

From Symmetric Pattern-Matching to Quantum Control

Amr Sabry¹, Benoît Valiron^{2*}, and Juliana Kaizer Vizzotto^{3*}

¹ Indiana University sabry@indiana.edu

² LRI, CentraleSupélec, Université Paris-Saclay benoit.valiron@lri.fr

³ Universidade Federal de Santa Maria juvizzotto@inf.ufsm.br

Abstract. One perspective on quantum algorithms is that they are classical algorithms having access to a special kind of memory with exotic properties. This perspective suggests that, even in the case of quantum algorithms, the control flow notions of sequencing, conditionals, loops, and recursion are entirely classical. There is however, another notion of control flow, that is itself quantum. The notion of quantum conditional expression is reasonably well-understood: the execution of the two expressions becomes itself a superposition of executions. The quantum counterpart of loops and recursion is however not believed to be meaningful in its most general form.

In this paper, we argue that, under the right circumstances, a reasonable notion of quantum loops and recursion is possible. To this aim, we first propose a classical, typed, reversible language with lists and fixpoints. We then extend this language to the *closed* quantum domain (without measurements) by allowing linear combinations of terms and restricting fixpoints to structurally recursive fixpoints whose termination proofs match the proofs of convergence of sequences in infinite-dimensional Hilbert spaces. We additionally give an operational semantics for the quantum language in the spirit of algebraic lambda-calculi and illustrate its expressiveness by modeling several common unitary operations.

1 Introduction

The control flow of a program describes how its elementary operations are organized along the execution. Usual primitive control mechanisms are sequences, tests, iteration and recursion. Elementary operations placed in sequence are executed in order. Tests allow conditionally executing a group of operations and changing the course of the execution of the program. Finally, iteration gives the possibility to iterate a process an arbitrary number of times and recursion generalizes iteration to automatically manage the history of the operations performed during iteration. The structure of control flow for conventional (classical) computation is well-understood. In the case of *quantum* computation, control flow is still subject to debate. This paper proposes a working notion of quantum control

* Partially funded by FoQCoss STIC AmSud project - STIC-AmSUD/Capes - Foundations of Quantum Computation: Syntax and Semantics.

in closed quantum systems, shedding new light on the problem, and clarifying several of the previous concerns.

Quantum computation. A good starting point for understanding quantum computation is to consider classical circuits over *bits* but replacing the bits with *qubits*, which are intuitively superpositions of bits weighed by complex number amplitudes. Computationally, a qubit is an abstract data type governed by the laws of quantum physics, whose values are normalized vectors of complex numbers in the Hilbert space \mathbb{C}^2 (modulo a global phase). By choosing an orthonormal basis, say the classical bits \mathbf{tt} and \mathbf{ff} , a qubit can be regarded as a complex linear combination, $\alpha \mathbf{tt} + \beta \mathbf{ff}$, where α and β are complex numbers such that $|\alpha|^2 + |\beta|^2 = 1$. This generalizes naturally to multiple qubits: the state of a system of n qubits is a vector in the Hilbert space $(\mathbb{C}^2)^{\otimes n}$.

The operations one can perform on a quantum memory are of two kinds: quantum gates and measurements. Quantum gates are unitary operations that are “purely quantum” in the sense that they modify the quantum memory without giving any feedback to the outside world: the quantum memory is viewed as a *closed system*. A customary graphical representation for these operations is the *quantum circuit*, akin to conventional boolean circuits: wires represent qubits while boxes represents operations to perform on them. One of the peculiar aspects of quantum computation is that the state of a qubit is non-duplicable [1], a result known as the *no-cloning theorem*. A corollary is that a quantum circuit is a very simple kind of circuit: wires neither split nor merge.

Measurement is a fundamentally different kind of operation: it queries the state of the quantum memory and returns a classical result. Measuring the state of a quantum bit is a probabilistic and destructive operation: it produces a classical answer with a probability that depends on the amplitudes α, β in the state of the qubit while projecting this state onto \mathbf{tt} or \mathbf{ff} , based on the result.

For a more detailed introduction to quantum computation, we refer the reader to recent textbooks (e.g., [2]).

Control flow in quantum computation. In the context of quantum programming languages, there is a well-understood notion of control flow: the so-called *classical control flow*. A quantum program can be seen as the construction, manipulation and evaluation of quantum circuits [3,4]. In this setting, circuits are simply considered as special kinds of data without much computational content, and programs are ruled by regular classical control.

One can however consider the circuit being manipulated as a program in its own right: a particular sequence of execution on the quantum memory is then seen as a closed system. One can then try to derive a notion of *quantum control* [5], with “quantum tests” and “quantum loops”. Quantum tests are a bit tricky to perform [5,6] but they essentially correspond to well-understood controlled operations. The situation with quantum loops is more subtle [6,7]. First, a hypothetical quantum loop *must* terminate. Indeed, a non-terminating quantum loop would entail an infinite quantum circuit, and this concept has so far no meaning. Second, the interaction of quantum loops with measurement is

problematic: it is known that the canonical model of *open* quantum computation based on superoperators [8, 9] is incompatible with such quantum control [6]. Finally, the mathematical operator corresponding to a quantum loop would need to act on an infinite-dimensional Hilbert space and the question of mixing programming languages with infinitary Hilbert spaces is still an unresolved issue.

Our contribution. In this paper, we offer a novel solution to the question of quantum control: we define a purely quantum language, inspired by Theseus [10], featuring tests and fixpoints in the presence of lists. More precisely, we propose (1) a typed, reversible language, extensible to linear combinations of terms, with a reduction strategy akin to algebraic lambda-calculi [11–13]; (2) a model for the language based on unitary operators over infinite-dimensional Hilbert spaces, simplifying the Fock space model of Ying [7]. This model captures lists, tests, and structurally recursive fixpoints. We therefore settle two longstanding issues. (1) We offer a solution to the problem of quantum loops, with the use of *terminating, structurally recursive, purely quantum* fixpoints. We dodge previously noted concerns (e.g., [6]) by staying in the closed quantum setting and answer the problem of the external system of quantum “coins” [7] with the use of lists. (2) By using a linear language based on patterns and clauses, we give an extensible framework for reconciling algebraic calculi with quantum computation [16, 12, 11].

In the remainder of the paper, we first introduce the key idea underlying our classical reversible language in a simple first-order setting. We then generalize the setting to allow second-order functions, recursive types (e.g., lists), and fixpoints. After illustrating the expressiveness of this classical language, we adapt it to the quantum domain and give a semantics to the resulting quantum language in infinite-dimensional Hilbert spaces. Technical material that would interrupt the flow or that is somewhat complementary has been relegated to an extended version of the paper [17].

2 Pattern-Matching Isomorphisms

The most elementary control structure in a programming language is the ability to conditionally execute one of several possible code fragments. Expressing such an abstraction using predicates and nested **if**-expressions makes it difficult for both humans and compilers to reason about the control flow structure. Instead, in modern functional languages, this control flow paradigm is elegantly expressed using *pattern-matching*. This approach yields code that is not only more concise and readable but also enables the compiler to easily verify two crucial properties: (i) non-overlapping patterns and (ii) exhaustive coverage of a datatype using a collection of patterns. Indeed most compilers for functional languages perform these checks, warning the user when they are violated. At a more fundamental level, e.g., in type theories and proof assistants, these properties are actually necessary for correct reasoning about programs. Our first insight, explained in this section, is that these properties, perhaps surprisingly, are sufficient to produce a simple and intuitive first-order reversible programming language.

```

f :: Either Int Int -> a      g :: (Bool,Int) -> a      h :: Either Int Int <-> (Bool,Int)
f (Left 0)    = undefined    g (False,n) = undefined    h (Left 0)   = (True,0)
f (Left (n+1)) = undefined    g (True,0)  = undefined    h (Left (n+1)) = (False,n)
f (Right n)   = undefined    g (True,n+1) = undefined    h (Right n)  = (True,n+1)

```

Fig. 1. A skeleton

Fig. 2. Another skeleton

Fig. 3. An isomorphism

2.1 An Example

We start with a small illustrative example, written in a Haskell-like syntax. Fig. 1 gives the skeleton of a function `f` that accepts a value of type `Either Int Int`; the patterns on the left-hand side exhaustively cover every possible incoming value and are non-overlapping. Similarly, Fig. 2 gives the skeleton for a function `g` that accepts a value of type `(Bool,Int)`; again the patterns on the left-hand side exhaustively cover every possible incoming value and are non-overlapping. Now we claim that since the types `Either Int Int` and `(Bool,Int)` are isomorphic, we can combine the patterns of `f` and `g` into *symmetric pattern-matching clauses* to produce a reversible function between the types `Either Int Int` and `(Bool,Int)`. Fig. 3 gives one such function; there, we suggestively use `<->` to indicate that the function can be executed in either direction. This reversible function is obtained by simply combining the non-overlapping exhaustive patterns on the two sides of a clause. In order to be well-formed in either direction, these clauses are subject to the constraint that each variable occurring on one side must occur exactly once on the other side (and with the same type). Thus it is acceptable to swap the second and third right-hand sides of `h` but not the first and second ones.

2.2 Terms and Types

We present a formalization of the ideas presented above using a simple typed first-order reversible language. The language is two-layered. The first layer contains values, which also play the role of patterns. These values are constructed from variables ranged over x and the introduction forms for the finite types a, b constructed from the unit type and sums and products of types. The second layer contains collections of pattern-matching clauses that denote isomorphisms of type $a \leftrightarrow b$. Computations are chained applications of isomorphisms to values:

(Value types)	$a, b ::= \mathbb{1} \mid a \oplus b \mid a \otimes b$
(Iso types)	$T ::= a \leftrightarrow b$
(Values)	$v ::= () \mid x \mid \text{inj}_l v \mid \text{inj}_r v \mid \langle v_1, v_2 \rangle$
(Isos)	$\omega ::= \{ \mid v_1 \leftrightarrow v'_1 \mid v_2 \leftrightarrow v'_2 \dots \}$
(Terms)	$t ::= v \mid \omega t$

The typing rules are defined using two judgments: $\Delta \vdash_v v : a$ for typing values (or *patterns*) and terms; and $\vdash_\omega \omega : a \leftrightarrow b$ for typing collections of

pattern-matching clauses denoting an isomorphism. As it is customary, we write $a_1 \otimes a_2 \otimes \dots \otimes a_n$ for $((a_1 \otimes a_2) \otimes \dots \otimes a_n)$, and similarly $\langle x_1, x_2, \dots, x_n \rangle$ for $\langle \langle x_1, x_2 \rangle, \dots, x_n \rangle$.

The typing rules for values are the expected ones. The only subtlety is the fact that they are linear: because values act as patterns, we forbid the repetition of variables. A typing context Δ is a set of typed variables $x_1 : a_1, \dots, x_n : a_n$. A value typing judgment is valid if it can be derived from the following rules:

$$\begin{array}{c} \overline{\vdash_v () : \mathbb{1}}, \quad \overline{x : a \vdash_v x : a}, \quad \frac{\Delta_1 \vdash_v v_1 : a \quad \Delta_2 \vdash_v v_2 : b}{\Delta_1, \Delta_2 \vdash_v \langle v_1, v_2 \rangle : a \otimes b} \\ \\ \frac{\Delta \vdash_v v : a}{\Delta \vdash_v \mathbf{inj}_l v : a \oplus b}, \quad \frac{\Delta \vdash_v v : b}{\Delta \vdash_v \mathbf{inj}_r v : a \oplus b}, \end{array}$$

The typing rule for term construction is simple and forces the term to be closed:

$$\frac{\vdash_v t : a \quad \vdash_\omega \omega : a \leftrightarrow b}{\vdash_v \omega t : b}$$

The most interesting type rule is the one for isomorphisms. We present the rule and then explain it in detail:

$$\frac{\begin{array}{c} \Delta_1 \vdash_v v_1 : a \quad \Delta_n \vdash_v v_n : a \quad \forall i \neq j, v_i \perp v_j \quad \dim(a) = n \\ \Delta_1 \vdash_v v'_1 : b \quad \dots \quad \Delta_n \vdash_v v'_n : b \quad \forall i \neq j, v'_i \perp v'_j \quad \dim(b) = n \end{array}}{\vdash_\omega \{ \mid v_1 \leftrightarrow v'_1 \mid v_2 \leftrightarrow v'_2 \dots \} : a \leftrightarrow b}, \quad (1)$$

The rule relies on two auxiliary conditions as motivated in the beginning of the section. These conditions are (i) the orthogonality judgment $v \perp v'$ that formalizes that patterns must be *non-overlapping* and (ii) the condition $\dim(a) = n$ which formalizes that patterns are *exhaustive*. The rules for deriving orthogonality of values or patterns are:

$$\begin{array}{c} \overline{\mathbf{inj}_l v_1 \perp \mathbf{inj}_r v_2} \quad \overline{\mathbf{inj}_r v_1 \perp \mathbf{inj}_l v_2} \\ \\ \frac{v_1 \perp v_2}{\mathbf{inj}_l v_1 \perp \mathbf{inj}_l v_2} \quad \frac{v_1 \perp v_2}{\mathbf{inj}_r v_1 \perp \mathbf{inj}_r v_2} \quad \frac{v_1 \perp v_2}{\langle v, v_1 \rangle \perp \langle v', v_2 \rangle} \quad \frac{v_1 \perp v_2}{\langle v_1, v \rangle \perp \langle v_2, v' \rangle} \end{array}$$

The idea is simply that the left and right injections are disjoint subspaces of values. To characterize that a set of patterns is exhaustive, we associate a *dimension* with each type. For finite types, this is just the number of elements in the type and is inductively defined as follows: $\dim(\mathbb{1}) = 1$; $\dim(a \oplus b) = \dim(a) + \dim(b)$; and $\dim(a \otimes b) = \dim(a) \cdot \dim(b)$. For a given type a , if a set of non-overlapping clauses has cardinality $\dim(a)$, it is exhaustive. Conversely, any set of exhaustive clauses for a type a either has cardinality $\dim(a)$ or can be extended to an equivalent exhaustive set of clauses of cardinality $\dim(a)$.

2.3 Semantics

We equip our language with a simple operational semantics on terms, using the natural notion of matching. To formally define it, we first introduce the notion of variable assignation, or valuation, which is a partial map from a finite set of variables (the support) to a set of values. We denote the matching of a value w against a pattern v and its associated valuation σ as $\sigma[v] = w$ and define it as follows:

$$\frac{}{\sigma[()] = ()} \quad \frac{\sigma = \{x \mapsto v\}}{\sigma[x] = v} \quad \frac{\sigma[v] = w}{\sigma[\mathbf{inj}_l v] = \mathbf{inj}_l w} \quad \frac{\sigma[v] = w}{\sigma[\mathbf{inj}_r v] = \mathbf{inj}_r w}$$

$$\frac{\sigma_2[v_1] = w_1 \quad \sigma_1[v_2] = w_2 \quad \text{supp}(\sigma_1) \cap \text{supp}(\sigma_2) = \emptyset \quad \sigma = \sigma_1 \cup \sigma_2}{\sigma[\langle v_1, v_2 \rangle] = \langle w_1, w_2 \rangle}$$

If σ is a valuation whose support contains the variables of v , we write $\sigma(v)$ for the value where the variables of v have been replaced with the corresponding values in σ .

Given these definitions, we can define the reduction relation on terms. The redex $\{ \mid v_1 \leftrightarrow v'_1 \mid v_2 \leftrightarrow v'_2 \dots \} v$ reduces to $\sigma(v'_i)$ whenever $\sigma[v_i] = v'_i$. Because of the conditions on patterns, a matching pattern exists by exhaustivity of coverage, and this pattern is unique by the non-overlapping condition. Congruence holds: $\omega t \rightarrow \omega t'$ whenever $t \rightarrow t'$. As usual, we write $s \rightarrow t$ to say that s rewrites in one step to t and $s \rightarrow^* t$ to say that s rewrites to t in 0 or more steps.

Because of the conditions set on patterns, the rewrite system is deterministic. More interestingly, we can swap the two sides of all pattern-matching clauses in an isomorphism ω to get ω^{-1} . The execution of ω^{-1} is the reverse execution of ω in the sense that $\omega^{-1}(\omega t) \rightarrow^* t$ and $\omega(\omega^{-1} t') \rightarrow^* t'$.

3 Second-Order Functions, Lists, and Recursion

The first-order reversible language from the previous section embodies symmetric-pattern matching clauses as its core notion of control. Its expressiveness is limited, however. We now show that it is possible to extend it to have more in common with a conventional functional language. To that end, we extend the language with the ability to parametrically manipulate isomorphisms, with a recursive type (lists), and with recursion.

3.1 Terms and Types

Formally, the language is now defined as follows.

$$\begin{array}{ll} \text{(Val \& term types)} & a, b ::= \mathbb{1} \mid a \oplus b \mid a \otimes b \mid [a] \\ \text{(Iso types)} & T ::= a \leftrightarrow b \mid (a \leftrightarrow b) \rightarrow T \\ \text{(Values)} & v ::= () \mid x \mid \mathbf{inj}_l v \mid \mathbf{inj}_r v \mid \langle v_1, v_2 \rangle \end{array}$$

$$\begin{array}{c}
\frac{}{\emptyset; \Psi \vdash_v () : \mathbb{1}} \quad \frac{}{x : a; \Psi \vdash_v x : a} \\
\\
\frac{\Delta; \Psi \vdash_v t : a}{\Delta; \Psi \vdash_v \mathbf{inj}_l t : a \oplus b} \quad \frac{\Delta; \Psi \vdash_v t : b}{\Delta; \Psi \vdash_v \mathbf{inj}_r t : a \oplus b} \quad \frac{\Delta_1; \Psi \vdash_v t_1 : a \quad \Delta_2; \Psi \vdash_v t_2 : b}{\Delta_1, \Delta_2; \Psi \vdash_v \langle t_1, t_2 \rangle : a \otimes b} \\
\\
\frac{\Psi \vdash_\omega \omega : a \leftrightarrow b \quad \Delta; \Psi \vdash_v t : a}{\Delta; \Psi \vdash_v \omega t : b} \quad \frac{\Delta; \Psi \vdash_v t_1 : a \otimes b \quad \Delta, x : a, y : b; \Psi \vdash_v t_2 : c}{\Delta; \Psi \vdash_v \mathbf{let} \langle x, y \rangle = t_1 \mathbf{in} t_2 : c}
\end{array}$$

Table 1. Typing rules for terms and values

(Products)	$p ::= () \mid x \mid \langle p_1, p_2 \rangle$
(Extended Values)	$e ::= v \mid \mathbf{let} p_1 = \omega p_2 \mathbf{in} e$
(Isos)	$\omega ::= \{ \mid v_1 \leftrightarrow e_1 \mid v_2 \leftrightarrow e_2 \dots \} \mid \lambda f. \omega \mid \mu f. \omega \mid f \mid \omega_1 \omega_2$
(Terms)	$t ::= () \mid x \mid \mathbf{inj}_l t \mid \mathbf{inj}_r t \mid \langle t_1, t_2 \rangle \mid \omega t \mid \mathbf{let} p = t_1 \mathbf{in} t_2$

We use variables f to span a set of iso-variables and variables x to span a set of term-variables. We extend the layer of isos so that it can be parameterized by a fixed number of other isos, i.e., we now allow higher-order manipulation of isos using $\lambda f. \omega$, iso-variables, and applications. Isos can now be used inside the definition of other isos with a let-notation. These let-constructs are however restricted to products of term-variables: they essentially serve as syntactic sugar for composition of isos. An extended value is then a value where some of its free variables are substituted with the result of the application of one or several isos. Given an extended value e , we define its *bottom value*, denoted with $\text{Val}(e)$ as the value “at the end” of the let-chain: $\text{Val}(v) = v$, and $\text{Val}(\mathbf{let} p = \omega p \mathbf{in} e) = \text{Val}(e)$. The orthogonality of extended values is simply the orthogonality of their bottom value.

As usual, the type of lists $[a]$ of elements of type a is a recursive type and is equivalent to $\mathbb{1} \oplus (a \times [a])$. We build the value $[]$ (empty list) as $\mathbf{inj}_l ()$ and the term $t_1 : t_2$ (cons of t_1 and t_2) as $\mathbf{inj}_r \langle t_1, t_2 \rangle$. In addition, to take full advantage of recursive datatypes, it is natural to consider recursion. Modulo a termination guarantee it is possible to add a fixpoint to the language: we extend isos with the fixpoint constructor $\mu f. \omega$. Some reversible languages allow infinite loops and must work with partial isomorphisms instead. Since we plan on using our language as a foundation for a quantum language we insist of termination.

Since the language features two kinds of variables, there are typing contexts (written Δ) consisting of base-level typed variables of the form $x : a$, and typing context (written Ψ) consisting of typed iso-variables of the form $f : T$. As terms and values contain both base-level and iso-variables, one needs two typing contexts. Typing judgments are therefore written respectively as $\Delta; \Psi \vdash_v t : a$. The updated rules for (\vdash_v) are found in Tab. 1. As the only possible free variables

$$\begin{array}{c}
\frac{\Delta_1; \Psi \vdash_v v_1 : a \quad \dots \quad \Delta_n; \Psi \vdash_v v_n : a \quad \text{OD}_a\{v_1, \dots, v_n\} \quad \Delta_1; \Psi \vdash_v e_1 : b \quad \dots \quad \Delta_n; \Psi \vdash_v e_n : b \quad \text{OD}_b^{\text{ext}}\{e_1, \dots, e_n\}}{\Psi \vdash_\omega \{ \mid v_1 \leftrightarrow e_1 \mid v_2 \leftrightarrow e_2 \quad \dots \} : a \leftrightarrow b.} \\
\\
\frac{\Psi, f : a \leftrightarrow b \vdash_\omega \omega : T}{\Psi \vdash_\omega \lambda f. \omega : (a \leftrightarrow b) \rightarrow T} \quad \frac{}{\Psi, f : T \vdash_\omega f : T} \\
\\
\frac{\Psi \vdash_\omega \omega_1 : (a \leftrightarrow b) \rightarrow T \quad \Psi \vdash_\omega \omega_2 : a \leftrightarrow b}{\Psi \vdash_\omega \omega_1 \omega_2 : T} \\
\\
\frac{\Psi, f : a \leftrightarrow b \vdash_\omega \omega : (a_1 \leftrightarrow b_1) \rightarrow \dots \rightarrow (a_n \leftrightarrow b_n) \rightarrow (a \leftrightarrow b) \quad \mu f. \omega \text{ terminates in any finite context}}{\Psi \vdash_\omega \mu f. \omega : (a_1 \leftrightarrow b_1) \rightarrow \dots \rightarrow (a_n \leftrightarrow b_n) \rightarrow (a \leftrightarrow b)}
\end{array}$$

Table 2. Typing rules for isos

in isos are iso-variables, their typing judgments only need one context and are written as $\Psi \vdash_\omega \omega : T$.

The rules for typing derivations of isos are in Tab. 2. It is worthwhile mentioning that isos are treated in a usual, non-linear way: this is the purpose of the typing context separation. The intuition is that an iso is the description of a closed computation with respect to inputs: remark that isos cannot accept value-types. As computations, they can be erased or duplicated without issues. On the other hand, value-types still need to be treated linearly.

In the typing rule for recursion, the condition “ $\mu f. \omega$ terminates in any finite context” formally refers to the following requirement. A well-typed fixpoint $\mu f. \omega$ of type $\Psi \vdash_\omega \mu f. \omega : (a_1 \leftrightarrow b_1) \rightarrow \dots \rightarrow (a_n \leftrightarrow b_n) \rightarrow (a \leftrightarrow b)$ is *terminating in a 0-context* if for all closed isos $\omega_i : a_i \leftrightarrow b_i$ not using fixpoints and for every closed value v of type a , the term $((\mu f. \omega)\omega_1 \dots \omega_n)v$ terminates. We say that the fixpoint is *terminating in an $(n+1)$ -context* if for all closed isos $\omega_i : a_i \leftrightarrow b_i$ terminating in n -contexts, and for every closed value v of type a , the term $((\mu f. \omega)\omega_1 \dots \omega_n)v$ terminates. Finally, we say that the fixpoint is *terminating in any finitary context* if for all n it is terminating in any n -context.

With the addition of lists, the non-overlapping and exhaustivity conditions need to be modified. The main problem is that we can no longer define the dimension of types using natural numbers: $[a]$ is in essence an infinite sum, and would have an “infinite” dimension. Instead, we combine the two conditions into the concept of *orthogonal decomposition*. Formally, given a type a , we say that a set S of patterns is an *orthogonal decomposition*, written $\text{OD}_a(S)$, when these patterns are pairwise orthogonal and when they cover the whole type. We formally define $\text{OD}_a(S)$ as follows. For all types a , $\text{OD}_a\{x\}$ is valid. For the unit type, $\text{OD}_\mathbb{1}\{\{\}\}$ is valid. If $\text{OD}_a(S)$ and $\text{OD}_b(T)$, then

$$\begin{array}{l}
\text{OD}_{a \oplus b}(\{\text{inj}_l v \mid v \in S\} \cup \{\text{inj}_r v \mid v \in T\}) \\
\text{and } \text{OD}_{a \otimes b}\{\langle v_1, v_2 \rangle \mid v_1 \in S, v_2 \in T, \text{FV}(v_1) \cap \text{FV}(v_2) = \emptyset\},
\end{array}$$

$$\begin{array}{c}
\frac{t_1 \rightarrow t_2}{C[t_1] \rightarrow C[t_2]} \text{ Cong} \quad \frac{\sigma[p] = v_1}{\mathbf{let } p = v_1 \mathbf{ in } t_2 \rightarrow \sigma(t_2)} \text{ LetE} \\
\frac{\sigma[v_i] = v}{\{ \mid v_1 \leftrightarrow t_1 \mid \dots \mid v_n \leftrightarrow t_n \} v \rightarrow \sigma(t_i)} \text{ IsoApp} \quad \frac{}{(\lambda f. \omega) \omega_2 \rightarrow \omega[\omega_2/f]} \text{ HIsoApp} \\
\frac{\Psi, f : a \leftrightarrow b \vdash_{\omega} \omega : (a_1 \leftrightarrow b_1) \rightarrow \dots \rightarrow (a_n \leftrightarrow b_n) \rightarrow (a \leftrightarrow b)}{\mu f. \omega \rightarrow \lambda f_1 \dots f_n. (\omega[(\mu f. \omega) f_1 \dots f_n / f]) f_1 \dots f_n} \text{ IsoRec}
\end{array}$$

Table 3. Reduction rules

where $\text{FV}(t)$ stands for the set of free value-variables in t . We then extend the notion of orthogonal decomposition to extended values as follows. If S is a set of extended values, $\text{OD}_a^{\text{ext}}(S)$ is true whenever $\text{OD}_a\{\text{Val}(e) \mid e \in S\}$. With this new characterization, the typing rule of iso in Eq. 1 still holds, and then can be re-written using this notion of orthogonal decomposition as shown in Tab. 2.

3.2 Semantics

In Tab. 3 we present the reduction rules for the reversible language. We assume that the reduction relation applies to well-typed terms. In the rules, the notation $C[-]$ stands for an *applicative context*, and is defined as: $C[-] ::= [-] \mid \mathbf{inj}_l C[-] \mid \mathbf{inj}_r C[-] \mid (C[-])\omega \mid \{\dots\} (C[-]) \mid \mathbf{let } p = C[-] \mathbf{ in } t_2 \mid \langle C[-], v \rangle \mid \langle v, C[-] \rangle$.

The inversion of isos is still possible but more subtle than in the first-order case. We define an inversion operation $(-)^{-1}$ on iso types with, $(a \leftrightarrow b)^{-1} := (b \leftrightarrow a)$, $((a \leftrightarrow b) \rightarrow T)^{-1} := ((b \leftrightarrow a) \rightarrow (T^{-1}))$. Inversion of isos is defined as follows. For fixpoints, $(\mu f. \omega)^{-1} = \mu f. (\omega^{-1})$. For variables, $(f)^{-1} := f$. For applications, $(\omega_1 \omega_2)^{-1} := (\omega_1)^{-1} (\omega_2)^{-1}$. For abstraction, $(\lambda f. \omega)^{-1} := \lambda f. (\omega^{-1})$. Finally, clauses are inverted as follows:

$$\left(\begin{array}{l} v_1 \leftrightarrow \mathbf{let } p_1 = \omega_1 p'_1 \mathbf{ in} \\ \dots \\ \mathbf{let } p_n = \omega_n p'_n \mathbf{ in } v'_1 \end{array} \right)^{-1} := \left(\begin{array}{l} v'_1 \leftrightarrow \mathbf{let } p'_n = \omega_n^{-1} p_n \mathbf{ in} \\ \dots \\ \mathbf{let } p'_1 = \omega_1^{-1} p_1 \mathbf{ in } v_1 \end{array} \right).$$

Note that $(-)^{-1}$ only inverts first-order arrows (\leftrightarrow), not second-order arrows (\rightarrow). This is reflected by the fact that iso-variable are non-linear while value-variables are. This is due to the clear separation of the two layers of the language.

The rewriting system satisfies the usual properties for well-typed terms: it is terminating, well-typed closed terms have a unique normal value-form, and it preserves typing.

Theorem 1. *The inversion operation is well-typed, in the sense that if $f_1 : a_1 \leftrightarrow b_1, \dots, f_n : a_n \leftrightarrow b_n \vdash_{\omega} \omega : T$ then we also have $f_1 : b_1 \leftrightarrow a_1, \dots, f_n : b_n \leftrightarrow a_n \vdash_{\omega} \omega^{-1} : T^{-1}$. \square*

Thanks to the fact that the language is terminating, we also recover the operational result of Sec. 2.3.

Theorem 2. Consider a well-typed, closed iso $\vdash_\omega \omega : a \leftrightarrow b$, and suppose that $\vdash_v v : a$ and that $\vdash_v w : b$, then $\omega^{-1}(\omega v) \rightarrow^* v$ and $\omega(\omega^{-1} w) \rightarrow^* w$. \square

4 Examples

In the previous sections, we developed a novel classical reversible language with a familiar syntax based on pattern-matching. The language includes a limited notion of higher-order functions and (terminating) recursive functions. We illustrate the expressiveness of the language with a few examples and motivate the changes and extensions needed to adapt the language to the quantum domain.

We encode booleans as follows: $\mathbb{B} = \mathbf{1} \oplus \mathbf{1}$, $\mathbf{tt} = \text{inj}_l()$, and $\mathbf{ff} = \text{inj}_r()$. One of the easiest function to define is $\mathbf{not} : \mathbb{B} \leftrightarrow \mathbb{B}$ which flips a boolean. The controlled-not gate which flips the second bit when the first is true can also be expressed:

$$\mathbf{not} : \mathbb{B} \leftrightarrow \mathbb{B} = \begin{pmatrix} \mathbf{ff} \leftrightarrow \mathbf{tt} \\ \mathbf{tt} \leftrightarrow \mathbf{ff} \end{pmatrix}, \quad \mathbf{cnot} : \mathbb{B} \otimes \mathbb{B} \leftrightarrow \mathbb{B} \otimes \mathbb{B} = \begin{pmatrix} \langle \mathbf{ff}, x \rangle \leftrightarrow \langle \mathbf{ff}, x \rangle \\ \langle \mathbf{tt}, \mathbf{ff} \rangle \leftrightarrow \langle \mathbf{tt}, \mathbf{tt} \rangle \\ \langle \mathbf{tt}, \mathbf{tt} \rangle \leftrightarrow \langle \mathbf{tt}, \mathbf{ff} \rangle \end{pmatrix}.$$

All the patterns in the previous two functions are orthogonal decompositions which guarantee reversibility as desired.

By using the abstraction facilities in the language, we can define higher-order operations that build complex reversible functions from simpler ones. For example, we can define a conditional expression parameterized by the functions used in the two branches:

$$\mathbf{if} : (a \leftrightarrow b) \rightarrow (a \leftrightarrow b) \rightarrow (\mathbb{B} \otimes a \leftrightarrow \mathbb{B} \otimes b) \\ \mathbf{if} = \lambda g. \lambda h. \begin{pmatrix} \langle \mathbf{tt}, x \rangle \leftrightarrow \text{let } y = g \ x \text{ in } \langle \mathbf{tt}, y \rangle \\ \langle \mathbf{ff}, x \rangle \leftrightarrow \text{let } y = h \ x \text{ in } \langle \mathbf{ff}, y \rangle \end{pmatrix}$$

Using \mathbf{if} and the obvious definition for the identity function \mathbf{id} , we can define $\mathbf{ctrl} :: (a \leftrightarrow a) \rightarrow (\mathbb{B} \otimes a \leftrightarrow \mathbb{B} \otimes a)$ as $\mathbf{ctrl} \ f = \mathbf{if} \ f \ \mathbf{id}$ and recover an alternative definition of \mathbf{cnot} as $\mathbf{ctrl} \ \mathbf{not}$. We can then define the controlled-controlled-not gate (aka the Toffoli gate) by writing $\mathbf{ctrl} \ \mathbf{cnot}$. We can even iterate this construction using fixpoints to produce an n -controlled-not function that takes a list of n control bits and a target bit and flips the target bit iff all the control bits are \mathbf{tt} :

$$\mathbf{cnot*} : ([\mathbb{B}] \otimes \mathbb{B}) \leftrightarrow ([\mathbb{B}] \otimes \mathbb{B}) \\ \mathbf{cnot*} = \mu f. \begin{pmatrix} \langle [], tb \rangle \leftrightarrow \text{let } tb' = \mathbf{not} \ tb \text{ in } \langle [], tb' \rangle \\ \langle \mathbf{ff} : cbs, tb \rangle \leftrightarrow \langle \mathbf{ff} : cbs, tb \rangle \\ \langle \mathbf{tt} : cbs, tb \rangle \leftrightarrow \text{let } \langle cbs', tb' \rangle = f \ \langle cbs, tb \rangle \text{ in } \langle \mathbf{tt} : cbs', tb' \rangle \end{pmatrix}$$

The language is also expressible enough to write conventional recursive (and higher-order) programs. We illustrate this expressiveness using the usual \mathbf{map} operation and an accumulating variant $\mathbf{mapAccu}$:

$$\mathbf{map} : (a \leftrightarrow b) \rightarrow ([a] \leftrightarrow [b]) \\ \lambda g. \mu f. \begin{pmatrix} [] \leftrightarrow [] \\ h : t \leftrightarrow \text{let } x = g \ h \text{ in} \\ \text{let } y = f \ t \text{ in } x : y \end{pmatrix}, \quad \mathbf{mapAccu} : (a \otimes b \leftrightarrow a \otimes c) \rightarrow (a \otimes [b] \leftrightarrow a \otimes [c]) \\ \lambda g. \mu f. \begin{pmatrix} \langle x, [] \rangle \leftrightarrow \langle x, [] \rangle \\ \langle x, (h : t) \rangle \leftrightarrow \text{let } \langle y, h' \rangle = g \ \langle x, h \rangle \text{ in} \\ \text{let } \langle z, t' \rangle = f \ \langle y, t \rangle \text{ in} \\ \langle z, (h' : t') \rangle \end{pmatrix}.$$

The three examples $\mathbf{cnot*}$, \mathbf{map} and $\mathbf{mapAccu}$ uses fixpoints which are clearly terminating in any finite context. Indeed, the functions are structurally recursive. A formal definition of this notion for the reversible language is as follows.

$$v'_i \begin{pmatrix} v_1 & v_2 & v_3 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Fig. 4. Classical iso

$$v'_i \begin{pmatrix} v_1 & v_2 & v_3 \\ a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

Fig. 5. Quantum iso

$$\langle \mathbf{tt}, x \rangle \begin{pmatrix} \langle \mathbf{tt}, x \rangle & \langle \mathbf{ff}, x \rangle \\ \frac{1}{\sqrt{2}} \text{Had} & \frac{1}{\sqrt{2}} \text{Id} \\ \frac{1}{\sqrt{2}} \text{Had} & \frac{-1}{\sqrt{2}} \text{Id} \end{pmatrix}$$

Fig. 6. Semantics of Gate

Definition 1. Define a *structurally recursive type* as a type of the form $[a] \otimes b_1 \otimes \dots \otimes b_n$. Let $\omega = \{v_i \leftrightarrow e_i \mid i \in I\}$ be an iso such that $f : a \leftrightarrow b \vdash_{\omega} \omega : a \leftrightarrow c$ where a is a structurally recursive type. We say that $\mu f.\omega$ is *structurally recursive* provided that for each $i \in I$, the value v_i is either of the form $\langle [], p_1, \dots, p_n \rangle$ or of the form $\langle h : t, p_1, \dots, p_n \rangle$. In the former case, e_i does not contain f as a free variable. In the latter case, e_i is of the form $C[f \langle t, p'_1, \dots, p'_n \rangle]$ where C is a context of the form $C[-] ::= [-] \mid \mathbf{let} p = C[-] \mathbf{in} t \mid \mathbf{let} p = t \mathbf{in} C[-]$.

This definition will be critical for quantum loops in the next section.

5 From Reversible Isos to Quantum Control

In the language presented so far, an iso $\omega : a \leftrightarrow b$ describes a bijection between the set \mathcal{B}_a of closed values of type a and the set \mathcal{B}_b of closed values of type b . If one regards \mathcal{B}_a and \mathcal{B}_b as the basis elements of some vector space $\llbracket a \rrbracket$ and $\llbracket b \rrbracket$, the iso ω becomes a 0/1 matrix.

As an example, consider an iso ω defined using three clauses of the form $\{ \mid v_1 \leftrightarrow v'_1 \mid v_2 \leftrightarrow v'_2 \mid v_3 \leftrightarrow v'_3 \}$. From the exhaustivity and non-overlapping conditions derives the fact that the space $\llbracket a \rrbracket$ can be split into the direct sum of the three subspaces $\llbracket a \rrbracket_{v_i}$ ($i = 1, 2, 3$) generated by v_i . Similarly, $\llbracket b \rrbracket$ is split into the direct sum of the subspaces $\llbracket b \rrbracket_{v'_i}$ generated by v'_i . One can therefore represent ω as the matrix $\llbracket \omega \rrbracket$ in Fig. 4: The “1” in each column v_i indicates to which subspace $\llbracket b \rrbracket_{v'_j}$ an element of $\llbracket a \rrbracket_{v_i}$ is sent to.

In Sec. 2.2 we discussed the fact that $v_i \perp v_j$ when $i \neq j$. This notation hints at the fact that $\llbracket a \rrbracket$ and $\llbracket b \rrbracket$ could be seen as Hilbert spaces and the mapping $\llbracket \omega \rrbracket$ as a unitary map from $\llbracket a \rrbracket$ to $\llbracket b \rrbracket$. The purpose of this section is to extend and formalize precisely the correspondence between isos and unitary maps.

The definition of clauses is extended following this idea of seeing isos as unitaries, and not only bijections on basis elements of the input space. We

therefore essentially propose to generalize the clauses to complex, linear combinations of values on the right-hand-side, such as shown on the left, with the side conditions on that the matrix of Fig. 5 is unitary. We define in Sec. 5.1 how this extends to second-order.

5.1 Extending the Language to Linear Combinations of Terms

The quantum unitary language extends the reversible language from the previous section by closing extended values and terms under complex, finite linear

combinations. For example, if v_1 and v_2 are values and α and β are complex numbers, $\alpha \cdot v_1 + \beta \cdot v_2$ is now an extended value.

Several approaches exist for performing such an extension. One can update the reduction strategy to be able to reduce these sums and scalar multiplications to normal forms [12, 18], or one can instead consider terms modulo the usual algebraic equalities [13, 18]: this is the strategy we follow for this paper.

When extending a language to linear combination of terms in a naive way, this added structure might generate inconsistencies in the presence of unconstrained fixpoints [13, 12, 18]. The weak condition on termination we imposed on fixpoints in the classical language was enough to guarantee reversibility. With the presence of linear combinations, we want the much stronger guarantee of unitarity. For this reason, we instead impose fixpoints to be *structurally recursive*.

The quantum unitary language is defined by allowing sums of terms and values and multiplications by complex numbers: if t and t' are terms, so is $\alpha \cdot t + t'$. Terms and values are taken modulo the equational theory of modules. We furthermore consider the value and term constructs $\langle -, - \rangle$, $\mathbf{let} p = - \mathbf{in} -$, $\mathbf{inj}_l(-)$, $\mathbf{inj}_r(-)$ distributive over sum and scalar multiplication. We do *not* however take iso-constructions as distributive over sum and scalar multiplication: $\{ \mid v_1 \leftrightarrow \alpha v_2 + \beta v_3 \}$ is *not* the same thing as $\alpha \{ \mid v_1 \leftrightarrow v_2 \} + \beta \{ \mid v_1 \leftrightarrow v_3 \}$. This is in the spirit of Lineal [12, 11].

The typing rules for terms and extended values are updated as follows. We only allow linear combinations of terms and values of the same type and of the same free variables. Fixpoints are now required to be *structurally recursive*, as introduced in Def. 1. Finally, an iso is now not only performing an “identity” as in Fig. 4 but a true unitary operation:

$$\frac{\begin{array}{l} \Delta_1; \Psi \vdash_v v_1 : a \quad \dots \quad \Delta_n; \Psi \vdash_v v_n : a \\ \Delta_1; \Psi \vdash_v e_1 : b \quad \dots \quad \Delta_n; \Psi \vdash_v e_n : b \\ \text{OD}_a \{v_1, \dots, v_n\} \quad \text{OD}_b^{\text{ext}} \{e_1, \dots, e_n\} \end{array} \quad \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix} \text{ is unitary}}{\Psi \vdash_\omega \left\{ \begin{array}{l} v_1 \leftrightarrow a_{11} \cdot e_1 + \dots + a_{1n} \cdot e_n \\ \dots \\ v_n \leftrightarrow a_{n1} \cdot e_1 + \dots + a_{nn} \cdot e_n \end{array} \right\} : a \leftrightarrow b.}$$

The reduction relation is updated in a way that it remains deterministic in this extended setting. It is split into two parts: the reduction of pure terms, i.e. non-extended terms or values, and linear combinations thereof. Pure terms and values reduce using the reduction rules found in Tab. 3. We do not extend applicative contexts to linear combinations. For linear combinations of pure terms, we simply ask that *all* pure terms that are not normal forms in the combination are reduced. This makes the extended reduction relation deterministic.

Example 1. This allows one to define an iso behaving as the Hadamard gate, or a slightly more complex iso conditionally applying another iso, whose behavior as a matrix is shown in Fig. 6.

$$\begin{array}{l} \text{Had} : \mathbb{B} \leftrightarrow \mathbb{B} \\ \left(\begin{array}{l} \mathbf{tt} \leftrightarrow \frac{1}{\sqrt{2}} \mathbf{tt} + \frac{1}{\sqrt{2}} \mathbf{ff} \\ \mathbf{ff} \leftrightarrow \frac{1}{\sqrt{2}} \mathbf{tt} - \frac{1}{\sqrt{2}} \mathbf{ff} \end{array} \right), \end{array} \quad \begin{array}{l} \text{Gate} : \mathbb{B} \otimes \mathbb{B} \leftrightarrow \mathbb{B} \otimes \mathbb{B} \\ \left(\begin{array}{l} \langle \mathbf{tt}, x \rangle \leftrightarrow \mathbf{let} y = \text{Had } x \mathbf{in} \frac{1}{\sqrt{2}} \langle \mathbf{tt}, y \rangle + \frac{1}{\sqrt{2}} \langle \mathbf{ff}, y \rangle \\ \langle \mathbf{ff}, x \rangle \leftrightarrow \mathbf{let} y = \text{Id } x \mathbf{in} \frac{1}{\sqrt{2}} \langle \mathbf{tt}, y \rangle - \frac{1}{\sqrt{2}} \langle \mathbf{ff}, y \rangle \end{array} \right). \end{array}$$

With this extension to linear combinations of terms, one can characterize normal forms as follows.

Lemma 1 (Structure of the normal forms). *Let ω be such that $\vdash_\omega \omega : a \leftrightarrow b$. For all closed values v of type a , the term ωv rewrites to a normal form $\sum_{i=1}^N \alpha_i \cdot w_i$ where $N < \infty$, each w_i is a closed value of type b and $\sum_i |\alpha_i| = 1$.*

Proof. The fact that ωv converges to a normal form is a corollary of the fact that we impose structural recursion on fixpoints. The property of the structure of the normal form is then proven by induction on the maximal number of steps it takes to reach it. It uses the restriction on the introduction of sums in the typing rule for clauses in isos and the determinism of the reduction. \square

In the classical setting, isos describe bijections between sets of closed values: it was proven by considering the behavior of an iso against its inverse. In the presence of linear combinations of terms, we claim that isos describe more than bijections: they describe unitary maps. In the next section, we discuss how types can be understood as Hilbert spaces (Sec. 5.2) and isos as unitary maps (Secs 5.3 and 5.4).

5.2 Modeling Types as Hilbert Spaces

By allowing complex linear combinations of terms, closed normal forms of finite types such as \mathbb{B} or $\mathbb{B} \otimes \mathbb{B}$ can be regarded as complex vector spaces with basis consisting of closed values. For example, \mathbb{B} is associated with $\llbracket \mathbb{B} \rrbracket = \{\alpha \cdot \mathbf{tt} + \beta \cdot \mathbf{ff} \mid \alpha, \beta \in \mathbb{C}\} \cong \mathbb{C}^2$. We can consider this space as a complex Hilbert space where the scalar product is defined on basis elements in the obvious way: $\langle v | v \rangle = 1$ and $\langle v | w \rangle = 0$ if $v \neq w$. The map \mathbf{Had} of Ex. 1 is then effectively a unitary map on the space $\llbracket \mathbb{B} \rrbracket$.

The problem comes from lists: the type $[\mathbb{1}]$ is inhabited by an infinite number of closed values: $[], [()], [(), ()], [(), (), ()], \dots$. To account for this case, we need to consider infinitely dimensional complex Hilbert spaces. In general, a complex Hilbert space [19] is a complex vector space endowed with a scalar product that is complete with respect the distance induced by the scalar product. The completeness requirement implies for example that the infinite linear combination $[\] + \frac{1}{2} \cdot [()] + \frac{1}{4} [(), ()] + \frac{1}{8} [(), (), ()] + \dots$ needs to be an element of $\llbracket [\mathbb{1}] \rrbracket$. To account for these limit elements, we propose to use the standard [19] Hilbert space ℓ^2 of infinite sequences.

Definition 2. Let a be a value type. As before, we write \mathcal{B}_a for the set of closed values of type a , that is, $\mathcal{B}_a = \{v \mid \vdash_v v : a\}$. The *span of a* is defined as the Hilbert space $\llbracket a \rrbracket = \ell^2(\mathcal{B}_a)$ consisting of sequences $(\phi_v)_{v \in \mathcal{B}_a}$ of complex numbers indexed by \mathcal{B}_a such that $\sum_{v \in \mathcal{B}_a} |\phi_v|^2 < \infty$. The scalar product on this space is defined as $\langle (\phi_v)_{v \in \mathcal{B}_a} | (\psi_v)_{v \in \mathcal{B}_a} \rangle = \sum_{v \in \mathcal{B}_a} \overline{\phi_v} \psi_v$.

We shall use the following conventions. A closed value v of $\llbracket a \rrbracket$ is identified with the sequence $(\delta_{v,v'})_{v' \in \mathcal{B}_a}$ where $\delta_{v,v} = 1$ and $\delta_{v,v'} = 0$ if $v \neq v'$. An element $(\phi_v)_{v \in \mathcal{B}_a}$ of $\llbracket a \rrbracket$ is also written as the infinite, formal sum $\sum_{v \in \mathcal{B}_a} \phi_v \cdot v$.

5.3 Modeling Isos as Bounded Linear Maps

We can now define what is the linear map associated to an iso.

Definition 3. For each closed iso $\vdash_{\omega} \omega : a \leftrightarrow b$ we define $\llbracket \omega \rrbracket$ as the linear map from $\llbracket a \rrbracket$ to $\llbracket b \rrbracket$ sending the closed value $v : a$ to the normal form of $\omega v : b$ under the rewrite system.

In general, the fact that $\llbracket \omega \rrbracket$ is well-defined is not trivial. If it is formally stated in Theorem 3, we can first try to understand what could go wrong. The problem comes from the fact that the space $\llbracket a \rrbracket$ is not finite in general. Consider the iso $\mathbf{map\ Had} : [\mathbb{B}] \leftrightarrow [\mathbb{B}]$. Any closed value $v : [\mathbb{B}]$ is a list and the term $(\mathbf{map\ Had}) v$ rewrites to a normal form consisting of a linear combination of lists. Denote the linear combination associated to v with L_v . An element of $\llbracket [\mathbb{B}] \rrbracket$ is a sequence $\phi = (\phi_v)_{v \in \mathcal{B}[\mathbb{B}]}$. From Definition 3, the map $\llbracket \omega \rrbracket$ sends the element $\phi \in \llbracket [\mathbb{B}] \rrbracket$ to $\sum_{v \in \mathcal{B}[\mathbb{B}]} \phi_v \cdot L_v$. This is an infinite sum of sums of complex numbers: we need to make sure that it is well-defined: this is the purpose of the next result. Because of the constraints on the language, we can even show that it is a *bounded* linear map.

In the case of the map $\mathbf{map\ Had}$, we can understand why it works as follows. The space $\llbracket [\mathbb{B}] \rrbracket$ can be decomposed as the direct sum $\sum_{i=0}^{\infty} E_i$, where E_i is generated with all the lists in \mathbb{B} of size i . The map $\mathbf{map\ Had}$ is acting locally on each finitely-dimensional subspace E_i . It is therefore well-defined. Because of the unitarity constraint on the linear combinations appearing in \mathbf{Had} , the operation performed by $\mathbf{map\ Had}$ sends elements of norm 1 to elements of norm 1. This idea can be formalized and yield the following theorem.

Theorem 3. *For each closed iso $\vdash_{\omega} \omega : a \leftrightarrow b$ the linear map $\llbracket \omega \rrbracket : \llbracket a \rrbracket \rightarrow \llbracket b \rrbracket$ is well-defined and bounded.* \square

5.4 Modeling Isos as Unitary Maps

In this section, we show that not only closed isos can be modeled as bounded linear maps, but that these linear maps are in fact unitary maps. The problem comes from fixpoints. We first consider the case of isos written without fixpoints, and then the case with fixpoints.

Without recursion. The case without recursion is relatively easy to treat, as the linear map modeling the iso can be compositionally constructed out of elementary unitary maps.

Theorem 4. *Given a closed iso $\vdash_{\omega} \omega : a \leftrightarrow b$ defined without the use of recursion, the linear map $\llbracket \pi \rrbracket : \llbracket a \rrbracket \rightarrow \llbracket b \rrbracket$ is unitary.* \square

The proof of the theorem relies on the fact that to each closed iso $\vdash_{\omega} \omega : a \leftrightarrow b$ one can associate an operationally equivalent iso $\vdash_{\omega} \omega' : a \leftrightarrow b$ that does not use iso-variables nor lambda-abstractions. We can define a notion of *depth* of an iso

as the number of nested isos. The proof is done by induction on this depth of the iso ω : it is possible to construct a unitary map for ω using the unitary maps for each ω_{ij} as elementary building blocks.

As an illustration, the semantics of `Gate` of Example 1 is given in Figure 6.

Isos with structural recursion. When considering fixpoints, we cannot rely anymore on this finite compositional construction: the space $\llbracket a \rrbracket$ cannot anymore be regarded as a *finite* sum of subspaces described by each clause.

We therefore need to rely on the formal definition of unitary maps in general, infinite Hilbert spaces. On top of being bounded linear, a map $\llbracket \omega \rrbracket : \llbracket a \rrbracket \rightarrow \llbracket b \rrbracket$ is unitary if (1) it preserves the scalar product: $\langle \llbracket \omega \rrbracket(e) | \llbracket \omega \rrbracket(f) \rangle = \langle e | f \rangle$ for all e and f in $\llbracket a \rrbracket$ and (2) it is surjective.

Theorem 5. *Given a closed iso $\vdash_{\omega} \omega : a \leftrightarrow b$ that can use structural recursion, the linear map $\llbracket \pi \rrbracket : \llbracket a \rrbracket \rightarrow \llbracket b \rrbracket$ is unitary. \square*

The proof uses the idea highlighted in Sec. 5.4: for a structurally recursive iso of type $[a] \otimes b \leftrightarrow c$, the Hilbert space $\llbracket [a] \otimes b \rrbracket$ can be split into a canonical decomposition $E_0 \oplus E_1 \oplus E_2 \oplus \dots$, where E_i contains only the values of the form $\langle [x_1 \dots x_i], y \rangle$, containing the lists of size i . On each E_i , the iso is equivalent to an iso without structural recursion.

6 Conclusion

In this paper, we proposed a reversible language amenable to quantum superpositions of values. The language features a weak form of higher-order that is nonetheless expressible enough to get interesting maps such as generalized Toffoli operators. We sketched how this language effectively encodes bijections in the classical case and unitary operations in the quantum case. It would be interesting to see how this relates to join inverse categories [14, 15].

In the vectorial extension of the language we have the same control as in the classical, reversible language. Tests are captured by clauses, and naturally yield quantum tests: this is similar to what can be found in QML [5, 6], yet more general since the QML approach is restricted to `if-then-else` constructs. The novel aspect of quantum control that we are able to capture here is a notion of *quantum loops*. These loops were believed to be hard, if not impossible. What makes it work in our approach is the fact that we are firmly within a closed quantum system, without measurements. This makes it possible to only consider unitary maps and frees us from the Löwer order on positive matrices [6]. As we restrict fixpoints to structural recursion, valid isos are regular enough to capture unitarity. Ying [7] also proposes a framework for quantum while-loops that is similar in spirit to our approach at the level of denotations: in his approach the control part of the loops is modeled using an external systems of “coins” which, in our case, correspond to conventional lists. Reducing the manipulation of this external coin system to iteration on lists allowed us to give a simple operational semantics for the language.

References

1. Wootters, W.K., Zurek, W.H.: A single quantum cannot be cloned. *Nature* **299** (October 1982) 802–803
2. Nielsen, M.A., Chuang, I.L.: *Quantum Computation and Quantum Information*. Cambridge University Press (2002)
3. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: A scalable quantum programming language. In: Proc. PLDI'13. (2013) 333–342
4. Paykin, J., Rand, R., Zdancewic, S.: QWIRE: A core language for quantum circuits. In: Proc. POPL'17. (2017) 846–858
5. Altenkirch, T., Grattage, J.: A functional quantum programming language. In: Proc. LICS'05. (2005) 249–258
6. Badescu, C., Panangaden, P.: Quantum alternation: Prospects and problems. In: Proceedings 12th International Workshop on Quantum Physics and Logic, QPL 2015, Oxford, UK, July 15-17, 2015. (2015) 33–42
7. Ying, M.: *Foundations of Quantum Programming*. Elsevier Science (2016)
8. Selinger, P.: Towards a quantum programming language. *Mathematical Structures in Computer Science* **14**(4) (August 2004) 527–586
9. Vizzotto, J.K., Altenkirch, T., Sabry, A.: Structuring quantum effects: superoperators as arrows. *Mathematical Structures in Computer Science* **16**(3) (2006) 453–468
10. James, R.P., Sabry, A.: Theseus: A high-level language for reversible computation. In: *Reversible Computation, Booklet of work-in-progress and short reports*. (2016)
11. Arrighi, P., Daz-Caro, A., Valiron, B.: The vectorial lambda-calculus. *Information and Computation* **254**(1) (2017) 105–139
12. Arrighi, P., Dowek, G.: Lineal: A linear-algebraic lambda-calculus. *Logical Methods in Computer Science* (2013)
13. Vaux, L.: The algebraic lambda calculus. *Mathematical Structures in Computer Science* **19**(5) (2009) 1029–1059
14. Glück, R., Kaarsgaard, R.: A categorical foundation for structured reversible flowchart languages: Soundness and adequacy. Available on arXiv:1710.03666 [cs.PL] (2017)
15. Kaarsgaard, R., Axelsen, H.B., Glück, R.: Join inverse categories and reversible recursion. *Journal of Logical and Algebraic Methods in Programming* **87** (2017) 33–50
16. van Tonder, A.: A lambda calculus for quantum computation. *SIAM Journal of Computing* **33**(5) (2004) 1109–1135
17. Sabry, A., Valiron, B., Vizzotto, J.K.: From symmetric pattern-matching to quantum control (extended version). To appear (2018)
18. Assaf, A., Díaz-Caro, A., Perdrix, S., Tasson, C., Valiron, B.: Call-by-value, call-by-name and the vectorial behaviour of the algebraic λ -calculus. *Logical Methods in Computer Science* **10**:4(8) (December 2014)
19. Young, N.: *An Introduction to Hilbert Space*. Cambridge University Press (1988)