Translating Finite-Typed Symmetric Pattern Matching into Ket

Flávio Borin Júnior
Universidade Federal de Santa Maria
Santa Maria, Brazil
fbjunior@inf.ufsm.br

Juliana Kaizer Vizzotto
Universidade Federal de Santa Maria
Santa Maria, Brazil
juvizzotto@inf.ufsm.br

Abstract

We present a translation procedure from Symmetric Pattern Matching (SPM) isomorphisms to executable quantum programs in Ket. Our approach exploits the preservation structure of pattern-matching clauses to identify control configurations, allowing high-level specifications to be rewritten as controlled quantum operations. Isomorphisms are first interpreted as unitary operators and then translated into Ket through a multiplexer-based decomposition, while single-qubit leaf operations are synthesized via ZYZ decomposition. As a first step toward compiling the full SPM framework, we restrict our attention to finite types and to isomorphisms whose non-controlled leaf operations act on a single qubit. This work contributes to bridging high-level reversible programming abstractions and practical quantum programming frameworks.

Keywords

Quantum Computing, Quantum Programming Languages, Reversible Computing

1 Introduction

Quantum programming languages provide high-level abstractions for expressing quantum algorithms through mechanisms such as functional programming, type systems, algebraic data types, and reversible computation [1, 6, 13, 16, 17]. However, quantum programs must ultimately be translated into executable circuit descriptions compatible with concrete software frameworks and hardware backends, making compilation a central challenge in quantum programming.

This work focuses on the compilation of Symmetric Pattern Matching (SPM) into Ket. SPM is a reversible and functional quantum programming paradigm in which computations are described through isomorphisms represented as collections of symmetric pattern-matching clauses [14]. The language supports quantum control and linear combinations of values while preserving reversibility at the syntactic level, combining ideas from reversible programming languages [8] and linear-algebraic approaches to quantum computation [2, 18, 19]. Ket provides a suitable compilation target due to its support for controlled operations, gate synthesis, and programmatic circuit construction [5].

The central observation explored in this work is that finite SPM isomorphisms admit a natural matrix representation [14]. By interpreting the clauses of an isomorphism over all basis values of its input type, a corresponding unitary matrix can be constructed. Moreover, preservation patterns appearing in the clauses reveal control structures that induce a quantum multiplexer interpretation [4, 7, 11]. This representation can be systematically translated into Ket code through controlled operations. For the subset of the language considered in this work, where non-controlled leaf operations act on a single qubit, the resulting operators are synthesized through ZYZ decomposition [12]. This compilation strategy is closely related to the Quantum Shannon Decomposition [15], but the multiplexing structure is inferred directly from the syntactic organization of clauses.

The compilation procedure presented here constitutes a first step towards connecting SPM to an executable quantum programming platform. Future extensions should address arbitrary multi-qubit operators through more general synthesis techniques such as KAK decomposition [9], as well as recursive structures and infinite types.

The remainder of this paper is organized as follows. Section 2 introduces the syntax of SPM. Section 3 presents the semantic interpretation of values and their initialization procedure. Section 4 describes the compilation pipeline, including matrix extraction, control identification, multiplexer construction, and gate synthesis. Section 5 discusses future work, and Section 6 concludes the paper.

2 Symmetric Pattern Matching

The language is constructed from a collection of basic and compound types [14]. At its core, SPM includes the primitive unit type $\mathbb{1}$, which has exactly one value denoted by $()$, together with the product type $a \otimes b$, representing pairs of values, and the sum type $a \oplus b$, representing a choice between values of type a or b. From these constructions, Boolean values emerge naturally as

$$\mathbb{B} = \mathbb{1} \oplus \mathbb{1}.$$

This definition induces the computational basis values

$$\texttt{ff} = \texttt{inj}_l\ () : \mathbb{B}, \quad \texttt{tt} = \texttt{inj}_r\ () : \mathbb{B},$$

which correspond respectively to the computational basis states $|0\rangle$ and $|1\rangle$. Through tensor composition and sum constructions, larger computational spaces can then be defined.

Quantum gates are represented in SPM as isomorphisms, defined through collections of reversible pattern-matching clauses. An isomorphism has type $a \leftrightarrow b$, representing a unitary transformation between the associated spaces $[\![a]\!]$ and $[\![b]\!]$, by $[\![\cdot]\!]$ meaning the semantic interpretation of a type.

As an example, the Hadamard transformation over Boolean values can be expressed as

$$H : \mathbb{B} \leftrightarrow \mathbb{B} := \begin{pmatrix} \mathrm{ff} \leftrightarrow \frac{1}{\sqrt{2}}\mathrm{ff} + \frac{1}{\sqrt{2}}\mathrm{tt} \\ \\ \mathrm{tt} \leftrightarrow \frac{1}{\sqrt{2}}\mathrm{ff} - \frac{1}{\sqrt{2}}\mathrm{tt} \end{pmatrix}.$$

To formalize these constructions, we now present the complete grammar of the language.

$$\begin{array}{rl}
\text{Val \& term types} & a, b ::= \mathbb{1} \mid a \oplus b \mid a \otimes b \\
\text{Iso types} & T ::= a \leftrightarrow b \\
\text{Pure values} & v ::= () \mid x \mid \mathrm{inj}_l\, v \mid \mathrm{inj}_r\, v \mid \langle v_1, v_2 \rangle \\
\text{Combination of values} & e ::= v \mid e_1 + e_2 \mid \alpha \cdot e \\
\text{Products} & p ::= x \mid \langle p_1, p_2 \rangle \\
\text{Extended Values} & e ::= v \mid \mathbf{let}\ p_1 = \omega\ p_2\ \mathbf{in}\ e \\
\text{Isomorphisms} & \omega ::= \left\{ \begin{array}{l} v_0 \leftrightarrow e_0 \\ v_1 \leftrightarrow e_1 \\ \cdots \end{array} \right. \\
\text{Terms} & t ::= v \mid \omega\, t
\end{array}$$

where $\alpha \in \mathbb{C}$ is a complex amplitude and x represents a variable. This grammar defines the syntactic structure necessary to construct unitary transformations and quantum states.

To verify the validity of an isomorphism, the semantics of SPM requires the linear transformation associated with an iso ω, denoted by $[\![\omega]\!]$, to be unitary. In practice, this corresponds to verifying the orthonormality of the quantum states induced by its clauses.

Formally, let $|v_i\rangle$ be the output state vector associated with the i-th clause of an isomorphism. The following conditions must hold:

$$\langle v_i \mid v_i \rangle = 1 \quad \forall i,$$
$$\langle v_i \mid v_j \rangle = 0 \quad \forall i \neq j.$$

The first condition ensures normalization preservation, while the second guarantees orthogonality between distinct clauses [8]. However, these conditions alone are not sufficient to characterize a valid unitary transformation, since the induced operator must additionally be exhaustive over the computational basis [14]. In particular, every basis value must be matched by exactly one clause.

When variables occur in patterns, linearity further requires that bound variables are preserved structurally [2]. In particular, a variable appearing on the left-hand side of a clause may occur exactly once on the right-hand side expression. Otherwise, the transformation could implicitly duplicate or erase quantum information, violating reversibility and unitarity. This restriction is consistent with the quantum no-cloning theorem, which states that arbitrary quantum states cannot be copied by a unitary transformation [12].

Together, normalization preservation, orthogonality, exhaustiveness, and the linearity of clauses ensure that their collection defines a valid unitary transformation over the computational basis.

For example, the controlled-X gate may be expressed as

$$CX : \mathbb{B} \otimes \mathbb{B} \leftrightarrow \mathbb{B} \otimes \mathbb{B} := \begin{pmatrix} \langle \mathrm{ff}, \mathrm{a} \rangle \leftrightarrow \langle \mathrm{ff}, \mathrm{a} \rangle & \\ \\ \langle \mathrm{tt}, \mathrm{a} \rangle \leftrightarrow & \mathbf{let}\ \mathrm{na} = X\ \mathrm{a} \\ & \mathbf{in}\ \langle \mathrm{tt}, \mathrm{na} \rangle \end{pmatrix}.$$

where the variable "a" is preserved linearly across both clauses. The **let**...**in**... construct simply binds the value resulting from the expression X a to a new variable "na", which is later used to construct the output $\langle \mathrm{tt}, \mathrm{na} \rangle$.

3 Translating Finite-Typed Values and Terms

For every finite type a, the SPM type system associates a finite Hilbert space $[\![a]\!]$, whose dimension is determined from its type structure [14]. We denote the dimension induced by a type a as $|a|$. The dimension induced by each type constructor is summarized in Table 1. When compiling SPM programs into Ket, the induced Hilbert spaces must be represented using qubit registers. Since a register of n qubits represents a space of dimension 2^n [12], arbitrary finite types must be embedded into the smallest power-of-two space capable of representing them.

Table 1: Dimension function induced by SPM types

Type	Dimension
$\mathbb{1}$	1
$a \oplus b$	$\lvert a\rvert + \lvert b\rvert$
$a \otimes b$	$\lvert a\rvert \cdot \lvert b\rvert$

Formally, the number of qubits required to represent all values of type a is $q(a) = \lceil \log_2 |a| \rceil$. Consequently, the effective Hilbert space used during execution has dimension $2^{q(a)}$. If $|a|$ is not itself a power of two, we introduce unused computational basis states.

For example, the type $\mathbb{T} = \mathbb{1} \oplus \mathbb{1} \oplus \mathbb{1}$ has a dimension $|\mathbb{T}| = 3$, but requires $q(\mathbb{T}) = \lceil \log_2 3 \rceil = 2$ qubits for representation. Thus, the original three-dimensional space is embedded into a $(2^2 = 4)$-dimensional Hilbert space, leaving one computational basis state unused.

The rewriting process of pure values therefore proceeds by allocating $q(a)$ qubits and embedding the basis vector induced by a into the corresponding Hilbert space.

Formally, we define a mapping

$$(\!|\cdot|\!)_a : \mathcal{B}_a \hookrightarrow \mathcal{H}^{2^{q(a)}},$$

where $\mathcal{B}_a = \{v_0, \ldots, v_{|a|-1}\}$ denotes the set of basis values generated by the type a, and $\mathcal{H}^{2^{q(a)}} = \left(\mathbb{C}^2\right)^{\otimes q(a)}$ is the required Hilbert space.

The mapping associates each value of a given type a with a unique basis vector in the target space. Rather than relying on a fixed syntactic encoding strategy, the conversion is defined inductively from the ordering induced by the type structure.

Let $\iota_a : \mathcal{B}_a \to \{0, \ldots, |a| - 1\}$ be an indexing function assigning a unique integer to each basis value of type a. Then the translation map is defined as

$$(\!|v|\!)_a = |\mathrm{bin}(\iota_a(v))\rangle\,,$$

where $\mathrm{bin}(n)$ denotes the binary representation of the integer n padded to length $q(a)$.

For example, Boolean values satisfy that

$$\iota_{\mathbb{B}}(\texttt{ff}) = 0, \quad \iota_{\mathbb{B}}(\texttt{tt}) = 1,$$

yielding

$$(\!|\texttt{ff}|\!)_{\mathbb{B}} = |0\rangle\,, \quad (\!|\texttt{tt}|\!)_{\mathbb{B}} = |1\rangle\,.$$

Similarly, for the type

$$\mathbb{T} = \mathbb{1} \oplus (\mathbb{1} \oplus \mathbb{1}),$$

one possible indexing is

$$\begin{aligned} \iota_{\mathbb{T}}(\texttt{inj}_l()) &= 0, \\ \iota_{\mathbb{T}}(\texttt{inj}_r(\texttt{inj}_l())) &= 1, \\ \iota_{\mathbb{T}}(\texttt{inj}_r(\texttt{inj}_r())) &= 2, \end{aligned}$$

which induces the embedding

$$\begin{aligned} (\!|v_0|\!)_{\mathbb{T}} &= |00\rangle\,, \\ (\!|v_1|\!)_{\mathbb{T}} &= |01\rangle\,, \\ (\!|v_2|\!)_{\mathbb{T}} &= |10\rangle\,. \end{aligned}$$

The remaining basis state $|11\rangle$ remains unused.

When appearing inside a term (as an isomorphism argument), a value must be physically prepared for execution. This motivates a state-preparation procedure for pure terms, which is induced by the computational basis translation introduced previously.

Since every pure value corresponds uniquely to a computational basis state, the translation process consists of allocating the required qubits and applying Pauli-X gates to initialize those that must be in state $|1\rangle$.

Formally, if $(\!|v|\!)_a = |b_1 \cdots b_{q(a)}\rangle$, then the generated Ket code allocates the qubits

$$b_1, \ldots, b_{q(a)} = \texttt{Process.alloc}(q(a)),$$

and applies

$$X(b_i)$$

for every position i such that $b_i = 1$.

Table 2 summarizes examples of the rewriting process, illustrating how SPM values of different types can be converted into their corresponding computational basis states and translated into Ket code.

The computational basis translation defined for pure values is also employed during clause compilation, since the left-hand side of every isomorphism clause is restricted to pure values. In addition, it is necessary to extend this translation to combinations of values and scalar multiplications, as expressions such as

$$\frac{1}{\sqrt{2}} \cdot \mathrm{tt} + \frac{1}{\sqrt{2}} \cdot \mathrm{ff}$$

may appear on the right-hand side of clauses.

In this case, the translation extends directly to linear combinations of values [3]. In particular, it is compatible with both vector addition and scalar multiplication:

$$\begin{aligned} (\!|v_1 + v_2|\!)_a &= (\!|v_1|\!)_a + (\!|v_2|\!)_a \\ (\!|\alpha v|\!)_a &= \alpha (\!|v|\!)_a \end{aligned}$$

This translation capability will play an important role in the next section.

4 Translating Finite-Typed Isomorphisms

Having established how finite SPM values are converted into computational basis states, we now describe the translation of finite isomorphisms into executable Ket programs.

The translation process is structured in three stages. First, the isomorphism is interpreted as a unitary matrix by evaluating its clauses over the computational basis of the input type. Next, the resulting matrix is analyzed in order to identify a quantum multiplexing structure through a reordering of the basis states, induced by its pattern-matching structure. This decomposition separates control configurations from the operational subspaces associated with each branch of the operation.

After the multiplexed structure is obtained, each operational block must be synthesized into elementary quantum operations. In the general case, this step may require decomposition techniques such as KAK, cosine-sine decomposition (CSD), or other unitary synthesis methods. In the restricted fragment considered in this work, however, all leaf blocks act on a single qubit, allowing them to be decomposed through

Table 2: Examples of SPM values, their semantic interpretation, and their corresponding state initialization in Ket

Type (a)	$\|a\|$	$q(a)$	Value (v)	$(\!\|v\|\!)_a$	Ket Code
$\mathbb{1}$	1	0	$()$	–	–
$\mathbb{B}$	2	1	$\mathtt{tt} = \mathtt{inj}_r\ ()$	$\|1\rangle$	`b0 = Process().alloc(1)` `X(b0)`
$\mathbb{B} \otimes \mathbb{B}$	4	2	$\langle \mathtt{inj}_l\ (), \mathtt{inj}_r\ () \rangle$	$\|01\rangle$	`b0, b1 = Process().alloc(2)` `X(b1)`
$\mathbb{1} \oplus \mathbb{B}$	3	2	$\mathtt{inj}_r\ (\mathtt{inj}_l\ ())$	$\|10\rangle$	`b0, b1 = Process().alloc(2)` `X(b0)`

the ZYZ decomposition and translated directly into Ket as sequences of elementary rotations and controlled operations.

As this procedure requires evaluating the isomorphism over every basis state induced by the input type, it is inherently restricted to finite types.

4.1 Unitary Extraction

An isomorphism defined by pattern matching can be translated into a linear operator constructed from its clauses interpretation.

Given an isomorphism

$$\omega : a \leftrightarrow b := \begin{pmatrix} p_0 \leftrightarrow e_0 \\ p_1 \leftrightarrow e_1 \\ \cdots \end{pmatrix},$$

each clause $p_i \leftrightarrow e_i$ specifies the image of the basis states matching the pattern p_i.

Let v be an element of the basis of the type a. By exhaustiveness, v matches exactly one pattern p_i.

The matching process induces a context

$$\Gamma_{i,v}$$

containing the bindings introduced by p_i. More precisely, if

$$(\!|v|\!)_a = |d_0\rangle \otimes \cdots \otimes |d_{q(a)-1}\rangle,$$

then every variable occurring in p_i is associated with the corresponding d_j subvalue extracted from v. The resulting context $\Gamma_{i,v}$ therefore assigns a concrete value to every variable bound by p_i. Applying these bindings to e_i and recursively eliminating `let`-expressions (described in the next subsections) eventually yields a pure value

$$e_i[\Gamma_{i,v}].$$

The resulting expression can then be translated into a vector of $[\![b]\!]$,

$$|e_{i,v}\rangle = (\!|e_i[\Gamma_{i,v}]|\!)_b.$$

Consequently, each basis vector $|v\rangle = (\!|v|\!)_a$ induces the transformation

$$|e_{i,v}\rangle \langle v| .$$

The preliminary semantics of the whole isomorphism is then given by the matrix

$$[\![\omega]\!]^\circ = \sum_{v \in \mathcal{B}_a} |e_{i,v}\rangle \langle v| .$$

4.1.1 Spatial Completion. As previously discussed, the encoding of a type may leave unused basis states in the corresponding Hilbert space. For instance, the type $\mathbb{T}$ requires two qubits for its representation, leaving one computational basis state without an associated value.

In such cases, the matrix obtained from the isomorphism semantics acts only on the subspace generated by valid values. To obtain a valid quantum operation, this matrix is extended to the entire Hilbert space by acting as the identity on all unused basis states.

Formally, if $[\![\omega : a \leftrightarrow b]\!]^\circ$ is a $n \times n$ matrix, the completed unitary is

$$[\![\omega]\!] = \begin{bmatrix} [\![\omega]\!]^\circ & 0 \\ 0 & I_{2^{q(a)}-n} \end{bmatrix}.$$

4.1.2 `Let-in` *Elimination.* Although `let`-expressions appear as a primitive syntactic construct, they merely provide a convenient notation for composing isomorphism applications. Their role is to name the result of an intermediate computation and make it available to subsequent expressions.

Consider an expression

$$\mathtt{let}\ s = f\ w\ \mathtt{in}\ e.$$

Before evaluating the application $f\ w$, the argument w is simplified using the bindings currently available in the given context, generating a value $w[\Gamma_{i,v}]$. The application $f\ (w[\Gamma_{i,v}])$ can then be evaluated using the same procedure described in this section.

However, the resulting value assigned to s is not necessarily a simple basis value. Since isomorphism applications may produce superpositions, the evaluation may yield a linear combination

$$\sum_j \alpha_j u_j,$$

where each u_j is a pure value. Each branch j induces a distinct continuation of the evaluation, corresponding to the quantum parallelism property. In this case, the body of the `let`-expression can be evaluated independently for every branch and the results are recombined linearly.

This behavior follows from the fact that `let`-expressions, like products, are distributive over addition and scalar multiplication. Precisely,

$$\texttt{let } s = a+b \texttt{ in } e \equiv (\texttt{let } s = a \texttt{ in } e)+(\texttt{let } s = b \texttt{ in } e),$$

and

$$\texttt{let } s = \alpha \cdot a \texttt{ in } e \equiv \alpha \cdot (\texttt{let } s = a \texttt{ in } e).$$

For each branch j, matching u_j against the pattern s produces bindings for the variables bound by s, generating a new context $\Gamma_{i,v,j}$. The body e is then evaluated recursively under $\Gamma_{i,v,j}$, producing a value

$$e[\Gamma_{i,v,j}].$$

The results of all branches are then recombined scaled to their amplitudes, producing

$$\sum_j \alpha_j \cdot e[\Gamma_{i,v,j}].$$

This process is applied recursively until no `let`-expressions remain. Since recursive isomorphisms are not considered, the evaluation eventually terminates in a `let`-free value for every branch. This procedure therefore provides a complete interpretation of right-hand side expressions appearing in clauses.

4.2 Multiplexing

The previous section showed how an isomorphism ω can be translated into a unitary matrix $[\![\omega]\!]$. To generate executable Ket code, however, a more structured representation is desirable. Rather than treating $[\![\omega]\!]$ as a mere unitary operator, we exploit the pattern-matching structure of the original isomorphism to decompose it into a collection of controlled operations.

The key observation comes from the semantics of controlled quantum operations: control is fundamentally a mechanism of syntactic invariance that selects computational branches.

In a controlled isomorphism, this appears at the level of product types, where a qubit acts as a controller when its value is preserved across the transformation. When a clause preserves the value of a given index in a product structure, that value does not contribute to the computation of the output state, instead, it works as a branch selector.

This notion extends naturally to sum types through the role of its constructors inj_l and inj_r, which play an analogous role to control qubits: patterns of the form $\text{inj}_l(v)$ and $\text{inj}_r(v)$ determine which clause of an isomorphism is selected to evaluation.

These preserved values and constructors tend to appear consistently across groups of clauses, forming what we call *matching clusters*. For example, note the preserving structure of the following Toffoli isomorphism, where matching clusters are highlighted in blue and green:

$$Toffoli : \mathbb{B} \otimes \mathbb{B} \otimes \mathbb{B} \leftrightarrow \mathbb{B} \otimes \mathbb{B} \otimes \mathbb{B}$$
$$:= \begin{pmatrix} \langle \text{ff}, \text{ff}, \text{a}\rangle \leftrightarrow \langle \text{ff}, \text{ff}, \text{a}\rangle \\ \langle \text{ff}, \text{tt}, \text{a}\rangle \leftrightarrow \langle \text{ff}, \text{tt}, \text{a}\rangle \\ \langle \text{tt}, \text{ff}, \text{a}\rangle \leftrightarrow \langle \text{tt}, \text{ff}, \text{a}\rangle \\ \langle \text{tt}, \text{tt}, \text{a}\rangle \leftrightarrow \begin{array}{l}\textbf{let } \text{na} = X\,\text{a} \\ \textbf{in } \langle \text{tt}, \text{tt}, \text{na}\rangle\end{array} \end{pmatrix}.$$

Let C be the set of indices corresponding to positions that are preserved within a matching cluster, given the semantic interpretation of the pattern $[\![p]\!]$, and let T denote the remaining target-values indices. Then

$$C \subseteq \{0, \ldots, q(a)-1\}, \qquad T = \{0, \ldots, q(a)-1\} \setminus C.$$

This induces a permutation of the computational basis, constructed so that all control qubits appear before the target qubits. Let $\Pi_{C,T}$ denote the corresponding permutation matrix. The reordered unitary is

$$\widetilde{[\![\omega]\!]} = \Pi_{C,T}\, [\![w]\!]\, \Pi_{C,T}^{\dagger}.$$

Under this reordering, the unitary acquires a quantum multiplexer interpretation. A quantum multiplexer is an operator of the form

$$\widetilde{[\![\omega]\!]} = \sum_{i=0}^{2^k-1} |i\rangle\langle i| \otimes V_i,$$

where the first k qubits act as control qubits and each V_i is a unitary operator acting on the remaining qubits [7]. This reordering procedure leads to a block decomposition. When the non-diagonal blocks are null, $\widetilde{[\![\omega]\!]}_{ij} = 0 \,\forall\, (i \neq j)$, the matrix represents a quantum multiplexer:

$$\widetilde{[\![\omega]\!]} = \begin{bmatrix} V_0 & 0 & \cdots & 0 \\ 0 & V_1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & V_{2^k-1} \end{bmatrix}.$$

Consequently, discovering a multiplexer structure involves finding a partition (C, T) for which the reordered matrix becomes block diagonal, which follows from matching cluster identification. Whenever such a partition exists, the indices in C are interpreted as control qubits and the indices in T as target qubits.

4.2.1 Rewriting. If $\omega : a \leftrightarrow b$ admits a multiplexer representation, then the corresponding Ket function has the form

$$\texttt{def } \omega(b_0, \ldots, b_{q(a)-1})\texttt{:}$$

where the qubits are partitioned into a control set C and a target set T.

The implementation of $[\![\omega]\!]$ is obtained by generating one controlled branch for each multiplexer block

$$|i\rangle\langle i| \otimes V_i.$$

Let the control and target qubit indexes be respectively

$$C = (c_0, \cdots, c_k) \text{ and, } T = (t_0, \cdots, c_m).$$

For a control configuration $i = d_0 \cdots d_k$, define

$$NC(i) = \{\, b_{c_r} \mid d_r = 0 \,\},$$

to be the set corresponding to negative controlling qubits. These are temporarily inverted using the Ket construct

```
with around(X, [NC(i)]):
```

The branch associated with V_i is then translated as

```
c_qbs = (b_{c_0}, ..., b_{c_k})
t_qbs = (b_{t_0}, ..., b_{t_m})
with around(X, [NC(i)]):
  ctrl(c_qbs, V_i)(t_qbs)
```

After the negative controls have been inverted, all control qubits are required to be in the state $|1\rangle$, allowing the branch to be implemented using Ket's standard `ctrl` operator. Repeating this procedure for every configuration i yields an implementation of the complete operation $[\![\omega]\!]$. Whenever NC(i) is empty, the `with around` construction can be suppressed, leaving only a controlled call. The controlled branch associated with a control configuration i can be represented as the circuit shown in Figure 1.

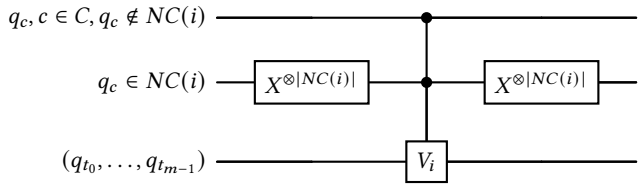

Figure 1: Circuit interpretation of a multiplexer branch associated with control configuration i.

The same procedure is then applied recursively to each block V_i. Whenever a block admits a multiplexer representation $\widetilde{V_i}$, a new Ket function is generated for its implementation and invoked within the corresponding controlled branch. This recursively constructs a hierarchy of controlled operations reflecting the structure of the original unitary matrix.

The recursion terminates when a block can no longer be expressed as a multiplexer. Such blocks correspond to leaf operations in the decomposition tree and must be synthesized using a general unitary decomposition technique. In particular, single-qubit operators can be translated using a ZYZ decomposition, while larger blocks may require more general synthesis procedures.

4.3 Decomposition

Several general-purpose decomposition techniques are available for this task, including the cosine-sine decomposition [10], and the KAK decomposition for two-qubit gates [9]. These approaches provide decompositions for arbitrary unitary matrices and are commonly used in quantum compilation toolchains.

In this work, however, we restrict ourselves to present the decomposition of single-qubit blocks, which arise naturally as leaf nodes in the recursive multiplexer structure. For leaf-blocks with higher qubit count, other decomposition methods should be applied.

Formally, the ZYZ decomposition states that any unitary operator $U \in U(2)$ can be written as

$$U = e^{i\alpha} R_Z(\beta) R_Y(\gamma) R_Z(\delta)$$

where $\alpha, \beta, \gamma, \delta \in \mathbb{R}$, and

$$R_Z(\theta) = \begin{bmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{bmatrix},\ R_Y(\theta) = \begin{bmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{bmatrix}.$$

This decomposition is particularly useful in our case because Ket already provides primitive support for rotation gates and explicit global phase annotations. Therefore, once the global phase is isolated, the resulting decomposition can be translated directly into an equivalent Ket definition. The extraction of the global phase is important because, although a global phase has no observable effect on an isolated qubit, it can become a relative phase when U is used within larger constructions, such as in controlled operations.

Given a unitary matrix, we first extract its global phase by computing

$$\alpha = \frac{1}{2} \arg(\det(U)).$$

Here, $\arg(z)$ denotes the principal argument of a complex number $z \in \mathbb{C}$, i.e., the phase angle $\theta \in (-\pi, \pi]$ such that $z = |z|e^{i\theta}$. The corresponding special unitary matrix is then obtained as

$$U_{SU(2)} = U e^{-i\alpha},$$

such that

$$\det(U_{SU(2)}) = 1.$$

Let

$$U_{SU(2)} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}.$$

The rotation angles can be extracted as follows.
First, compute

$$\gamma = 2\arccos(|a|).$$

If

$$\sin\left(\frac{\gamma}{2}\right) \approx 0,$$

then the decomposition is degenerate, and we choose

$$\beta = 0, \qquad \delta = \arg(d) - \arg(a).$$

Otherwise, the angles are given by

$$\beta = \arg(c) - \arg(a), \qquad \delta = -(\arg(c) + \arg(a)).$$

Therefore, the final decomposition is

$$U = e^{i\alpha} R_Z(\beta) R_Y(\gamma) R_Z(\delta).$$

By this approach, every single-qubit gate can be represented in Ket as a composition of elementary rotations together with an explicit global phase annotation:

```
@global_phase(α)
def U(q):
  RZ(δ, q)
  RY(γ, q)
  RZ(β, q)
```

Table 3 presents examples illustrating the proposed compilation procedure. Each entry shows the semantic interpretation of an SPM isomorphism alongside the corresponding Ket code generated through controlled structures.

5 Further Work

Although the compilation procedure presented in this work is sufficient for a subset of SPM, several extensions are required to support the full language and broader classes of quantum programs.

5.1 Multi-Qubit Non-Controlled Gates

The current framework assumes that all non-controlled leaf operations act on a single qubit. Under this restriction, every leaf block can be synthesized directly through the ZYZ decomposition, yielding a sequence of elementary rotation gates. While this is sufficient for demonstrating the overall compilation strategy, it excludes a large class of quantum operations that act on multiple qubits.

A natural extension is therefore to apply more general synthesis techniques once the multiplexer extraction procedure reaches a multi-qubit operator. Possible approaches include the KAK decomposition for two-qubit operators, the cosine-sine decomposition (CSD), and recursive unitary synthesis methods.

Just as syntactic analysis is used to identify control and target qubits, it can also guide the translation of multi-qubit leaf blocks. In particular, some classes of transformations admit a direct syntactic interpretation. Basis-reordering operations, such as SWAP gates, correspond to permutations of variables, while tensor-product operators can often be identified through independent transformations acting on disjoint subsets of variables.

The difficulty arises when these structural operations are combined with genuinely quantum transformations. Consider, for example, the operator $H \otimes SWAP$ defined as:

$$H \otimes SWAP : \mathbb{B} \otimes \mathbb{B} \otimes \mathbb{B} \leftrightarrow \mathbb{B} \otimes \mathbb{B} \otimes \mathbb{B}$$

$$:= \begin{pmatrix} \langle \text{ff}, \text{a}, \text{b}\rangle \leftrightarrow \frac{1}{\sqrt{2}}\langle \text{ff}, \text{b}, \text{a}\rangle + \frac{1}{\sqrt{2}}\langle \text{tt}, \text{b}, \text{a}\rangle \\ \langle \text{tt}, \text{a}, \text{b}\rangle \leftrightarrow \frac{1}{\sqrt{2}}\langle \text{ff}, \text{b}, \text{a}\rangle - \frac{1}{\sqrt{2}}\langle \text{tt}, \text{b}, \text{a}\rangle \end{pmatrix}.$$

Although the procedure presented in this work is capable of constructing the corresponding matrix $[\![H \otimes SWAP]\!]$, the resulting operator does not admit a direct multiplexer interpretation.

In this case, neither a syntactic analysis nor a matrix-based analysis is sufficient to decompose the resulting operator. From a syntactic perspective, the isomorphism clearly contains structural information corresponding to a variable permutation. From a semantic perspective, however, the operator also contains a quantum transformation acting on a distinct subsystem, and thus would require a decomposition method.

As a consequence, a complete translation procedure would likely require both analysis to join their forces: syntactic techniques to identify structural transformations such as permutations and tensor-product decompositions, and matrix-based techniques to synthesize the remaining quantum operators.

It is also important to note that not all permutation operators can be identified through syntactic analysis alone. For example, a programmer is not required to define a SWAP operation through mere variable permutations. The same operator may be specified by enumerating all basis values, as shown below.

$$SWAP : \mathbb{B} \otimes \mathbb{B} \leftrightarrow \mathbb{B} \otimes \mathbb{B} := \begin{pmatrix} \langle \text{ff}, \text{ff}\rangle \leftrightarrow \langle \text{ff}, \text{ff}\rangle \\ \langle \text{ff}, \text{tt}\rangle \leftrightarrow \langle \text{tt}, \text{ff}\rangle \\ \langle \text{tt}, \text{ff}\rangle \leftrightarrow \langle \text{ff}, \text{tt}\rangle \\ \langle \text{tt}, \text{tt}\rangle \leftrightarrow \langle \text{tt}, \text{tt}\rangle \end{pmatrix}.$$

Such a representation does not expose any syntactic structure that can be exploited by the compiler. Therefore, from a compiler's perspective, it is equivalent to an explicit matrix representation. Consequently, a fully general compiler

Table 3: Examples of SPM isomorphism compilation into Ket. Matching clusters are highlighted in blue and green, while purple indicates the identity extension introduced during space completion.

Iso (ω)	Type	Clauses	$[\![\omega]\!]$	Ket Code
CX	$\mathbb{B}\otimes\mathbb{B}\leftrightarrow\mathbb{B}\otimes\mathbb{B}$	$\langle \mathrm{ff}, \mathrm{a}\rangle \leftrightarrow \langle \mathrm{ff}, \mathrm{a}\rangle$ $\langle \mathrm{tt}, \mathrm{a}\rangle \leftrightarrow$ **let** na $= X$ a **in** $\langle \mathrm{tt}, \mathrm{na}\rangle$	$\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{bmatrix}$	`def cx(q0, q1):` `  c_set = q0` `  t_set = q1` `  ctrl(c_set,x)(t_set)` *@global_phase(pi/2)* `def x(q):` `  RZ(pi/2, q)` `  RY(pi, q)` `  RZ(-pi/2, q)`
$Toffoli$	$\mathbb{B}\otimes\mathbb{B}\otimes\mathbb{B}$ $\leftrightarrow$ $\mathbb{B}\otimes\mathbb{B}\otimes\mathbb{B}$	$\langle \mathrm{ff}, \mathrm{ff}, \mathrm{a}\rangle \leftrightarrow \langle \mathrm{ff}, \mathrm{ff}, \mathrm{a}\rangle$ $\langle \mathrm{ff}, \mathrm{tt}, \mathrm{a}\rangle \leftrightarrow \langle \mathrm{ff}, \mathrm{tt}, \mathrm{a}\rangle$ $\langle \mathrm{tt}, \mathrm{ff}, \mathrm{a}\rangle \leftrightarrow \langle \mathrm{tt}, \mathrm{ff}, \mathrm{a}\rangle$ $\langle \mathrm{tt}, \mathrm{tt}, \mathrm{a}\rangle \leftrightarrow$ **let** na $= X$ a **in** $\langle \mathrm{tt}, \mathrm{tt}, \mathrm{na}\rangle$	$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{bmatrix}$	`def toffoli(q0, q1, q2):` `  c_set = (q0, q1)` `  t_set = q2` `  ctrl(c_set, x)(t_set)`
$Toffoli+$	$(\mathbb{B}\oplus\mathbb{B})\oplus\mathbb{B}$ $\leftrightarrow$ $(\mathbb{B}\oplus\mathbb{B})\oplus\mathbb{B}$	$\mathrm{inj}_l(\mathrm{a}) \leftrightarrow \mathrm{inj}_l(\mathrm{a})$ $\mathrm{inj}_r(\mathrm{a}) \leftrightarrow$ **let** xa $= X$ a **in** $\mathrm{inj}_r(\mathrm{xa})$	$\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\end{bmatrix}$	`def toffoli+(q0, q1, q2):` `  c_set = (q0, q1)` `  t_set = q2` `  with around(X, [q1]):` `  ctrl(c_set, x)(t_set)`

for the framework must rely on a general-purpose quantum circuit synthesis routine, such as the Quantum Shannon Decomposition, which operates on the unitary matrix directly and has exponential cost in the number of qubits.

5.2 Infinite Types and Recursion

The approach presented in this work focuses exclusively on finite types and relies fundamentally on the ability to enumerate all basis values of a type.

However, this methodology does not extend to the full SPM language, which also includes types with infinite dimension and recursive isomorphisms. In particular, list types are interpreted over infinite-dimensional Hilbert spaces. Consequently, the semantic interpretation of an isomorphism can no longer be obtained through enumeration, making the matrix-construction procedure proposed in this work infeasible.

The difficulties introduced by these constructs become apparent even in simple examples. Consider the following recursively-defined controlled operation, where the control qubits come from a list.

$$CX^* : [\mathbb{B}]\otimes\mathbb{B}\leftrightarrow[\mathbb{B}]\otimes\mathbb{B} := \mu f.\begin{pmatrix} \langle [], \mathrm{a}\rangle \leftrightarrow & \textbf{let } \mathrm{xa} = X\,\mathrm{a} \textbf{ in } \langle [], \mathrm{xa}\rangle \\ \langle \mathrm{ff} : \mathrm{cs}, \mathrm{a}\rangle \leftrightarrow & \langle \mathrm{ff} : \mathrm{cs}, \mathrm{a}\rangle \\ \langle \mathrm{tt} : \mathrm{cs}, \mathrm{a}\rangle \leftrightarrow & \textbf{let } \langle \mathrm{cs}', \mathrm{a}'\rangle = f\langle \mathrm{cs}, \mathrm{a}\rangle \textbf{ in } \langle \mathrm{tt} : \mathrm{cs}', \mathrm{a}'\rangle \end{pmatrix}.$$

In this specific case, a *matching cluster* can be identified, inducing a simple rewritten Ket code, since the Ket operator `ctrl` already supports lists. However, a general translation method should be capable of identifying more complicated recursion schemes. This suggests that the resulting Ket program may itself be recursive, mirroring the structure of the source isomorphism, or alternatively be transformed into iterative constructs, such as for-loops. Developing semantics-preserving transformations of this kind remains an important direction for extending the compilation framework.

6 Conclusion

This work presented a procedure for translating quantum isomorphisms expressed in Symmetric Pattern Matching into executable Ket code. The translation was developed for a restricted fragment of finite types and non-controlled single-qubit quantum operations, while preserving the high-level structure of the original isomorphic specification.

The proposed compilation pipeline proceeds through a sequence of semantic and structural transformations. First, the semantics of an isomorphism is obtained by interpreting its clauses over the basis values of the corresponding type, yielding a unitary representation. Next, pattern-preservation structures are exploited to identify control information, allowing the unitary to be rewritten as a hierarchy of multiplexed controlled operations. Finally, single-qubit leaf blocks are synthesized through ZYZ decomposition, producing executable Ket definitions.

The resulting approach demonstrates how declarative reversible specifications based on symmetric pattern matching can be systematically compiled into quantum programs. In particular, the extraction of matching clusters provides a direct correspondence between high-level pattern matching and controlled quantum operations, bridging the gap between semantic descriptions and circuit-oriented implementations.

Several directions remain open for future work. The most immediate extension is the support for general multi-qubit leaf blocks. Additional work includes the treatment of recursive isomorphisms, and the investigation of more expressive data structures, such as lists, enabling the specification and compilation of higher-level quantum algorithms. Achieving these goals will likely require a compilation technique that combines syntactic analyses of isomorphism structure with semantic manipulations of their extracted operators.

Artifact Availability

The implementation of the compilation procedure described in this paper, together with the SPM language type checker and interpreter, is currently under development and will be made available in the project's public repository: https://github.com/Fleivio/spm.

References

[1] T. Altenkirch and J. Grattage. 2005. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 249–258. doi:10.1109/LICS.2005.1

[2] Pablo Arrighi and Gilles Dowek. 2017. Lineal: A linear-algebraic Lambda-calculus. *Logical Methods in Computer Science* Volume 13, Issue 1 (03 2017). doi:10.23638/LMCS-13(1:8)2017

[3] Ali Assaf, Alejandro Díaz-Caro, Simon Perdrix, Christine Tasson, and Benoît Valiron. 2010. Call-by-value, call-by-name and the vectorial behaviour of the algebraic λ-calculus. *Log. Methods Comput. Sci.* 10 (2010). https://api.semanticscholar.org/CorpusID:11306235

[4] Ville Bergholm, Juha J. Vartiainen, Mikko Möttönen, and Martti M. Salomaa. 2005. Quantum circuits with uniformly controlled one-qubit gates. *Phys. Rev. A* 71 (May 2005), 052330. Issue 5. doi:10.1103/PhysRevA.71.052330

[5] Evandro Chagas Ribeiro Da Rosa and Rafael De Santiago. 2021. Ket Quantum Programming. *J. Emerg. Technol. Comput. Syst.* 18, 1, Article 12 (Oct. 2021), 25 pages. doi:10.1145/3474224

[6] Liliane-Joy Dandy, Emmanuel Jeandel, and Vladimir Zamdzhiev. 2023. Type-safe Quantum Programming in Idris. In *Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 507–534. doi:10.1007/978-3-031-30044-8_19

[7] José Alex de Carvalho, Carlos Batista, Tiago de Veras, Israel Araujo, and Adenilton José da Silva. 2025. Quantum Multiplexer Simplification for State Preparation. *ACM Transactions on Quantum Computing* 6, 4, Article 26 (Aug. 2025), 12 pages. doi:10.1145/3748260

[8] Roshan P. James. 2014. Theseus : A High Level Language for Reversible Computing. https://api.semanticscholar.org/CorpusID:10756936

[9] Navin Khaneja and Steffen J. Glaser. 2001. Cartan decomposition of SU(2n) and control of spin systems. *Chemical Physics* 267, 1 (2001), 11–23. doi:10.1016/S0301-0104(01)00318-4

[10] Mikko Möttönen, Juha J. Vartiainen, Ville Bergholm, and Martti M. Salomaa. 2004. Quantum Circuits for General Multiqubit Gates. *Phys. Rev. Lett.* 93 (Sep 2004), 130502. Issue 13. doi:10.1103/PhysRevLett.93.130502

[11] Mikko Möttönen, Juha J. Vartiainen, Ville Bergholm, and Martti M. Salomaa. 2005. Transformation of quantum states using uniformly controlled rotations. *Quantum Info. Comput.* 5, 6 (Sept. 2005), 467–473.

[12] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press.

[13] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 846–858. doi:10.1145/3009837.3009894

[14] Amr Sabry, Benoît Valiron, and Juliana Kaizer Vizzotto. 2018. From Symmetric Pattern-Matching to Quantum Control. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 348–364.

[15] V.V. Shende, S.S. Bullock, and I.L. Markov. 2006. Synthesis of quantum-logic circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 25, 6 (2006), 1000–1010.

[16] Benoît Valiron. 2024. On Quantum Programming Languages. doi:10.48550/arXiv.2410.13337

[17] André van Tonder. 2004. A Lambda Calculus for Quantum Computation. *SIAM J. Comput.* 33, 5 (2004), 1109–1135. arXiv:https://doi.org/10.1137/S0097539703432165 doi:10.1137/S0097539703432165

[18] Andre van Tonder and Miquel Dorca. 2003. Quantum computation, categorical semantics and linear logic. (10 2003). arXiv:quant-ph/0312174

[19] LIONEL VAUX. 2009. The algebraic lambda calculus. *Mathematical Structures in Computer Science* 19, 5 (2009), 1029–1059. doi:10.1017/S0960129509990089