 12** (Weakening). *Let $\Gamma; \Delta \vdash t : T$ be a well-typed term. Then $\Gamma, \Gamma'; \Delta \vdash t : T$ holds for any context Γ' compatible with Γ, Δ.*

The reduction relation $\rightsquigarrow$ is confluent up to equivalence for well-typed terms.

▶ **Theorem 13** (Confluence). *Given a well-typed term t, if there exist t_1 and t_2 such that $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$, then there exist t_3 and t_4 such that $t_1 \rightsquigarrow^* t_3$, $t_2 \rightsquigarrow^* t_4$ and $t_3 \equiv t_4$.*

Due to rule (Equiv) of Table 4, confluence is up to equivalence $\equiv$. Moreover, confluence only holds on well-typed terms. A consequence of Theorem 13 is that any terminating term has a unique normal form, up to equivalence, which will be useful for progress (Theorem 15).

▶ **Lemma 14** (Canonical form for typed terms). *Let $\Gamma; \Delta \vdash t : T$ be a well-typed term. Then t has a canonical form $\sum_{i=1}^{n} \alpha_i \cdot p_i$, and this canonical form is unique, up to equivalence on each p_i and term reordering. Furthermore, $\forall\ 1 \leq i \leq n,\ \Gamma; \Delta \vdash p_i : T$.*

As there are no distinct and equivalent pure values, Lemma 14 implies that the canonical forms of Definition 8 exist and are unique.

Typing implies that the values are the normal forms of the language, up to equivalence.

▶ **Theorem 15** (Progress). *Let $\varnothing; \varnothing \vdash t : T$ be a closed term, either t is equivalent to a value, or t reduces.*

In contrast to standard progress lemmas, we require t to be a value up to equivalence. For instance, the term $|0\rangle + 0 \cdot (\lambda y.y)\,|1\rangle$ does not reduce and is not a value, though it is equivalent to the value $|0\rangle$.

Finally, typing is also preserved by reduction, provided we consider pure or terminating terms.

▶ **Theorem 16** (Subject reduction). *Let $\Gamma; \Delta \vdash t : T$ be a well-typed term, and $t \rightsquigarrow t'$. If t terminates or t is pure, then $\Gamma; \Delta \vdash t' : T$.*

▶ Remark 17. Theorem 16 does not hold on non-terminating and non-pure terms, as we cannot type non-terminating superpositions. This is the case with the following example, with the Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}} \cdot |0\rangle \otimes |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle \otimes |1\rangle$:

$$\left(\lambda y.\texttt{qcase}\left((\texttt{letrec}\, f\, x = f x)\,|0\rangle\right)\left\{|0\rangle \to |0\rangle \otimes y\,, |1\rangle \to |1\rangle \otimes y\right\}\right)|\Phi^+\rangle$$

While such a term is well-typed, it reduces to a superposition of non-terminating pure terms, which is impossible to type through the rule (sup) from Table 5, as the orthogonality predicate requires termination. This is a counterpart of having a general language with few restrictions; however, as resource analysis will be done on terminating terms, such ill-behaved cases will not be considered.

3.2 Decidability of Orthogonality

The orthogonality predicate of Definition 8 is extensionally complete. As a consequence, orthogonality is undecidable. Indeed, it is as hard as the Universal Halting Problem [24], that is Π_2^0-complete in the arithmetical hierarchy [38]. Orthogonality uses the inner product, which requires computing additions, multiplications, and nullity checks on complex numbers. Recall that algebraic numbers are complex numbers in $\mathbb{C}$ that are roots of a polynomial in $\mathbb{Q}[X]$, and write their sets as $\bar{\mathbb{C}}$. In the field of algebraic numbers, equality is decidable, and product and sum are computable in polynomial time [30]. We thus restrict ourselves to terms that only contain amplitudes $\alpha \in \bar{\mathbb{C}}$.

▶ **Theorem 18** (Undecidability of orthogonality). *Deciding orthogonality between two well-typed terms is Π_2^0-complete.*

This result is not surprising as `Hyrql` allows for general recursion. Decidability can be recovered under some slight restrictions. Towards that end, we introduce for basic types a *type depth* d as a partial function $d : \{\text{Qbit}\} \cup \mathbb{B} \to \mathbb{N}$, defined as follows:

$$d(\text{Qbit}) \triangleq 1$$
$$d(B) \triangleq \max \left\{ 1 + \sum_{i=1}^{n} d(B_i^{|\rangle}) \mid c^{B_1^{|\rangle},\dots,B_n^{|\rangle} \to B} \in \text{Cons}(B) \right\}$$

Note that $d(B^{|\rangle})$ is undefined on inductive types. We call *finite types* any basic type with a defined depth.

▶ **Proposition 19.** *Given a finite type $B^{|\rangle}$, and two terminating terms $\varnothing;\varnothing \vdash s : B^{|\rangle}$, $\varnothing;\varnothing \vdash t : B^{|\rangle}$, it is decidable whether they are orthogonal.*

In previous works of `Spm` [14, 13], the syntax of the language was restrictive enough to ensure the decidability of orthogonality. As `Spm` is a strict sublanguage of `Hyrql`, orthogonality can also be decided on this fragment.

3.3 Complexity Properties

We finally discuss how quantum circuits and their size are related to terms of `Hyrql` and their runtime-complexity. Towards that end, we introduce a sublanguage of `Hyrql` (`Hyrql`(T), Definition 26) based on three restrictions: one on the runtime-complexity of the program, to ensure terminating terms (`Time`(T), Definition 20); one on types, to obtain terms that can be represented as circuits (`CircuitTerms`, Definition 21); and one on recursive calls, to have a faithful relation between the size of the circuit and the runtime-complexity (`Faithful`, Definition 24). These restrictions allow, for terms terminating in polynomial time, to characterize precisely the class `FBQP`. We first define the time termination of a term.

▶ **Definition 20.** *Given a term $\varnothing;\varnothing \vdash t : Q \multimap Q'$ and a function $T : \mathbb{N} \to \mathbb{N}$, we say that t* terminates in time *T, if for any value $\varnothing;\varnothing \vdash v : Q$, tv terminates in at most $T(|v|)$ steps. Let* `Time`(T) *be the set of terms terminating in time T.*

Note that this definition should not be confused with termination defined in Section 2.2.

To obtain terms that can be compiled to a circuit form, we introduce a typed restriction `CircuitTerms`, which relies on the following type grammar:

$$B_r ::= \text{Qbit} \mid \texttt{nat} \mid \texttt{list}(B_r) \mid B_r \otimes B_r \mid B_r \multimap B_r \mid B_r \Rightarrow B_r$$

▶ **Definition 21** (Circuit terms). *We define* `CircuitTerms` *as the set of well-typed terms* $\pi \triangleright \varnothing; \varnothing \vdash s : Q \multimap Q'$ *satisfying the following conditions:*

- *Any type in* π *is generated by the grammar of* B_r*;*
- *Any rule* $\pi \rhd (\mathrm{sup})[t_1, \ldots, t_n]$ *satisfies* $t_i \in \mathrm{Value}$*;*
- *Any rule* $\pi \rhd (\mathrm{qcase})[t, t_0, t_1]$ *satisfies either* $t_0, t_1 \in \mathrm{Value}$*; or* $t_0 = c(|0\rangle, s_0)$ *and* $t_1 = c(|1\rangle, s_1)$*, for* $c \in \{\otimes, ::\}$.

While the first condition yields terms that can be represented by a circuit, the two others allow for an efficient compilation and guarantees that any superposition can be compiled effectively to circuits. Note that while terms in `CircuitTerms` only take inputs of type Q, to represent them clearly in a quantum circuit, they can nonetheless contain multiple inputs and higher-order constructs in their body. This restriction alone is sufficient to compile t to a family of circuits in Theorem 23, provided t terminates for a given time T.

▶ **Example 22.** The following term, which is an adaptation of the program from Example 4, belongs to `CircuitTerms`.

$$\pi \triangleright \varnothing; \varnothing \vdash \lambda x.\mathtt{match}\, x \left\{ q \otimes n \to \mathtt{bqwalk}\, q\, n \right\} : \mathrm{Qbit} \otimes \mathtt{nat} \multimap \mathtt{list}(\mathrm{Qbit})$$

Any type in π is derived from B_r. The only term superposition happens in `Had` and is between values. Furthermore, there are two qcase constructs, either between values, or between two terms $|0\rangle :: s_0$ and $|1\rangle :: s_1$, thus it satisfies all properties of `CircuitTerms`. Note that as B_r allows for tensor products, any program taking multiple inputs can always be rewritten using the same technique so that it belongs to `CircuitTerms`.

In the literature, quantum algorithms do not correspond to a single circuit, but rather to a family of quantum circuits, which are parameterized, e.g., by the size of their input for the Quantum Fourier Transform, or by the natural number we want to factor in Shor's algorithm. In `Hyrql`, such parameterization is done by the *shape* of the input, which will give us the necessary information. Formally, we define

$$\mathtt{S}(Q) \triangleq \{\, w \mid \varnothing; \varnothing \vdash w : \mathtt{shape}(Q) \,\}$$
$$\mathtt{S}(w) \triangleq \{\, v \mid \varnothing; \varnothing \vdash v : Q \wedge \mathtt{shape}(v) \rightsquigarrow^* w \,\}, \forall w \in \mathtt{S}(Q),$$

where $\mathtt{S}(Q)$ is the set of closed values of type $\mathtt{shape}(Q)$, and $\mathtt{S}(w)$ the set of closed values of shape w.

Assuming an encoding of $t \in \mathtt{CircuitTerms}$ to a quantum state $|t\rangle$, which can be done naturally, a circuit C is said to *approximate* v on input v' with probability p, if C evaluates to $|\phi\rangle$, on input $|v'\rangle$ (up to ancillary qubits), and $|\langle v|\phi\rangle|^2 \geq p$.

▶ **Theorem 23** (Circuit compilation). *Let* $t \in \mathtt{CircuitTerms} \cap \mathtt{Time}(T)$ *be a term of type* $Q \multimap Q'$ *for a given* $T : \mathbb{N} \to \mathbb{N}$*. Then, we can generate a family of circuits* $\mathtt{C}(t) \triangleq (C_w)_{w \in \mathtt{S}(Q)}$ *such that for any* $w \in \mathtt{S}(Q)$ *and any* $v \in \mathtt{S}(w)$*,* C_w *approximates* v' *with probability* $\frac{2}{3}$ *on input* v*, where* $tv \rightsquigarrow^* v'$*.*

We are now interested in the *size* of each circuit of the family, i.e., the total number of gates acting on the circuit; in particular, the goal is to exhibit a relation between the size and the runtime of the initial term t. To avoid a blow-up due to recursive calls, we introduce a syntactic restriction `Faithful`. It relies on the notion of the *width* of a variable f in a term t, written $w(f, t)$ and defined inductively as follows:

$$
\begin{aligned}
w(f,x) &\triangleq \delta_{x,f}\\
w(f,|i\rangle) &\triangleq 0\\
w(f,\lambda x.t) &\triangleq w(f,t)\\
w(f,\texttt{letrec}\, g\, x = t) &\triangleq w(f,t)\\
w(f,t_1t_2) &\triangleq w(f,t_1)+w(f,t_2)\\
w(f,\sum_{i=1}^{n}\alpha_i\cdot t_i) &\triangleq \max_{1\le i\le n} w(f,t_i)\\
w(f,c(t_1,\dots,t_n)) &\triangleq \sum_{i=1}^{n} w(f,t_i)\\
w(f,\texttt{shape}(t)) &\triangleq w(f,t)\\
w(f,\texttt{qcase}\, t\left\{|0\rangle\to t_0\,,|1\rangle\to t_1\right\}) &\triangleq w(f,t)+\max(w(f,t_0),w(f,t_1))\\
w(f,\texttt{match}_{1\le i\le n}\, t\,\{c_i(\overrightarrow{x_i})\to t_i\}) &\triangleq w(f,t)+\max_{1\le i\le n}(w(f,t_i))
\end{aligned}
$$

▶ **Definition 24.** *A term t is said to* compile faithfully*, if for any term $t' = (\texttt{letrec}\, f\, x = s)$ such that $t' \trianglelefteq t$, the following conditions hold:*

- $w(f,s) \le 1$*;*
- $\exists!\, t', \forall t'', ft'' \trianglelefteq s \implies t' = t''$*;*
- $\neg\exists\, t', t'f \trianglelefteq s$*.*

Let `Faithful` *be the set of terms that compile faithfully.*

The first two conditions are derived from [29], and ensure that, for each recursive program $\texttt{letrec}\, f\, x = s$, all recursive calls are done in different branches of a `qcase` or `match`, with the same input. Furthermore, the last condition ensures that this behaviour is preserved, even in presence of higher-order constructs.

▶ **Example 25.** The program of Example 22 belongs to `Faithful`, as both `bqwalk` and `repeat` only perform one recursive call. All the examples across the paper also satisfy this criterion.

▶ **Definition 26.** *Let $T : \mathbb{N} \to \mathbb{N}$. We define* $\texttt{Hyrql}(T)$ *as follows:*

$$\texttt{Hyrql}(T) \triangleq \texttt{CircuitTerms} \cap \texttt{Faithful} \cap \texttt{Time}(T)$$

Any term in this subset compiles to quantum circuits of size bounded polynomially by T.

▶ **Theorem 27** (Bounded circuit compilation)**.** *Let $T : \mathbb{N} \to \mathbb{N}$, and let $t \in \texttt{Hyrql}(T)$ be a term of type $Q \multimap Q'$. Then, we can generate a family of circuits $\texttt{C}(t) \triangleq (C_w)_{w\in\texttt{S}(Q)}$ and a polynomial P such that, for any $w \in \texttt{S}(Q)$:*

- $|C_w| \le P(T(|w|))$*;*
- $\forall\, v \in \texttt{S}(w)$*, C_w approximates v' with probability $\frac{2}{3}$ on input v, where $tv \rightsquigarrow^* v'$.*

We now want to use Theorem 27 to obtain a characterization of `FBQP` (Definition 28). We recall that a family of circuits $(C_n)_{n\in\mathbb{N}}$ is said to be *uniform polynomially-sized* with $\alpha : \mathbb{N} \to \mathbb{N}$ if there exists $P \in \mathbb{N}[X]$ such that $|C_n| \le P(n)$ and C_n has exactly $\alpha(n)$ ancillary qubits, and there is a polynomial-time Turing machine, which takes n as input, and outputs a representation of C_n for any $n \in \mathbb{N}$.

▶ **Definition 28** ([10]). *A binary function $f : \{0,1\}^* \to \{0,1\}^*$ is said to be computed by a family of circuits $(C_n)_{n\in\mathbb{N}}$ if for any $x \in \{0,1\}^*$, $C_{|x|}$ approximates $f(x)$ with probability $\frac{2}{3}$ on input x.* FBQP *is defined as the set of binary functions computed by a uniform polynomially-sized family of circuits.*

To characterize FBQP, we define the following set of terms, which terminate in polynomial time, and have list of qubits as input:

$$\texttt{Hyrql}(\text{Poly})_{\texttt{QList}} \triangleq \cup_{P\in\mathbb{N}[X]} \{\, t \in \texttt{Hyrql}(P) \mid \varnothing;\varnothing \vdash t : \texttt{list}(\text{Qbit}) \multimap Q \,\}$$

These generated families of circuits will be indexed by natural numbers instead of shape, as one can remark that $\texttt{S}(\texttt{list}(\text{Qbit}))$ is in bijection with $\mathbb{N}$. Altogether, $\texttt{Hyrql}(\text{Poly})_{\texttt{QList}}$ contain terms that are compiled to circuits approximating functions of FBQP.

▶ **Theorem 29** (FBQP soundness). *Let $t \in \texttt{Hyrql}(\text{Poly})_{\texttt{QList}}$, and let $f : \{0,1\}^* \to \{0,1\}^*$. If f is computed by $\texttt{C}(t)$, then $f \in$ FBQP.*

For example, the Quantum Fourier Transform program from Example 3 belongs to FBQP, and can thus be compiled to circuits of polynomial size.

We now prove FBQP completeness, meaning that every function of FBQP can be represented by a function in $\texttt{Hyrql}(\text{Poly})_{\texttt{QList}}$. This is proven by writing any function of Yamakami's algebra [50], which is known to capture FBQP, into $\texttt{Hyrql}(\text{Poly})_{\texttt{QList}}$.

▶ **Theorem 30** (FBQP completeness). *Let $f : \{0,1\}^* \to \{0,1\}^* \in$ FBQP. Then, there exists $t \in \texttt{Hyrql}(\text{Poly})_{\texttt{QList}}$ such that $\texttt{C}(t)$ computes f.*

We have given a typed, syntactic and semantic restriction of Hyrql, that compiles into a bounded family of quantum circuits. Such restriction could be refined, either to accept more terms by lifting requirements in Definition 24, as done in [29]; or by adding requirements to characterize more precise complexity classes [26]. We are nonetheless able to encapsulate FBQP, which demonstrates the expressivity of the $\texttt{Hyrql}(\text{Poly})_{\texttt{QList}}$ fragment.

4 Towards Automatization

Theorem 27 establishes that, given a program that terminates with a specified complexity bound, one can extract a concrete quantum circuit exhibiting the same space complexity. This result is particularly significant, as it enables the characterization of quantum polynomial time, as shown in Theorem 29 and Theorem 30. However, the burden of proving termination of a Hyrql program currently rests entirely on the programmer. This naturally raises the following question: to what extent is the termination of Hyrql programs semi automatable?

We argue that such automatization is feasible by leveraging techniques from Term Rewrite Systems (TRS)[1], as they are equipped with a rich toolbox for analyzing both termination and computational complexity. Approaches to termination analysis include non-exhaustively recursive path orderings [21, 22], dependency pairs [4], and size-change termination [34]. Similarly, several methods address complexity analysis, e.g., interpretation methods [12, 36, 7], and polynomial path orders [37, 5].

Although a full formal treatment is left to future work, we present an illustrative example supporting our claim. Specifically, we illustrate how a Hyrql program can be efficiently

[1] More specifically, a higher-order version called Simply-Typed TRS [49], but we do not enter this level of details here.

translated into a (quantum) TRS, how the reduction behaviour of the original program is faithfully simulated by the resulting rewriting system while preserving complexity bounds, and how termination and complexity results established for the TRS can be transferred back to the corresponding `Hyrql` program.

For example, the standard NOT gate is represented by the quantum TRS whose set of rewrite rules is $\{\texttt{NOT}\ |0\rangle \to |1\rangle, \texttt{NOT}\ |1\rangle \to |0\rangle\}$. The application of the `NOT` function symbol to $|0\rangle$ rewrites to $|1\rangle$. Similarly, the Hadamard gate H can be defined using a superposition of terms $\{\texttt{H}\ |0\rangle \to \frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{1}{\sqrt{2}} \cdot |1\rangle, \texttt{H}\ |1\rangle \to \frac{1}{\sqrt{2}} \cdot |0\rangle - \frac{1}{\sqrt{2}} \cdot |1\rangle\}$. The evaluation of a quantum term will follow a parallel strategy, similar to `Hyrql` operational semantics (Table 4), which reduces in parallel the distinct branches of a superposition, when possible (otherwise, the reduction will have an exponential blowup).

We claim that we can automatize the translation of any term of `Hyrql` to a corresponding TRS. This translation works as follows:

- Produce a set of rules, inductively on the syntax of the term;
- Yield a function symbol `f` for the initial term t;
- When encountering $\texttt{qcase}\, x \left\{|0\rangle \to t_0, |1\rangle \to t_1\right\}$ (same for `match`), merge the two sets of rules generated for t_0 and t_1, while keeping track of which path was taken.

Coming back on Example 4, such an algorithm would generate the following rewrite rules:

$$\left\{\begin{array}{cc} \texttt{Repeat}\, 0 \to [\,] & \texttt{BQWalk}\ |0\rangle\ m \to |0\rangle :: \texttt{Repeat}\, m \\ \texttt{Repeat}\, S(m) \to |0\rangle :: \texttt{Repeat}\, m & \texttt{BQWalk}\ |1\rangle\ 0 \to |1\rangle :: [\,] \\ & \texttt{BQWalk}\ |1\rangle\ S(m) \to |1\rangle :: \texttt{BQWalk}\, (|-\rangle)\, m \end{array}\right\}$$

One can remark that this set of rules captures completely the behaviour of `bqwalk`, and they both compute the same function; furthermore, one reduction step for a given rewrite rule will correspond to $1 \leq k \leq |\texttt{bqwalk}|$ reduction steps in `Hyrql`. Therefore, a certificate on this TRS yields directly a certificate on the original term.

Now, it is possible to apply tools we mentioned to the obtained TRS. In particular, we can use recursive path orderings [21] to show termination, coupled with quasi-interpretations [12] to get that this TRS terminates in time $\mathcal{O}(n)$ for any input of size n. While such properties are not defined for quantum TRS, i.e., with superpositions and parallel reduction, they can be adapted to fit the quantum setting; a formal adaptation is left for future work. Altogether with Theorem 27, we can conclude that `bqwalk` can be compiled to a family of quantum circuits, whose size is polynomial in the size of the input.

Using the same ideas, we claim that we can describe a semantic-preserving translation algorithm from `Hyrql` to TRS, with no blowup on the termination complexity in either direction. Such work, along with a proper development of TRS in the quantum setting and an adaptation of existing techniques, would give a semi-automatized way to prove the feasibility/tractability of terms in `Hyrql`.

While this section was focused on termination and time complexity, some of the mentioned techniques also provide results to bound the space complexity of a TRS program [12]. Given a good definition of quantum space, this would make it possible to characterize quantum space complexities, and obtain bounds on the depth or width of the circuit.

References

1 Alastair A. Abbott, Julian Wechs, Dominic Horsman, Mehdi Mhalla, and Cyril Branciard. Communication through coherent control of quantum channels. *Quantum*, 4:333, September 2020. doi:10.22331/q-2020-09-24-333.

2 Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *Logic in Computer Science, LICS'05*, pages 249–258. IEEE Computer Society, 2005. doi:10.1109/lics.2005.1.

3 Mateus Araújo, Fabio Costa, and Časlav Brukner. Computational advantage from quantum-controlled ordering of gates. *Phys. Rev. Lett.*, 113:250402, December 2014. doi:10.1103/PhysRevLett.113.250402.

4 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1):133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.

5 Martin Avanzini and Georg Moser. Dependency pairs and polynomial path orders. In *International Conference on Rewriting Techniques and Applications, RTA'09*, pages 48–62. Springer, 2009. doi:10.1007/978-3-642-02348-4_4.

6 Martin Avanzini, Georg Moser, Romain Péchoux, Simon Perdrix, and Vladimir Zamdzhiev. Quantum expectation transformers for cost analysis. In Christel Baier and Dana Fisman, editors, *LICS '22: Symposium on Logic in Computer Science*, pages 10:1–10:13. ACM, 2022. doi:10.1145/3531130.3533332.

7 Patrick Baillot and Ugo Dal Lago. Higher-order interpretations and program complexity. *Information and Computation*, 248:56–81, 2016. doi:10.1016/j.ic.2015.12.008.

8 Henk Barendregt. The lambda calculus: its syntax and semantics. *Studies in logic and the foundations of Mathematics*, 1984. doi:10.1016/c2009-0-14341-6.

9 Kathleen Barsse, Romain Péchoux, and Simon Perdrix. A quantum programming language for coherent control, 2025. doi:10.48550/arXiv.2507.10466.

10 Ethan Bernstein and Umesh Vazirani. Quantum Complexity Theory. *SIAM Journal on Computing*, 26(5):1411–1473, October 1997. doi:10.1137/S0097539796300921.

11 Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '20, pages 286–300. ACM, June 2020. doi:10.1145/3385412.3386007.

12 Guillaume Bonfante, Jean-Yves Marion, and Jean-Yves Moyen. Quasi-interpretations a way to control resources. *Theoretical Computer Science*, 412(25):2776–2796, June 2011. doi:10.1016/j.tcs.2011.02.007.

13 Kostia Chardonnet, Louis Lemonnier, and Benoît Valiron. Semantics for a Turing-Complete Reversible Programming Language with Inductive Types. In Jakob Rehof, editor, *Formal Structures for Computation and Deduction (FSCD 2024)*, volume 299 of *LIPIcs*, pages 19:1–19:19, 2024. doi:10.4230/LIPIcs.FSCD.2024.19.

14 Kostia Chardonnet, Alexis Saurin, and Benoît Valiron. A Curry-Howard Correspondence for Linear, Reversible Computation. *LIPIcs, Volume 252, CSL 2023*, 252:13:1–13:18, 2023. doi:10.4230/LIPIcs.CSL.2023.13.

15 Giulio Chiribella, Giacomo Mauro D'Ariano, Paolo Perinotti, and Benoit Valiron. Quantum computations without definite causal structure. *Phys. Rev. A*, 88:022318, August 2013. doi:10.1103/PhysRevA.88.022318.

16 Andrea Colledan and Ugo Dal Lago. Flexible type-based resource estimation in quantum circuit description languages. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704883.

17 Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017. arXiv:1707.03429.

18 Andrew W. Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. OpenQASM 3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3):1–50, September 2022. doi:10.1145/3505636.

19 Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theor. Comput. Sci.*, 411(2):377–409, 2010. doi:10.1016/J.TCS.2009.07.045.

20 Kinnari Dave, Louis Lemonnier, Romain Péchoux, and Vladimir Zamdzhiev. Combining quantum and classical control: syntax, semantics and adequacy. In *Foundations of Software Science and Computation Structures, FoSSaCS 2025*, pages 155–175, Berlin, Heidelberg, 2025. Springer-Verlag. doi:10.1007/978-3-031-90897-2_8.

21 Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical computer science*, 17(3):279–301, 1982. doi:10.1016/0304-3975(82)90026-3.

22 Nachum Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1-2):69–115, February 1987. doi:10.1016/S0747-7171(87)80022-6.

23 Alejandro Díaz-Caro and Octavio Malherbe. Quantum control in the unitary sphere: Lambda-s1 and its categorical model. *Log. Methods Comput. Sci.*, 18(3), 2022. doi:10.46298/LMCS-18(3:32)2022.

24 Jörg Endrullis, Herman Geuvers, and Hans Zantema. *Degrees of Undecidability in Term Rewriting*, pages 255–270. Springer Berlin Heidelberg, 2009. doi:10.1007/978-3-642-04027-6_20.

25 Yuan Feng and Mingsheng Ying. Quantum Hoare logic with classical variables. *ACM Transactions on Quantum Computing*, 2(4):1–43, 2021. doi:10.1145/3456877.

26 Florent Ferrari, Emmanuel Hainry, Romain Péchoux, and Mário Silva. Quantum programming in polylogarithmic time. In *Mathematical Foundations of Computer Science, MFCS 2025*, volume 345 of *LIPIcs*, pages 47:1–47:17, 2025. doi:10.4230/LIPIcs.MFCS.2025.47.

27 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 333–342. ACM, June 2013. doi:10.1145/2491956.2462177.

28 Emmanuel Hainry, Romain Péchoux, and Mário Silva. A programming language characterizing quantum polynomial time. In *Foundations of Software Science and Computation Structures, Fossacs 2023*, pages 156–175. Springer, April 2023. doi:10.1007/978-3-031-30829-1_8.

29 Emmanuel Hainry, Romain Péchoux, and Mário Silva. Branch sequentialization in quantum polytime. In Maribel Fernández, editor, *Formal Structures for Computation and Deduction, FSCD 2025*, volume 337 of *LIPIcs*, pages 22:1–22:22, 2025. doi:10.4230/LIPIcs.FSCD.2025.22.

30 Vesa Halava, Tero Harju, Mika Hirvensalo, and Juhani Karhumäki. Skolem's problem - on the border between decidability and undecidability. Technical Report 683, Turku Center for Computer Science, 2005. URL: https://tucs.fi/publications/view/?pub_id=tHaHaHiKa05a.

31 Emanuel Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Lab., 1996. arXiv:2211.02559.

32 Hlér Kristjánsson, Tatsuki Odake, Satoshi Yoshida, Philip Taranto, Jessica Bavaresco, Marco Túlio Quintino, and Mio Murao. Exponential separation in quantum query complexity of the quantum switch with respect to simulations with standard quantum circuits, 2024. arXiv:2409.18420.

33 Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1-2):163–180, 2004. doi:10.1016/J.TCS.2003.10.018.

34 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Symposium on Principles of Programming Languages, POPL 2001*, pages 81–92, New York, NY, USA, January 2001. ACM. doi:10.1145/373243.360210.

35 Louis Lemonnier. *The Semantics of Effects : Centrality, Quantum Control and Reversible Recursion*. PhD thesis, Université Paris-Saclay, June 2024. doi:10.70675/0db41738z16f8z447fz8352z236e657df8ee.

36 Jean-Yves Marion and Romain Péchoux. Sup-interpretations, a semantic method for static analysis of program resources. *ACM Trans. Comput. Logic*, 10(4), August 2009. doi:10.1145/1555746.1555751.

37 Georg Moser. *Proof Theory at Work: Complexity Analysis of Term Rewrite Systems*. Cumulative habilitation thesis, University of Innsbruck, 2009. arXiv:0907.5527.

38 André Nies. *Computability and Randomness*. Oxford University Press, January 2009. doi:10.1093/acprof:oso/9780199230761.001.0001.

39 Romain Péchoux. *Complexité implicite : bilan et perspectives*. Accreditation to supervise research, Université de Lorraine, October 2020. URL: https://hal.univ-lorraine.fr/tel-02978986.

40 Lorenzo M. Procopio, Amir Moqanaki, Mateus Araújo, Fabio Costa, Irati Alonso Calafell, Emma G. Dowd, Deny R. Hamel, Lee A. Rozema, Časlav Brukner, and Philip Walther. Experimental superposition of orders of quantum gates. *Nature Communications*, 6(1), August 2015. doi:10.1038/ncomms8913.

41 Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. From symmetric pattern-matching to quantum control. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 348–364, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-89366-2_19.

42 Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, August 2004. doi:10.1017/s0960129504004256.

43 Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. In Paweł Urzyczyn, editor, *Typed Lambda Calculi and Applications, TLCA 2005*, pages 354–368. Springer Berlin Heidelberg, 2005. doi:10.1007/11417170_26.

44 Peter Selinger and Benoît Valiron. Semantic techniques in quantum computation. In Simon Gay and Ian Mackie, editors, *Quantum lambda calculus*, pages 135–172. Cambridge University Press, 2009. doi:10.1017/cbo9781139193313.005.

45 Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. doi:10.1137/S0097539795293172.

46 Márcio M. Taddei, Jaime Cariñe, Daniel Martínez, Tania García, Nayda Guerrero, Alastair A. Abbott, Mateus Araújo, Cyril Branciard, Esteban S. Gómez, Stephen P. Walborn, Leandro Aolita, and Gustavo Lima. Computational advantage from the quantum superposition of multiple temporal orders of photonic gates. *PRX Quantum*, 2:010320, February 2021. doi:10.1103/PRXQuantum.2.010320.

47 Peter van Emde Boas. *Machine Models and Simulations*, pages 1–66. Handbook of Theoretical Computer Science. Elsevier, 1990. doi:10.1016/b978-0-444-88071-0.50006-0.

48 Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. Qunity: A unified language for quantum and classical computing. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571225.

49 Toshiyuki Yamada. Confluence and Termination of Simply Typed Term Rewriting Systems. In Aart Middeldorp, editor, *Rewriting Techniques and Applications*, pages 338–352, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-45127-7_25.

50 Tomoyuki Yamakami. A Schematic Definition of Quantum Polynomial Time Computability. *The Journal of Symbolic Logic*, 85(4):1546–1587, December 2020. doi:10.1017/jsl.2020.45.

51 Mingsheng Ying. *Foundations of quantum programming*. Elsevier, 2024. doi:10.1016/C2014-0-02660-3.

52 Mingsheng Ying and Zhicheng Zhang. Verification of recursively defined quantum circuits, 2024. doi:10.48550/arXiv.2404.05934.

53 Charles Yuan, Agnes Villanyi, and Michael Carbin. Quantum control machine: The limits of control flow in quantum programming. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1–28, April 2024. doi:10.1145/3649811.