Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics

Benjamin Bichsel  
ETH Zurich, Switzerland  
benjamin.bichsel@inf.ethz.ch

Maximilian Baader  
ETH Zurich, Switzerland  
mbaader@inf.ethz.ch

Timon Gehr  
ETH Zurich, Switzerland  
timon.gehr@inf.ethz.ch

Martin Vechev  
ETH Zurich, Switzerland  
martin.vechev@inf.ethz.ch

1 Introduction

Quantum algorithms leverage the principles of quantum mechanics to achieve an advantage over classical algorithms. In recent years, researchers have continued proposing increasingly complex quantum algorithms [5, 9, 12, 13, 24, 33], driving the need for expressive, high-level quantum languages.

The Need for Uncomputation. Analogously to the classical setting, quantum computations often produce temporary values. However, as a key challenge specific to quantum computation, removing such values from consideration induces an implicit measurement collapsing the state [21, §4.4]. In turn, collapsing can result in unintended side-effects on the state due to the phenomenon of entanglement. Surprisingly, due to the quantum principle of deferred measurement [21, §4.4], preserving values until computation ends is equivalent to measuring them immediately after their last use, and hence cannot prevent this problem.

To remove temporary values from consideration without inducing an implicit measurement, algorithms in existing languages must explicitly uncompute all temporary values, i.e., modify their state to enable ignoring them without side-effects. This results in a significant gap from quantum to classical languages, where discarding temporary values typically requires no action (except for heap values not garbage-collected). This gap is a major roadblock preventing the adoption of quantum languages as the implicit side-effects resulting from uncomputation mistakes, such as silently dropping temporary values, are highly unintuitive.

This Work. We present Silq, a high-level quantum language which bridges this gap by automatically uncomputing temporary values. To this end, Silq’s type system exploits a fundamental pattern in quantum algorithms, stating that uncomputation can be done safely if (i) the original evaluation of the uncomputed value can be described classically, and (ii) the variables used to evaluate it are preserved and can thus be leveraged for uncomputation.

As uncomputation happens behind the scenes and is always safe, Silq is the first quantum language to provide intuitive semantics: if a program type-checks, its semantics
2 Benefit of Automatic Uncomputation

Next, we show the benefit of automatic uncomputation compared to explicit uncomputation in existing languages, including Q# [30], Quipper [7], and QWire [22].

Explicit Uncomputation. Fig. 1 shows code snippets which compute the OR of three qubits. This is easily expressed in Silq (top left), which leverages automatic uncomputation for $a \lor b$. In contrast, Q# (right) requires (i) allocating a new qubit $t$ initialized to $\text{false}$ in Line 1, (ii) using OR to store the result of $a \lor b$ in $t$ in Line 2, (iii) using OR to store the result of $t \lor c$ in the pre-allocated qubit $d$ in Line 3, and (iv) uncomputing $t$ by reversing the operation from Line 2 to Line 4. Here, $\text{Adjoint OR}$ is the inverse of OR and thus resets $t$ to its original value of $\text{false}$. Hence, the implicit measurement induced by removing $t$ from consideration in Line 5 always measures the value $\text{true}$, which has no side-effects (see §3). We note that we cannot allocate $t$ within OR, as Q# enforces that qubits must be deallocated in the function that allocates them.

Explicit uncomputation is even more tedious in Fig. 2, which shows part of a triangle finding algorithm originally encoded\(^5\) by the authors of Quipper [7] (middle). The condition is easily expressed in Silq (left, Lines 4–5) using nested expressions. In contrast, the equivalent Quipper code is obfuscated by uncomputation of sub-expressions.

Convenience Functions. As uncomputation is a common task, various quantum languages try to reduce its boilerplate code by introducing convenience functions such as ApplyWith in Q#. Fig. 1 shows a Quipper implementation using a similar function with_computed, which (i) evaluates $a \lor b$ in Line 1, (ii) uses the result $t$ to compute $t \lor c$ in Line 2, and (iii) implicitly uncomputes $t$. However, this still requires explicitly triggering uncomputation using with_computed and introducing a name $t$ for the result of $a \lor b$. In particular, this does not enable a natural nesting of expressions, as the sub-expression OR $a \lor b$ needs to be managed by with_computed. Moreover, with_computed cannot ensure safety: we can make the uncomputation unsafe by flipping the bit stored in $b$ between Line 1 and Line 2, triggering an implicit measurement.

Non-Linear Type Systems. Most quantum languages cannot ensure that all temporary values are safely uncomputed for a fundamental reason: they support reference sharing in a non-linear type system and hence cannot statically detect when values are removed from consideration (which happens when the last reference to the value goes out of scope). Besides Quipper, which we discuss in more detail, there are many other works of this flavor, including LQuID [32], ProjectQ [29], Cirq [31], and QisKit [1].

---

4. Since Q# does not support OR natively, we would need to implement it too.
5. Taken from: https://www.mathstat.dal.ca/~selinger/quipper/doc/src/Algorithms/TF/QWTFP.html#line-494
**Linear Type Systems.** Other languages, like QPL [27] and QWire, introduce a linear type system to prevent accidentally removing values from consideration, which corresponds to not using a value. However, linear type systems still require explicit uncomputation that ends in *assertive termination* [7]: the programmer must (manually) assert that uncomputation correctly resets temporary values to 0. Most recently, ReQWire [23] introduced syntactic conditions sufficient to verify assertive termination. However, ReQWire can only verify explicitly provided uncomputation (except for purely classical oracle functions, see below), and cannot statically reason across function boundaries as its type system does not address uncomputation — a key contribution of Silq.

Further, linear type systems introduce significant syntactic overhead for constant (i.e., read-only) variables where enforcing linearity is not necessary. Fig. 2 demonstrates this in QWire code (right), where encoding only Lines 4–6 from Silq (left) requires 19 lines of code, even when we generously assume built-in primitives and omit parts of the required type annotations. We note that while QWire [22] does not explicitly claim to be high-level, we are not aware of more high-level quantum languages that achieve a level of safety similar to QWire — even though it cannot prevent implicit measurement caused by incorrect manual uncomputation.

In contrast, Silq uses a linear type system to detect values removed from consideration (which are automatically uncomputed), but reduces notational overhead by treating constant variables non-linearly.

**Bennett’s Construction.** Various languages, like Quipper, ReVerC [2], and ReQWire, support Bennett’s construction [3], which can lift purely classical (oracle) functions to quantum inputs, automatically uncomputing all temporary values computed in the function. Concretely, this standard approach (i) lifts all primitive classical operations in the oracle function to quantum operations, (ii) evaluates the function while preserving all temporary values, (iii) uses the function’s result, and (iv) uncomputes temporary values by reversing step (ii). Bennett’s construction is also supported by Qumquat\(^6\), which skips step (i) above by annotating quantum functions as `@qq.garbage` and calling them with notation analogous to Quipper’s `with_computed`.

However, Bennett’s construction is unsafe when the oracle function contains quantum operations: as we demonstrate in App. C, it can fail to drop temporary values without side-effects. In contrast, Silq safely uncomputes temporary values in functions containing quantum operations.

Importantly, Silq’s workflow when defining oracle functions is different from existing languages: while the latter typically require programmers to define a purely classical oracle function and then apply Bennett’s construction, Silq programmers can define oracle functions directly using primitive quantum operations, implicitly relying on Silq’s automatic uncomputation.

**Summary.** In contrast to other languages, Silq (i) enables intuitive yet physical semantics and (ii) statically prevents errors that are not detected in existing languages, while (iii) avoiding the notational overhead associated with languages that achieve (less) static safety (e.g., QWire).

3 Background on Quantum Computation

We now provide a short review of the core concepts in quantum computation relevant to this work.

**Qubit.** The state of a quantum bit (qubit) is a superposition (linear combination) \( \phi = \gamma_0 |0\rangle + \gamma_1 |1\rangle \), where \( \gamma_0, \gamma_1 \in \mathbb{C} \), and \( \|\phi\|^2 = \|\gamma_0\|^2 + \|\gamma_1\|^2 \) denotes the probability of being in state \( \phi \). In particular, we allow \( \|\phi\| < 1 \) to indicate that a measurement yields state \( \phi \) with probability \( \|\phi\| \) — a common convention [27, Convention 3.3].

**Hilbert Space, Ground Set, Basis State.** More generally, assume a variable on a classical computer can take on values from a finite *ground set* \( S \). Then, the quantum states induced by \( S \) form the Hilbert space \( \mathcal{H}(S) \) consisting of the formal complex linear combinations [25, p. 379] over \( S \):

\[
\mathcal{H}(S) := \left\{ \sum_{v \in S} y_v |v\rangle \mid y_v \in \mathbb{C} \right\}.
\]

Here, each element \( v \in S \) corresponds to a (computational) basis state \( |v\rangle \). For \( S = \{0, 1\} \), we obtain the Hilbert space of a single qubit \( \mathcal{H}(\{0, 1\}) = \{0\rangle \oplus \langle 1\} \mid y_0, y_1 \in \mathbb{C} \}, \) with computation basis states \( |0\rangle \) and \( |1\rangle \).

We note that we use the (standard) inner product \( \langle \cdot | \cdot \rangle \) throughout this work, defined by

\[
\left( \sum_{v \in S} y_v |v\rangle \right) \left( \sum_{u \in S} y_u |u\rangle \right) = \sum_{v \in S} y_v y_u^*.
\]

**Tensor Product.** A system of multiple qubits can be described using the tensor product \( \otimes \). For example, for two qubits \( \phi_0 = |0\rangle \) and \( \phi_1 = \frac{1}{\sqrt{2}} |0\rangle - i\frac{1}{\sqrt{2}} |1\rangle \), the composite state is \( \phi_0 \otimes \phi_1 = \frac{1}{\sqrt{2}} |0\rangle \otimes |0\rangle - i\frac{1}{\sqrt{2}} |0\rangle \otimes |1\rangle = \frac{1}{\sqrt{2}} |0\rangle |0\rangle - i\frac{1}{\sqrt{2}} |0\rangle |1\rangle \). Here, we first used the linearity of \( \otimes \) in its first argument and then omitted \( \otimes \) for convenience. Simplifying notation further, we may also write \( |0\rangle |0\rangle \) as \( |0, 0\rangle \).

**Entanglement.** A composite state is called *entangled* if it cannot be written as a tensor product of single qubit states, but needs to be written as a sum of tensor products. For example, the above composite state \( \phi_0 \otimes \phi_1 \) is entangled, while \( \Phi^+ = \frac{1}{\sqrt{2}} |0\rangle |0\rangle + \frac{1}{\sqrt{2}} |1\rangle |1\rangle \) is entangled.

**Measurement.** To acquire information about a quantum state, we can (partially) measure it. Measurement has a probabilistic nature; if we measure \( \phi = \sum_{v \in S} y_v |v\rangle \), we obtain

---

\(^6\)Available at https://github.com/patrickrall/Qumquat, commit 27d6794
the value \( v' \in S \) with probability \( ||y_{v'}||^2 \). As a fundamental law of quantum mechanics, if we measure the value \( v' \), the state after the measurement is \( y_{v'} |v'\rangle \) (we do not normalize this state to preserve linearity). This is referred to as the collapse of \( \phi \) to \( y_{v'} |v'\rangle \), since superposition is lost.

Importantly, measuring part of a state can affect the whole state. To illustrate the effect of measuring the first part \( |v\rangle \) of \( \sum_{v,w} y_{v,w} |v\rangle |w\rangle \), we first rewrite it to \( \sum_{v} y_{v} |v\rangle \otimes \tilde{\phi}_v \), separating out the remainder \( \tilde{\phi}_v \) of the state, where \( \|\tilde{\phi}_v\| = 1 \). This is a common technique and always possible for appropriate choices of \( y_v \) and \( \tilde{\phi}_v \). Then, measuring the first part to be \( v' \) yields state \( y_{v'} |v'\rangle \otimes \tilde{\phi}_{v'} \), also collapsing the remainder of the state.

**Linear Isometries.** Besides measurements, we can also manipulate quantum states using **linear isometries**, i.e., linear functions \( f : \mathcal{H}(S) \rightarrow \mathcal{H}(S') \) preserving inner products: for all \( \phi, \phi' \in \mathcal{H}(S), \langle f(\phi) | f(\phi') \rangle = \langle \phi | \phi' \rangle \). Linear isometries generalize the commonly used notion of unitary operations, which additionally require that vector spaces \( \mathcal{H}(S) \) and \( \mathcal{H}(S') \) have the same dimension. As this prevents dynamically allocating and deallocating qubits, we use the more general notion of linear isometries in this work.

**QRAM.** As a computational model for quantum computers, this work assumes a quantum random access machine (QRAM) [11]. A QRAM consists of a classical computer extended with quantum storage supporting state preparation, some unitary gates, and measurement. QRAMs can be extended to support linear isometries, by (i) padding input and output space to have the same dimension (using state preparation) and (ii) approximating the resulting unitary operation arbitrarily well using a standard set of universal quantum gates [21, §4.5.3].

**No-Cloning.** The no-cloning theorem states that cloning an arbitrary quantum state is unphysical: we cannot achieve the operation \( \phi \mapsto \phi \otimes \phi \). Silq’s type system prevents cloning.

### 4 Overview of Silq

We now illustrate Silq on Grover’s algorithm, a widely known quantum search algorithm [8], [21, §6.1]. It can be applied to any NP problem, where finding the solution may be hard, but verification of a solution is easy.

Fig. 3 shows a Silq implementation of grover. Its input is an oracle function \( f \) from (quantum) unsigned integers represented with \( n \) qubits to (quantum) booleans, mapping all but one input \( w^* \) to \( 0 \). Here, Silq uses the generic parameter \( n \) to parametrize the input type \( \text{unit}[n] \) of \( f \). Then, grover outputs an \( n \)-bit unsigned integer \( w \) which is equal to \( w^* \) with high probability.

### 4.1 Silq Annotations

**Classical Types.** The first argument of grover is a **generic parameter** \( n \), used to parametrize \( f \). It has type \( \mathbb{N} \), which indicates classical natural numbers of arbitrary size. Here, annotation \( ! \) indicates \( n \) is classically known, i.e., it is in a basis state (not in superposition), and we can manipulate it classically. For example, \( \theta \) has type \( \mathbb{B} \). In contrast, \( H(0) \) applies Hadamard \( H \) (defined shortly) to \( \theta \) and yields \( \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) \).

Thus, \( H(0) \) is of type \( \mathbb{B} \) and not of (classical) type \( \mathbb{B} \).

In general, we can liberally use classical variables like normal variables on a classical computer: we can use them multiple times, or drop them. We also annotate parameter \( f \) as classical, writing the annotation as \( !r \rightarrow !r' \) instead of \( !r \rightarrow r' \) to avoid the ambiguity between \( !r \rightarrow r' \) and \( ![r] \rightarrow ![r'] \).

**qfree Functions.** The type of \( f \) is annotated as \( \text{qfree} \), which indicates the semantics of \( f \) can be described classically: we can capture the semantics of a \( \text{qfree} \) function \( g \) as a function \( \overline{g} : S \rightarrow S' \) for ground sets \( S \) and \( S' \). Note that since \( S' \) is a ground set, \( \overline{g} \) can never output superpositions. Then, \( g \) acting on \( \sum_{v} y_v |v\rangle \) yields \( \sum_{v} y_v |\overline{g}(v)\rangle \), where for simplicity \( \sum_{v} y_v |v\rangle \) does not consider other qubits untouched by \( g \). For example, the \( \text{qfree} \) function \( X \) flips the bit of its input, mapping \( \sum_{v=0}^{1} y_v |v\rangle \) to \( \sum_{v=1}^{0} y_v |\overline{1}\rangle \), for \( \overline{1} = 1 - v \). In contrast, the Hadamard transform \( H \) maps \( \sum_{v=0}^{1} y_v |v\rangle \) to \( \sum_{v=0}^{1} y_v \frac{1}{\sqrt{2}} (|0\rangle + (-1)^v |1\rangle) \). As this semantics cannot be described by a function on ground sets, \( H \) is not \( \text{qfree} \).

**Constant Parameters.** Note that \( X \) (introduced above) transforms its input—it does not preserve it. In contrast, the parameter of \( f \) is annotated as \( \text{const} \), indicating \( f \) preserves its input, i.e., treats it as a read-only control. Thus, running \( f \) on \( \sum_{v} y_v |v\rangle \) yields \( \sum_{v} y_{f(v)} |\overline{f}(v)\rangle \), where \( \overline{f} \) follows the semantics of \( f \). Because \( f \) is also \( \text{qfree} \), \( |v\rangle \otimes \phi_v = |\overline{f}(v)\rangle \) for some \( \overline{f} : S \rightarrow S \times S' \). Combining both, we conclude that \( \overline{f}(v) = (v, \overline{f}(v)) \) for some function \( \overline{f} : S \rightarrow S' \).

An example of a possible instantiation of \( f \) is NOT, which maps \( y_0 |0\rangle + y_1 |1\rangle \) to \( y_0 |0\rangle + y_1 |1, 0\rangle \). Here, \( \text{NOT} : \{0, 1\} \rightarrow \{0, 1\} \) maps \( v \mapsto 1 - v \) and \( \text{NOT} : \{0, 1\} \rightarrow \{0, 1\} \times \{0, 1\} \) maps \( v \mapsto (1 - v) = (v, \text{NOT}(v)) \).

Function parameters not annotated as \( \text{const} \) are not accessible after calling the function — the function **consumes** them. For example, \( \text{groverDiff} \) consumes its argument (see top-right box in Fig. 3). Hence, the call in Line 10 consumes \( \text{cand} \), transforms it, and writes the result into a new variable with the same name \( \text{cand} \). Similarly, measure in Line 12 consumes \( \text{cand} \) by measuring it.

---

\footnote{Annotating functions as classical indicates that their function bodies are classically known (at runtime). We note that classical functions can still perform quantum operations: for example, \( H : \text{qfree} \rightarrow \mathbb{B} \) is classical, meaning that the quantum operations performed by \( H \) are classically known.}
**Lifted Functions.** We introduce the term lifted to describe qfree functions with exclusively const parameters, as such functions are crucial for uncomputation. In particular, we could write the type of \( f \) as uint[n] \( \xrightarrow{\text{lifted}} \) B.

### 4.2 Silq Semantics

**Input State.** In Fig. 3, the state of the system after Line 1 is \( \psi_1 \), where the state of \( f: \text{uint}[n] \xrightarrow{\text{qfree}} B \) is described as a function \( \bar{f} : \{0, \ldots, 2^n - 1\} \rightarrow \{0, 1\} \). We note that later, our formal semantics represents the state of functions as Silq-core expressions (§7). However, as the semantics of \( f \) can be captured by \( \bar{f} \), this distinction is irrelevant here. Next, Line 2 initializes the classical variable nIterations, yielding \( \psi_2 \).

**Superpositions.** Lines 3–4 result in state \( \psi_3 \), where cand holds the equal superposition of all n-bit unsigned integers. To this end, Line 4 updates the \( k \)-th bit of cand by applying the Hadamard transform \( H \) to it.

**Loops.** The loop in Line 6 runs nIterations times. Each loop iteration increases the probability of measuring \( w^* \) in Line 12. We now discuss the first loop iteration (\( k = 0 \)). It starts from state \( \psi_6^{(0)} \) which introduces variable \( k \). For convenience, \( \psi_6^{(0)} \) splits the superposition into \( w^* \) and all other values.

**Conditionals.** Intuitively, Lines 7–9 flip the sign of those coefficients for which \( f(\text{cand}) \) returns true. To this end, we first evaluate \( f(\text{cand}) \) and place the result in a temporary variable \( f(\text{cand}) \), yielding state \( \psi_7 \). Here and in the following, we write \( e \) for a temporary variable that contains the result of evaluating \( e \). Then, we determine those summands of \( \psi_8^{(0)} \) where \( f(\text{cand}) \) is true (marked as “then branch” in Fig. 3), and run phase(\( \pi \)) on them. This yields \( \psi_8^{(0)} \), as phase(\( \pi \)) flips the sign of coefficients. Lastly, we drop \( f(\text{cand}) \) from the state, yielding \( \psi_9^{(0)} \).

**Grover’s Diffusion Operator.** Completing the explanations of our example, Line 10 applies Grover’s diffusion operator to cand. Its implementation consists of 6 lines of code (see App. B). It increases the weight of solution \( w^* \), obtaining \( \| \psi_9^{(0)} \| = 1 \), and decreases the weight of non-solutions \( v \neq w^* \), obtaining \( \| \psi_9^{(1)} \| < \| \psi_9^{(0)} \| \). After one loop iteration, this results in state \( \psi_9^{(0)} \). Repeated iterations of the loop in Lines 6–11 further increase the coefficient of \( w^* \), until it is approximately 1. Thus, measuring cand in Line 12 returns \( w^* \) with high probability.

### 4.3 Uncomputation

While dropping the temporary value \( f(\text{cand}) \) from \( \psi_8^{(0)} \) is intuitive, achieving this physically requires uncomputation.

Without uncomputation, simply removing \( f(\text{cand}) \) from consideration in Line 9 would induce an implicit measurement. 9 Concretely, measuring and dropping \( f(\text{cand}) \) would collapse \( \psi_8^{(0)} \) to one of the following two states (ignoring \( f \), \( n \), and \( k \)):

\[
\psi_8^{(0,0)} = \sum_{v \neq w^*} \frac{1}{\sqrt{2^n}} |v\rangle_{\text{cand}} \quad \text{or} \quad \psi_8^{(0,1)} = -\frac{1}{\sqrt{2^n}} |w^*\rangle_{\text{cand}}.
\]

In this case, as the probability of obtaining \( \psi_8^{(0,1)} \) is only \( \frac{1}{2^n} \), grover returns the correct result \( w^* \) with probability \( \frac{1}{2^n} \), i.e., it degrades to random guessing.

\( ^9 \)Formally, this corresponds to taking the partial trace over \( f(\text{cand}) \).
Without correct intervention from the programmer, all existing quantum languages would induce an implicit measurement in Line 9, or reject grover. This is unfortunate as grover cleanly and concisely captures the programmer’s intent. In contrast, Silq achieves the intuitive semantics of dropping \( f(\text{cand}) \) from \( \psi_{\text{8}}^{(0)} \), using uncomputation. In general, uncomputing \( x \) is possible whenever in every summand of the state, the value of \( x \) can be reconstructed (i.e., determined) from all other values in this summand. Then, reversing this reconstruction removes \( x \) from the state.

**Automatic Uncomputation.** To ensure that uncomputing \( f(\text{cand}) \) is possible, the type system of Silq ensures that \( f(\text{cand}) \) is lifted, i.e., (i) \( f \) is qfree and (ii) cand is const: it is preserved until uncomputation in Line 9.

Fig. 4 illustrates why this is sufficient. Evaluating \( f \) in Line 7 adds a temporary variable \( f(\text{cand}) \) to the state, whose value can be computed from cand using \( f \) (as \( f \) is qfree and cand is const). Then, Line 8 transforms the remainder \( \tilde{\psi}_v \) of the state to \( X_{\text{8}, f(\text{cand})} \). The exact effect of Line 8 on the state is irrelevant for uncomputation, as long as it preserves cand, ensuring we can still reconstruct \( f(\text{cand}) \) from cand in \( \psi_{\text{8}}^{(0)} \). Thus, reversing the operations of this reconstruction (i.e., reversing \( f \)) uncomputes \( f(\text{cand}) \) and yields \( \psi_{\text{9}}^{(0)} \).

**4.4 Preventing Errors: Rejecting Invalid Programs**

Fig. 5 demonstrates how the type system of Silq rejects invalid programs. We note that the presented examples are not exhaustive — we discuss additional challenges in §6.

**Error: Using Consumed Variables.** In useConsumed, \( H \) consumes \( x \) and stores its result in \( y \). Then, it accesses \( x \), which leads to a type error as \( x \) is no longer available.

Assuming we want to preserve \( x \), we can fix this code by marking \( x \) as const (see useConsumedFixed). Then, instead of consuming \( x \) in the call to \( H \) (which is disallowed as \( x \) must be preserved), Silq implicitly duplicates \( x \), resulting in \( \psi_{\text{2}} \), and then only consumes the duplicate \( x \).

**Implicit Duplication.** It is always safe to implicitly duplicate constant variables, as such duplicates can be uncomputed (in useConsumedFixed, uncomputation is not necessary as the duplicate is consumed). In contrast, it is typically impossible to uncompute duplicates of consumed quantum variables, which may not be available for uncomputation later. Hence, Silq treats constant variables non-linearly (they can be duplicated or ignored), but treats non-constant variables linearly (they must be used exactly once).

We note that duplication \( \sum_{v} |v\rangle \rightarrow \sum_{v} |v\rangle \otimes |v\rangle \) is physical and can be implemented using CNOT, unlike the unphysical cloning \( \sum_{v} |v\rangle \rightarrow (\sum_{v} |v\rangle \otimes |v\rangle) = \sum_{v,w} |v\rangle |v\rangle |w\rangle \) discussed earlier.

**Error: Discarding Variables.** Function discard does not annotate \( x \) as const, meaning that its callers expect it to consume \( x \). However, the body of discard does not consume \( x \), hence calling discard would silently discard \( x \). As the callee does not know if \( x \) can be uncomputed, Silq rejects this code. A possible fix is annotating \( x \) as const, which would be in line with preserving \( x \) in the function body.
**Error: Uncomputation Without Qfree.** Silq rejects the function nonQfree, as \( H(x) \) is not lifted (since \( H \) is not qfree), and hence its result cannot be automatically uncomputed. Indeed, automatic uncomputation of \( H(x) \) is not possible in this case, intuitively because \( H \) introduces additional entanglement preventing uncomputation in the end. We provide a more detailed mathematical derivation of this subtle fact in App. C. To prevent this case, Silq only supports uncomputing qfree expressions.

We note that because \( x \) is const in nonQfree, \( H \) does not consume it, but a duplicate of \( x \).

**Error: Uncomputation Without Const.** Silq rejects the function nonConst, as \( X(c) \) is not lifted (since it consumes \( c \)). Indeed, automatic uncomputation is not possible in this case, as the original value of \( c \) is not available for uncomputation of \( X(c) \). To get this code to type-check, we can mark \( c \) as const (see nonConstFixed) to clarify that \( c \) should remain in the context. Then, Silq automatically duplicates \( c \) before calling \( X \), which thus consumes a duplicate of \( c \), leaving the original \( c \) available for later uncomputation.

**Temporary Constants.** In contrast to nonConst, which consumes \( c \), grover does not consume cand in Line 7 (Fig. 3), even though cand is not annotated as const either. This is because Silq temporarily annotates cand as const in grover. In general, Silq allows temporarily annotating some variables as const for the duration of a statement or a consumed subexpression. Our implementation determines which variables to annotate as const as follows: If a variable is encountered in a position where it is not expected to be const (as in \( X(c) \)), it is consumed, and therefore any further occurrence of that variable will result in an error (whether const or not). If a variable is encountered in a position where it is expected to be const (as in \( f(c) \)), we temporarily mark it as const until the innermost enclosing statement or consumed subexpression finishes type checking.

**Mfree.** Silq’s main advantage over existing quantum languages is its safe, automatic uncomputation, enabled by its novel annotations const and qfree. To ensure all Silq programs are physical (i.e., can be physically realized on a QRAM), we leverage one additional annotation mfree, indicating a function does not perform measurements. This allows us to detect (and thus prevent) attempts to reverse measurements and to apply measurements conditioned on quantum values.

**Error: Conditional Measurement.** Silq rejects condMeas, as it applies a measurement conditioned on quantum variable \( c \). This is not realizable on a QRAM, as the then-branch requires a physical action and we cannot determine whether or not we need to carry out the physical action without measuring the condition. However, changing the type of \( c \) to \( | \Box \) would fix this error, as conditional measurement is possible if \( c \) is classical. We note that Silq could also detect this error if measurement was hidden in a function passed to condMeas.

**Reverse.** Silq additionally also supports reversing functions, where expression reverse \((f)\) returns the inverse of function \( f \). In general, all quantum operations except measurement describe linear isometries (see §3) and are thus injective. Hence, if \( f \) is also surjective (and thus bijective), we can reverse it, meaning reverse \((f)\) is well-defined on all inputs.

**Reverse Returns Unsafe Functions.** When \( f \) is not surjective, reverse \((f)\) is only well-defined on the range of \( f \). Hence, it is the programmer’s responsibility to ensure reversed functions never operate on invalid inputs.

For example, \( y := \text{dup}(x) \) duplicates \( x \), mapping \( \sum_i y_i |v⟩_i \) to \( \sum_i y_i |v⟩_i |v⟩_i \). Thus, reverse \((\text{dup})\) \((x, y)\) operates on states \( \sum_i y_i |v⟩_i |v⟩_i |ψ⟩_i \), for which it yields \( \sum_i y_i |v⟩_i |ψ⟩_i |ψ⟩_i \), uncomputing \( y \). On other states, reverse \((\text{dup})\) is undefined. As reverse \((\text{dup})\) is generally useful for (unsafe) uncomputation, we introduce its (unsafe) shorthand forget.

When realizing a reversed function on a QRAM, the resulting program is defined on all inputs but only behaves correctly on valid inputs. For example, we can implement reverse \((\text{dup})\) \((x, y)\) by running if \( x \text{=} \text{X}(y) \) and discarding \( y \), which has unintended side-effects (due to implicit measurement) unless originally \( x \text{=} y \).

**Error: Reversing Measurement.** Silq rejects revMeas as it tries to reverse a measurement, which is physically impossible according to the laws of quantum mechanics. Thus, reverse only operates on mfree functions.

**Discussion: Annotations as Negated Effects.** We can view annotations mfree and qfree as indicating the absence of effects: mfree indicates a function does not perform a measurement, while qfree indicates the function does not introduce quantum superposition. As we will see later, all qfree functions in Silq are also mfree.

5 The Silq-Core Language Fragment

In this section, we present the language fragment Silq-core of Silq, including syntax (§5.1) and types (§5.2).

Silq-core is selected to contain Silq’s key features, in particular all its annotations. Compared to Silq, Silq-core omits features (such as the imperative fragment and dependent types) that distract from its key insights. We note that in our implementation, we type-check and simulate full Silq.

5.1 Syntax of Silq-Core

Fig. 6 summarizes the syntax of Silq-core.

**Expressions.** Silq-core expressions include constants and built-in functions (c), variables (x), measurement (measure),
and reversing quantum operations (\texttt{reverse}). Further, its if-then-else construct \texttt{if } e \texttt{ then } \epsilon_1 \texttt{ else } \epsilon_2 is syntactically standard, but supports both classical (\texttt{IF}) and quantum (\texttt{B}) condition \epsilon. Function application \texttt{e’} (\texttt{f}) explicitly takes multiple arguments. Likewise, lambda abstraction \texttt{\lambda \beta.x_1 (t_1), \ldots, x_n (t_n). e} describes a function with multiple parameters \{(x_i : \tau_i)\}_{i=1}^n \text{ of types } \{\tau_i\}_{i=1}^n, \text{ annotated by } \{\beta\}_{i=1}^n, \text{ as discussed in \S5.2 (next).}

We note that Silq-core can support typing as a built-in function \texttt{f}.

\textbf{Universality.} Assuming built-in functions \texttt{c} include \texttt{X} (enabling \texttt{CNOT} by \texttt{if } \texttt{x (y Z(x y))}) and arbitrary operations on single qubits (e.g., enabled by \texttt{rotX}, \texttt{rotY}, and \texttt{rotZ}), Silq-core is universal for quantum computation, i.e., it can approximate any quantum operation to arbitrary accuracy [21].

\subsection{5.2 Types and Annotations of Silq-Core}

Further, Fig. 6 introduces the types \rho of Silq-core.

\textbf{Primitive Types.} Silq-core types include standard primitive types, including \texttt{1}, the singleton type that only contains the element \texttt{0}, and \texttt{B}, the Boolean type describing a single qubit. We note that it is straightforward to add other primitive types like integers or floats to Silq-core.

\textbf{Products and Functions.} Silq-core also supports products, where we often write \tau_1 \times \ldots \times \tau_n, and functions, where \texttt{!} emphasizes that functions are classically known (i.e., we do not discuss superpositions of functions). Function parameters and functions themselves may be annotated by \beta_i and \alpha, respectively, as discussed shortly. As usual, \times binds stronger than \texttt{!}.

Finally, Silq-core supports annotating types as classical.

\textbf{Annotations.} Fig. 6 also lists all Silq-core annotations. Our annotations express restrictions on the computations of Silq-core expressions and functions, ensuring the physicality of its programs. For example, for quantum variable \texttt{x : B}, the expression \texttt{if } \texttt{x then f(0) else f(1)} is only physical if \texttt{f} is \texttt{mfree} (note that \texttt{x} does not appear in the two branches).

\section{Typing Rules}

In this section, we introduce the typing rules of Silq. Most importantly, they ensure that every sub-expression that is not consumed can be uncomputed, by ensuring these sub-expressions are lifted.

\textbf{Format of Typing Rules.} In Fig. 7, \Gamma ?\rightarrow e : \tau indicates an expression \texttt{e} has type \tau under context \Gamma, and the evaluation of \texttt{e} is \texttt{a} \subseteq \{\texttt{qfree}, \texttt{mfree}\}. For example, \texttt{x : B ?\rightarrow H(x) : B} for \texttt{a = {mfree}}, where \texttt{mfree} \epsilon \texttt{a} since evaluating \texttt{H(x)} does not induce a measurement, and \texttt{qfree} \epsilon \texttt{a} since the effect of evaluating \texttt{H(x)} cannot be described classically. We note that in general, \texttt{x : \tau \texttt{f (x) : \tau’} if \texttt{f} has type \texttt{r !\rightarrow r’}, i.e., the annotation of \texttt{f} determines the annotation of the turnstile !\rightarrow.

A context \Gamma is a multiset \texttt{(\beta}_{i=1}^n) \texttt{I} that assigns a type \tau_i to each variable \texttt{x_i}, where \texttt{I} is a finite index set, and \texttt{x_i} may be annotated by \texttt{const} \texttt{\epsilon} \texttt{\beta}, indicating that it will not be consumed during evaluation of \texttt{e}. As a shorthand, we often write \Gamma = \texttt{\beta}_{i=1}^n.

We write \Gamma, \texttt{\beta x : \tau for } \Gamma \uplus \texttt{\beta x : \tau}, where \uplus denotes the union of multisets. Analogously, \Gamma, \Gamma’ denotes \Gamma \uplus \Gamma’. In general, we require that types and annotations of contexts can never conflict, i.e., \texttt{\beta x : \tau } \texttt{\epsilon } \texttt{\Gamma} \texttt{and} \texttt{\beta’x : \tau’} \texttt{\epsilon } \texttt{\Gamma’ implies} \texttt{\beta = \beta’} \texttt{and} \texttt{\tau = \tau’}.

\subsection{6.1 Typing Constants and Variables}

If \texttt{c} is a constant of type \tau, its typing judgement is given by \texttt{\emptyset \texttt{nfree,qfree c : \tau}. For example, \emptyset \texttt{nfree,qfree H : B !\rightarrow B}. Here, we annotate the turnstile \texttt{!\rightarrow as qfree, because evaluating expression \texttt{H}} maps the empty state \texttt{|}\rangle to \texttt{|}\rangle \texttt{\otimes |H}\rangle, which can be described classically by \texttt{\hat{f}(|}\rangle) = |H}\rangle. We provide the types of other selected built-in functions in App. E.2.

Likewise, the typing judgement of variables carries annotations \texttt{qfree} and \texttt{mfree} (rule \texttt{var} in Fig. 8), as all constants \texttt{c} and variables \texttt{x} in Silq-core can be evaluated without measurement, and their semantics can be described classically. Further, both rules assume an empty context (for constants \texttt{c}) or a context consisting only of the evaluated variable (for variables), preventing ignoring variables from the context. To drop constant and classical variables from the context, we introduce an explicit weakening rule, discussed next.

\textbf{Weakening and Contraction.} Fig. 8 further shows weakening and contraction typing rules for classical and constant variables. These rules allow us to drop classical and constant variables from the context (weakening rules \texttt{!, W} and \texttt{W}) and duplicate them (contraction rules \texttt{\&C} and \texttt{C}). For weakening, the interpretation of "dropping variable \texttt{x}" arises from reading the rule bottom-up, which is also the way our semantics operates (analogously for contraction). In our semantics (\S7), dropping constant variable \texttt{x} in the body of a function \texttt{f} can be handled by uncomputing it at the end of \texttt{f}. 

\begin{figure}[ht]
\centering
\includegraphics[width=\columnwidth]{silq-core.png}
\caption{Syntax, types, and annotations.}
\end{figure}

\begin{figure}[ht]
\centering
\includegraphics[width=\columnwidth]{typing-judgments.png}
\caption{Typing judgments.}
\end{figure}
We note that non-constant arguments (as indicated by \( \alpha \)), as reconstructing classical components of types without classical components is again standard (App. E.1). In terms of annotations, the rule enforces multiple constraints. First, it ensures that the annotation of the abstracted function follows the annotation \( \alpha \) of the original typing judgment. Second, we tag the resulting type judgment as \( \alpha \), since function abstraction requires neither measurement nor quantum operations. Third, the rule allows capturing classical variables (\( y_i \) has type \( \alpha \)), but not quantum variables. This ensures that all functions in Silq-core are classically known, i.e., can be described by a classical state.

6.4 Lambda Abstraction

Fig. 10 shows the rule for lambda abstraction. Its basic pattern without annotations is again standard (App. E.1). In terms of annotations, the rule enforces multiple constraints. First, it ensures that the annotation of the abstracted function follows the annotation \( \alpha \) of the original typing judgment. Second, we tag the resulting type judgment as \( \alpha \), since function abstraction requires neither measurement nor quantum operations. Third, the rule allows capturing classical variables (\( y_i \) has type \( \alpha \)), but not quantum variables. This ensures that all functions in Silq-core are classically known, i.e., can be described by a classical state.

6.5 Reverse

Fig. 11 shows the type of \textit{reverse}. We only allow reversing functions without classical components in input or output types (indicated by \( \alpha \)), as reconstructing classical components of inputs is typically impossible. Concretely, types without classical components are (i) \( \alpha \), (ii) \( \mathbb{B} \), and (iii) products of types without classical components. In particular, this rules out all classical types \( \alpha \), function types, and products of types with classical components.

The input to \textit{reverse}, i.e., the function \( f \) to be reversed must be measure-free, because measurement is irreversible. Further, the function \( f \) may or may not be \textit{qfree} (as indicated by a callout). Then, the type rule for \textit{reverse} splits the input types of \( f \) into constant and non-constant ones. The depicted rule assumes the first parameters of \( f \) are annotated as constant, but we can easily generalize this rule to other
orders. Based on this separation, reverse returns a function which starts from the constant input types and the output types of f, and returns the non-constant input types. The returned function reverse(f) is measure-free, and qfree if f is qfree.

6.6 Control Flow

Even though if e then e₁ else e₂ is syntactically standard, it supports both classical and quantum conditions e. A classical condition induces classical control flow, while a non-classical (i.e., quantum) condition induces quantum control flow. In Fig. 12, we provide the typing rules for both cases, which follow the standard basic patterns when ignoring annotations (App. E.1).

Quantum Control Flow. Constraint (4) ensures that e is lifted and can thus be uncomputed after the conditional, analogously to uncomputing constant arguments in Constraint (1). Constraint (5) requires both branches to be mfree, which is important because we cannot condition a measurement on a quantum value (this would violate physicality). Further, it also requires the condition to be mfree (which is already implicitly ensured by Constraint (4) as all qfree expressions are also mfree), meaning the whole expression is mfree. Constraint (6) ensures that the resulting typing judgment gets tagged as qfree if all subexpressions are qfree. Finally, the rule does not allow the return type τ to contain classical components (indicated by f), as otherwise we could introduce unexpected superpositions of classical values.

Classical Control Flow. Classical control flow requires the condition to be classical, in addition to our usual restrictions on annotations. Concretely, Constraints (7) and (8) propagate mfree and qfree annotations.

7 Semantics of Silq-Core

In this section, we discuss the operational semantics of Silq-core. We use big-step semantics, as this is more convenient to define reverse and control flow.

7.1 Semantics of Types

We build the semantics [r] of type τ from a classical set [r]c and a quantum ground set [r]q as [r] = [r]c × H ([r]q). Note that [r] stores the classical and quantum parts of τ separately, which is in line with how a QRAM can physically store values of type τ. In particular, [r]q contains the ground set from which we build the Hilbert space H ([r]q).

Classical Set and Quantum Ground Set. Fig. 13 defines both the classical set [r]c and the quantum ground set [r]q for all possible types τ. For type ℝ, both the classical set and the quantum ground set are the singleton set {0}. The (quantum) Boolean type B stores no classical information and hence, its classical set is again the singleton set. In contrast, its quantum ground set is {0, 1}, for which H ([r]q) contains all superpositions of {0} and {1}. The sets associated with the product type are standard. Functions store no quantum information, and hence their quantum ground set is {0}.

In contrast, the classical set associated with a function type contains all expressions e of this type, and a state σ storing the variables captured in e. Finally, classical types !τ store no quantum information and hence their quantum ground set is {0}. As a straightforward consequence of our definition, duplicate classical annotations do not affect the semantics: [[!τ]] ≃ [[τ]].
We formally prove in Thm. 7.1 where classical values are in superposition. In contrast, do not put classical values in superposition, i.e., the classical and quantum part of elements \( \{ 0 \} \times B \) are handled analogously and (ii) operation \( \tau \) is not a vector space. 

\[ \text{Semantics of Expressions.} \] Fig. 15 provides the semantics of context \( [\tilde{\beta} x : \tilde{\tau}] \). Here, \( (v_i)_x \) indicates that variable \( x \) stores value \( v_i \). Fig. 14 provides semantics for an example context, where we write \( | 0 \rangle_x \) as a short-hand for \( | 0 \rangle_x \).

Analogously to \( [[r]]^\ast \), Fig. 15 also introduces the extended semantics \( [\tilde{\beta} x : \tilde{\tau}]^\ast \) for contexts, and a standard representation that stores the classical and quantum value of variable \( x \) together in a single location \( | v \rangle_x \). We use this representation throughout this work (including Fig. 3). Again, we illustrate this extended semantics in Fig. 14.

For contexts, the embedding \( \iota : [\tilde{\beta} x : \tilde{\tau}] \rightarrow [\tilde{\beta} x : \tilde{\tau}]^\ast \) is

\[ \iota((\tilde{v})_x, \sum_{(\tilde{v}')} | \tilde{v}' \rangle \langle \tilde{v}' |) = \sum_{(\tilde{v}')} | \tilde{v}' \rangle \langle \tilde{v}' | \tilde{x} \],

for \( (\tilde{v})_x \in [\tilde{\beta} x : \tilde{\tau}] \) and \( \sum_{(\tilde{v}')} | \tilde{v}' \rangle \langle \tilde{v}' | \tilde{x} \in \mathcal{H}([\tilde{\beta} x : \tilde{\tau}]^\ast) \). We illustrate this in Fig. 14 on an example context.

### 7.2 Semantics of Expressions

Our operational semantics evaluates an expression \( e \) in state \( \psi \) by constructing derivation trees whose structure follows the structure of our type derivations. Since \( e \) may contain measurements with probabilistic outcome, we provide an evaluation \( \Gamma \vdash_e : \tau | \psi \rightarrow^n \psi' \) for each possible sequence of measurement results, indicating that evaluating \( e \) (typed as \( \Gamma \vdash_e : \tau \)), on state \( \psi \) yields state \( \psi' \) with probability \( ||\psi'||^2 \).
Weakening of constant variables postpones uncomputing: we must evaluate \( x \) independently, as their values may be entangled in \( \psi' \) (meaning that the classical values in \( \psi'' \) can never be in superposition (since \( \iota \) only returns valid elements)).

**Theorem 7.1 (Type Preservation).** If \( \Gamma = \texttt{const}\ x: \tau; \bar{y}: \tau' \) then \( \psi' \) lies in \( \iota((\psi,\Delta)) \).

We provide a proof for Thm. 7.1 in App. G. Here, \( \iota((\psi,\Delta)) \) contains all elements of \( [\Gamma,\Delta]^+ \) where classical values are in nonuperposition.

### 7.4 Semantics of Annotations

In the following, we show theorems formalizing the guarantees of annotations of Silq-core expressions. We do not formally discuss the guarantees of annotations of Silq-core functions, which are analogous. We note that the guarantees of \( \iota \) were already discussed in §7.3.

**Preserving Constants.** Thm. 7.2 ensures that constant variables are indeed preserved by Silq-core.

**Theorem 7.2 (Const Semantics).** If \( \Gamma = \texttt{const}\ x: \tau; \bar{y}: \tau' \), then \( \psi' \) lies in \( \iota((\psi,\Delta)) \).

We provide a proof for Thm. 7.2 in App. G.

**Mfree Expressions.** We want to ensure that mfree expressions correspond to linear isometries, which in turn ensures we can physically implement their effect with quantum gates. However, this correspondence is non-trivial.

**Theorem 7.3 (Mfree Semantics).** If \( mfree \in \alpha, \sigma \in [\Gamma,\Delta]^c \), then \( |(\sigma,\psi_1)\rangle \rightarrow |(\sigma,\psi_1')\rangle \) for \( \psi_1'' \) in \( \mathcal{H}(\Gamma,\Delta)^c \) and \( \psi_2'' \) in \( \mathcal{H}(\Gamma,\Delta)^c \).

We provide a proof for Thm. 7.3 in App. G.
A useful interpretation of Thm. 7.3 states that \( \text{run} \) acts like an isometry on the subspace consistent with a fixed classical component \( \sigma \in [\Gamma, \Delta]^c \),

\[
\{ (\sigma, \chi) \mid \chi \in \mathcal{H} ([\Gamma, \Delta]^\theta) \} \subseteq [\Gamma, \Delta]^+.
\]

This corresponds to the intuition that in order to evaluate \( e \) on \( \psi \), we can (i) extract the classical component \( \sigma \) from \( \psi \), (ii) build a circuit \( C \) that realizes the linear isometry for this classical component and (iii) run \( C \), yielding \( \psi' \).

**Qfree Expressions.** Thm. 7.4 ensures that qfree expressions can be described by a function \( f \) on the ground sets.

**Theorem 7.4** (Qfree Semantics). If \( \Gamma \vdash e : \tau'' \) for qfree \( e \in \alpha \) and context \( \Gamma = \text{const} \tilde{x} : \tilde{t}, \tilde{y} : \tilde{t}' \), then there exists a function \( f : [\Gamma]^s \rightarrow [\text{const} \tilde{x}] : [\tilde{t}, e : \tau'']^s \) on ground sets such that

\[
\begin{array}{c}
\Gamma \vdash e : \tau'' \\
\sum_{\sigma \in [\Gamma]} \gamma_{\sigma} [\sigma] \otimes \tilde{\psi}_{\sigma} \end{array} \xrightarrow{\text{run}} \sum_{\sigma \in [\Gamma]} \gamma_{\sigma} [f(\sigma)] \otimes \tilde{\psi}_{\sigma},
\]

where \([\Gamma]^s\) is a shorthand for the ground set \([\Gamma]^c \times [\Gamma]^p\) on which the Hilbert space \([\Gamma]^s = \mathcal{H} ([\Gamma]^p)\) is defined.

We provide a proof for Thm. 7.4 in App. G.

**7.5 Physicality**

Thm. 7.5 ensures Silq-core programs can be physically realized on a QRAM. If we would change our semantics to abort on operations that are not physical, we could re-interpret Thm. 7.5 to guarantee progress, i.e., the absence of errors due to unphysical operations.

**Theorem 7.5** (Physicality). The semantics of well-typed Silq programs is physically realizable on a QRAM.

We provide a proof for Thm. 7.5 in App. G, which heavily relies on the semantics of annotations. As a key part of the proof, we show that we can compute temporary values by reversing the computation that computed them. Reversing a computation is possible on a QRAM (and supported by most existing quantum languages) by (i) producing the gates that perform this computation and (ii) reversing them.

**8 Evaluation of Silq**

Next, we experimentally compare Silq to other languages. Our comparison focuses on Q#, because (i) it is one of the most widely used quantum programming languages, (ii) we consider it to be more high-level than Cirq or QisKit, and (iii) the Q# coding contest [15, 16] provides a large collection of Q# implementations we can leverage for our comparison. To check if our findings can generalize to other languages, we also compare Silq to Quipper (§8.2).

**Table 1.** Silq compared to Q#.

<table>
<thead>
<tr>
<th>Lines of code</th>
<th>Silq</th>
<th>Q#</th>
</tr>
</thead>
<tbody>
<tr>
<td>S18</td>
<td>99</td>
<td>251</td>
</tr>
<tr>
<td>W19</td>
<td>168</td>
<td>242</td>
</tr>
<tr>
<td>Both</td>
<td>267</td>
<td>493</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Quantum primitives</th>
<th>Silq</th>
<th>Q#</th>
</tr>
</thead>
<tbody>
<tr>
<td>S18</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>W19</td>
<td>10</td>
<td>19</td>
</tr>
<tr>
<td>Both</td>
<td>10</td>
<td>22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Annotations</th>
<th>Silq</th>
<th>Q#</th>
</tr>
</thead>
<tbody>
<tr>
<td>S18</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>W19</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Both</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Low-level quantum gates</th>
<th>Silq</th>
<th>Q#</th>
</tr>
</thead>
<tbody>
<tr>
<td>S18</td>
<td>14</td>
<td>33</td>
</tr>
<tr>
<td>W19</td>
<td>23</td>
<td>54</td>
</tr>
<tr>
<td>Both</td>
<td>37</td>
<td>87</td>
</tr>
</tbody>
</table>

**Implementation.** We implemented a publicly available parser, type-checker, and simulator for Silq as a fork of the PSI probabilistic programming language [6]. Specifically, Silq’s AST and type checker are based on PSI, while Silq’s simulator is independent of PSI. Our implementation handles all valid Silq code examples in this paper, while rejecting invalid programs. We also provide a development environment for Silq, in the form of a Visual Studio Code extension.\(^\text{10}\)

Compared to Silq-core, Silq supports an imperative fragment (including automatic uncomputation), additional primitives, additional convenience features (e.g., unpacking of tuples), additional types (e.g., arrays), dependent types (which only depend on classical values, as shown in Fig. 3), type equivalences (e.g., \(!! \tau \equiv !! \tau \)), subtyping, and type conversions.

**8.1 Comparing Silq to Q#**

To compare Silq to Q#, we solved all 28 tasks of Microsoft’s Q# Summer 2018 and Winter 2019 [15, 16] coding contest in Silq. We compared the Silq solutions to the Q# reference solutions provided by the language designers [17, 18] (Tab. 1) and the top 10 contestants (App. H).

Our results indicate that algorithms expressed in Silq are far more concise compared to the reference solution (−46%) and the average top 10 contestants (−59%). We stress that we specifically selected these baselines to be written by experts in Q# (for reference solutions) or strong programmers well-versed in Q# (for top 10 contestants). We did not count empty lines, comments, import statements, namespace statements, or lines that were unreachable for the method solving the task. This greatly benefits Q#, as it requires various imports.

Because the number of lines of code heavily depends on the available language features, we also counted (i) the number of different quantum primitives, (ii) the number of different annotations in both Q# (controlled auto, adjacent self, Controlled,...) and Silq (\textit{mfree}, \textit{qfree}, \textit{const}, \textit{lifted}, and !), as well as (iii) the number of low-level quantum circuit gates used to encode all programs in Q# and Silq (for details, see App. H).

Our results demonstrate that Silq is not only significantly more concise, but also requires only half as many quantum primitives, annotations, and low-level quantum gates compared to Q#. As a consequence, we believe Silq programs are easier to read and write. In fact, we conjecture that the

\(^{10}\)https://marketplace.visualstudio.com/items?itemName=eth-sri.vscode-silq
code of the top 10 contestants was longer than the reference solutions because they had difficulties choosing the right tools out of Q#’s large set of quantum primitives. We note that Silq is better in abstracting away standard low-level quantum circuit gates: they occur only half as often in Silq.

### 8.2 Comparing Silq to Quipper

The language designers of Quipper provide an encoding [26] of the triangle finding algorithm [4, 14]. We encoded this algorithm in Silq and found that again, we need significantly less code (∼38%; Quipper: 378 LOC, Silq: 236 LOC). An excerpt of this, on which we achieve even greater reduction (∼64%), was already discussed in Fig. 2.

The intent of the algorithm in Fig. 2 is naturally captured in Silq: it iterates over all $j,k$ with $0 \leq j < k < 2^{\text{bar}}$, and counts how often $\text{ee}[\text{tau}[j]][\text{tau}[k]] \& \& \text{ee}[j] \& \& \text{ee}[k]$ is true, where we use quantum indexing into $\text{ee}$. In contrast, Quipper’s code is cluttered by explicit uncomputation (e.g., of $\text{eedd}_k$), custom functions aiding uncomputation (e.g., $\& \& \&$), and separate initialization and assignment (e.g., $\text{eedd}_k$), because Quipper lacks automatic uncomputation.

Similarly to Q#, Quipper offers an abundance of built-in and library functions. It supports 76 basic gates and 8 types of reverse, while Silq only provides 10 basic gates and 1 type of reverse, without sacrificing expressivity. Some of Quippers overhead is due to double definitions for programming in declarative and imperative style, e.g., it offers both $\text{gate}_T$ and $\text{gate}_T\_at$ or due to definition of inverse gates, e.g., $\text{gate}_T\_inv$.

### 8.3 Further Silq Implementations

To further illustrate the expressiveness of Silq on interesting quantum algorithms, we provide Silq implementations of (i) Wiesner’s quantum money scheme [34], (ii) a naive (unsuccessful) attack on it, and (iii) a recent (successful) attack on it [20] in App. H.3.

### 9 Related Work

Various quantum programming languages aim to simplify development of quantum algorithms. Tab. 2 shows the key language features of the languages most related to ours.

**Const.** To our knowledge, Silq is the first quantum language to mark variables as constant. We note that for Q#, so-called immutable variables can still be modified (unlike const variables), for example by applying the Hadamard transform $H$.

Silq’s constant annotation is related to ownership type systems guaranteeing read-only references [19]. As a concrete example, the Rust programming language supports a single mutable borrow and many const borrows [10](§4.2). However, the quantum setting induces additional challenges: only guaranteeing read-only access to variables is insufficient as we must also ensure safe uncomputation. To this end, Silq supports a combination of const and qfree.

**Qfree.** To our knowledge, no existing quantum language annotates qfree functions. ReverC’s language fragment contains qfree functions (e.g., $X$), and ReQWire’s syntactic conditions cover some qfree operations, but neither language explicitly introduces or annotates qfree functions.

**Mfree.** Of the languages in Tab. 2, only Q# can prevent reversing measurement and conditioning measurement (via special annotations). However, as Q# cannot detect implicit measurements, reverse and conditionals may still induce unexpected semantics. For other languages, reversal may fail at runtime when reversing measurements, and control may fail at runtime on conditional measurement.

We note that QWire’s reverse returns safe functions, but only when given unitary functions (otherwise, it reports a runtime error by outputting None). Thus, it for example cannot reverse $\text{dup}$, which is linearly isometric but not unitary.

**Semantics.** The semantics of Silq is conceptually inspired by Selinger and Valiron, who describe an operational semantics of a lambda calculus that operates on a separate quantum storage [28]. However, as a key difference, Silq’s semantics is more intuitive due to automatic uncomputation.

All other languages in Tab. 2 support semantics in terms of circuits that are dynamically constructed by the program.

### 10 Conclusion

We presented Silq, a new high-level statically typed quantum programming language which ensures safe uncomputation. This enables an intuitive semantics that is physically realizable on a QRAM.

Our evaluation shows that quantum algorithms expressed in Silq are significantly more concise and less cluttered compared to their version in other quantum languages.
A Comparing Silq to Quipper and QWire

Fig. 18 and Fig. 19 provide full versions of the programs shown in Fig. 2.

B Grover’s Algorithm

Fig. 20 shows an implementation of Grover’s algorithm, including Grover’s diffusion operator in Silq.

C Uncomputing Non-Qfree Expressions

Here, we show why uncomputing the condition in function nonQfree in Fig. 5 is not possible (in particular also not by following Bennett’s construction). Fig. 22a provides a rewritten version of nonQfree that makes its individual operations more explicit.

Without uncomputation, nonQfree produces x (implicitly duplicated before applying H), a modified y, and a temporary control t, hence uncomputation should remove t without uncomputing x or the modified y.

The most natural way to try to uncompute t is running Bennett’s construction by (i) running nonQfree, (ii) duplicating the modified y, and (iii) reversing nonQfree. However, this would result in x, the original y, and the modified y, instead of just x and the modified y.

Fig. 22 shows that more generally, dropping t from the state is unphysical. Specifically, dropping t from the state (which is the goal of correct uncomputation) can result in the invalid state 0.

D Notational Conventions

Fig. 23 summarizes the notational conventions used in this work.

E Typing Rules

In the following, we provide additional information on typing rules of Silq-core.

E.1 Basic Pattern of Typing Rules

Fig. 25 shows the basic patterns of our typing rules without annotations.

E.2 Types of Selected Built-in Functions

Fig. 24 shows the type of some built-in functions. In the following, we only discuss its most interesting aspects.
\[
\frac{1}{\sqrt{2}} |1\rangle_x |0\rangle_y + \frac{1}{\sqrt{2}} |1\rangle_x |1\rangle_y
\]

\[
\frac{1}{\sqrt{2}} |1\rangle_x |0\rangle_y |1\rangle_t + \frac{1}{\sqrt{2}} |1\rangle_x |1\rangle_y |1\rangle_t
\]

\[
\frac{1}{\sqrt{2}} |1\rangle_x |0\rangle_y |1\rangle_t + \frac{1}{\sqrt{2}} |1\rangle_x |1\rangle_y |1\rangle_t
\]

\[
\frac{1}{2} |1\rangle_x |0\rangle_y |0\rangle_t - \frac{1}{2} |1\rangle_x |0\rangle_y |1\rangle_t + \frac{1}{2} |1\rangle_x |1\rangle_y |0\rangle_t - \frac{1}{2} |1\rangle_x |1\rangle_y |1\rangle_t
\]

\[
\frac{1}{2} |1\rangle_x |0\rangle_y - \frac{1}{2} |1\rangle_x |1\rangle_y + \frac{1}{2} |1\rangle_x |1\rangle_y - \frac{1}{2} |1\rangle_x |0\rangle_y = 0
\]

**Figure 20.** Grover’s diffusion operator in Silq.

**Figure 21.** Period Finding and Quantum Fourier Transform in Silq.

**Figure 22.** Semantics of non\$Q\$free on input \(\frac{1}{\sqrt{2}} |1\rangle_x |0\rangle_y + \frac{1}{\sqrt{2}} |1\rangle_x |1\rangle_y\) when uncomputing the condition.

**Figure 23.** Notational conventions used in this work.
In the following, we provide formal semantics for Silq-core expressions.

**Constants, Variables.** Fig. 26 first shows the rule for constants, which adds the constant to the state. Then, it shows the rules for variables. For consumed variables, \( \mathbb{I}_{x \rightarrow x} \) renames \( x \) to \( x \) in \( \psi \) without affecting other variables in \( \psi \) (shortly discussed in more detail). In contrast, the rule for constant variables preserves \( x \) and introduces an explicit duplicate \( x \) by \([\text{dup}]\), which maps \(|v\rangle\) to \(|v, v\rangle\) (cp. Fig. 33).

**Operating on Named States with Context.** We provide a more detailed example demonstrating the effect of subscript “\(x \rightarrow x\)” in Fig. 31, which shows how to apply \([\mathbf{X}]\) to state \(|b\rangle \otimes |\psi\rangle\), where the formal definition of \([\mathbf{X}]\) is \([\mathbf{X}](|b\rangle) = |1 - b\rangle\) (see App. F.2).

Here, the subscript \(x \rightarrow y\) of \(\mathbf{X}\) ensures that we (i) preserve \(|\psi\rangle\) (cp. Eq. (10–11)) and (ii) run \(\mathbf{X}\) on \(|b\rangle\) and name the output \(\gamma\) (cp. Eq. (12)).

Here, it is crucial that we assume the standard representation introduced in Fig. 15, which ensures that classical and quantum components of variable \(x\) are stored together as \(|(\psi, \psi')\rangle\). As a consequence, we know that if \(\mathbb{I}\) contains one or more occurrences of \(x\), these represent duplicates of \(x\), as opposed to classical or quantum components of \(x\).

**Contraction, Weakening.** Next, Fig. 27 shows the semantics of contraction and weakening.

If the weakening rule drops a classical variable \(x\) from the context (rule \(!W\)), the semantics drops \(x\) from the state, using drop\((\cdot)^x\) \(|\psi\rangle = |\psi_{\neg x}\rangle\). If the context contains multiple occurrences of \(x\), only the first occurrence of \(x\) is dropped.

If the rule drops a constant variable (rule \(W\)), the semantics ignores this. Instead, it waits until the end of the function to uncompute all constant variables.

The contraction rule for classical variables (rule \!'C) duplicates the contracted variable \(x\). In contrast, the contraction rule for quantum variables (rule \(C\)), duplicating the contracted variable \(x\), and removes the duplicate after evaluating \(e\). This removal of duplicates is not needed for constant variables, as only constant variables are preserved after their last usage.

**Function Calls.** The first rule in Fig. 28 shows the semantics of a generic function call \(e'(\bar{e})\). First, the rule evaluates all arguments, resulting in state \(\psi_n\). Second, the rule evaluates \(e'\), resulting in state \(\psi_{n+1}\) containing the function \(e''\) to be evaluated, which may capture variables \(\sigma\). We note that the rule implicitly assumes that the function to be evaluated is classically known — a property guaranteed by our type system. Third, it evaluates the function using a transition rule of the form \(|e''(\bar{e})\rangle \sigma |\psi_{n+1}\rangle\xrightarrow{\text{eval}}|\psi_{n+2}\rangle\). In contrast to run-transitions, eval-transitions assume that all arguments \(\bar{e}\) are already evaluated in \(\psi_{n+1}\) (as guaranteed by run-transitions).

Finally, the rule drops the \(\text{const}\) arguments of \(e''\) by uncomputing them, and renames the output value from \(\text{ret}\) to \(e'(\bar{e})\)
Figure 26. Semantics of constants and variables.

\[
\begin{array}{ll}
\text{eval arguments} & \quad \text{determine function} & \quad \text{captured in } e'' \\
\Gamma, \Gamma' \vdash e_1 : \tau & \quad \Gamma' \vdash e' : \tau & \quad \Gamma' \vdash e' : \tau \\
\psi_1 = \psi' & \quad \psi_n = \psi' & \quad \psi_n = \psi' \\
\end{array}
\]

Figure 27. Semantics of contraction and weakening.

\[
\begin{array}{ll}
\text{evaluate arguments} & \quad \text{evaluate function} & \quad \text{Name result} & \quad \text{Uncompute} \\
\Gamma, \Gamma' \vdash e_1 : \tau & \quad e''(\xi) : \Gamma' \vdash e' : \tau & \quad \psi' \quad \psi'' & \quad \psi'' \\
\psi_1 = \psi' & \quad \psi_n = \psi' & \quad \psi_n = \psi' & \quad \psi_n = \psi' \\
\end{array}
\]

Figure 28. Semantics of function calls.

\[
\begin{array}{ll}
\text{measure} & \quad \text{built-in eval} \\
\text{measure} & \quad \text{call-rev} \\
\end{array}
\]

Figure 29. Semantics of reverse.

\[
\begin{array}{ll}
\text{reverse} & \quad \text{re-q} \\
\text{reverse} & \quad \text{else} \\
\end{array}
\]

Figure 30. Semantics of control flow. The rule is analogous for \( e_\text{c} : 1 \# B \).
\[
\left( \left[ x \rightarrow y \right] \right) (b) x \otimes |\tilde{w}⟩_2 \quad (9)
\]

\[
= \left[ [x \rightarrow y] \left( |b⟩x \otimes |\tilde{w}⟩_2 \right) \right] \quad (10)
\]

\[
= \left[ [x \rightarrow y] \left( |b⟩x \right) \otimes |\tilde{w}⟩_2 \right] \quad (11)
\]

\[
\left( \left[ (b) \right] \right) y \otimes |\tilde{w}⟩_2 \quad x \rightarrow y \quad (12)
\]

\[
= [1 - b] y \otimes |\tilde{w}⟩_2 \quad x \quad (13)
\]

**Eval-transitions.** All remaining rules in Fig. 28 are evaltransitions. The rules modify their input state \( \psi \) according to the called function, and then store the result in \( \text{ret} \).

**Measurement.** The rule for measurement selects one possible measurement \( \psi’ \) and collapses the state to this value. Note that measurement allows multiple transitions, one for each possible measured value \( \psi’ \). Here, and in various other eval-transitions, the state of captured variables is \( \sigma = \emptyset \), as measurement cannot capture variables.

**Built-in Functions.** The rule for evaluating built-in functions \( c \) relies on the semantics \( \llbracket c \rrbracket \) of these functions, as discussed in App. F.2. The subscript to \( \llbracket c \rrbracket \) ensures that the function operates on input values named \( e_1, \ldots, e_n \), and names the output values \( e_i \) (if the \( i \)-th argument of \( c \) is \texttt{const}) and \( \text{ret} \) (to indicate the return value).

**Evaluating Lambda Abstraction.** The last rule in Fig. 28 evaluates a lambda abstraction. First, it adds the variables captured in \( e’’ \) to the current state \( \psi \). Second, it renames the values of the evaluated arguments to the names of the parameters of \( e’’ \). Third, it runs \( e’’ \) on the resulting state, obtaining \( \psi’ \). Finally, it resets the variable names of constant parameters to \( e’’ \) back to \( e_i \) and names the return value \( \text{ret} \).

**Reverse.** We show the semantics of \texttt{reverse} in Fig. 29. Expression \texttt{reverse}(e) does not immediately reverse \( e \) (which evaluates to function \( e_{\text{func}} \), but instead records that \( e_{\text{func}} \) should be reversed, by storing \texttt{reverse}(e_{\text{func}}) and the state \( \sigma \) captured by \( e_{\text{func}} \) under \( \text{ret} \).

The actual reversal is performed upon a call to the reversed function, also shown in Fig. 29. Here, we explicitly split the arguments into \( \tilde{e}’ \) (the \texttt{const} arguments) and \( \tilde{e}’’ \) (the non-\texttt{const} arguments), as assumed by Fig. 11. Intuitively, rule call-reversed maps \( \psi \) to \( \psi’ \), if running \( e_{\text{func}} \) on \( \psi’ \) yields \( \psi’' \). However, it must also account for naming mismatches: Running \( e_{\text{func}} \) on \( \psi’' \) yields \( \text{ret} \) instead of \( \tilde{e}’’ \), and the name of the returned value must be \( \text{ret} \).

We note that it is possible that there is no \( \psi’’ \) satisfying the premise of call-reversed, when \( e_{\text{func}} \) is not surjective. In this case, \texttt{reverse}(e_{\text{func}}) is undefined on \( \psi’ \), which intuitively happens if \( \psi’ \) is not in the range of \( e_{\text{func}} \).

For \( f : n \text{ const } \epsilon_i x \times m \text{ const } \epsilon_j \mapsto r_{ij} \) we have \( \llbracket f \rrbracket : \epsilon_i \times \epsilon_j \mapsto r_{ij} \) with \( r_{ij} = \alpha (\epsilon_i \times \epsilon_j) \) for \( i = 1, \ldots, n \) and \( j = 1, \ldots, m \).

(a) Semantics of a general built-in function \( f \).

\[
\llbracket f \rrbracket : \epsilon_i \times \epsilon_j \mapsto r_{ij}
\]

(b) Semantics of \( X \).

**Control Flow.** Fig. 30 shows the semantics of control flow, handling both classical and quantum control flow. The rule (i) evaluates condition \( e \) and (ii) splits the resulting state into two states based on the value of \( e \). Then, it evaluates \( e_1 \) in the first state and \( e_2 \) in the second. Finally, it adds both resulting states and drops \( e \) from the state.

**F.2 Semantics of Built-in Functions**

Fig. 32a shows the semantic space of built-in functions \( f \) in terms of partial linear functions \( \llbracket f \rrbracket \), where being a partial function allows us to support undefined behavior for some inputs.

Note that the function space of \( \llbracket f \rrbracket \) in principle admits functions (i) violating \texttt{const} by modifying constant arguments and even (ii) violating the rules of quantum physics as in \( a |0⟩ + \beta |1⟩ \mapsto (α + β) |0⟩ \). Thus, we must ensure that these violations do not occur for the built-in functions defined by Silq-core.

As an example, Fig. 32b shows the semantics of \( X \) on basis states (Eq. (14)), the quantum semantics are given by linear extension (Eq. (15)). For simplicity, the semantics in Fig. 32a (i) operates on states with unnamed indices and (ii) does not take context into account. However, our operational semantics operates on states with named indices involving context. Fig. 31 shows how to bridge this gap when applying \( X \) to state \( |b⟩x \otimes |\tilde{w}⟩_2 \). The subscript \( x \rightarrow y \) of \( X \) ensures (i) we preserve \( |\tilde{w}⟩_2 \) (cp. Eq. (10–11)) and (ii) we run \( X \) on \( |b⟩x \) and name the output \( y \) (cp. Eq. (12)).

**Semantics of Selected Built-in Functions.** Fig. 32 shows the semantics of selected built-in functions in Silq-core.

The semantics of \texttt{forget}(· = ·) is only defined if its two arguments evaluate to the same value.

**F.3 Semantics Example**

We provide an example semantic derivation tree in Fig. 34. It demonstrates weakening, contraction, and function evaluation.
We recall all theorems presented in §7 in the following.

![Figure 33](image-url)

**Theorem 7.1 (Type Preservation)**

\[ \text{if } \Gamma \vdash t : \tau' \text{ then } \psi \vdash t : \tau' \]

**Theorem 7.2 (Const Semantics)**

\[ \text{if } \Gamma \vdash t : \tau' \text{ then } \psi \vdash t : \tau' \]

**Theorem 7.3 (Mfree Semantics)**

\[ \text{if } \text{mfree} \in \alpha, \sigma \in [\Gamma, \Delta]^c, \]

\[ \Gamma \vdash e : \tau'' \text{ then } \psi \vdash e : \sigma \]

**Theorem 7.4 (Qfree Semantics)**

\[ \text{if } \Gamma \vdash e : \tau'' \text{ for qfree } \in \alpha \text{ and context } \Gamma \vdash \text{const} \bar{\sigma} \vdash \tau' \]

\[ \text{then there exists a function } \tilde{f} : [\Gamma]^p \rightarrow [\Gamma]^q \text{ on ground sets such that} \]

\[ \Gamma \vdash e : \tau'' \text{ then } \psi \vdash e : \sigma \text{ and } \tilde{f} \text{ on ground sets} \]

\[ \text{where } [\Gamma]^p \text{ is a shorthand for the ground set } [\Gamma]^p \times [\Gamma]^q \text{ on which the Hilbert space } [\Gamma]^p = \mathcal{H}(\Gamma)^p \text{ is defined.} \]

We will prove a different formulation of this theorem to improve presentation. Because of Thm. 7.2, we know that the constant part of \( \Gamma \) is preserved, hence it suffices to prove that there exists a function \( \tilde{f} : [\Gamma, \Delta]^p \rightarrow [\Gamma]^q \) such that

\[ \psi = \sum_{\bar{\sigma}} Y_{\bar{\sigma}} (|\tilde{\psi}\rangle) \otimes [\psi]_{\bar{\sigma}} \]

gets mapped to

\[ \psi' = \sum_{\bar{\sigma}} Y_{\bar{\sigma}} (|\tilde{\psi}\rangle) \otimes [\psi']_{\bar{\sigma}} \]

**Theorem 7.5 (Physicality)**

The semantics of well-typed Silq programs is physically realizable on a QRAM.

We use the following helper lemma to prove Thm. 7.5.

**Lemma G.1.** *Any well-typed mfree expression \( e \) can be implemented on a QRAM which maps \( \psi \in i([\Gamma, \Delta]) \) to \( \psi' \) if \( \Gamma \vdash e : \tau' \rightarrow \psi' \) run \( \Gamma \).

Proof. Let \( \Gamma = \text{const} \bar{\sigma} \vdash \bar{\tau}, \bar{\delta} : \bar{\tau}' \) and \( \sigma \in [\Gamma, \Delta]^c \). From Thm. 7.3, we know that there exists a linear isometry \( M_{\sigma} : \mathcal{A} \rightarrow i([\text{const} \bar{\sigma} : \bar{\tau}, \bar{\delta} : \tau, \Delta]) \),

where \( \mathcal{A} := \{i(\sigma, \tilde{\psi}) \mid \tilde{\psi} \in \mathcal{H}(\Gamma, \Delta)^{\Delta}\} \). Hence, given \( \psi \), a QRAM can (i) extract the classical components of \( \psi \), (ii) determine \( M_{\sigma} \) based on those classical components \( \sigma \), and (iii) run \( M_{\sigma} \) on \( \psi \), yielding \( \psi' \). \( \square \)

**G2 Proofs for Run**

To improve presentation, we prove all theorems simultaneously in one large inductive proof. In the following, we discuss each semantic rule, e.g., the rules in Fig. 26. For each rule, we will mark the part for type-preservation (Thm. 7.1) by \([T]\), the part for preserving constants (Thm. 7.2) by \([C]\), the part for mfree expressions (Thm. 7.3) by \([M]\), the part for qfree expressions (Thm. 7.4) by \([Q]\), and the part for physicality (Thm. 7.5) by \([P]\).

**G2.1 [const].** The rule

\[ \theta \vdash c : \tau \quad \psi \quad \text{run} \quad \psi \otimes |c\rangle \]

maps \( \psi \) to \( \psi \otimes |c\rangle \).

\[ [T] \text{ Since } \Gamma = \emptyset \text{ we have that } \psi \in i([\Delta]) \text{. Hence we have immediately } \psi' = \psi \otimes |c\rangle \in i([\tau, \Delta]) \]

\[ [C] \text{ Since } \Gamma = \emptyset \text{ we have that } \psi = \tilde{\psi} \text{. Hence we have immediately } \psi' = \tilde{\psi} \otimes \chi = \psi \otimes |c\rangle \chi, \text{ where } \chi = |c\rangle \chi \]

\[ [M] \text{ We have } \]

\[ (\psi_1 \otimes |c\rangle \chi) (\psi_2 \otimes |c\rangle \chi) = \psi_1 \psi_2, \]

where \( \psi_1 \psi_2 \) denotes the inner product \( \langle \psi_1 | \psi_2 \rangle \).

\[ [Q] \text{ Function } f(\cdot) = c \text{ has the correct behavior.} \]

\[ [P] \text{ A QRAM can prepare prepare state } c \text{ in variable } c. \]
\[ \begin{align*}
\Gamma &\vdash x: B, x: x, \text{var} \\
\Delta &\vdash x: B^\alpha, x: x, \text{var} \\
\Theta &\vdash \text{const} \times \text{const} \vdash x: (0)_x (0)_x (0)_y (0)_y \quad \text{run} \quad (0)_x (0)_x (0)_y (0)_y \\
\end{align*} \]

(a) Subtrees of full semantic derivation tree (provided separately due to space constraints).

Assuming: \([\ | \ ](a) (b) = [a] (b) [a] [b])

\[ \begin{align*}
\alpha &\vdash [\ | \ ](x, y): [\ | \ ] \times [\ | \ ] \vdash \text{const} \times \text{const} \vdash x: (0)_x (0)_x (0)_y (0)_y \quad \text{run} \quad (0)_x (0)_x (0)_y (0)_y \\
\end{align*} \]

(b) Full semantic derivation tree for \([\ | \ ] x: B, \text{const} y: B^\alpha x \vdash [\ | \ ] x: B \).

\[ \begin{align*}
\alpha &\vdash \text{const} x: B \vdash [\ | \ ] \times [\ | \ ] \vdash \text{const} \times \text{const} \vdash \text{func-eval} \quad \text{run} \quad [\ | \ ] \times [\ | \ ] \times [\ | \ ] \times [\ | \ ] \\
\end{align*} \]

(c) Type derivation tree for \([\ | \ ] x: B, \text{const} y: B^\alpha x \vdash [\ | \ ] x: B \).

Figure 34. Semantics of \([\ | \ ] x: B, \text{const} y: B^\alpha x \vdash [\ | \ ] x: B \) on input state \((0)_x (1)_y \). Here, \(\alpha = \text{qfree, mfree} \) and gray parts of states correspond to the additional context \(\Delta\).

**G.2.2 [var].** The rule
\[
\begin{array}{c}
\text{x: } \tau \vdash x: \tau \\
\psi \quad \text{run} \quad \text{eval} \psi
\end{array}
\]
maps \(\psi = \sum_y y \cdot \langle \cdot \rangle_x \otimes \tilde{\psi}_w\) to \(\psi' = \sum_y y \cdot \langle \cdot \rangle_x \otimes \tilde{\psi}_w\).

**[T]** Since \(\psi \in \iota(\langle x: \tau, \Delta \rangle)\), we have that \(\psi' \in \iota(\langle x: \tau, \Delta \rangle)\).

**[C]** The claim follows immediately.

**[M]** We have
\[
\sum_y y \cdot \langle \cdot \rangle_x \otimes \tilde{\psi}_w = \sum_y y \cdot \langle \cdot \rangle_x \otimes \tilde{\psi}_w
\]

**G.2.3 [var-const].** The rule is
\[
\begin{array}{c}
\text{const } \times \text{const} \vdash x: B \vdash x: x, \text{var-const} \\
\psi \quad \text{run} \quad \text{dup} \psi
\end{array}
\]

maps \(\psi = \sum_y y \cdot \langle \cdot \rangle_x \otimes \tilde{\psi}_w\) to \(\psi' = \sum_y y \cdot \langle \cdot \rangle_x \otimes \tilde{\psi}_w\).

**[Q]** Function \(\bar{f}(\cdot) = \cdot\) has the correct behavior.

**[P]** A QRAM can simply rename variable \(x\) to \(x\).

**G.2.3 [var-const].** The rule
\[
\begin{array}{c}
\text{const } \times \text{const} \vdash x: B \vdash x: x, \text{var-const} \\
\psi \quad \text{run} \quad \text{dup} \psi
\end{array}
\]

maps \(\psi = \sum_y y \cdot \langle \cdot \rangle_x \otimes \tilde{\psi}_w\) to \(\psi' = \sum_y y \cdot \langle \cdot \rangle_x \otimes \tilde{\psi}_w\).

**[Q]** Function \(\bar{f}(\cdot) = \cdot\) has the correct behavior.

**[P]** A QRAM can run the linear isometry \(\text{dup}\).
The general form for $\psi \in \iota(\llbracket \Gamma, x: \tau, \Delta \rrbracket)$ is

$$
\psi = \sum_{\bar{v}, \bar{w}} \lambda_{\bar{v}, \bar{w}} |\bar{v}\rangle_2 \otimes |\bar{w}\rangle_2 \otimes \tilde{\psi}_{\bar{v}, \bar{w}},
$$

where $\Gamma = \text{const } \bar{x}, \bar{v}, \bar{w}$. Note that

$$
drop^{(x)}(\psi_1) \downarrow drop^{(x)}(\psi_2) = \psi_1^\dagger \psi_2.
$$

Further, because $x: \tau$, $\psi_1 = |v_1\rangle_x \otimes drop^{(x)}(\psi_1)$ and similarly $\psi_2 = |v_2\rangle_x \otimes drop^{(x)}(\psi_2)$. Thus the claim

$$
\psi_1^\dagger \psi_2 = drop^{(x)}(\psi_1)^\dagger drop^{(x)}(\psi_2) = \psi_1^\dagger \psi_2.
$$

By the induction hypothesis, we know that

$$
drop^{(x)}(\psi_1)^\dagger drop^{(x)}(\psi_2) = \psi_1^\dagger \psi_2.
$$

Hence, because $\psi_1^\dagger \psi_2$ satisfies

$$
\psi' = \sum_{\bar{v}, \bar{w}} \lambda_{\bar{v}, \bar{w}} |\bar{v}\rangle_2 \otimes |\bar{w}\rangle_2 \otimes \tilde{\psi}_{\bar{v}, \bar{w}},
$$

therefore $\tilde{f}(v, \bar{v}, \bar{w})$ suffices.

A QRAM can remove $x$ from consideration (which has the semantics of $drop^{(x)}(\cdot)$ for classical values), and then compute $\psi'$ by the induction hypothesis.

The general form for $\psi \in \iota(\llbracket \Gamma, \text{const } x: \tau, \Delta \rrbracket)$ is

$$
\psi = \sum_{\bar{v}, \bar{w}} \lambda_{\bar{v}, \bar{w}} |\bar{v}\rangle_2 \otimes |\bar{w}\rangle_2 \otimes \tilde{\psi}_{\bar{v}, \bar{w}},
$$

where $\Gamma = \text{const } \bar{x}, \bar{v}, \bar{w}$.

We can apply the induction hypothesis by considering $\text{const } x: \tau$ as part of the remainder $\Delta' = \text{const } x: \tau, \Delta$, yielding $\psi' \in \iota(\llbracket \text{const } \bar{x}, \bar{v}, \bar{w}, \tau', \Delta \rrbracket)$, hence the claim.

Similarly, this claim follows immediately by applying the induction hypothesis after grouping $|v\rangle_x$ with $\tilde{\psi}_{\bar{v}, \bar{w}}$.

The induction hypothesis immediately yields the claim.

Here $\tilde{f}(v, \bar{v}, \bar{w}) := \tilde{f}'(v, \bar{v}, \bar{w})$, where $\tilde{f}'$ is the function from the induction hypothesis.

A QRAM can compute $\psi'$ by the induction hypothesis.

The general form for $\psi \in \iota(\llbracket \Gamma, x: \tau, \Delta \rrbracket)$ is

$$
\psi = \sum_{\bar{v}, \bar{w}} \lambda_{\bar{v}, \bar{w}} |\bar{v}\rangle_2 \otimes |\bar{w}\rangle_2 \otimes \tilde{\psi}_{\bar{v}, \bar{w}}.
$$

Thus, the induction hypothesis yields

$$
\psi' \in \iota(\llbracket \text{const } x: \tau, \text{const } \bar{x}, \bar{v}, \bar{w}, \tau', \Delta \rrbracket).
$$

The claim follows by applying $\text{drop}^{(x)}(\cdot)$ to $\psi'$.

A straightforward calculation and the induction hypothesis yield the claim.

Similar to before, $\tilde{f}(v, \bar{v}, \bar{w}) := \tilde{f}'(v, \bar{v}, \bar{w})$, where $\tilde{f}'$ is the function from the induction hypothesis.

A QRAM can duplicate $x$ using the linear isometry $[\text{dup}]$, yielding a state of the form $\sum_v |v\rangle_x \otimes \tilde{\psi}_v$. By the induction hypothesis, the QRAM can then compute $\psi'$ of the form $\psi' = \sum_v \lambda_v |v\rangle_x \otimes \chi_v$ (Thm. 7.2). Hence, reversing $\text{dup}$ yields

$$
[\text{dup}]^{-1}_{\text{dup}}(\psi') = \sum_v \lambda_v |v\rangle_x \otimes \chi_v = \text{drop}^{(x)}(\psi').
$$
G.2.8 [ite]. The rule is depicted in Fig. 30.

[T] We first consider quantum control flow ($e_c : \mathbb{B}$).

Using the induction hypothesis, we know that the state after evaluating the condition is

$$\psi' \in i\left(\left[\Gamma_c, e_c : \mathbb{B}, \Gamma, \Delta\right]\right),$$

hence we can write

$$\psi' = \psi_t \otimes |1\rangle_{\mathsf{e}_c} + \psi_f \otimes |0\rangle_{\mathsf{e}_c}.$$

Next we show that $I_{\mathsf{e}_c} \rightarrow (\psi_t') + I_{\mathsf{e}_c} \rightarrow (\psi_f')$ is in

$$i\left(\left[\mathsf{const} \ x_c : \Gamma_c, e_c : \mathbb{B}, \Gamma, \Delta\right]\right).$$

The induction hypothesis yields that

$$\psi_t' \in i\left(\left[\mathsf{const} \ x_c : \Gamma_c, e_c : \mathbb{B}, \Gamma, \Delta\right]\right),$$

$$\psi_f' \in i\left(\left[\mathsf{const} \ x_c : \Gamma_c, e_c : \mathbb{B}, \Gamma, \Delta\right]\right).$$

Because $\tau$ does not have any classical components, the classical components of $\psi_t'$ and $\psi_f'$ coincide, hence they can be added. This yields, after renaming, the claim.

Next we consider classical control flow ($e_c : !\mathbb{B}$). It is clear that $\psi'$ originating from

$$\left[\Gamma_c, e_c : !\mathbb{B}\right] \xrightarrow{\text{run}} \psi',$$

where $\psi \in i(\left[\Gamma_c, \Gamma, \Delta\right])$, can be written as

$$\psi' = \psi_t \otimes |1\rangle_{\mathsf{e}_c} + \psi_f \otimes |0\rangle_{\mathsf{e}_c} \in i\left(\left[\mathsf{const} \ x_c : \Gamma_c, e_c : !\mathbb{B}, \Gamma, \Delta\right]\right).$$

We assume w.l.o.g. $e_c$ evaluates to true, thus the $\psi_f$ part has amplitude 0.

The induction hypothesis yields that

$$\psi_t' \in i\left(\left[\mathsf{const} \ x_c : \Gamma_c, e_c : \mathbb{B}, \Gamma, \Delta\right]\right),$$

which is exactly what we would like to have after renaming $e_f$ to $e$. The second summand can be neglected due to having 0 amplitude.

[C] From the induction hypothesis for $e_c$, we know that

$$\psi_t \otimes |1\rangle_{\mathsf{e}_c} + \psi_f \otimes |0\rangle_{\mathsf{e}_c}$$

are of the correct form. Thus, due to the induction hypotheses for $e_t$ and $e_f$, $\psi_t' + \psi_f'$ is of the correct form. This proves the claim, up to renaming of variables.

[M] First, we consider quantum control flow. The induction hypothesis on $e_c$ yields that

$$\psi_t' \psi_f' = (\psi_t' \otimes |1\rangle_{\mathsf{e}_c} + \psi_f' \otimes |0\rangle_{\mathsf{e}_c}) (\psi_t \otimes |1\rangle_{\mathsf{e}_c} + \psi_f \otimes |0\rangle_{\mathsf{e}_c})$$

$$\psi_t' \psi_f = \psi_t' \psi_f ',$$

hence we can write

$$\psi_1 \otimes |1\rangle_{\mathsf{e}_c} + \psi_2 \otimes |0\rangle_{\mathsf{e}_c}.$$

The induction hypothesis on $e_t$ and $e_f$ yields that

$$\psi_t' \psi_f' = \psi_t' \psi_f ',$$

$$\psi_t' \psi_f = \psi_t' \psi_f '.$$
[P] For quantum control flow $(e_c : \mathbb{B})$, expression

$$\text{if } e_c \text{ then } e_t \text{ else } e_f$$

is $\text{mfree}$ (ensured by our type system), hence Lem. G.1 applies.

For classical control flow $(e_c : \mathbb{B})$, a QRAM can first evaluate $e_c$ (by the induction hypothesis). Then, as $e_c : \mathbb{B}$, by Thm. 7.1, its value is classical, meaning the QRAM can classically determine what this value is, and run the appropriate branch (by induction hypothesis). This yields the correct state up to renaming of variables.

G.3 Proofs for Eval

In order to prove our theorems for rules involving $\frac{\text{eval}}{}$, we strengthen our theorems to also cover the following:

For $\tilde{e} = \tilde{\epsilon}, \tilde{\epsilon}$, split into constant and non-constant arguments, assume

$$\left[ e'(\tilde{e}) : \prod_{i=1}^n a_i t_i \xrightarrow{\alpha} t' \right] : \Gamma \mid \psi \xrightarrow{\text{eval}} \psi',$$

where the general form of $\psi$ is

$$\psi = \sum_{\tilde{v}, \tilde{w}} Y_{\tilde{v}, \tilde{w}} \otimes |\tilde{v}\rangle_{\tilde{v}} \otimes |\tilde{w}\rangle_{\tilde{w}} \otimes \tilde{\psi}_{\tilde{v}, \tilde{w}}.$$

Then, we have the following:

[T] If $\psi \in (\tilde{e} : \tilde{\tau}, \tilde{\pi}, \Delta)$, then

$$\psi' \in (\tilde{e}' : \tilde{\tau}', \tilde{\pi}' : \tilde{\tau}', \tilde{\tau}', \Delta').$$

[C]

$$\psi' = \sum_{\tilde{v}, \tilde{w}} Y_{\tilde{v}, \tilde{w}} \otimes |\tilde{v}\rangle_{\tilde{v}} \otimes |\tilde{w}\rangle_{\tilde{w}} \otimes \tilde{\psi}_{\tilde{v}, \tilde{w}}.$$

[M] If $\text{mfree} \in \alpha, \rho \in (\tilde{e} : \tilde{\tau}, \tilde{\pi}, \Delta)^{\circ}$,

$$\Gamma \tilde{e} : \tilde{\tau}' \mid \sigma : \Gamma \xrightarrow{\text{eval}} \psi'_1$$

for $\psi_1 \in \mathcal{H}(\tilde{e} : \tilde{\tau}, \tilde{\pi}, \Delta)$ and

$$\Gamma \tilde{e} : \tilde{\tau}' \mid \sigma : \Gamma \xrightarrow{\text{eval}} \psi'_2$$

for $\psi_2 \in \mathcal{H}(\tilde{e} : \tilde{\tau}, \tilde{\pi}, \Delta)$, then it holds that

$$\psi'_1 \psi_2 = \psi'_2 \psi_1.$$

[Q] If $\text{qfree} \in \alpha$, then there exists $\tilde{f} : \tilde{\tau} \rightarrow \tilde{\tau}'$, such that

$$\psi' = \sum_{\tilde{v}, \tilde{w}} Y_{\tilde{v}, \tilde{w}} \otimes |\tilde{v}\rangle_{\tilde{v}} \otimes |\tilde{w}\rangle_{\tilde{w}} \otimes \tilde{\psi}_{\tilde{v}, \tilde{w}}$$

where $\tilde{f}$ can depend on $\sigma$.

[P] Then there exists a QRAM implementing this, i.e., maps input $\psi \otimes |e'\rangle, \sigma \in \tilde{\tau}$ to the correct output $\psi'$.

G.3.1 [built-in-eval]. We require that all built-ins behave correctly, thus no further reasoning is needed.

G.3.2 [measure]. The rule is provided in Fig. 28. The general form of $\psi$ is

$$\psi = \sum_w y_w |w\rangle_c \otimes \hat{\psi}_w$$

[T] Let $w' \in \tilde{\tau} \rightarrow \tilde{\tau}'$. Then immediately

$$\psi' = y_w' |w\rangle_c \otimes \hat{\psi}_w' \in (\text{ret} : !r, \Delta).$$

[C] The claim follows immediately from the semantics of measure.

[M] Nothing to prove as measure is not mfree.

[Q] Nothing to prove as measure is not qfree.

[P] Measuring the appropriate value yields the correct semantics.

G.3.3 [rev].

[T] We see that

$$\text{reverse}(e_{\text{func}}) : \prod_{i=1}^n \text{cons} t_i \times \prod_{k=1}^m \text{mfree, } \alpha \xrightarrow{\text{rev}} \prod_{j=1}^m t_j',$$

hence the claim.

[C] The claim follows immediately from the semantics of reverse.

[M] The classical components of $\psi_1$ and $\psi_2$ coincide, hence in particular the expression $e$ and the captured values $\sigma$ coincide. The non-classical part of $\psi_1$ and $\psi_2$ does not get modified, thus the inner product is preserved.

[Q] The appropriate $\hat{f}$ is

$$\hat{f}(e_{\text{func}}, \sigma) = (\text{reverse}(e_{\text{func}}), \sigma).$$

[P] A QRAM can prepare the correct state by purely classical operations, replacing $e_{\text{func}}$ by reverse($e_{\text{func}}$).

G.4 [call-rev]

[T] Using the induction hypothesis, we know that

$$\psi' \in (\tilde{e} : \tilde{\tau}, \tilde{\pi}, \tilde{\tau}', \tilde{\tau}', \Delta').$$

We need to show $\psi' \in (\tilde{e} : \tilde{\tau}, \tilde{\pi}, \tilde{\tau}', \tilde{\tau}', \Delta')$, then the claim follows immediately after renaming.

By contradiction: Let $\psi' \notin (\tilde{e} : \tilde{\tau}, \tilde{\pi}, \tilde{\tau}', \tilde{\tau}', \Delta')$, then there exists a classical component of $\psi'$ which is in superposition. The typing rule of reverse enforces that the arguments and the return value of reversed function are not classical, hence the classical component in superposition needs to lie in context $\Delta$. By the induction hypothesis (specifically [C]), we know that evaluating $e_{\text{func}(\tilde{e}, \tilde{\pi})}$ leaves $\Delta$ unchanged, hence the classical component in superposition is also a classical component in superposition of $\psi$, which is a contradiction to $\psi \in (\tilde{e} : \tilde{\tau}, \tilde{\pi}, \Delta')$.

[C] We know that $\psi' \in (\tilde{e} : \tilde{\tau}, \tilde{\pi}, \tilde{\tau}', \tilde{\tau}', \Delta')$. Further, the linear map sending $\psi'$ to $\psi$ is

$$\sum_{\tilde{v}} |\tilde{v}\rangle \otimes M_{\tilde{v} \rightarrow \text{ret}} \otimes I_{\Delta},$$
where $M_v : \mathbb{F}^v \rightarrow \mathbb{F}^v$ is an isometry. The map becomes unitary by restricting the codomain to its image, which can be inverted resulting in
\[
\sum_{\tilde{v}} |\tilde{v}\rangle \langle \tilde{v}| M_{\tilde{v}, \tilde{v}}^{-1}(\tilde{v}^*) \otimes \mathbb{1}_\Lambda.
\]
which preserves $\tilde{v}^* \otimes \Lambda$, hence the claim.

[M] As renaming does not change the inner product, this claim follows from the induction hypothesis.

(Q) Let $q_{\mathrm{free}} \in \alpha$. By the induction hypothesis, we get that
\[
\psi' = \sum_{\tilde{v}, \tilde{w}} y_{\tilde{v}, \tilde{w}} |\tilde{v}\rangle \langle \tilde{w}| \otimes |\tilde{w}\rangle \hat{\psi}| \tilde{w}\rangle.
\]
and that there exists an $\hat{f}'$ such that after renaming $\hat{f}' \tilde{v}$ we have
\[
\psi = \sum_{\tilde{v}, \tilde{w}} y_{\tilde{v}, \tilde{w}} |\tilde{v}\rangle \langle \tilde{v}| \tilde{v} \hat{f}'(\tilde{v}, \tilde{w}) \rangle \hat{\psi}| \tilde{w}\rangle.
\]

We note that $\hat{f}$ is injective by Thm. 7.3, since non-injectivity would violate the semantics of the $\epsilon_{\mathrm{fun}}$ being a linear isometry. Thus, there exists a function $\hat{f} = \hat{f}^{-1}$ satisfying $\hat{f}^{-1}(\tilde{v}, \hat{f}'(\tilde{v}, \tilde{w})) = \tilde{w}'$, hence the claim.

(P) As $\epsilon_{\mathrm{fun}}$ is $\mathbb{free}$, a QRAM can implement it, according to Lem. G.1. As $\epsilon_{\mathrm{fun}}$ has no classical components in its type, the implementation depends only on $\epsilon_{\mathrm{fun}}$, not on classical components of the input state. Then, applying the reverse of the implementation to $\psi$ yields the correct result (up to renaming).

G.4.1 [eval-\lambda-abs].

[T] We know that $\psi \in \iota\left(\mathbb{F}: \tilde{v}, \tilde{\Lambda}\right)$, thus
\[
\psi \otimes |\sigma\rangle \in \iota\left(\mathbb{F}: \tilde{v}, \tilde{\Lambda}, \tilde{\Gamma}\right),
\]
which leads to $\mathbb{1}_{\tilde{v}, \Gamma, \Lambda}(\psi \otimes |\sigma\rangle) \in \iota\left(\mathbb{F}: \tilde{v}, \tilde{\Lambda}, \tilde{\Gamma}\right)$. The induction hypothesis yields now that
\[
\psi' \in \iota\left(\mathbb{F}': \tilde{v}', \tilde{\Lambda}'\right),
\]
thus after renaming, $\psi' \in \iota\left(\mathbb{F}': \tilde{v}', \tilde{\Lambda}'\right)$. Hence the claim.

[M] It is immediate that $\psi_1^N \psi_2 = \psi_1^N \otimes |\sigma\rangle \psi_2 \otimes |\sigma\rangle$. Further, renaming does not change the inner product, hence by the induction hypothesis, we get that $\psi_1^N \psi_2 = \psi_1^N \psi_2$. Renaming again leaves the inner product invariant, hence the claim.

(Q) The $\hat{f}$ obtained from the induction hypothesis behaves correctly, up to adding $\sigma$ to the state and renaming variables.

[P] Given input $\psi \otimes |\sigma''\rangle |\sigma\rangle$, a QRAM can rename variables ($\tilde{v} \rightarrow \tilde{x}$), run $e''$ (by induction hypothesis), and rename variables in the result again.
Lines of Code. When counting lines of code, we did not count empty lines, lines that only consist of comments, contain import or namespace statements or code that is unreachable for the solving operation.

Quantum Primitives and Annotations. We counted both the number of quantum primitives and annotations. Note that annotations are called functors in Q#. The summary in Tab. 3 shows how many quantum primitives and annotations were used at least once, measuring how many concepts a programmer needs to know.

Low-level quantum gates. We also counted low-level quantum gates, which are marked as ♣ in the detailed results. The summary in Tab. 3 shows how many low-level quantum gates were used in total, measuring how often the programmer has to resort to low-level operations.

For Q#, we did not include the counts of operations like ControlledOnInt, as they are more high-level. For the same reason, for Silq, we did not include phase, if, or forget.

Further, we did not add the counts for M or Measure (Q#) or measure (Silq), because measure can be applied to any data structure, and is thus more high-level, but gets often used similarly to M in Q#.

Top 10 Contestants. In order to compare the Silq solutions against the solutions of the top 10 contestants of the Q# Summer 2018 and Winter 2019 coding contest, we evaluate the submissions of the top 10 contestants using the same methods as before. We provide detailed results in Tab. 8, 9, 10, and 11.

<table>
<thead>
<tr>
<th></th>
<th>Silq S18</th>
<th>Silq W19</th>
<th>Silq Both</th>
<th>Q# reference solution S18</th>
<th>Q# reference solution W19</th>
<th>Q# reference solution Both</th>
<th>Q# average of top 10 S18</th>
<th>Q# average of top 10 W19</th>
<th>Q# average of top 10 Both</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of code</td>
<td>99</td>
<td>168</td>
<td>267</td>
<td>251</td>
<td>242</td>
<td>493</td>
<td>282.9</td>
<td>372.7</td>
<td>655.6</td>
</tr>
<tr>
<td>Quantum primitives</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>12</td>
<td>19</td>
<td>22</td>
<td>8.1</td>
<td>12.0</td>
<td>-</td>
</tr>
<tr>
<td>Annotations</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>1.0</td>
<td>4.0</td>
<td>-</td>
</tr>
<tr>
<td>Low-level quantum gates</td>
<td>14</td>
<td>23</td>
<td>37</td>
<td>33</td>
<td>54</td>
<td>87</td>
<td>38.2</td>
<td>102.9</td>
<td>141.1</td>
</tr>
</tbody>
</table>

Table 3. Silq compared to Q#. Two entries in the last column are missing, because the top 10 contestants are not the same for both competitions and the number of used annotation and built-in and library functions where calculated per contestant.
<table>
<thead>
<tr>
<th>Quantum primitives</th>
<th>Summer 2018</th>
<th>Winter 2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplyToEach</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>ApplyToEachC</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ApplyToEachCA</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>CCNOT</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>CNOT</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ControlledOnBitString</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ControlledOnInt</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CNOT</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CCNOT</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MResetZ</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MeasureInteger</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PrepareUniformSuperposition</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>R*</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ResetAll</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ResultAsInt</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>H*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Y*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Z*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Annotations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adjoint</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Controlled</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>adjoint auto</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>adjoint self</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>classical</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Low-level quantum gates (marked by *)</td>
<td>33</td>
<td>33</td>
</tr>
<tr>
<td>Lines of code</td>
<td>493</td>
<td>493</td>
</tr>
</tbody>
</table>

Table 4. Evaluation of the solutions provided by the Q# language designers for the Summer 2018 and Winter 2019 coding contest.

<table>
<thead>
<tr>
<th>Quantum primitives</th>
<th>Summer 2018</th>
<th>Winter 2019</th>
</tr>
</thead>
<tbody>
<tr>
<td>dup</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>forget</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>if*</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>if!</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>if?</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>measure</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>phase</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>reverse</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rotY</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>X*</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Annotations</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mfree</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lifted</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Low-level quantum gates (marked by *)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Lines of code</td>
<td>167</td>
<td>167</td>
</tr>
</tbody>
</table>

Table 5. Evaluation of the Silq solutions for Q# Summer 2018 and Winter 2019 coding contest.
<table>
<thead>
<tr>
<th>Quantum primitives</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplyToEach</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.2</td>
</tr>
<tr>
<td>BoolArrFromResultArr</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.3</td>
</tr>
<tr>
<td>BoolFromResult</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.1</td>
</tr>
<tr>
<td>CNOT</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>12</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td>CNOT*</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>8</td>
<td>9</td>
<td>6</td>
<td>6</td>
<td>7.6</td>
</tr>
<tr>
<td>H*</td>
<td>14</td>
<td>11</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>12.6</td>
</tr>
<tr>
<td>IsResultZero</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.1</td>
</tr>
<tr>
<td>M</td>
<td>12</td>
<td>11</td>
<td>12</td>
<td>16</td>
<td>12</td>
<td>10</td>
<td>12</td>
<td>3</td>
<td>12</td>
<td>11.8</td>
<td></td>
</tr>
<tr>
<td>MultiM</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.6</td>
</tr>
<tr>
<td>R*</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>Reset</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.3</td>
</tr>
<tr>
<td>ResetAll</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td>1.8</td>
</tr>
<tr>
<td>ResultAsInt</td>
<td>5</td>
<td>1</td>
<td></td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.8</td>
</tr>
<tr>
<td>Ry*</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>SWAP</td>
<td>7</td>
<td>12</td>
<td>15</td>
<td>5</td>
<td>42</td>
<td>11</td>
<td>9</td>
<td>10</td>
<td>6</td>
<td>10</td>
<td>12.7</td>
</tr>
<tr>
<td>Z*</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0.2</td>
</tr>
</tbody>
</table>

| Annotations        |     |     |     |     |     |     |     |     |     |     |         |
|--------------------|     |     |     |     |     |     |     |     |     |     |         |
| Controlled         | 2 | 5 | 1 | 8 | 3 | 2 | 2 | 2 | 2 | 3 | 2.3     |
| controlled auto    | 1 |    |    |    |    |    |    |    |    |    | 0.3     |

Quantum primitives (number of non-zero rows) | 7 | 6 | 10 | 6 | 6 | 7 | 10 | 13 | 10 |    | 8.1     |
Annotations (number of non-zero rows)       | 2 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 1   | 1.0     |
Low-level quantum gates (marked by *)        | 35 | 35 | 39 | 29 | 65 | 39 | 30 | 46 | 30 | 34 | 38.2    |
Lines of code                                | 181 | 312 | 259 | 313 | 313 | 387 | 228 | 271 | 280 | 285 | 282.9    |

Table 6. Summer 2018: Overview of the evaluation of the Q# solution provided by the top 10 contestants.
<table>
<thead>
<tr>
<th>Quantum primitives</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplyPauliFromBitString</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.4</td>
</tr>
<tr>
<td>ApplyToEach</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>ApplyToEachA</td>
<td>8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.1</td>
</tr>
<tr>
<td>ApplyToEachC</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.1</td>
</tr>
<tr>
<td>ApplyToEachCA</td>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.9</td>
</tr>
<tr>
<td>CCNOT*</td>
<td>6</td>
<td>10</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>31</td>
<td>1</td>
<td>4</td>
<td>6</td>
<td></td>
<td>6.4</td>
</tr>
<tr>
<td>CNOT*</td>
<td>5</td>
<td>15</td>
<td>12</td>
<td>14</td>
<td>20</td>
<td>17</td>
<td>13</td>
<td>23</td>
<td>18</td>
<td>15</td>
<td>15.2</td>
</tr>
<tr>
<td>ControlledOnBitString</td>
<td>2</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.0</td>
</tr>
<tr>
<td>ControlledOnInt</td>
<td>4</td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>H*</td>
<td>13</td>
<td>11</td>
<td>14</td>
<td>13</td>
<td>10</td>
<td>13</td>
<td>12</td>
<td>15</td>
<td></td>
<td></td>
<td>12.7</td>
</tr>
<tr>
<td>IntegerIncrementLE</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>0.6</td>
</tr>
<tr>
<td>M</td>
<td>3</td>
<td>3</td>
<td>12</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td>3.8</td>
</tr>
<tr>
<td>Measure</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>MeasureInteger</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td>0.3</td>
</tr>
<tr>
<td>MultiM</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>0.3</td>
</tr>
<tr>
<td>MultiX</td>
<td>9</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>QFT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.1</td>
</tr>
<tr>
<td>RX*</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>1.1</td>
</tr>
<tr>
<td>Reset</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>ResetAll</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td>0.4</td>
</tr>
<tr>
<td>ResultAsInt</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>0.3</td>
</tr>
<tr>
<td>RX*</td>
<td>2</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>0.2</td>
</tr>
<tr>
<td>Ry*</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>6</td>
<td>2</td>
<td>3.7</td>
</tr>
<tr>
<td>Rz*</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>1.5</td>
</tr>
<tr>
<td>S*</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>SWAP*</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>StatePreparationComplexCoefficients</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0.3</td>
</tr>
<tr>
<td>StatePreparationPositiveCoefficients</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>0.1</td>
</tr>
<tr>
<td>WithA</td>
<td>18</td>
<td>62</td>
<td>50</td>
<td>36</td>
<td>98</td>
<td>65</td>
<td>119</td>
<td>64</td>
<td>58</td>
<td>27</td>
<td>59.7</td>
</tr>
<tr>
<td>X*</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Annotations</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adjoint</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1.9</td>
</tr>
<tr>
<td>Controlled</td>
<td>14</td>
<td>26</td>
<td>27</td>
<td>25</td>
<td>37</td>
<td>27</td>
<td>30</td>
<td>57</td>
<td>36</td>
<td>12</td>
<td>29.1</td>
</tr>
<tr>
<td>adjoint self</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.5</td>
</tr>
<tr>
<td>adjoint auto</td>
<td>2</td>
<td>5</td>
<td>23</td>
<td>15</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>8.2</td>
</tr>
<tr>
<td>controlled auto</td>
<td>2</td>
<td>10</td>
<td>28</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.2</td>
</tr>
<tr>
<td>controlled adjoint auto</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.9</td>
</tr>
</tbody>
</table>

| Quantum primitives (number of non-zero rows) | 15| 9 | 10| 21| 10| 8 | 8 | 12| 12| 15| 12.0   |
| Annotations (number of non-zero rows) | 4 | 4 | 4 | 6 | 2 | 5 | 2 | 5 | 4 | 4 | 4.0    |
| Low-level quantum gates (marked by *) | 47| 107|84| 72| 143|111|182|115|102|66|102.9   |
| Lines of code | 163|322|461|298|367|358|543|610|323|282|372.7   |

Table 7. Winter 2019: Overview of the evaluation of the Q# solution provided by the top 10 contestants.
Table 8. Evaluation of the submissions of the top 10 contestants of the Q# Summer 2018 coding contest.
Table 9. Evaluation of the submissions of the top 10 contestants of the Q# Summer 2018 coding contest.
Table 10. Evaluation of the submissions of the top 10 contestants of the Q# Winter 2018 coding contest.
Table 11. Evaluation of the submissions of the top 10 contestants of the Q# Winter 2018 coding contest.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10 average</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplyToEachCA</td>
<td>2</td>
<td>2</td>
<td>0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApplyToEachCA</td>
<td>2</td>
<td>2</td>
<td>0.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNOT</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNOT</td>
<td>2</td>
<td>4</td>
<td>0.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ControlledOnInt</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IntegerIncrementLE</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SWAP</td>
<td></td>
<td>4</td>
<td>1</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>10</td>
<td>13</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>Controlled</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>adjoint auto</td>
<td>1</td>
<td>1</td>
<td>0.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>controlled auto</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1.9</td>
</tr>
<tr>
<td>Lines of code</td>
<td>20</td>
<td>27</td>
<td>40</td>
<td>35</td>
<td>31</td>
<td>33</td>
<td>36</td>
<td>39</td>
<td>31</td>
<td>32.4</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10 average</td>
</tr>
<tr>
<td>ApplyToEachCA</td>
<td>1</td>
<td>1</td>
<td>0.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ControlledOnInt</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ry</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Controlled</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>adjoint auto</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>controlled auto</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lines of code</td>
<td>5</td>
<td>14</td>
<td>16</td>
<td>13</td>
<td>14</td>
<td>17</td>
<td>14</td>
<td>9</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10 average</td>
</tr>
<tr>
<td>ApplyToEach</td>
<td>6</td>
<td>0.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApplyToEachCA</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNOT</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>8</td>
<td>4</td>
<td>2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IntegerIncrementLE</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MultiX</td>
<td>3</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>7</td>
<td>15</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>Controlled</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adjoint auto</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>controlled auto</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lines of code</td>
<td>10</td>
<td>29</td>
<td>44</td>
<td>35</td>
<td>28</td>
<td>51</td>
<td>153</td>
<td>25</td>
<td>22</td>
<td>43.5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10 average</td>
</tr>
<tr>
<td>ApplyToEach</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CNOT</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ry</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>3</td>
<td>15</td>
<td>13</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Controlled</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>adjoint auto</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>7</td>
<td>33</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>controlled auto</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>controlled adjoint auto</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lines of code</td>
<td>22</td>
<td>37</td>
<td>33</td>
<td>26</td>
<td>40</td>
<td>36</td>
<td>141</td>
<td>179</td>
<td>77</td>
<td>60</td>
</tr>
</tbody>
</table>

(a) Winter 19: C3
(b) Winter 19: D1
(c) Winter 19: D2
(d) Winter 19: D3
(e) Winter 19: D4
(f) Winter 19: D5
(g) Winter 19: D6
H.2 Evaluation against Quipper

In order to compare the amount of features, we counted the definitions provided in Quipper’s core library and list them by rubric and type in Tab. 12.

H.3 Further Algorithms

In the following, we provide further algorithms implemented in Silq.

// Wiesner’s quantum money: Conjugate coding, Stephen Wiesner,
// https://dl.acm.org/citation.cfm?id=1008920

```python
// create new bill
def create_bill[n:;!N](){
    // generate new bill and verifier
    secret:=uniform[4,n]();
    bill:=encode(secret)(0:uint[n]);
    verifier:=λ(b:uint[n]). verify(b,secret);
    return (bill,verifier);
}
```

```python
// verify a given bill
check:=reverse(encode(secret))(bill);
if measure(check)==0 { // ok, give money back
    return (encode(secret)(0:uint[n]),true);
} else { // forged!
    return (0:uint[n],false);
}
```

```python
// ENCODING FUNCTIONS

def encode[n:;!N](secret:;!N^n)(bill:uint[n])mfree{
    for k in [0..n]{
        bill[k]:=encode_B[secret[k]](bill[k]);
    }
    return bill;
}

def encode_B[state:;!N](b:B)mfree{
    // 0→0, 1→1, 2→+, 3→-
    if state%2==1 { b:=X(b); }
    if state>=2 { b:=H(b); } // switch basis to +/-
    return b;
}
```

```python
// SIMPLE TEST

```python
// verify new test
def verify_new_test[n:;!N](){
    // verify a new bill twice
    (bill,verifier):=create_bill[n]();
    // verify twice it is genuine
    (bill,ok1):=verifier(bill);
    assert(ok1);
    (bill,ok2):=verifier(bill);
    assert(ok2);
    // discard the bill
    measure(bill);
}
```

```python
// HELPER FUNCTIONS

def uniform[range:;!N,length:;!N](){
```

12https://www.mathstat.dal.ca/~selinger/quipper/doc/Quipper.html
// returns (x_1, ..., x_length) with x_i \in \{0, ..., \text{range}-1\}

n:=round(log(range)/log(2)) coerce \(\mathbb{N}\);
assert(2^n==range);
r:=vector(length,0:coerce !N);
for l in [0..length]{
    for k in [0..n]{
        r[l]+=2^k*rand();
    }
}
return r;
}
def rand(){
    // quantum number generator
    return measure(H(false));
}

import quantum_money;

// PRIMITIVE FORGE ATTEMPT
// The attempt does not work due to the no-cloning theorem

def forge_primitive[n:\mathbb{N}](bill:uint[n]){ forged:=dup(bill);
    return (bill,forged);
}

// SIMPLE TEST

def forge_primitive_test[n:\mathbb{N}](){
    // create new bill
    (bill,verifier):=create_bill[n]();
    // try to duplicate it
    (bill,forged):=forge_primitive(bill);
    // verify both
    (bill,ok_original):=verifier(bill);
    (forged,ok_forged):=verifier(forged);
    assert(ok_original == !ok_forged);
    // discard bills
    measure(bill);
    measure(forged);
}
def main(){
    forge_primitive_test[4]();
}


import quantum_money;

def forge_nagaj[n:\mathbb{N}](bill:uint[n],verifier:uint[n]!\rightarrow\mathbb{N}×\mathbb{B}){
    secret:=vector(n,0);
    for k in [0..n]{
        (bill,is_plus):=determine(bill,verifier,k,true);
        if is_plus{
            secret[k]=2;
        }else{
            secret[k]=measure(bill[k]);
        }
    }

    return (bill, encode(secret)(0:uint[n]));
}

def determine[n:\mathbb{N}](bill:uint[n],verifier:uint[n]!\rightarrow\mathbb{N}×\mathbb{B},k:\mathbb{N},check_plus:\mathbb{B}):uint[n]×\mathbb{B} {
    // determine the value of the k-th bit of the quantum bill
    // - check_plus=true: return 1 iff bit is plus
    // - check_plus=false: return 1 iff bit is minus
    fail_prob:=0.01;
    N:=ceil(\pi^2*n/(2*fail_prob)); // choose N
    if N%2==1{ N+=1; } // ensure N is even
    // choose \(\delta\)
    \(\delta:=\pi/(2*N)\);
    probe:=0:\mathbb{B};
    repeat N{
        probe:=rotY(\delta*2,probe); // rotate slightly towards 1
        if probe{
            // entangle
            bill[k]:=X(bill[k]);
            if !check_plus{ phase(\pi); }
        }
    }

    (bill,ok):=verifier(bill); // project back by verification
    assert(ok==1); // we should not be caught

    return (bill, measure(probe));
}

def forge_nagaj_test[n:\mathbb{N}](){
    // create a new bill
    (bill,verifier):=create_bill[n]();
    // forge
    (bill,forged):=forge_nagaj(bill,verifier);
    // verify both bills
    (bill,ok_original):=verifier(bill);
    (forged,ok_forged):=verifier(forged);
    assert(ok_original == !ok_forged);
    // discard both bills
    measure(bill);
    measure(forged);
}
def main(){
    forge_nagaj_test[2]();
}