TRA105 - Building and Programming a Quantum Computer

Compiling Quantum Conditionals in a Functional Language

Nicklas Botö Fabian Forslund

1 Abstract

We present the functional quantum programming language Signe. The language features type inference, quantum conditionals, and compiles to multiple low-level representations. A compiler for Signe is presented with an implementation in Haskell. The compiler uses a Hindley-Milner type system, linearity checking, compilation to circuit and matrix data types, and a modern, user-friendly syntax in the functional style. The operational semantics are described in terms of the category of finite quantum computations **FQC**. Signe's semantics and the **FQC** category are based on the QML language by Altenkirch and Grattage [1, 5, 4].

2 Introduction

With the rise of quantum computers in the last few years, the search for quantum programming languages is very much alive. Previously defined programming languages mainly base their semantics on the circuit model, meaning that most of the language's primitives are qubits and gates, where the gates are applied to qubits in order to build a final circuit, much like classical circuits. Probably the most used circuit-describing language is IBM's OpenQASM[2] which is described as a quantum assembly language and is used to construct circuits working directly with wires and gates. Another example is Silq which, again, works with circuits but features higher-level constructs such as function definition, loops, and classical data types.

It is the opinion of the authors of this report that this model should make way for more higher-level languages. An ideal language of this kind would include concepts from conventional classical programming, such as higher levels of abstraction and loops, as well as novel constructs emerging from quantum computing; and may or may not include gate primitives. The benefits of such a language should be comparable to the benefits of higher level classical programming languages over classical assembly languages.

While these languages would possibly improve the expressiveness of quantum programs, it is uncertain at which scale of quantum programs they would surpass the abilities of circuit-describing languages. For instance, with the low number of qubits most quantum computers today can run, it is possible that a high-level programming language may be too big and inefficient in comparison to a circuit-describing language. The difficulty of designing such a language is also relevant. While providing high levels of abstraction is relatively easy, constructs such as quantum recursion and looping are problems that are not yet solved, instead having to make do with using classical bits to loop over, which has no clear representation in quantum computers.

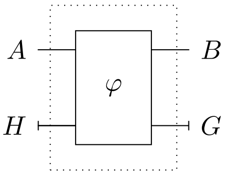
With this in mind, we present Signe which is a compilable language that features the use of quantum conditionals, purely quantum primitives, and a powerful type system.

3 Background

This chapter presents relevant background in the fields covered in this report. First we introduce an important concept that is used in the modelling of Signe programs, the category of finite quantum computations. Then we introduce type theory for the reader unfamiliar with the concept. Further reading on the discussed subjects can be found in appendix A.

3.1 Finite quantum computations

We describe the operational semantics of Signe in terms of finite quantum computation (**FQC**) morphisms. An **FQC** morphism from a to b is defined by the three-tuple (h, g, φ), where each of the terms $a, b, g, h \in \mathbb{N}$ generate Hilbert spaces $A, B, G, H \equiv \mathbb{C}^a$, $\mathbb{C}^b$, $\mathbb{C}^g$, $\mathbb{C}^h$, respectively; while the unitary operator $\varphi \in A \otimes H \multimapboth B \otimes G$ describes the operation on the input A and heap H producing the output B and garbage G. A morphism is written as $(h, g, \varphi) \in \mathbf{FQC}\, a\, b$ and can be visualised as in figure 1.



$(h, g, \varphi) \in \mathbf{FQC}\ a\ b$

Figure 1: Visualisation of an FQC. Note that the heap and garbage are affixed with vertical pipes to distinguish them from sequentially composable wires.

Given two morphisms $\alpha \in \mathbf{FQC}\ a\ b$ and $\beta \in \mathbf{FQC}\ b\ c$, the sequential composition is denoted $\beta \circ \alpha \in \mathbf{FQC}\ a\ c$. Similarly, given morphisms $\alpha \in \mathbf{FQC}\ a\ b$ and $\beta \in \mathbf{FQC}\ c\ d$, the parallel composition is denoted $\alpha \otimes \beta \in \mathbf{FQC}\ (a \otimes c)\ (b \otimes d)$.

The motivation for this model lies in its modularity and composability. It allows a recursive description of the building blocks of a morphism. This leads to a clear way to describe how programs in Signe are compiled to a morphism by composition of smaller, simpler morphisms. Chapter 4.4 uses these concepts to describe how language primitives are compiled to **FQC** morphisms by specific composition of the morphisms generated by their arguments.

The intuitiveness of the model is also apparent when viewing **FQC** morphisms as quantum circuits. A morphism $(h, g, \varphi) \in \mathbf{FQC}\, a\, b$ can intuitively be described as a quantum circuit with h auxiliary qubits, initialised to $|0\rangle^{\otimes h}$, g qubits to be measured and discarded, and a gate whose action is described by the unitary operation φ. This circuit expects some quantum state in $\mathbb{C}^a$ from a previous computation, and outputs a state in $\mathbb{C}^b$. The unitary operator φ can in fact be defined in terms of circuit-like operations (e.g. figure 2), and can trivially be viewed as a quantum circuit.

$$\varphi \ ::= \ \varphi \otimes \varphi \mid \varphi \odot \varphi \mid \varphi \oplus \varphi \mid \mathcal{P} \mid \mathcal{U}$$

Figure 2: Recursive definition of a unitary operator where $\otimes$ is parallel execution, $\odot$ is sequential execution, $\oplus$ is a conditional connective, $\mathcal{P}$ is a permutation pattern over wires, and $\mathcal{U}$ is a unitary matrix in $\mathbb{C}^{2\times2}$. Specifically, $a \oplus b$ represents that a is one-controlled and b is zero-controlled by the same wire, executed in series.

When specifying unitary operators, unitary matrices are often denoted by their standard gate name and permutation patterns by $\{p_0 \ldots p_n\} = \{0 \mapsto p_0, \ldots, n \mapsto p_n\}$. For example $X \equiv \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, $I \equiv \{0\}$, $CNOT \equiv X \oplus \{0\}$, and $SWAP \equiv \{1\ 0\}$.

3.1.1 Subcategories of FQC

Defined below are three subcategories of the **FQC** category. The purpose of these becomes apparent in the methodology chapter.

- **Canonical**: $(h, g, \varphi) \in \mathbf{FQC}\ a\ b$. This is the unrestricted category of morphisms.
- **Strict**: $(h, 0, \varphi) \in \mathbf{FQC}^{\circ}\ a\ b$. Strictness in an **FQC** morphism entails that the computation produces no garbage, and therefore no measurement.
- **Executable**: $(h^{>0}, g, \varphi) \in \mathbf{FQC}^{\Downarrow}\ 0\ b$. Executable morphisms are the ones that are able to be run on quantum hardware. The restriction here is that the morphism has no input, and that the heap is non-empty. That is, a circuit with a defined input state.

3.1.2 Type arity

When describing **FQC** morphisms, the inputs and outputs are often denoted as types of contexts of types. By doing this, we can discuss computations independent of values. To describe the sizes of these types and contexts, the operation $[\![\_]\!] : \mathit{Type} \cup \mathrm{Set}\ \mathit{Type} \rightharpoonup \mathbb{N}$ is defined as

$$\begin{aligned}
[\![\top]\!] &= 0 \\
[\![\mathcal{Q}]\!] &= 1 \\
[\![\sigma \otimes \tau]\!] &= [\![\sigma]\!] + [\![\tau]\!] \\
[\![\Gamma]\!] &= \sum_{x:\sigma \in \Gamma} [\![\sigma]\!] \\
[\![\Gamma \otimes \Delta]\!] &= \sum_{x:\sigma \in \Gamma \cup \Delta} [\![\sigma]\!].
\end{aligned}$$

Note that the function type $\sigma \to \tau$ will be interpreted as its own **FQC** morphism and is therefore not included in the domain of the type arity operator.

3.2 Type theory

Type theory is the study of type systems. A type system can describe a program which can derive a type for each term in a language. This type is then typically ascribed some meaning, for example $0 : \mathbb{N}$ says that the number 0 is a natural number. From this, one can build more complex typing rules, describing how to assign a type to larger expressions. For example, we can derive that $1 + 1 : \mathbb{N}$ by the typing rule

$$\frac{\Gamma \vdash x, y : \mathbb{N}}{\Gamma \vdash x + y : \mathbb{N}} \text{ ADD}$$

.

Such a rule is constructed by its premises above the line, conclusion below the line, and name to the right. The Γ refers to the context in which the derivation is made. This context consists of type assignments on the form $x : \sigma$, which states what type a certain variable is. The judgement $\Gamma \vdash M : \sigma$ states that in this context, one can derive that M is of type σ. Using a simple variable rule,

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \text{ VAR}$$

one can derive that $\{x : \sigma\} \vdash x : \sigma$ since $x : \sigma \in \{x : \sigma\}$.

Some type systems (such as Hindley-Milner used later in this paper) feature polymorphic types. These are types that are quantified by a universal quantifier (such as in first-order logic). An assignment on the form $f : \forall\alpha.\alpha \to \alpha$ states that for all types α, the f is a function from α to α; in polymorphic type systems this is true for the identity function $\lambda x.x$, for example.

Later in this paper we use the notation free(σ) to denote the set of variables that are not bound by a quantifier. This is defined by

$$\begin{aligned}
&\text{free} &&: \ \mathit{Type} \cup \text{Set } \mathit{Type} \to \text{Set } \mathit{Type}\\
&\text{free}(\alpha) &&= \{\alpha\}\\
&\text{free}(\tau) &&= \emptyset\\
&\text{free}(\forall\alpha.\sigma) &&= \text{free}(\sigma) \setminus \{\alpha\}\\
&\text{free}(C\ \tau_0 \ldots \tau_n) &&= \bigcup_{i=0}^{n} \text{free}(\tau_i)\\
&\text{free}(\Gamma) &&= \bigcup_{x:\sigma\in\Gamma} \text{free}(\sigma),
\end{aligned}$$

where C is any connective, α is a type variable, σ is a polymorphic type, τ is a monomorphic type, and Γ is a context.

4 Methodology

This chapter presents descriptions and specifications of the language Signe, as well as presentations of the operational semantics used by the code generator. The intention is to present the language in a formal, technical way, without heavy implementation details.

4.1 Architecture

A high-level overview of the compiler architecture is presented in figure 3. The compiler is divided into three stages, a front-end, generator, and back-end. The front-end handles the lexing, parsing, as well as static checks on the abstract syntax. Code generation and dynamic checks are handled by the generator, and the back-end handles translation into some target representation.

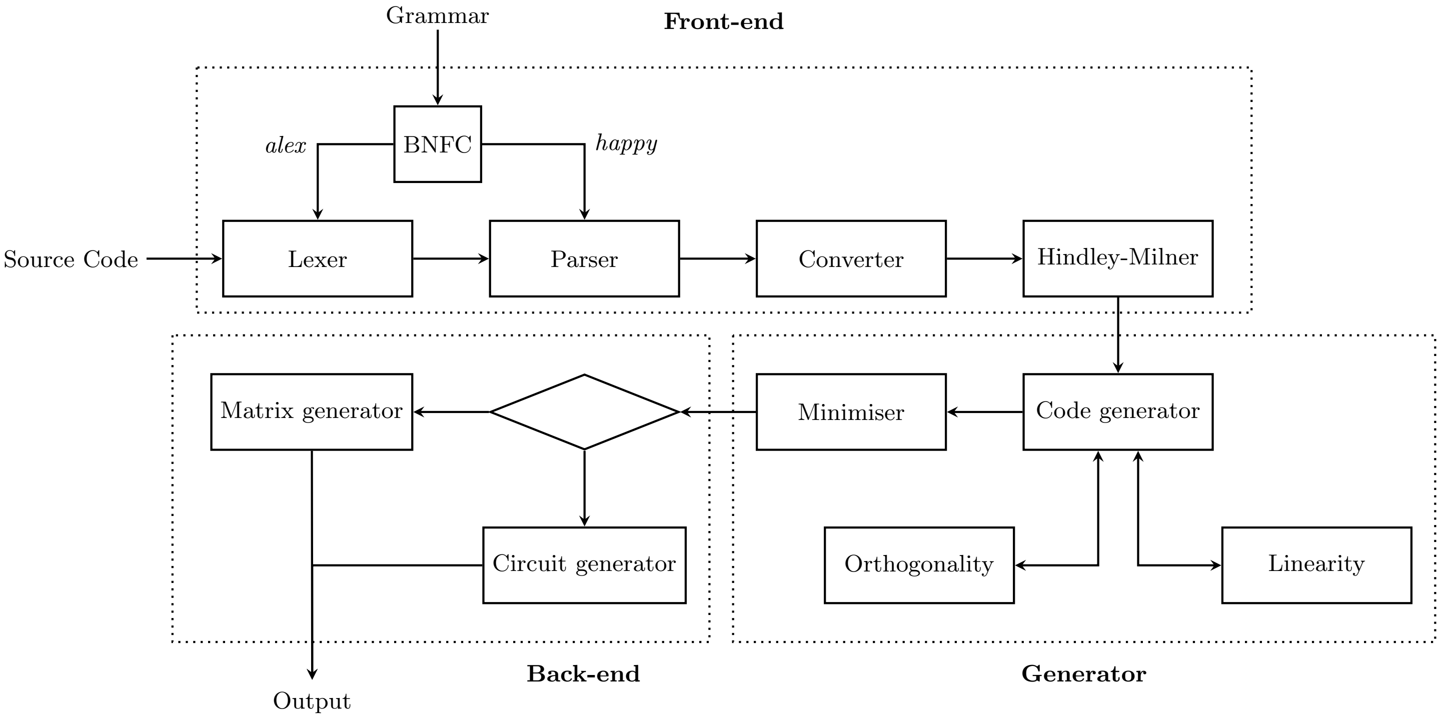


Figure 3: High-level overview of the compiler architecture

4.2 Syntax and parsing

The syntax of Signe is described in Backus-Naur form in figure 4. The types are defined using a subset of monomorphic types. Note that the function definitions of a program need not be typed.

$$
\begin{array}{lrl}
\mathit{Term} & M,\ N,\ P & ::= x \mid \lambda x.M \mid MN \mid \text{if}^{\circ} P \text{ then } M \text{ else } N \mid \text{if } P \text{ then } M \text{ else } N \\
 & & \mid M + N \mid M - N \mid \kappa * M \mid \langle M, N \rangle \mid |0\rangle \mid |1\rangle \mid \langle\rangle \mid \text{let } x = M \text{ in } N \\
 & & \\
\mathit{Mono} & \tau,\ \varphi & ::= \alpha \mid \tau \to \varphi \mid \tau \otimes \varphi \mid \mathcal{Q} \mid \top \\
 & & \\
\mathit{Type} & \sigma & ::= \tau \mid \forall \alpha.\sigma \\
 & & \\
\mathit{Program} & P & ::= f x : \sigma := M \mathbin{\text{↴}} P \mid \mathbf{eof}
\end{array}
$$

$$f, x, \alpha \in \mathit{String} \quad \kappa \in \mathbb{C} \quad \tau \in \mathit{Mono} \quad \sigma \in \mathit{Type} \quad M \in \mathit{Term}$$

Figure 4: The syntax of Signe

The lexer and the parser are generated by *alex* and *happy* using the BNFC tool. The abstract syntax tree (AST) is then converted to an internal simplified AST.

The syntax is inspired by Haskell and the ML family of languages.

```
MyGate x : qubit -> qubit
 := if° x
      then e^(12πi) * ~0
      else i * ~1
```

```
ControlledGate g c t
 : ∀ a . (a -> a) -> qubit -> a -> qubit * a
 := if° c
      then (~1, g t)
      else (~0,   t)
```

Figure 5: Signe syntax examples

4.3 Type checking

The compiler uses two distinct type systems used in separate stages of compilation. The first type system as described in figure 6 is checked statically directly after parsing and conversion. It is the usual description of a Hindley-Milner type system [3] extended with the atoms and connectives from our language, notably also the addition of introduction and elimination of a product type ($\forall\otimes_i$, $\forall\otimes_e$). The usual rules for specialisation and generalisation are described in (GEN) and (INST). The definition of the type order is defined by the $\sqsubseteq$ relation in the (SPEC) rule. The statement $\sigma' \sqsubseteq \sigma$ would mean that σ' is a more general type than σ in the sense that a substitution can be applied to the quantified variables in σ' to yield σ. Another notational convention of note is that σ always describes a polymorphic type, while τ is a monomorphic one (this notation is only used in figure 6, however).

$$\frac{}{\Gamma \vdash |0\rangle : \mathcal{Q}}\ \forall|0\rangle \qquad \frac{}{\Gamma \vdash |1\rangle : \mathcal{Q}}\ \forall|1\rangle \qquad \frac{}{\Gamma \vdash \langle\rangle : \top}\ \forall\top \qquad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}\ \forall\ \textsc{var}$$

$$\frac{\Gamma \vdash P : \mathcal{Q} \qquad \Gamma \vdash M, N : \tau}{\Gamma \vdash \text{if } P \text{ then } M \text{ else } N : \tau}\ \forall\ \textsc{if} \qquad \frac{\Gamma \vdash P : \mathcal{Q} \qquad \Gamma \vdash M, N : \tau}{\Gamma \vdash \text{if}^\circ P \text{ then } M \text{ else } N : \tau}\ \forall\ \textsc{if}^\circ \qquad \frac{\Gamma \vdash M, N : \tau}{\Gamma \vdash M + N : \tau}\ \forall\textsc{sup}_+$$

$$\frac{\Gamma \vdash M, N : \tau}{\Gamma \vdash M - N : \tau}\ \forall\textsc{sup}_- \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \kappa * M : \tau}\ \forall\kappa \qquad \frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \tau'}{\Gamma \vdash \langle M, N\rangle : \tau \otimes \tau'}\ \forall\otimes_i$$

$$\frac{\Gamma \vdash M : \tau \to \tau' \qquad \Gamma \vdash N : \tau}{\Gamma \vdash MN : \tau'}\ \forall\textsc{app} \qquad \frac{\Gamma, x : \tau \vdash M : \tau'}{\Gamma \vdash \lambda x.M : \tau \to \tau'}\ \forall\lambda \qquad \frac{\Gamma \vdash M : \sigma \qquad \Gamma, x : \sigma \vdash N : \tau}{\Gamma \vdash \text{let } x = M \text{ in } N : \tau}\ \forall\textsc{let}$$

$$\frac{\Gamma \vdash M : \sigma \otimes \sigma' \qquad \Gamma, x : \sigma, y : \sigma' \vdash N : \tau}{\Gamma \vdash \text{let } \langle x, y\rangle = M \text{ in } N : \tau}\ \forall\otimes_e \qquad \frac{\Gamma \vdash M : \sigma' \qquad \sigma' \sqsubseteq \sigma}{\Gamma \vdash M : \sigma}\ \textsc{inst}$$

$$\frac{\Gamma \vdash M : \sigma \qquad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash M : \forall\alpha.\sigma}\ \textsc{gen} \qquad \frac{\tau' = \{\alpha_i \mapsto \tau_i\}\tau \qquad \beta_i \notin \text{free}(\forall\alpha_1 \ldots \alpha_n.\tau)}{\forall\alpha_0 \ldots \forall\alpha_n.\tau \sqsubseteq \forall\beta_0 \ldots \forall\beta_m.\tau'}\ \textsc{spec}$$

Figure 6: Hindley-Milner type system

Note that some rules are repeated in the different typing rules (figure 6, 8), ($\forall\lambda$) and ($\multimap \lambda$) for example are essentially identical. The derivation of types is in fact handled only by the Hindley-Milner system. During code generation the linear type checker can then make useful assumptions about the program (since it passed the first type checker), and is left solely to enforce the linearity and strictness constraints posed by the linear type system. The linear type checker is also responsible for dynamic checks of orthogonality ($\multimap$ IF$^\circ$, $\multimap$ SUP$_+$, $\multimap$ SUP$_+$) and scalars ($\multimap$ SUP$^0_+$, $\multimap$ SUP$^0_-$, $\multimap$ SUP$_+$, $\multimap$ SUP$_-$, $\multimap \kappa$).

$$\frac{\dfrac{\dfrac{\dfrac{x : \alpha \in \{x : \alpha\}}{x : \alpha \vdash x : \alpha}\ \forall\text{VAR} \qquad \dfrac{x : \alpha \in \{x : \alpha\}}{x : \alpha \vdash x : \alpha}\ \forall\text{VAR}}{x : \alpha \vdash \langle x, x\rangle : \alpha \otimes \alpha}\ \forall\otimes_i}{\vdash \lambda x.\langle x, x\rangle : \alpha \to \alpha \otimes \alpha}\ \forall\lambda \qquad \alpha \notin \text{free}(\emptyset)}{\vdash \lambda x.\langle x, x\rangle : \forall\alpha.\alpha \to \alpha \otimes \alpha}\ \forall\text{GEN}$$

Figure 7: Derivation in Hindley-Milner system using contraction

The rules in figure 8 essentially describe how contexts need to be split and discarded to keep track of expression linearity. A derivation that can be done strictly, that is without discarding anything, is written $\Gamma \vdash^{\circ} M : \sigma$. For any strictness a and b, $a \sqcap b$ is the combined strictness of the two, only being strict when both a and b are. The derivation made in figure 7 in the Hindley-Milner system cannot be made in the linear type system, since the context $\{x : \sigma\}$ cannot be split into Γ and Δ in the $(\multimap \otimes_i)$ rule. This results in an internal call to the contraction function $\delta x = \langle x, x \rangle$ to create the separate contexts.

Similarly, the derivation $\overline{x : \sigma \vdash |1\rangle : \mathcal{Q}}^{\forall|1\rangle}$ uses weakening to discard the variable x. In the linear type system, one has to use the $(\multimap |1\rangle)$ rule, which instead handles explicit weakening with the function $\pi x = \langle\rangle$. This unit can then be discarded strictly using the $(\multimap \top_e^{\circ})$ rule. The circuit for this is also described in section 4.4. Discarding of the entire context Γ in a derivation for M is denoted by $M^{\mathrm{dom}\ \Gamma}$, used in rules ($\multimap \top_i$, $\multimap |0\rangle$, $\multimap |1\rangle$, $\multimap$VAR). This can never be done strictly. The circuit handling this is described in section 4.4.

$$\frac{}{\vdash^{\circ} \langle\rangle : \top} \multimap \top_i^{\circ} \qquad \frac{}{\Gamma \vdash \langle\rangle^{\mathrm{dom}\ \Gamma} : \top} \multimap \top_i \qquad \frac{\Gamma \vdash^{\circ} M : \sigma}{\Gamma, x : \top \vdash^{\circ} M : \sigma} \multimap \top_e^{\circ} \qquad \frac{}{\vdash^{\circ} |0\rangle : \mathcal{Q}} \multimap |0\rangle^{\circ}$$

$$\frac{}{\Gamma \vdash |0\rangle^{\mathrm{dom}\ \Gamma} : \mathcal{Q}} \multimap |0\rangle \qquad \frac{}{\vdash^{\circ} |1\rangle : \mathcal{Q}} \multimap |1\rangle^{\circ} \qquad \frac{}{\Gamma \vdash |1\rangle^{\mathrm{dom}\ \Gamma} : \mathcal{Q}} \multimap |1\rangle \qquad \frac{}{x : \sigma \vdash^{\circ} x : \sigma} \multimap\mathrm{VAR}^{\circ}$$

$$\frac{}{\Gamma, x : \sigma \vdash x^{\mathrm{dom}\ \Gamma} : \sigma} \multimap\mathrm{VAR} \qquad \frac{\Gamma \vdash^{a} M : \sigma \qquad \Delta, x : \sigma \vdash^{b} N}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \text{let } x = M \text{ in } N : \tau} \multimap\mathrm{LET} \qquad \frac{\Gamma \vdash M : \sigma \qquad \Delta \vdash N : \tau}{\Gamma \otimes \Delta \vdash \langle M, N \rangle : \sigma \otimes \tau} \multimap \otimes_i$$

$$\frac{\Gamma \vdash^{a} M : \sigma \otimes \tau \qquad \Delta, x : \sigma, y : \tau \vdash^{b} N : \psi}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \text{let } \langle x, y \rangle = M \text{ in } N : \psi} \multimap \otimes_e \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Delta \vdash N : \sigma}{\Gamma \otimes \Delta \vdash MN : \tau} \multimap\mathrm{APP}$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \multimap \lambda \qquad \frac{\Gamma \vdash P : \mathcal{Q} \qquad \Delta \vdash M, N : \sigma}{\Gamma \otimes \Delta \vdash \text{if } P \text{ then } M \text{ else } N : \sigma} \multimap\mathrm{IF}$$

$$\frac{\Gamma \vdash^{\circ} P : \mathcal{Q} \qquad \Delta \vdash^{\circ} M, N : \sigma \qquad M \perp N}{\Gamma \otimes \Delta \vdash^{\circ} \text{if}^{\circ} P \text{ then } M \text{ else } N : \sigma} \multimap\mathrm{IF}^{\circ} \qquad \frac{|\kappa|^2 + |\lambda|^2 = 1}{\Gamma \vdash (\kappa * |0\rangle + \lambda * |1\rangle)^{\mathrm{dom}\ \Gamma} : \mathcal{Q}} \multimap\mathrm{SUP}_{+}^{0}$$

$$\frac{|\kappa|^2 + |\lambda|^2 = 1}{\Gamma \vdash (\kappa * |0\rangle - \lambda * |1\rangle)^{\mathrm{dom}\ \Gamma} : \mathcal{Q}} \multimap\mathrm{SUP}_{-}^{0} \qquad \frac{\Gamma \vdash^{\circ} M, N : \sigma \qquad |\kappa|^2 + |\lambda|^2 = 1 \qquad M \perp N}{\Gamma \vdash \kappa * M + \lambda * N : \sigma} \multimap\mathrm{SUP}_{+}$$

$$\frac{\Gamma \vdash^{\circ} M, N : \sigma \qquad |\kappa|^2 + |\lambda|^2 = 1 \qquad M \perp N}{\Gamma \vdash \kappa * M - \lambda * N : \sigma} \multimap\mathrm{SUP}_{-} \qquad \frac{\Gamma \vdash M : \sigma \qquad |\kappa| = 1}{\Gamma \vdash \kappa * M : \sigma} \multimap \kappa \qquad \frac{\Gamma \vdash^{\circ} M : \sigma}{\Gamma \vdash M : \sigma} \multimap \Gamma^{\circ}$$

Figure 8: Linear type system

The orthogonality judgement is defined to follow the orthogonality of the state vectors the expressions represent. The rules for orthogonality are presented in figure 9.

$$\frac{}{|0\rangle \perp |1\rangle} \perp |0\rangle \qquad \frac{}{|1\rangle \perp |0\rangle} \perp |1\rangle \qquad \frac{M \perp N}{\langle M, V \rangle \perp \langle N, W \rangle} \perp \otimes_1 \qquad \frac{M \perp N}{\langle V, M \rangle \perp \langle W, M \rangle} \perp \otimes_2$$

$$\frac{M \perp N \qquad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\lambda_0 * M + \lambda_1 * N \perp \kappa_0 * M + \kappa_1 * N} \perp\mathrm{SUP} \qquad \frac{M \perp N}{\lambda * M \perp \kappa * N} \perp\mathrm{MUL}$$

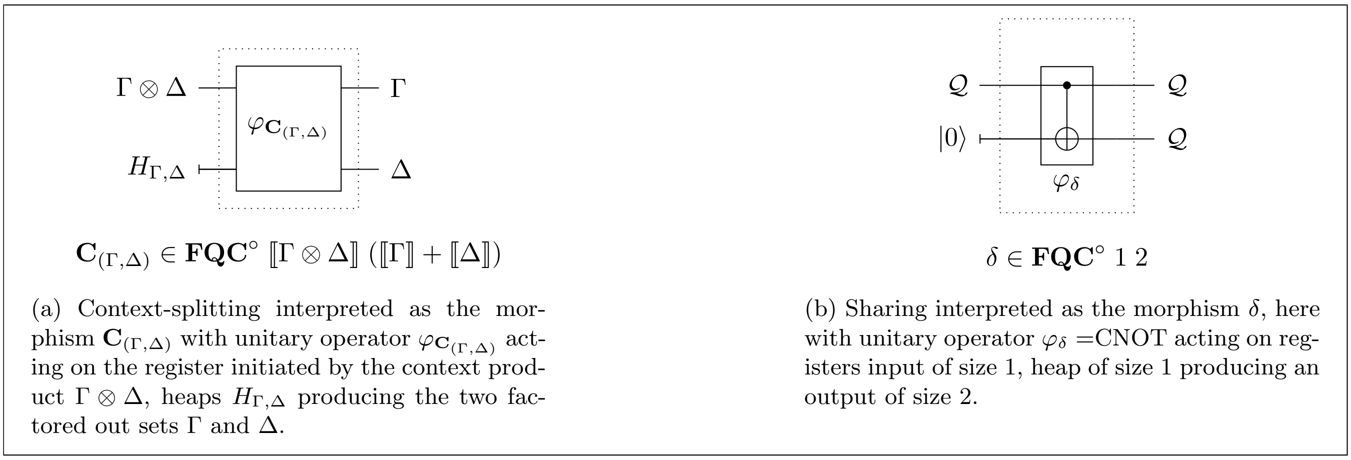
Figure 9: Orthogonality rules

4.4 Code generation

This section illustrates the operations that describe the compilation of Signe programs. These operations mainly consist of typing rules and **FQC** morphisms.

4.4.1 Contraction

While the linear type system of Signe does not explicitly allow contraction, a form of contraction that is instead handled implicitly by the compiler is included. The contraction operation is modelled contextually as an $\mathbf{FQC}^{\circ}$ morphism $\mathbf{C}_{(\Gamma,\Delta)}$ acting on contexts Γ and Δ and is constructed as shown in figure 10a. If identical variables appear in the both Γ and Δ, the variables are shared so that each of the two split contexts receives an instance of the shared variable. On terms, contraction is interpreted as a sharing operation acting on qubit registers $\mathcal{Q}$ and is facilitated by a CNOT gate. A visualisation of the sharing operation is shown in figure 10b.



$\mathbf{C}_{(\Gamma,\Delta)} \in \mathbf{FQC}^{\circ}\ [\![\Gamma \otimes \Delta]\!]\ ([\![\Gamma]\!] + [\![\Delta]\!])$

(a) Context-splitting interpreted as the morphism $\mathbf{C}_{(\Gamma,\Delta)}$ with unitary operator $\varphi_{\mathbf{C}_{(\Gamma,\Delta)}}$ acting on the register initiated by the context product $\Gamma \otimes \Delta$, heaps $H_{\Gamma,\Delta}$ producing the two factored out sets Γ and Δ.

$\delta \in \mathbf{FQC}^{\circ}\ 1\ 2$

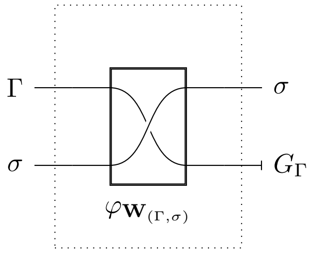
(b) Sharing interpreted as the morphism δ, here with unitary operator φ_{δ} =CNOT acting on registers input of size 1, heap of size 1 producing an output of size 2.

Figure 10: Contraction FQC morphisms

4.4.2 Weakening

Similar to how contraction is enabled in the strict linear type system, discarding of variables in Signe is handled explicitly by the compiler as a weakening operation, modelled by the **FQC** morphism $\mathbf{W}_{(\Gamma,\sigma)}$. In the semantics of Signe, whenever any variable or context is discarded the qubits they point to are automatically measured before being discarded. Measurements on the discarded qubits can affect the meaning of the rest of the program, so care has to be taken whenever weakening is used.

The weakening **FQC** morphism is visualised below in Figure 11.

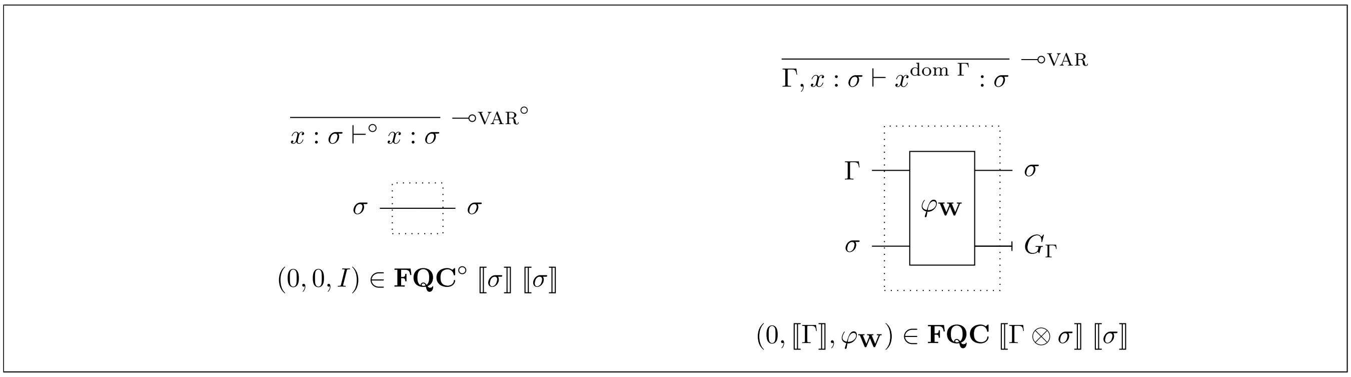


$\mathbf{W}_{(\Gamma,\sigma)} \in \mathbf{FQC}\ ([\![\Gamma]\!] + [\![\sigma]\!])\ [\![\sigma]\!]$

Figure 11: Weakening FQC morphism

4.4.3 Atoms and variables

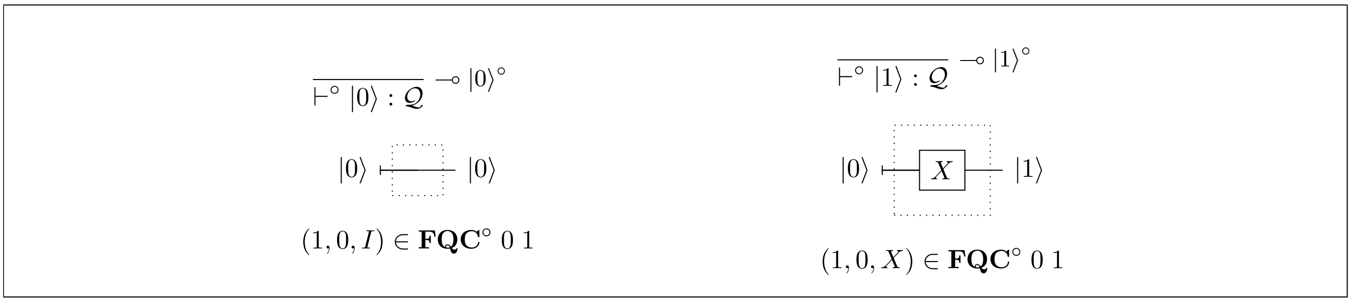
All atoms have separate rules in strict and non-strict versions ($\multimap \top_i^\circ$, $\multimap \top_i$,$\multimap |0\rangle^\circ$,$\multimap |0\rangle$,$\multimap |1\rangle^\circ$,$\multimap |1\rangle$,$\multimap \mathrm{VAR}^\circ$,$\multimap \mathrm{VAR}$). Both forms are presented for $\top$ and VAR, and the strict forms for $|0\rangle$ and $|1\rangle$. The remaining non-strict variants follow the same definition as for VAR, by using the weakening morphism $\mathbf{W}_{(\Gamma,|0\rangle)}$.



$$\frac{}{x : \sigma \vdash^\circ x : \sigma}\ \multimap\mathrm{VAR}^\circ \qquad \frac{}{\Gamma, x : \sigma \vdash x^{\mathrm{dom}\ \Gamma} : \sigma}\ \multimap\mathrm{VAR}$$

$$(0, 0, I) \in \mathbf{FQC}^\circ\ [\![\sigma]\!]\ [\![\sigma]\!] \qquad (0, [\![\Gamma]\!], \varphi_{\mathbf{W}}) \in \mathbf{FQC}\ [\![\Gamma \otimes \sigma]\!]\ [\![\sigma]\!]$$

Figure 12: Operational semantics of variables

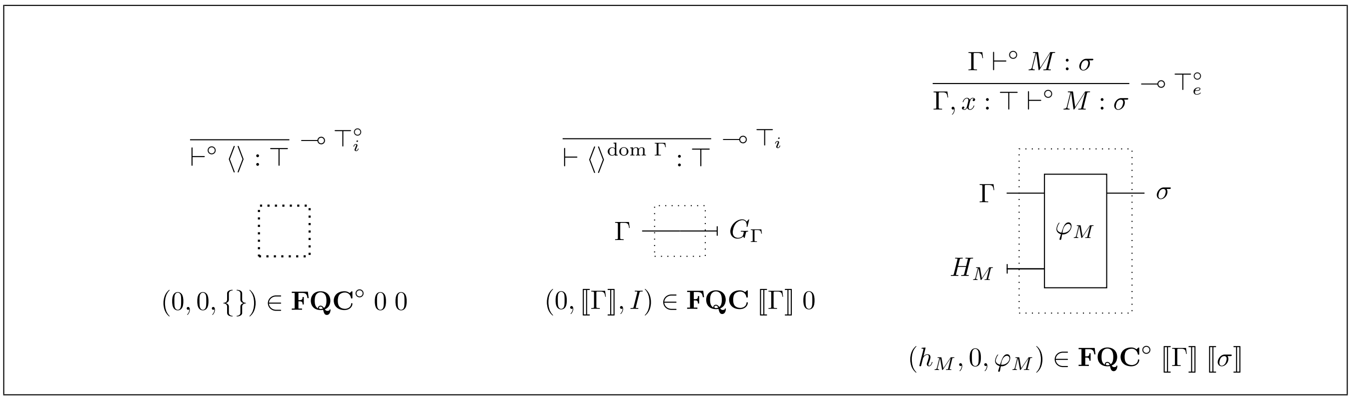


$$\frac{}{\vdash^\circ |0\rangle : \mathcal{Q}}\ \multimap |0\rangle^\circ \qquad \frac{}{\vdash^\circ |1\rangle : \mathcal{Q}}\ \multimap |1\rangle^\circ$$

$$(1, 0, I) \in \mathbf{FQC}^\circ\ 0\ 1 \qquad (1, 0, X) \in \mathbf{FQC}^\circ\ 0\ 1$$

Figure 13: Operational semantics of qubits

The unit atom $\langle\rangle$ generates the empty **FQC** morphism $\bullet = (0, 0, \{\}) \in \mathbf{FQC}^\circ\ 0\ 0$. Note also the elimination rule for unit atoms, which states that unlike any other term, unit atoms can be discarded strictly as they carry no computational meaning.



$$\frac{}{\vdash^\circ \langle\rangle : \top}\ \multimap \top_i^\circ \qquad \frac{}{\vdash \langle\rangle^{\mathrm{dom}\ \Gamma} : \top}\ \multimap \top_i \qquad \frac{\Gamma \vdash^\circ M : \sigma}{\Gamma, x : \top \vdash^\circ M : \sigma}\ \multimap \top_e^\circ$$

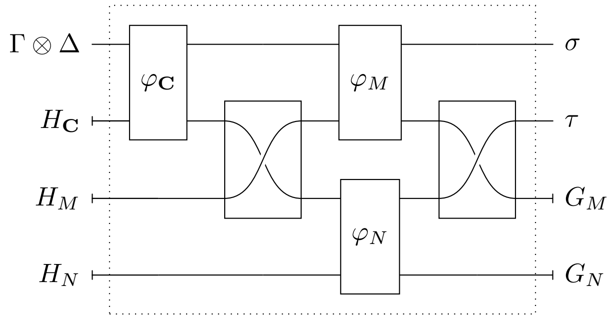
$$(0, 0, \{\}) \in \mathbf{FQC}^\circ\ 0\ 0 \qquad (0, [\![\Gamma]\!], I) \in \mathbf{FQC}\ [\![\Gamma]\!]\ 0 \qquad (h_M, 0, \varphi_M) \in \mathbf{FQC}^\circ\ [\![\Gamma]\!]\ [\![\sigma]\!]$$

Figure 14: Operational semantics of unit atoms

4.4.4 Tuples

Tuples in Signe are represented as a product of terms $\langle M, N\rangle$ and is formalised in the $\multimap \otimes_i$ rule. The circuit describing the tuple introduction **FQC** morphism begins in the same manner as that for let expressions, with a contraction operation taking place through $\varphi_{\mathbf{C}}$ on inputs $\Gamma \otimes \Delta$ if needed; and heaps $H_{C,M,N}$. The output of φ_C is sent to the corresponding unitary operators $\varphi_{M,N}$ that generate the terms making up the tuple. The outputs of these correspond to the output of the morphism and are denoted σ, τ. The overall action of the tuple rule is to split variables in a shared context register into a pair of context registers. The $\multimap \otimes_i$ rule and corresponding **FQC** morphism are shown below.

$$\frac{\Gamma \vdash M : \sigma \qquad \Delta \vdash N : \tau}{\Gamma \otimes \Delta \vdash \langle M, N\rangle : \sigma \otimes \tau} \multimap \otimes_i$$

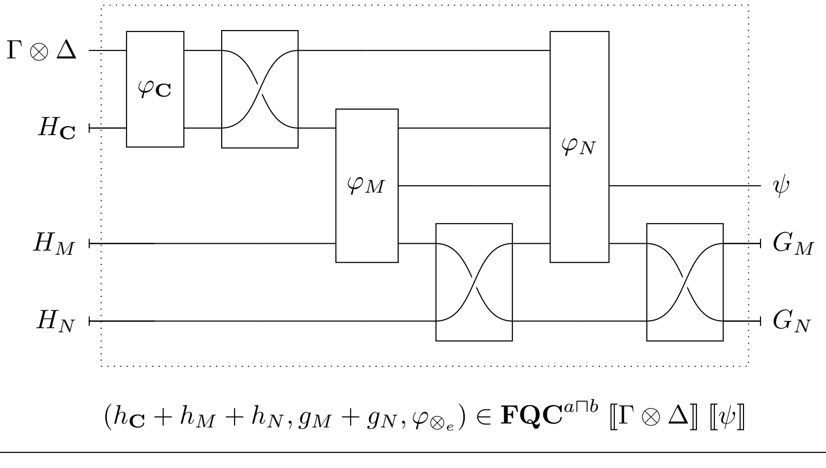


$$(h_{\mathbf{C}} + h_M + h_N, g_M + g_N, \varphi_{\otimes_i}) \in \mathbf{FQC}\ [\![\Gamma \otimes \Delta]\!]\ ([\![\sigma]\!] + [\![\tau]\!])$$

Figure 15: Operational semantics of tuple introduction

Tuple elimination is enabled through pattern-matching in let expressions, where contents of a variable denoting a tuple are embedded in a tuple term. The corresponding typing rule ($\multimap \otimes_e$) and **FQC** morphism are shown below in figure 16.

$$\frac{\Gamma \vdash^a M : \sigma \otimes \tau \qquad \Delta, x : \sigma, y : \tau \vdash^b N : \psi}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \text{let } \langle x, y\rangle = M \text{ in } N : \psi} \multimap \otimes_e$$



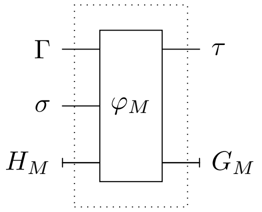
$$(h_{\mathbf{C}} + h_M + h_N, g_M + g_N, \varphi_{\otimes_e}) \in \mathbf{FQC}^{a \sqcap b}\ [\![\Gamma \otimes \Delta]\!]\ [\![\psi]\!]$$

Figure 16: Operational semantics of tuple elimination

4.4.5 Abstraction and application

Interpretations for λ-abstraction and application are added to improve the extensibility of the language. The morphism for λ-abstraction simply interprets the body and adds an expected argument as an additional input.

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau} \multimap \lambda$$

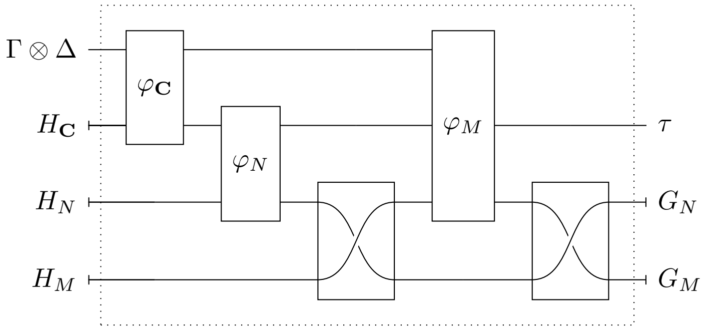


$$(h_M, g_M, \varphi_M) \in \mathbf{FQC}\ (\llbracket\Gamma\rrbracket + \llbracket\sigma\rrbracket)\ \llbracket\tau\rrbracket$$

Figure 17: Operational semantics of lambda abstraction

The interpretation of application is similar to that of the let expression. The difference here is the order of the operators for the function M and its argument N, as well as the context management. The order of evaluation is, however, the same to preserve the call-by-name semantics (i.e. M is evaluated before N). Whether this has any effect on laziness is debatable.

$$\frac{\Gamma \vdash^{a} M : \sigma \to \tau \qquad \Delta \vdash^{b} N : \sigma}{\Gamma \otimes \Delta \vdash^{a \sqcap b} MN : \tau} \multimap\text{APP}$$



$$(h_{\mathbf{C}} + h_M + h_N, g_M + g_N, \varphi_{\text{LET}}) \in \mathbf{FQC}^{a \sqcap b}\ \llbracket\Gamma \otimes \Delta\rrbracket\ \llbracket\tau\rrbracket$$

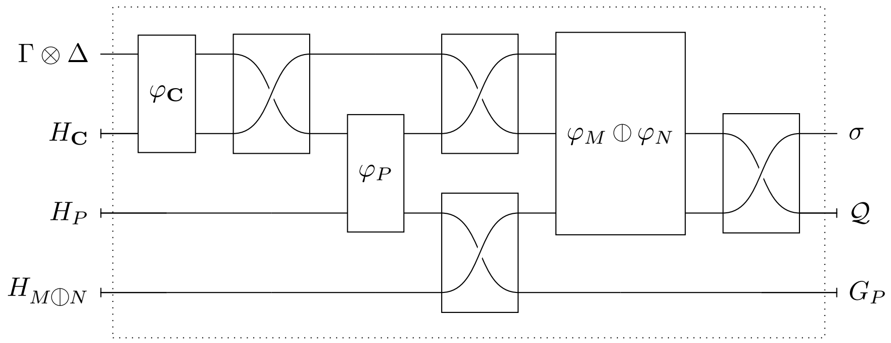
Figure 18: Operational semantics of application

4.4.6 Conditionals

The interpretation of conditionals are split into their respective rules. The additional restrictions, namely strictness and orthogonality, added to the quantum conditional ($\multimap$IF°) results in more powerful programming capabilities as it makes it possible to modify qubits without measurement.

The interpretation of a classical conditional ($\multimap$IF) is defined in figure 19. Contexts are split to the conditional qubit P and the branches M and N respectively. The resulting output from φ_P is then sent to the branch interpretation. Finally, the conditional qubit is discarded.

$$\frac{\Gamma \vdash P : \mathcal{Q} \qquad \Delta \vdash M, N : \sigma}{\Gamma \otimes \Delta \vdash \text{if } P \text{ then } M \text{ else } N : \sigma} \multimap\text{IF}$$

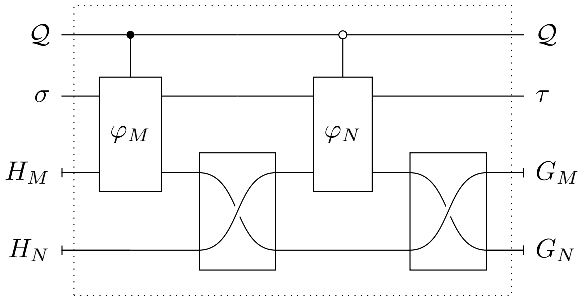


$$(h_{\mathbf{C}} + h_P + h_{M \obar N}, 1 + g_P, \varphi_{\text{IF}}) \in \mathbf{FQC}\ [\![\Gamma \otimes \Delta]\!]\ [\![\sigma]\!]$$

Figure 19: Operational semantics of classical conditionals

Where the conditional circuit $\varphi_M \obar \varphi_N$ is defined as

$\varphi_M \obar \varphi_N \quad \equiv$

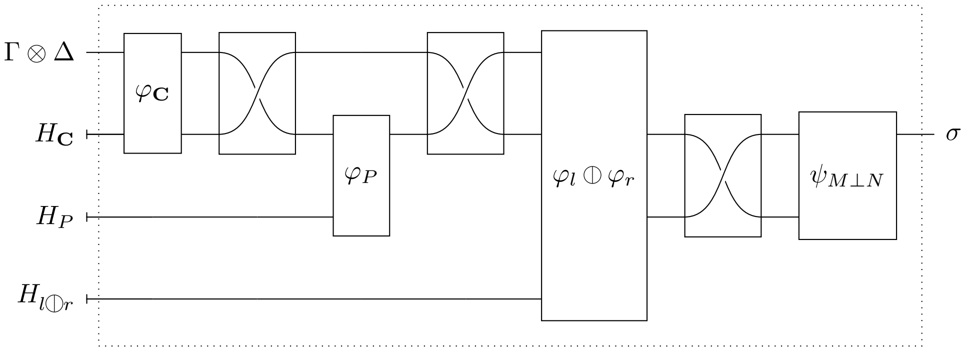


.

$$(h_M + h_N, g_M + g_N, \varphi_{M \obar N}) \in \mathbf{FQC}\ (1 + [\![\sigma]\!])\ (1 + [\![\tau]\!])$$

The interpretation of the quantum conditional is defined similarly to the classical one. One notable difference is the lack of garbage, as required by the strictness restriction. The final unitary operation $M \perp N$ produces the operations φ_l and φ_r used in the conditional circuit, the description of this operation is described in section 4.5.

$$\frac{\Gamma \vdash^{\circ} P : \mathcal{Q} \qquad \Delta \vdash^{\circ} M, N : \sigma \qquad M \perp N}{\Gamma \otimes \Delta \vdash^{\circ} \text{if}^{\circ} P \text{ then } M \text{ else } N : \sigma} \multimap \text{IF}^{\circ}$$



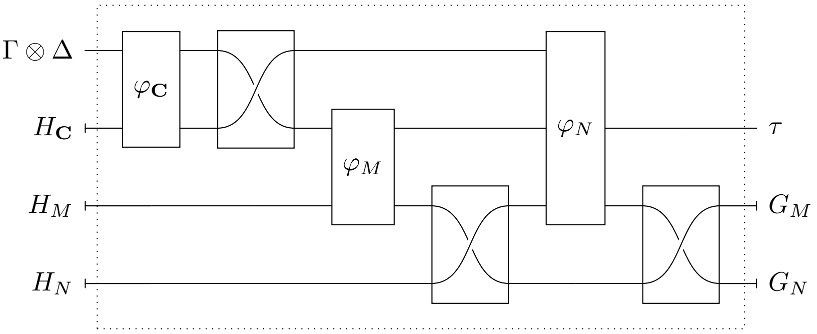
$$(h_{\mathbf{C}} + h_P + h_{l \obar r}, 0, \varphi_{\text{IF}^{\circ}}) \in \mathbf{FQC}^{\circ}\ [\![\Gamma \otimes \Delta]\!]\ [\![\sigma]\!]$$

Figure 20: Operational semantics of quantum conditionals

4.4.7 Let expressions

The rule for let expressions ($\multimap$ LET) is introduced below on the top of figure 21, together with its **FQC** morphism and corresponding circuit. The circuit may be interpreted as the following: The morphism takes in a pair of contexts $\Gamma \otimes \Delta$ and heaps $H_{C,M,N}$ corresponding to the terms of the let expression. If contraction is needed, $\varphi_{\mathbf{C}}$ acts on the context pair. The operations corresponding to the terms of the let expression are then generated by the $\varphi_{M,N}$ operators which finally produces an output τ and if non-strict may produce garbage in registers $G_{M,N}$.

$$\frac{\Gamma \vdash^{a} M : \sigma \qquad \Delta, x : \sigma \vdash^{b} N}{\Gamma \otimes \Delta \vdash^{a \sqcap b} \text{let } x = M \text{ in } N : \tau} \multimap \text{LET}$$



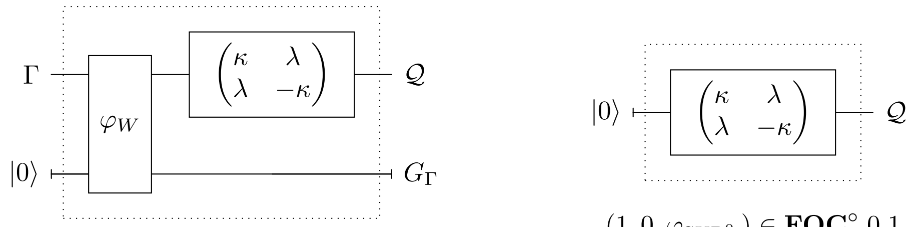
$$(h_{\mathbf{C}} + h_M + h_N, g_M + g_N, \varphi_{\text{LET}}) \in \mathbf{FQC}^{a \sqcap b}\ [\![\Gamma \otimes \Delta]\!]\ [\![\tau]\!]$$

Figure 21: Operational semantics of let expressions

4.4.8 Superpositions and rotations

The rules for superpositions are interpreted as desugaring in the general case, and as a **FQC** morphism in the base qubit case. A superposition can be desugared as $\kappa * M + \lambda * N \equiv \text{if}^{\circ} \lambda * |0\rangle + \kappa * |1\rangle$ then M else N. The base case interpretation is described in figure 22.

$$\frac{|\kappa|^2 + |\lambda|^2 = 1}{\Gamma \vdash (\kappa * |0\rangle + \lambda * |1\rangle)^{\text{dom}\ \Gamma} : \mathcal{Q}} \multimap \text{SUP}^0_+$$


Γ
|0⟩
φ_W
$\begin{pmatrix} \kappa & \lambda \\ \lambda & -\kappa \end{pmatrix}$
$\mathcal{Q}$
G_Γ
|0⟩
$\begin{pmatrix} \kappa & \lambda \\ \lambda & -\kappa \end{pmatrix}$
$\mathcal{Q}$

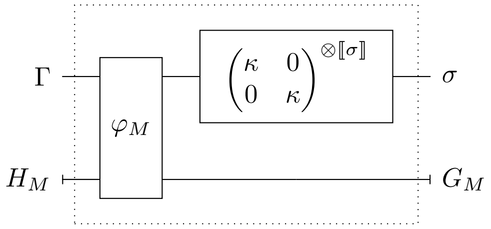
$(1, 0, \varphi_{\text{SUP}^0_+}) \in \mathbf{FQC}^{\circ}\ 0\ 1$

$(1, [\![\Gamma]\!], \varphi_{\text{SUP}^0_+}) \in \mathbf{FQC}\ [\![\Gamma]\!]\ 1$

Figure 22: Operational semantics of single qubit superpositions

Rotations are interpreted similarly to superpositions, but acting on a single term. Figure 23 describes this interpretation, where $\otimes n$ is the n-fold Kronecker product of the matrix.

$$\frac{\Gamma \vdash M : \sigma \qquad |\kappa| = 1}{\Gamma \vdash \kappa * M : \sigma} \multimap \kappa$$


Γ
H_M
φ_M
$\begin{pmatrix} \kappa & 0 \\ 0 & \kappa \end{pmatrix}^{\otimes [\![\sigma]\!]}$
σ
G_M

$(h_M, g_M, \varphi_\kappa) \in \mathbf{FQC}\ [\![\Gamma]\!]\ [\![\sigma]\!]$

Figure 23: Operational semantics of rotations

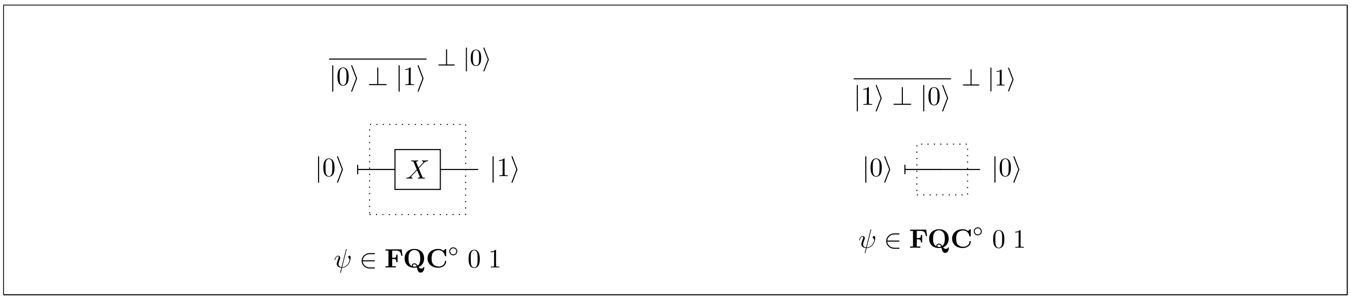
4.5 Orthogonality implementation

The semantics of an orthogonality judgement is as follows. Given two terms M and N such that $\Gamma \vdash^{\circ} M, N : \sigma$, the judgement $M \perp N$ holds if $\mathcal{O}$ can be derived. Here, $\mathcal{O}$ is defined as

$$\mathcal{O} \ \equiv\ (c \in \mathbb{N},\ l \in \mathbf{FQC}^{\circ}\ [\![\Gamma]\!]\ c,\ r \in \mathbf{FQC}^{\circ}\ [\![\Gamma]\!]\ c,\ \psi \in \mathbf{FQC}\ (1+c)\ [\![\sigma]\!]).$$

4.5.1 Orthogonality of qubits

Orthogonality of qubit atoms is interpreted by $\mathcal{O} = (0, \bullet, \bullet, \psi)$, where $\bullet$ is the empty **FQC** morphism $(0, 0, \{\}) \in \mathbf{FQC}^{\circ}\ 0\ 0$ and ψ is defined as in figure 24.



$\dfrac{}{|0\rangle \perp |1\rangle} \perp |0\rangle$
$|0\rangle$
X
$|1\rangle$
$\psi \in \mathbf{FQC}^{\circ}\ 0\ 1$
$\dfrac{}{|1\rangle \perp |0\rangle} \perp |1\rangle$
$|0\rangle$
$|0\rangle$
$\psi \in \mathbf{FQC}^{\circ}\ 0\ 1$

Figure 24: Orthogonality circuits for qubits

4.5.2 Orthogonality of tuples

For an orthogonality judgement $\langle M, V\rangle \perp \langle N, W\rangle$ of types $\Gamma \vdash^{\circ} \langle M, V\rangle, \langle N, W\rangle : \sigma \otimes \tau$, first produce the interpretation (c, l, r, ψ) for $M \perp N$. Then construct the final interpretation $\mathcal{O} = (c + [\![\tau]\!], \hat{l}, \hat{r}, \hat{\psi})$ where $\hat{l}$ and $\hat{r}$ are shown in figure 25 and $\hat{\psi}$ in figure 26.

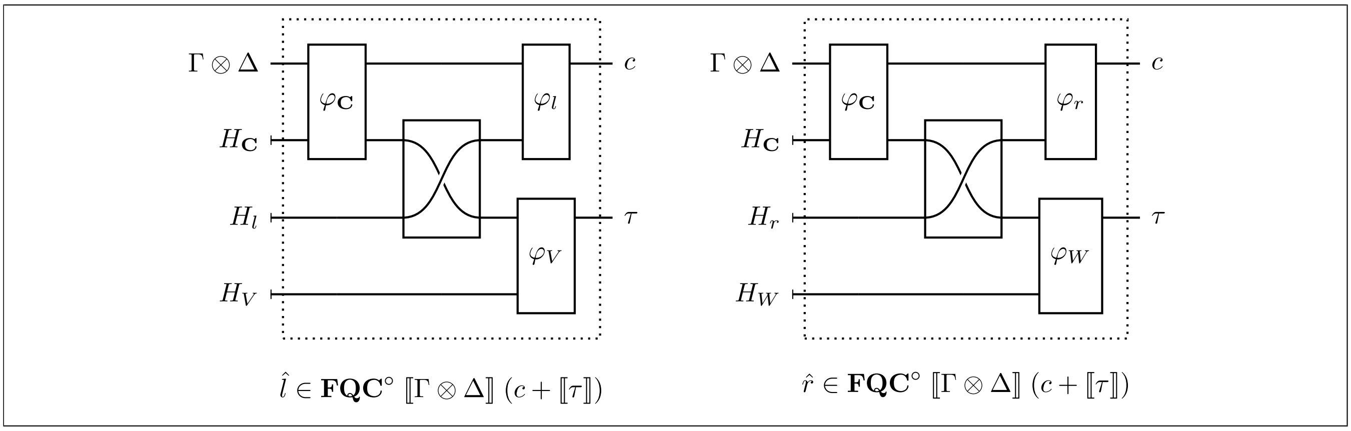
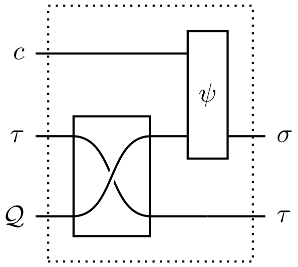


Figure 25: Orthogonality subcircuits for tuples

In the reverse rule $(\perp \otimes_2)$ the $\hat{l}$ and $\hat{r}$ circuits are flipped. In either case $\hat{\psi}$ is given by figure 26.



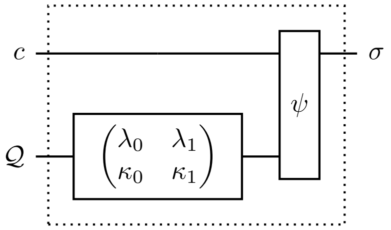
$$\hat{\psi} \in \mathbf{FQC}^\circ\ (c + [\![\tau]\!] + 1)\ ([\![\sigma]\!] + [\![\tau]\!])$$

Figure 26: Orthogonality final circuit for tuples

4.5.3 Orthogonality of superpositions and rotations

For an orthogonality judgement $\lambda_0 * M + \lambda_1 * N \perp \kappa_0 * M + \kappa_1 * N$ of types $\Gamma \vdash^\circ M, N : \sigma$, produce the interpretation (c, l, r, ψ). The final interpretation is constructed by $\mathcal{O} = (c, l, r, \hat{\psi})$, where $\hat{\psi}$ is defined in figure 27. Rotations are interpreted similarly by the modified morphism $\hat{\psi}$ as defined in figure 28.

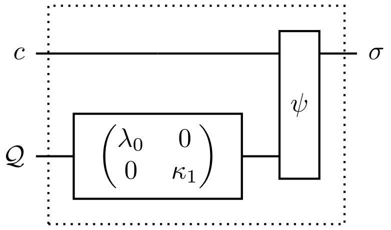
$$\frac{M \perp N \qquad \lambda_0^* \kappa_0 = -\lambda_1^* \kappa_1}{\lambda_0 * M + \lambda_1 * N \perp \kappa_0 * M + \kappa_1 * N}\ \perp\text{SUP}$$


c
σ
ψ
$\mathcal{Q}$
$\begin{pmatrix}\lambda_0 & \lambda_1 \\ \kappa_0 & \kappa_1\end{pmatrix}$

$$\hat{\psi} \in \mathbf{FQC}^\circ\ (c + 1)\ [\![\sigma]\!]$$

Figure 27: Orthogonality circuit for superpositions

$$\frac{M \perp N}{\lambda * M \perp \kappa * N}\ \perp\text{MUL}$$



$$\hat{\psi} \in \mathbf{FQC}^\circ\ (c + 1)\ [\![\sigma]\!]$$

Figure 28: Orthogonality circuit for rotations

5 Results

This project has resulted in a compiler capable of generating circuit representations of the expressions specified by the syntax of Signe. The compiler features a static type checker that can verify correctness properties and infer types for the entire language, without requiring type annotations. The code generator handles constraints on linearity and orthogonality as specified by the operational semantics. Finally, compilation results can be simulated using the built-in matrix generator.

A complete implementation of the specified language, fully utilising the power of the Hindley-Milner type system, requires a definition of top-level function handling, as well as extending the static type checker with an application-arity expression. A rigorous testing framework, or preferably formal verification, of the compiler would also be required for a complete result.

5.1 Performance

The compiler was benchmarked on programs to estimate its time complexity. In particular, it was tested on a program consisting of the definition of a tuple containing a number of quantum and classical conditionals as well as superpositions. The choice of conditionals and superpositions in testing is due to their relatively complex implementation. In the first compilation run, there were one of each of the conditionals; in the following runs the sizes were each doubled until the sizes were each 2^{14} for a total of 2^{15} terms, eliciting an **FQC** heap size of $3 \cdot 2^{15}$ qubits[1].

With the compilation times and number of qubits recorded, a curve $T(n)$ was fitted to the data points using a second degree linear regression, yielding the parameters $\theta = [0.146467780, -2.29823951 \cdot 10^{-5}, 9.33694378 \cdot 10^{-9}]$. The function estimating the compilation time per number of qubits is then given by $T(n) = \sum_{i=0}^{2} \theta_i n^i$. This yields an estimated time complexity of $\mathcal{O}(n^2)$ for the compiler using circuit output.

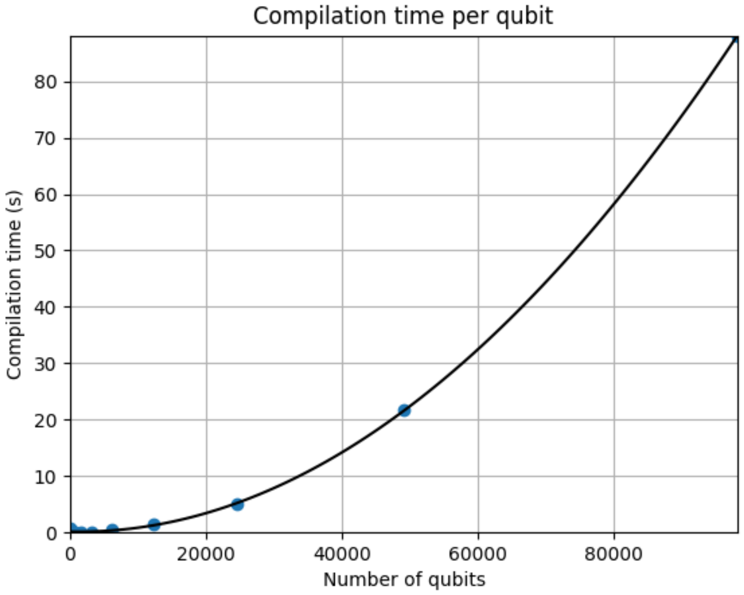


Figure 29: Time taken for each of the 16 programs to compile, plotted against the number of qubits in said program. The curve that was fitted to the data points is also shown.

For each of the runs after the first one, the exponent c of the time complexity $\mathcal{O}(n^c)$ term was calculated using

$$c = \log_2 \frac{T_i}{T_{i-1}},$$

where T_i correspond to the time measurements. A table of these measurements is included in appendix B.

[1] The reason for the number 3 appearing is that for each pair of conditionals, three qubits are needed from the heap. The quantum conditional generates one qubit; while the classical conditional generates two, one for control (which is subsequently measured and discarded) and one for the target.

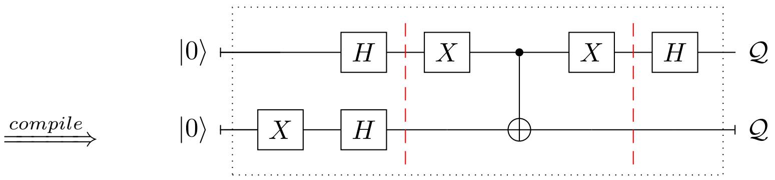
5.2 Programs

Figure 31 illustrates functions in Signe and the resulting **FQC** morphism returned by the compiler. Note that some trivial optimisations have been made to the unitary operators, specifically removal of identity permutations, for the sake of brevity. Top-level definitions have been substituted using λ-abstractions.

```
Balanced (x,y)
 := if° X x
      then (~0, X y)
      else (~1,   y)

Deutsch oracle
 := let (x,y) = oracle (~+, ~-)
      in H x

Main := Deutsch Oracle
```



$$(2, 1, \delta) \in \mathbf{FQC}^{\Downarrow}\ 0\ 1$$

$$\delta = ((H \odot X) \otimes (X \odot H)) \odot (X \oplus \{0\}) \odot ((X \odot H) \otimes \{0\})$$

Figure 30: Compilation of Deutsch's algorithm with slices indicating state preparation and the oracle.

An implementation of Deutsch's algorithm is presented in figure 30. Since the compilation result is an executable morphism, it can be simulated using Signe's matrix translation. Applying the generated matrix on the heap state gives

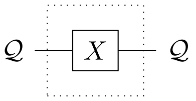
$$\begin{pmatrix} 0 & \frac{-1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} & 0 \\ 0 & \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 \\ \frac{1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} & 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{pmatrix}.$$

Measuring and discarding the second qubit as per the morphism results in the output state $|1\rangle$, which is the expected result of Deutsch's algorithm on a balanced oracle.

Pauli-X

```
X q := if° q
         then ~0
         else ~1
```

$\xRightarrow{compile}$

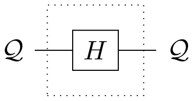


$(0, 0, X) \in \mathbf{FQC}^\circ\ 1\ 1$

Hadamard

```
H q := if° q
         then ~-
         else ~+
```

$\xRightarrow{compile}$

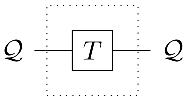


$(0, 0, H) \in \mathbf{FQC}^\circ\ 1\ 1$

T-gate

```
T q := if° q
         then e^(iπ/4) * ~1
         else ~0
```

$\xRightarrow{compile}$

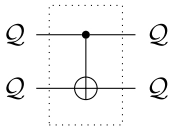


$(0, 0, T) \in \mathbf{FQC}^\circ\ 1\ 1$

Controlled-Not

```
CNOT c t
 := if° c
      then (~1, X t)
      else (~0,   t)
```

$\xRightarrow{compile}$

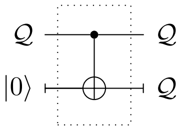


$(0, 0, X \obar \{0\}) \in \mathbf{FQC}^\circ\ 2\ 2$

Measurement

```
Measure q : qubit -> qubit
 := let (x,y) = (q,q)
      in q
```

$\xRightarrow{compile}$

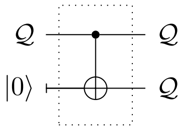


$(1, 1, X \obar \{0\}) \in \mathbf{FQC}\ 1\ 1$

Sharing

```
Share q : qubit -> qubit * qubit
 := (q,q)
```

$\xRightarrow{compile}$



$(1, 0, X \obar \{0\}) \in \mathbf{FQC}^\circ\ 1\ 2$

Figure 31: Collection of Signe programs and the **FQC** morphisms they compile to.

6 Discussion

With the most complicated and essential parts of the compiler having been implemented, we feel confident in saying that the project has been successful. A complete compilation chain from source code to execution by simulation is implemented, and the language is indeed in a usable state.

6.1 Performance, power, and expressiveness

With Signe being a compiled language, the exponentially increasing amount of time needed to simulate quantum states is entirely evaded. While this is neither unique nor unexpected we gather that the relative efficiency together with the strict and high-level nature of Signe could be of use in the analysis of quantum programs. Specifically it allows properties of programs to be studied within a simpler system, either in the system itself using the type checker, or by using some metalanguage.

The estimated time complexity of $\mathcal{O}(n^2)$, while not entirely unmanageable, puts some restrictions on the scalability of the compiler. This poses little threat to the current usability as the compiler seems to handle currently executable programs with relative ease. Whether or not the performance, or lack thereof, is inherent to compilation of this kind, or the fault of poor programming, has not been investigated in this project. Further analysis and testing of the compiler, and especially the code generator, is needed but was not prioritised due to time restrictions.

It is hard to judge the expressiveness of the language, being its creator. There is a definite bias that develops from complete knowledge of the description and implementation of the language. Feedback from programmers from a wider background would provide much greater insight into possible improvements. We do feel, however, that the language at its current state is more natural and easier to program in than many other quantum programming languages we've encountered. The large separation from circuit programming seems to make it easier to translate experience in classical programming to quantum programming. Together with Signe's familiar syntax and type system, this advantage becomes even more apparent. Freeing the programmer from the constraints of linearity is also advantageous in this regard.

Nevertheless, our intention is that Signe could serve as a modern and well-defined platform for further exploration into the applications of functional programming and type theory in quantum computing.

6.2 Comparison of Signe with QML

While we call Signe its own language, the similarities between Signe and QML are more numerous than their differences. The theoretical definition of QML provided by its original creators Grattage and Altenkirch, and in particular the operational semantics in terms of **FQC** , has been the main source of inspiration for Signe. It is therefore natural that the two languages are similar. However, there are some differences that we want to highlight.

One of the more apparent differences of the two languages is the type systems used. The type system of QML is based on strict linear logic, which is similar to the linear type system used in Signe. It is however simply typed and requires explicit type declaration. Herein lies the most striking modification with regards to typing. Signe's primary type system, a Hindley-Milner system, features full type inference and as such doesn't require type declaration. This type system was chosen with the prospect of polymorphic types in mind, and establishing polymorphic types in a language that is otherwise similar to QML would contribute significantly to differentiating the two. Full integration of polymorphic types in the code generator is, however, not completely implemented. While the implementation of the system is capable of correctly inferring polymorphic types for the whole language, further analysis during static type checking, along with annotating arity (in the sense of type arity described in section 3.1.2) in applications of polymorphic functions is needed to utilise this power. This is currently being worked on.

In the way of semantics, the languages are indeed similar with the distinction coming down to the differences in expressions included in the language. The interpretation of rotations, λ-abstraction, application, and atoms have been designed specifically for Signe. These extensions also affect the judgement of orthogonality, and more extensions to this have been discussed and investigated, though not to the extent of being included in the definition used in this report.

The implementation of the compilers for the languages is also very different, though both written in Haskell. With the state of Haskell having evolved significantly since QML's conception, this is not unexpected, but their differences are apparent beyond this too. Signe is implemented more in line with modern compilers and has more developed error handling and overall structure as compared to QML.

7 Conclusion

A compiled functional quantum programming language, without circuit primitives, using a Hindley-Milner type system, and quantum conditionals has been defined and implemented. The language is in a usable, though incomplete, state. The operational semantics is defined in, and compilation facilitated by, the model **FQC** of finite quantum computations. The compiler is implemented in Haskell[2] in a relatively modern and user-oriented way, featuring helpful and extensive error handling. Correctness, both in regards to the rules of Signe and physicality, is checked by the compiler. This is done statically to a high extent by the type system, with the remaining checks begin performed during code generation. A lack of rigorous testing and verification of the compiler itself prevents us from stating if these checks contain any implementation errors, or if they are indeed sufficient to fulfil specification.

7.1 Future work

Perhaps most the most obvious continuation is the completion of the implementation according to the language's description in this paper. This is currently being worked on by the authors, and a more formal specification of Signe is being written. An interesting extension would be to include some classical data types in the language. While the compilation of these is not entirely clear, we propose two separate but complementary interpretations. Firstly, one could ascribe otherwise classical data types a quantum interpretation; a trivial example being quantum integers (i.e. $|5\rangle = |101\rangle$). It would also be possible to use these syntactically to parameterise quantum functions in some way, similar to Silq's generic parameters. Secondly, it would be interesting do define a classical semantics separate of the one already established. This would of course not be compilable but by running a pre-compiler (or more correctly a pre-compilation interpreter) that tries to evaluate classical expressions, which are capable of containing quantum subroutines, before returning a purely quantum program to the rest of the compiler. This would likely resemble syntactic macros from classical programming languages, and could perhaps (though not to the extent of homoiconicity) be implement in a similar way.

Since the internal abstract syntax of Signe represents a relatively standard functional language it would be possible to expose a generalised version of it in a Haskell library. This would then be connected to the generator and back-end of Signe's compiler, enabling language designers to define their own syntax and type system using the operational semantics of Signe. This would essentially serve as a general internal representation for functional quantum programming languages, greatly increasing production speed for such languages.

[2] The source code is available at `https://github.com/NicklasBoto/signe`

8 Acknowledgements

We would like to thank our wonderful supervisors Miroslav Dobsicek and David Wahlstedt for their valuable guidance and feedback. We would also like to thank the examiners of the course Simone Gasparinetti and Attila Geresdi for letting us do the project at all. Let us not forget the language's greatest inspiration, Signe the dog (figure 32), who finally provided a quantum programming language without a Q in its name, previously thought to be impossible. Finally, Nicklas would like to thank Robert Rand for his advice on, and introduction to, the field of quantum programming languages.

This project required no funding at all, especially not from any evil cult, only an unholy amount of time.

Figure 32: Signe, the dog

References

[1] T. Altenkirch and J. Grattage. "A Functional Quantum Programming Language". In: *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)* (). DOI: `10.1109/lics.2005.1`. URL: `http://dx.doi.org/10.1109/LICS.2005.1`.

[2] Andrew W. Cross et al. *Open Quantum Assembly Language*. 2017. arXiv: `1707.03429 [quant-ph]`.

[3] Luis Damas and Robin Milner. "Principal Type-Schemes for Functional Programs". In: *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '82. Albuquerque, New Mexico: Association for Computing Machinery, 1982, pp. 207–212. ISBN: 0897910656. DOI: `10.1145/582153.582176`. URL: `https://doi.org/10.1145/582153.582176`.

[4] Jonathan Grattage. "An Overview of QML with a Concrete Implementation in Haskell". In: *Electronic Notes in Theoretical Computer Science* 270.1 (2011). Proceedings of the *4^{th} Workshop on Developments in Computational Models* (DCM '08), doi:10.1016/J.ENTCS.2011.01.015, arXiv:0806.2735, pp. 165–174. ISSN: 1571-0661. DOI: `10.1016/j.entcs.2011.01.015`. URL: `http://fop.cs.nott.ac.uk/qml`.

[5] Jonathan Grattage. "QML: A functional quantum programming language". PhD thesis. School of Computer Science and School of Mathematical Sciences, The University of Nottingham, Sept. 2006. URL: `http://etheses.nottingham.ac.uk/archive/00000250/`.

A Reading material

- Type theory: `https://www.cs.ru.nl/~herman/PUBS/IntroTT-improved.pdf`
- Functional programming: `https://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/all.pdf`
- Compiler design: `https://www.grammaticalframework.org/ipl-book/`

B Table of complexity coefficients

Qubits (№)	**Time (s)**	$\mathbf{c} = \log_2 \frac{T_i}{T_{i-1}}$
3	1.60×10^{-4}	N/A
6	1.84×10^{-4}	2.03×10^{-1}
12	7.77×10^{-1}	1.20×10^{1}
24	6.18×10^{-2}	-3.65
48	1.56×10^{-2}	-1.99
96	1.18×10^{-3}	-3.72
192	3.34×10^{-1}	8.15
384	7.90×10^{-2}	-2.08
768	7.42×10^{-2}	-8.96×10^{-2}
1536	3.07×10^{-2}	-1.27
3072	1.43×10^{-1}	2.21
6144	4.20×10^{-1}	1.56
12288	1.40×10^{0}	1.74
24576	5.03×10^{0}	1.85
49152	2.17×10^{1}	2.11
98304	8.81×10^{1}	2.02

Table 1: Time complexity coefficients c for increasing number of qubits