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

Abstract

Existing quantum languages force the programmer to work at a low level of abstraction leading to unintuitive and cluttered code. A fundamental reason is that dropping temporary values from the program state requires explicitly applying quantum operations that safely *uncompute* these values.

We present Silq, the first quantum language that addresses this challenge by supporting safe, automatic uncomputation. This enables an intuitive semantics that implicitly drops temporary values, as in classical computation. To ensure physicality of Silq's semantics, its type system leverages novel annotations to reject unphysical programs.

Our experimental evaluation demonstrates that Silq programs are not only easier to read and write, but also significantly shorter than equivalent programs in other quantum languages (on average -46% for Q#, -38% for Quipper), while using only half the number of quantum primitives.

CCS Concepts: • **Software and its engineering** → *Formal language definitions*; *Language features*; • **Computer systems organization** → *Quantum computing*.

Keywords: Quantum Language, Uncomputation, Semantics

ACM Reference Format:
Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 36 pages. https://doi.org/10.1145/3385412.3386007

PLDI '20, June 15–20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06…$15.00
https://doi.org/10.1145/3385412.3386007

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

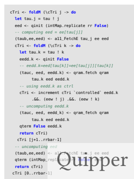
```
1 d := a || b || c;
```
Silq

```
1 with_computed (OR a b) $
2   \t -> OR t c
```
Quipper

```
1 using(t=Qubit()){
2   OR(a,b,t);
3   OR(t,c,d);
4   Adjoint OR(a,b,t);
5 }
```
Q#

Figure 1. Benefit of Silq's automatic uncomputation.

```
1 cTri := 0:int[rrbar];
2 for j in [0..rrbar) {
3   for k in [j+1..rrbar) {
4     if ee[tau[j]][tau[k]]
5        && eew[j] && eew[k]{
6       cTri += 1;
7 } } }
```
Silq


Quipper

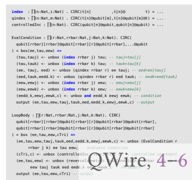

QWire, 4–6

Figure 2. Comparing Silq to Quipper and QWire code, more readable version in App. A.

follows an intuitive recipe that simply drops temporary values. Importantly, Silq's semantics is *physical*, i.e., can be realized on a quantum random access machine (QRAM) [11].

Overall, Silq allows expressing quantum algorithms more safely and concisely than existing quantum programming languages, while typically using only half the number of quantum primitives. In our evaluation (§8), we show that across 28 challenges from recent coding contests [15, 16], Silq programs require on average 46% less code than Q# [30]. Similarly, expressing the triangle finding algorithm [26] in Silq requires 38% less code than Quipper [7].

Main Contributions. Our main contributions are:

- Silq[1], a high-level quantum language enabling safe, automatic uncomputation (§4).
- A full formalization of Silq's key language fragment Silq-core (§5), including its type system (§6) and semantics (§7), whose physicality relies on its type system.
- An implemented type-checker[2], proof-of-concept simulator[2], and development environment[3] for Silq.
- An evaluation, showing that Silq code is more concise and readable than code in existing languages (§8).

Future Benefits. We expect Silq to advance various tools central to programming quantum computers. First, Silq compilers may require fewer qubits, e.g., by more flexibly picking the time of uncomputation. Second, static analyzers for Silq can safely assume (instead of explicitly proving) temporary values are discarded safely. Third, Silq simulators may improve performance by dropping temporary values instead of explicitly simulating uncomputation. Finally, Silq's key language features are relevant beyond Silq, as they can be incorporated into existing languages such as QWire or Q#.

[1]http://silq.ethz.ch/
[2]https://github.com/eth-sri/silq/tree/pldi2020
[3]https://marketplace.visualstudio.com/items?itemName=eth-sri.vscode-silq

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||b`. In contrast, Q# (right) requires (i) allocating a new qubit `t` initialized to `0` in Line 1, (ii) using `OR`[4] to store the result of `a||b` in `t` in Line 2, (iii) using `OR` to store the result of `t||c` in the pre-allocated qubit `d` in Line 3, and (iv) uncomputing `t` by reversing the operation from Line 2 in Line 4. Here, `Adjoint OR` is the inverse of `OR` and thus resets `t` to its original value of `0`. Hence, the *implicit measurement* induced by removing `t` from consideration in Line 5 always measures the value `0`, 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||b` in Line 1, (ii) uses the result `t` to compute `t||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||b`. In particular, this does not enable a natural nesting of expressions, as the sub-expression `OR a 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 LIQuiD [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) $\varphi = \gamma_0 |0\rangle + \gamma_1 |1\rangle$, where $\gamma_0, \gamma_1 \in \mathbb{C}$, and $\|\varphi\|^2 = \|\gamma\|^2 = \|\gamma_0\|^2 + \|\gamma_1\|^2$ denotes the probability of being in state φ. In particular, we allow $\|\varphi\| < 1$ to indicate that a measurement yields state φ with probability $\|\varphi\|$ — 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:

$$\mathcal{H}(S) := \left\{ \sum_{v \in S} \gamma_v |v\rangle \;\middle|\; \gamma_v \in \mathbb{C} \right\}.$$

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 $\mathcal{H}(\{0,1\}) = \{\gamma_0 |0\rangle + \gamma_1 |1\rangle \mid \gamma_0, \gamma_1 \in \mathbb{C}\}$, 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

$$\left\langle \sum_{v \in S} \gamma_v |v\rangle \;\middle|\; \sum_{v \in S} \gamma'_v |v\rangle \right\rangle = \sum_{v \in S} \gamma_v \gamma'_v.$$

Tensor Product. A system of multiple qubits can be described using the tensor product $\otimes$. For example, for two qubits $\varphi_0 = |0\rangle$ and $\varphi_1 = \frac{1}{\sqrt{2}} |0\rangle - \mathrm{i} \frac{1}{\sqrt{2}} |1\rangle$, the composite state is $\varphi_0 \otimes \varphi_1 = \frac{1}{\sqrt{2}} |0\rangle \otimes |0\rangle - \mathrm{i} \frac{1}{\sqrt{2}} |0\rangle \otimes |1\rangle = \frac{1}{\sqrt{2}} |0\rangle |0\rangle - \mathrm{i} \frac{1}{\sqrt{2}} |0\rangle |1\rangle$. 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 $\varphi_0 \otimes \varphi_1$ is unentangled, 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 $\varphi = \sum_{v \in S} \gamma_v |v\rangle$, we obtain

[6] Available at https://github.com/patrickrall/Qumquat, commit 27d6794

the value $v' \in S$ with probability $\|\gamma_{v'}\|^2$. As a fundamental law of quantum mechanics, if we measure the value v', the state after the measurement is $\gamma_{v'}\,|v'\rangle$ (we do not normalize this state to preserve linearity). This is referred to as the *collapse* of φ to $\gamma_{v'}\,|v'\rangle$, 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 $\sum_{v,w}\gamma_{v,w}\,|v\rangle\otimes|w\rangle$, we first rewrite it to $\sum_v \gamma_v\,|v\rangle\otimes\tilde{\varphi}_v$, separating out the remainder $\tilde{\varphi}_v$ of the state, where $\|\tilde{\varphi}_v\| = 1$. This is a common technique and always possible for appropriate choices of γ_v and $\tilde{\varphi}_v$. Then, measuring the first part to be v' yields state $\gamma_{v'}\,|v'\rangle\otimes\tilde{\varphi}_{v'}$, also collapsing the remainder of the state.

Linear Isometries. Besides measurements, we can also manipulate quantum states using *linear isometries*, i.e., linear functions $f\colon \mathcal{H}(S) \to \mathcal{H}(S')$ preserving inner products: for all $\varphi, \varphi' \in \mathcal{H}(S)$, $\langle f(\varphi)|f(\varphi')\rangle = \langle\varphi|\varphi'\rangle$. 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 [7], 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 $\varphi \mapsto \varphi\otimes\varphi$. 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 $w^\star$ to `0`. Here, Silq uses the generic parameter `n` to parametrize the input type `uint[n]` of `f`. Then, `grover` outputs an `n`-bit unsigned integer w which is equal to $w^\star$ with high probability.

[7]Since given input space $\mathcal{H}(S)$, allocating qubits leads to a larger output space $\mathcal{H}(S')$, requiring $\mathcal{H}(S)$ and $\mathcal{H}(S')$ to have the same dimension (as enforced by unitary operations) prevents dynamically allocating qubits.

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, `0` has type $!\mathbb{B}$. In contrast, **`H`**(0) applies Hadamard **`H`** (defined shortly) to `0` and yields $\frac{1}{\sqrt{2}}\big(\,|0\rangle + |1\rangle\,\big)$. 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 $\tau\,!{\to}\,\tau'$ instead of $!\tau \to \tau'$ to avoid the ambiguity between $!(\tau \to \tau')$ and $(!\tau) \to \tau'$. [8]

Qfree Functions. The type of `f` is annotated as **`qfree`**, which indicates the semantics of `f` can be described classically: we can capture the semantics of a **`qfree`** function `g` as a function $\overline{g}\colon S \to S'$ for ground sets S and S'. Note that since S' is a ground set, $\overline{g}$ can never output superpositions. Then, `g` acting on $\sum_{v\in S}\gamma_v\,|v\rangle$ yields $\sum_{v\in S}\gamma_v\,|\overline{g}(v)\rangle$, where for simplicity $\sum_{v\in S}\gamma_v\,|v\rangle$ does not consider other qubits untouched by `g`.

For example, the **`qfree`** function **`X`** flips the bit of its input, mapping $\sum_{v=0}^{1}\gamma_v\,|v\rangle$ to $\sum_{v=0}^{1}\gamma_v\,\big|\overline{\mathtt{X}}(v)\big\rangle$, for $\overline{\mathtt{X}}(v) = 1 - v$. In contrast, the Hadamard transform **`H`** maps $\sum_{v=0}^{1}\gamma_v\,|v\rangle$ to $\sum_{v=0}^{1}\gamma_v\frac{1}{\sqrt{2}}\big(\,|0\rangle + (-1)^v\,|1\rangle\,\big)$. As this semantics cannot be described by a function on ground sets, **`H`** is not **`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 **`const`**, indicating `f` preserves its input, i.e., treats it as a read-only control. Thus, running `f` on $\sum_{v\in S}\gamma_v\,|v\rangle$ yields $\sum_{v\in S}\gamma_v\,|v\rangle\otimes\varphi_v$, where φ_v follows the semantics of `f`. Because `f` is also **`qfree`**, $|v\rangle\otimes\varphi_v = \Big|\overline{f}(v)\Big\rangle$ for some $\overline{f}\colon S \to S \times S'$. Combining both, we conclude that $\overline{f}(v) = (v, \tilde{f}(v))$ for some function $\tilde{f}\colon S \to S'$.

An example of a possible instantiation of f is NOT, which maps $\gamma_0|0\rangle + \gamma_1|1\rangle$ to $\gamma_0|0,1\rangle + \gamma_1|1,0\rangle$. Here, $\widetilde{\mathrm{NOT}}\colon \{0,1\} \to \{0,1\}$ maps $v \mapsto 1 - v$ and $\overline{\mathrm{NOT}}\colon \{0,1\} \to \{0,1\}\times\{0,1\}$ maps $v \mapsto (v, 1-v) = (v, \widetilde{\mathrm{NOT}}(v))$.

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

[8]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`**: $\mathbb{B}!\xrightarrow{\mathtt{mfree}}\mathbb{B}$ is classical, meaning that the quantum operations performed by **`H`** are classically known.

generic parameter n — f preserves argument — n-bit uint — boolean

```
1  def grover[n:!ℕ](f:const uint[n]!→qfree 𝔹){
2    nIterations:=⌊π/4 / asin(2^(-n/2))⌋;
3    cand:=0:uint[n];
4    for k in [0..n){ cand[k]:=H(cand[k]); }
5
6    for k in [0..nIterations){
7      if f(cand){
8        phase(π);
9      }
10     cand:=groverDiff[n](cand);
11   }
12   return measure(cand);
13 }
```

!: classical — ℕ: natural number — f preserves superpositions — variable n holds value n

$$\psi_1 = \left|\tilde{f}\right\rangle_{\texttt{f}} \otimes |n\rangle_{\texttt{n}}$$

$$\psi_2 = \psi_1 \otimes \left|\left\lfloor \tfrac{\pi}{4}/\mathrm{asin}\left(2^{-n/2}\right)\right\rfloor\right\rangle_{\texttt{nIterations}}$$

$$\psi_4 = \psi_2 \otimes \sum_v \tfrac{1}{\sqrt{2^n}} |v\rangle_{\texttt{cand}}$$

$$\psi_6^{(0)} = \psi_2 \otimes |0\rangle_{\texttt{k}} \otimes \Big(\sum_{v \neq w^\star} \tfrac{1}{\sqrt{2^n}} |v\rangle_{\texttt{cand}} + \tfrac{1}{\sqrt{2^n}} |w^\star\rangle_{\texttt{cand}} \Big)$$

then branch

$$\psi_7^{(0)} = \psi_2 \otimes |0\rangle_{\texttt{k}} \otimes \Big(\sum_{v \neq w^\star} \tfrac{1}{\sqrt{2^n}} |v\rangle_{\texttt{cand}} |0\rangle_{\underline{\texttt{f(cand)}}} + \tfrac{1}{\sqrt{2^n}} |w^\star\rangle_{\texttt{cand}} |1\rangle_{\underline{\texttt{f(cand)}}} \Big)$$

$$\psi_8^{(0)} = \psi_2 \otimes |0\rangle_{\texttt{k}} \otimes \Big(\sum_{v \neq w^\star} \tfrac{1}{\sqrt{2^n}} |v\rangle_{\texttt{cand}} |0\rangle_{\underline{\texttt{f(cand)}}} + -\tfrac{1}{\sqrt{2^n}} |w^\star\rangle_{\texttt{cand}} |1\rangle_{\underline{\texttt{f(cand)}}} \Big)$$

$$\psi_9^{(0)} = \psi_2 \otimes |0\rangle_{\texttt{k}} \otimes \Big(\sum_{v \neq w^\star} \tfrac{1}{\sqrt{2^n}} |v\rangle_{\texttt{cand}} + -\tfrac{1}{\sqrt{2^n}} |w^\star\rangle_{\texttt{cand}} \Big)$$

$$\psi_{10}^{(0)} = \psi_2 \otimes |0\rangle_{\texttt{k}} \otimes \Big(\sum_{v \neq w^\star} \gamma_v^- |v\rangle_{\texttt{cand}} + \gamma_{w^\star}^+ |w^\star\rangle_{\texttt{cand}} \Big)$$

$$\begin{aligned} \texttt{H}:&\ \mathbb{B}\,! \xrightarrow{\texttt{mfree}} \mathbb{B} \\ \texttt{groverDiff[n]}:&\ \textbf{uint}[n]\,! \xrightarrow{\texttt{mfree}} \textbf{uint}[n] \\ \texttt{measure}:&\ \tau\,! \longrightarrow !\tau \\ \texttt{phase}:&\ !\textbf{float}\,! \xrightarrow{\texttt{mfree}} \mathbb{1} \end{aligned}$$

unit type

Figure 3. Grover's algorithm in Silq. We provide an unannotated version, including `groverDiff`, in App. B. The top-right box shows the type of all used functions. The shown sums range over all n-bit unsigned integers $\{0, \ldots, 2^n - 1\}$.

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 $\textbf{uint}[n] \xrightarrow{\texttt{lifted}} \mathbb{B}$.

4.2 Silq Semantics

Next, we discuss the semantics of Silq on `grover`.

Input State. In Fig. 3, the state of the system after Line 1 is ψ_1, where the state of `f:uint[n]!`$\xrightarrow{\texttt{qfree}} \mathbb{B}$ is described as a function $\tilde{f}\colon \{0, \ldots, 2^n - 1\} \to \{0, 1\}$. 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 $\tilde{f}$, this distinction is irrelevant here. Next, Line 2 initializes the classical variable `nIterations`, yielding ψ_2.

Superpositions. Lines 3–4 result in state ψ_4, 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 coefficient of $|w^\star\rangle$, thus increasing the probability of measuring $w^\star$ 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 $w^\star$ and all other values.

Conditionals. Intuitively, Lines 7–9 flip the sign of those coefficients for which `f(cand)` returns true. To this end, we first evaluate `f(cand)` and place the result in a temporary variable $\underline{\texttt{f(cand)}}$, yielding state $\psi_7^{(0)}$. Here and in the following, we write $\underline{e}$ for a temporary variable that contains the result of evaluating e. Then, we determine those summands of $\psi_7^{(0)}$ where $\underline{\texttt{f(cand)}}$ is true (marked as "then branch" in Fig. 3), and run **`phase`**(π) on them. This yields $\psi_8^{(0)}$, as **`phase`**(π) flips the sign of coefficients. Lastly, we drop $\underline{\texttt{f(cand)}}$ 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 $w^\star$, obtaining $\left\|\gamma_{w^\star}^+\right\| > \left\|\frac{1}{\sqrt{2^n}}\right\|$, and decreases the weight of non-solutions $v \neq w^\star$, obtaining $\left\|\gamma_v^-\right\| < \left\|\frac{1}{\sqrt{2^n}}\right\|$. After one loop iteration, this results in state $\psi_{10}^{(0)}$. Repeated iterations of the loop in Lines 6–11 further increase the coefficient of $w^\star$, until it is approximately 1. Thus, measuring `cand` in Line 12 returns $w^\star$ with high probability.

4.3 Uncomputation

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

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

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

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

[9] Formally, this corresponds to taking the partial trace over $\underline{\texttt{f(cand)}}$.

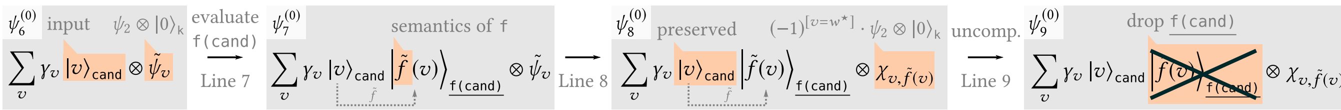


Figure 4. Uncomputation of `f(cand)` is safe. The term $(-1)^{[v=w^\star]}$ equals -1 if $v = w^\star$ and 1 otherwise.

```
def useConsumed(x:𝔹){
  y := H(x); // consumes x
  return (x,y);
} // undefined identifier x

def useConsumedFixed(const x:𝔹){
  // ψ1 = Σ_{v=0}^1 γv |v⟩x
  // ψ2 = Σ_{v=0}^1 γv |v⟩x ⊗ |v⟩x
  y := H(x);
  // ψ3 = Σ_{v=0}^1 γv |v⟩x ⊗ (|0⟩y + (−1)^v |1⟩y)
  return (x,y);
}
```

```
def discard[n:!ℕ](x:uint[n]){
  y := x % 2; // '%' supports quantum inputs
  return y;
} // parameter 'x' is not consumed

def nonQfree(const x:𝔹,y:𝔹){
  if H(x) { y := X(y); }
  return y;
} // non-lifted quantum expression must be consumed

def nonConst(c:𝔹){
  if X(c) { phase(π); } // X consumes c
} // non-lifted quantum expression must be consumed
```

```
def nonConstFixed(const c:𝔹){
  // ψ1 = Σ_{v=0}^1 γv |v⟩c
  if X(c) { phase(π); }
  // ψ2 = Σ_{v=0}^1 (−1)^{1−v} γv |v⟩c
}

def condMeas(const c:𝔹,x:𝔹){
  if c { x := measure(x); }
} // cannot call function
// 'measure[𝔹]' in 'mfree' context

def revMeas(){
  return reverse(measure);
} // reversed function must be mfree
```

Figure 5. Examples of invalid Silq programs, their error messages, and possible fixes (where applicable).

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(cand)` from $\psi_8^{(0)}$, 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(cand)` is possible, the type system of Silq ensures that `f(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(cand)` to the state, whose value can be computed from `cand` using $\tilde{f}$ (as `f` is **`qfree`** and `cand` is **`const`**). Then, Line 8 transforms the remainder $\tilde{\psi}_v$ of the state to $\chi_{v,\tilde{f}(v)}$. 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(cand)` from `cand` in $\psi_8^{(0)}$. Thus, reversing the operations of this reconstruction (i.e., reversing `f`) uncomputes `f(cand)` and yields $\psi_9^{(0)}$.

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 ψ_2, and then only consumes the duplicate $\underline{\texttt{x}}$.

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 $\sum_v \gamma_v |v\rangle \mapsto \sum_v \gamma_v |v\rangle |v\rangle$ is physical and can be implemented using `CNOT`, unlike the unphysical cloning $\sum_v \gamma_v |v\rangle \mapsto (\sum_v \gamma_v |v\rangle) \otimes (\sum_v \gamma_v |v\rangle) = \sum_{v,w} \gamma_v \gamma_w |v\rangle |w\rangle$ 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(cand)`), 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 $!\mathbb{B}$ 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`, as this function would not be **`mfree`**. Here, it is crucial that Silq disallows implicit measurement — otherwise, it would be hard to determine which functions are **`mfree`**.

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 its 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:=`**`dup`**`(x)` duplicates `x`, mapping $\sum_v \gamma_v \left|v\right\rangle_{\mathsf{x}}$ to $\sum_v \gamma_v \left|v\right\rangle_{\mathsf{x}} \left|v\right\rangle_{\mathsf{y}}$. Thus, **`reverse(dup)`**`(x,y)` operates on states $\sum_v \gamma_v \left|v\right\rangle_{\mathsf{x}} \left|v\right\rangle_{\mathsf{y}} \otimes \tilde{\psi}_v$, for which it yields $\sum_v \gamma_v \left|v\right\rangle_{\mathsf{x}} \otimes \tilde{\psi}_v$, uncomputing `y`. On other states, **`reverse`**(**`dup`**) is undefined. As **`reverse(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(dup)`**`(x,y)` by running **`if`** `x { y:=X(y); }` and discarding `y`, which has unintended side-effects (due to implicit measurement) unless originally `x==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`**),

$$e ::= c \mid x \mid \texttt{measure} \mid \texttt{reverse} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \mid e'(\underbrace{e_1,\ldots,e_n}_{(\vec{e})}) \mid \lambda(\underbrace{\beta_1 x_1\colon \tau_1,\ldots,\beta_n x_n\colon \tau_n}_{(\vec{\beta}\vec{x}\colon\vec{\tau})}).e$$

expressions

types

$$\tau ::= \mathbb{1} \mid \mathbb{B} \mid \bigtimes_{k=1}^{n} \tau_k \mid \bigtimes_{k=1}^{n} \beta_k \tau_k \; ! \xrightarrow{\alpha} \tau' \mid \; !\tau$$

annotations

$$\alpha \subseteq \{\texttt{mfree}, \texttt{qfree}\}$$
$$\beta \subseteq \{\texttt{const}\}$$

Figure 6. Syntax, types, and annotations.

$$\Gamma ::= \beta_1 x_1\colon \tau_1, \ldots, \beta_n x_n\colon \tau_n \qquad \Gamma \vdash^{\alpha} e\colon \tau$$

Figure 7. Typing judgments.

and reversing quantum operations (**`reverse`**). Further, its if-then-else construct **`if`** e **`then`** e_1 **`else`** e_2 is syntactically standard, but supports both classical ($!\mathbb{B}$) and quantum ($\mathbb{B}$) condition e. Function application $e'(\vec{e})$ explicitly takes multiple arguments. Likewise, lambda abstraction $\lambda(\vec{\beta}\vec{x}\colon \vec{\tau}).e$ describes a function with multiple parameters $\{x_i\}_{i=1}^{n}$ of types $\{\tau_i\}_{i=1}^{n}$, annotated by $\{\beta\}_{i=1}^{n}$, as discussed in §5.2 (next).

We note that Silq-core can support tupling as a built-in function c.

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

5.2 Types and Annotations of Silq-Core

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

Primitive Types. Silq-core types include standard primitive types, including $\mathbb{1}$, the singleton type that only contains the element "()", and $\mathbb{B}$, 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.

Products and Functions. Silq-core also supports products, where we often write $\tau_1 \times \cdots \times \tau_n$ for $\bigtimes_{k=1}^{n} \tau_k$, and functions, where ! emphasizes that functions are classically known (i.e., we do not discuss superpositions of functions). Function parameters and functions themselves may be annotated by β_i and α, respectively, as discussed shortly. As usual, $\times$ binds stronger than $\rightarrow$.

Finally, Silq-core supports annotating types as classical.

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 $x\colon \mathbb{B}$, the expression **`if`** x **`then`** $f(0)$ **`else`** $f(1)$ is only physical if f is **`mfree`** (note that x does not appear in the two branches).

6 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`**.

Format of Typing Rules. In Fig. 7, $\Gamma \vdash^{\alpha} e\colon \tau$ indicates an expression e has type τ under context Γ, and the evaluation of e is $\alpha \subseteq \{\texttt{qfree}, \texttt{mfree}\}$. For example, $x\colon \mathbb{B} \vdash^{\alpha} \texttt{H}(x)\colon \mathbb{B}$ for $\alpha = \{\texttt{mfree}\}$, where $\texttt{mfree} \in \alpha$ since evaluating $\texttt{H}(x)$ does not induce a measurement, and $\texttt{qfree} \notin \alpha$ since the effect of evaluating $\texttt{H}(x)$ cannot be described classically. We note that in general, $x\colon \tau \vdash^{\alpha} f(x)\colon \tau'$ if f has type $\tau ! \xrightarrow{\alpha} \tau'$, i.e., the annotation of f determines the annotation of the turnstile $\vdash$.

A context Γ is a multiset $\{\beta_i x_i\colon \tau_i\}_{i \in \mathcal{I}}$ that assigns a type τ_i to each variable x_i, where $\mathcal{I}$ is a finite index set, and x_i may be annotated by $\texttt{const} \in \beta_i$, indicating that it will not be consumed during evaluation of e. As a shorthand, we often write $\Gamma = \vec{\beta}\vec{x}\colon \vec{\tau}$.

We write $\Gamma, \beta x\colon \tau$ for $\Gamma \uplus \{\beta x\colon \tau\}$, where $\uplus$ denotes the union of multisets. Analogously Γ, Γ' denotes $\Gamma \uplus \Gamma'$. In general, we require that types and annotations of contexts can never be conflicting, i.e., $\beta x\colon \tau \in \Gamma$ and $\beta' x\colon \tau' \in \Gamma$ implies $\beta = \beta'$ and $\tau = \tau'$.

6.1 Typing Constants and Variables

If c is a constant of type τ, its typing judgement is given by $\emptyset \vdash^{\texttt{mfree},\texttt{qfree}} c\colon \tau$. For example, $\emptyset \vdash^{\texttt{mfree},\texttt{qfree}} \texttt{H}\colon \mathbb{B} ! \xrightarrow{\texttt{mfree}} \mathbb{B}$. Here, we annotate the turnstile $\vdash$ as **`qfree`**, because evaluating expression **`H`** maps the empty state $|\rangle$ to $|\rangle \otimes |\texttt{H}\rangle_{\underline{\texttt{H}}}$, which can be described classically by $\overline{f}(|\rangle) = |\texttt{H}\rangle_{\underline{\texttt{H}}}$. We provide the types of other selected built-in functions in App. E.2.

Likewise, the typing judgement of variables carries annotations **`qfree`** and **`mfree`** (rule var in Fig. 8), as all constants c and variables 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 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.

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 !W and W) and duplicate them (contraction rules !C and C). For weakening, the interpretation of "dropping variable x" arises from reading the rule bottom-up, which is also the way our semantics operates (analogously for contraction). In our semantics (§7), dropping constant variable x in the body of a function f can be handled by uncomputing it at the end of f.

$$\frac{}{\beta x\colon \tau \vdash^{\texttt{mfree,qfree}} x\colon \tau}\text{ var}\qquad \frac{\Gamma \vdash^{\alpha} e\colon \tau'}{\Gamma, x\colon !\tau \vdash^{\alpha} e\colon \tau'}\text{ !W}\qquad \frac{\Gamma \vdash^{\alpha} e\colon \tau'}{\Gamma, \texttt{const}\ x\colon \tau \vdash^{\alpha} e\colon \tau'}\text{ W}\qquad \frac{\Gamma, x\colon !\tau, x\colon !\tau \vdash^{\alpha} e\colon \tau'}{\Gamma, x\colon !\tau \vdash^{\alpha} e\colon \tau'}\text{ !C}\qquad \frac{\Gamma, \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau \vdash^{\alpha} e\colon \tau'}{\Gamma, \texttt{const}\ x\colon \tau \vdash^{\alpha} e\colon \tau'}\text{ C}$$

Figure 8. Typing variables, including weakening and contraction.

$$\frac{\boxed{\Gamma_i \vdash^{\alpha_i} e_i\colon \tau_i} \quad \boxed{\Gamma' \vdash^{\alpha'} e'\colon \bigtimes_{i=1}^{n} \beta_i\tau_i \mathrel{!}\xrightarrow{\alpha'_{\text{func}}} \tau'}}{\boxed{\Gamma_1,\dots,\Gamma_n,\Gamma' \vdash^{\alpha''} e'(e_1,\dots,e_n)\colon \tau'}}\text{ func-eval}$$

$$\texttt{const} \in \beta_i \implies \texttt{qfree} \in \alpha_i \land \Gamma_i = \texttt{const}\ \vec{x}\colon \vec{\tau}'' \tag{1}$$

$$\texttt{qfree} \in \alpha'' \iff \texttt{qfree} \in \bigcap_i \alpha_i \cap \alpha' \cap \alpha'_{\text{func}} \tag{2}$$

$$\texttt{mfree} \in \alpha'' \iff \texttt{mfree} \in \bigcap_i \alpha_i \cap \alpha' \cap \alpha'_{\text{func}} \tag{3}$$

Figure 9. Typing rule and constraints for function calls.

We note that variables with classical type can be used more liberally than **const** variables (e.g., as `if`-conditions). Hence, annotating a classical variable as **const** has no effect. We annotate variables (not types) as **const** as our syntax does not allow partially consuming variables.

6.2 Measurement

We type **measure** as $\emptyset \vdash^{\texttt{mfree,qfree}} \texttt{measure}\colon \tau \mathrel{!}\!\rightarrow\ !\tau$, where the lack of **const** annotation for τ indicates **measure** consumes its argument, and $!\tau$ indicates the result is classical. We annotate the judgement itself as **mfree**, as evaluating the expression **measure** simply yields the function **measure**, without inducing a measurement. In contrast, the function type of **measure** itself is not **mfree** (indicated by $! \rightarrow$), as evaluating the function **measure** on an argument induces a measurement. Thus, **measure**(0) is not **mfree**, as evaluating it induces a measurement: $\emptyset \vdash \texttt{measure}(0)\colon !\mathbb{B}$.

6.3 Function Calls

Fig. 9 shows the typing rule for function calls $e'(e_1,\dots,e_n)$. Ignoring annotations, the rule follows the standard pattern, which we provide for convenience in App. E.1 (Fig. 25).

We now discuss the annotation Constraints (1)–(3). Constraint (1) ensures that if a function leaves its argument constant ($\texttt{const} \in \beta_i$), e_i is **lifted**, i.e., **qfree** and depending only on **const** variables. In turn, this ensures that we can automatically uncompute e_i when it is no longer needed. We note that non-constant arguments ($\texttt{const} \notin \beta_i$) do not need to be uncomputed, as they are no longer available after the call. To illustrate that Constraint (1) is critical, consider a function $f\colon \texttt{const}\ \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, and a call $f(x+1, \texttt{H}(x))$ with non-constant variable x in context Γ_1. This call must be rejected by Silq-core as there is no way to uncompute $x+1$ after the call to f, since **H** consumes x. Indeed, since x is not **const** in Γ_1 even though $\texttt{const} \in \beta_1$, (1) does not hold.

$$\frac{\boxed{\vec{\beta}\vec{x}\colon \vec{\tau}, \vec{\beta}'\vec{y}\colon !\vec{\tau}' \vdash^{\alpha} e\colon \tau''}}{\boxed{\vec{\beta}'\vec{y}\colon !\vec{\tau}' \vdash^{\texttt{mfree,qfree}} \lambda(\vec{\beta}\vec{x}\colon \vec{\tau}).e\colon \bigtimes_{i=1}^{n} \beta_i\tau_i \mathrel{!}\xrightarrow{\alpha} \tau''}}\ \lambda\text{-abs}$$

Figure 10. Typing lambda abstraction.

Constraint (2) ensures an expression is only **qfree** if all its components are **qfree**, and if the evaluated function only uses **qfree** operations. Constraint (3) is analogous for **mfree**.

We note that in order to allow temporarily marking variables as **const** (as discussed in §4.4), we could replace the top-left Γ_i in Fig. 9 by $\Gamma_i, \star_i$, where $\star_i = \texttt{const}\ \Gamma_{i+1},\dots,\texttt{const}\ \Gamma_n$ if $\texttt{const} \notin \beta_i$, and $\star_i = \emptyset$ otherwise. This would allow us to temporarily treat variables as **const**, if they appear in a consumed expression e_i and they are consumed in a later expression e_j for $j > i$. Fig. 9 omits this for conciseness.

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 α of the original typing judgment. Second, we tag the resulting type judgment as **mfree** and **qfree**, since function abstraction requires neither measurement nor quantum operations. Third, the rule allows capturing classical variables (y_i has type $!\tau_i$), 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 **reverse**. We only allow reversing functions without classical components in input or output types (indicated by ✗), as reconstructing classical components of inputs is typically impossible. Concretely, types without classical components are (i) $\mathbb{1}$, (ii) $\mathbb{B}$, and (iii) products of types without classical components. In particular, this rules out all classical types $!\tau$, function types, and products of types with classical components.

The input to **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 **qfree** (as indicated by a callout). Then, the type rule for **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

$$\left(\bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{j=1}^{m} \tau'_j\ ! \xrightarrow{\texttt{mfree},\,\alpha} \bigtimes_{k=1}^{l} \tau''_k\right)$$

$$! \xrightarrow{\texttt{mfree},\,\texttt{qfree}}$$

$$\left(\bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{k=1}^{l} \tau''_k\ ! \xrightarrow{\texttt{mfree},\,\alpha} \bigtimes_{j=1}^{m} \tau'_j\right)$$

Figure 11. Type of **reverse**.

$$\frac{\Gamma_c \overset{\alpha_c}{\vdash} e\colon \mathbb{B} \quad \Gamma \overset{\alpha_1}{\vdash} e_1\colon \tau \quad \Gamma \overset{\alpha_2}{\vdash} e_2\colon \tau}{\Gamma_c, \Gamma \overset{\alpha}{\vdash} \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2\colon \tau}\ \text{ite-q} \qquad \frac{\Gamma_c \overset{\alpha_c}{\vdash} e\colon !\mathbb{B} \quad \Gamma \overset{\alpha_1}{\vdash} e_1\colon \tau \quad \Gamma \overset{\alpha_2}{\vdash} e_2\colon \tau}{\Gamma_c, \Gamma \overset{\alpha}{\vdash} \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2\colon \tau}\ \text{ite-c}$$

$$\texttt{qfree} \in \alpha_c \wedge \Gamma_c = \texttt{const}\ \vec{x}\colon \vec{\tau}' \tag{4}$$

$$\texttt{mfree} \in \alpha_c \cap \alpha_1 \cap \alpha_2 \cap \alpha \tag{5}$$

$$\texttt{qfree} \in \alpha \iff \texttt{qfree} \in \alpha_c \cap \alpha_1 \cap \alpha_2 \tag{6}$$

$$\texttt{mfree} \in \alpha \iff \texttt{mfree} \in \alpha_c \cap \alpha_1 \cap \alpha_2 \tag{7}$$

$$\texttt{qfree} \in \alpha \iff \texttt{qfree} \in \alpha_c \cap \alpha_1 \cap \alpha_2 \tag{8}$$

Figure 12. Typing quantum (left) and classcial (right) control flow

Classical set $\llbracket\tau\rrbracket^{\mathsf{c}}$

$$\llbracket\mathbb{1}\rrbracket^{\mathsf{c}} := \{()\} \quad \llbracket\mathbb{B}\rrbracket^{\mathsf{c}} := \{()\} \quad \left\llbracket\bigtimes_{k=1}^{n} \tau_k\right\rrbracket^{\mathsf{c}} := \bigtimes_{k=1}^{n} \llbracket\tau_k\rrbracket^{\mathsf{c}} \quad \left\llbracket\bigtimes_{k=1}^{n} \beta_k\tau_k\ ! \overset{\alpha}{\longrightarrow} \tau'\right\rrbracket^{\mathsf{c}} := \left\{e, \sigma \;\middle|\; \begin{array}{c} \vec{y}\colon !\vec{\tau}'' \overset{\alpha'}{\vdash} e\colon \bigtimes_{k=1}^{n} \beta_k\tau_k\ ! \overset{\alpha}{\longrightarrow} \tau', \\ \sigma \in \llbracket\vec{y}\colon !\vec{\tau}''\rrbracket^{\mathsf{c}} \end{array}\right\} \quad \llbracket!\tau\rrbracket^{\mathsf{c}} := \llbracket\tau\rrbracket^{\mathsf{c}} \times \llbracket\tau\rrbracket^{\mathsf{q}}$$

Quantum ground set $\llbracket\tau\rrbracket^{\mathsf{q}}$

$$\llbracket\mathbb{1}\rrbracket^{\mathsf{q}} := \{()\} \quad \llbracket\mathbb{B}\rrbracket^{\mathsf{q}} := \{0, 1\} \quad \left\llbracket\bigtimes_{k=1}^{n} \tau_k\right\rrbracket^{\mathsf{q}} := \bigtimes_{k=1}^{n} \llbracket\tau_k\rrbracket^{\mathsf{q}} \quad \left\llbracket\bigtimes_{k=1}^{n} \beta_k\tau_k ! \overset{\alpha}{\longrightarrow} \tau'\right\rrbracket^{\mathsf{q}} := \{()\} \quad \llbracket!\tau\rrbracket^{\mathsf{q}} := \{()\}$$

Figure 13. Classical set $\llbracket\tau\rrbracket^{\mathsf{c}}$ and quantum ground set $\llbracket\tau\rrbracket^{\mathsf{q}}$ to build the semantics $\llbracket\tau\rrbracket = \llbracket\tau\rrbracket^{\mathsf{c}} \times \mathcal{H}\left(\llbracket\tau\rrbracket^{\mathsf{q}}\right)$ of type τ.

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** e_1 **else** e_2 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 $\not!$), 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 $\llbracket\tau\rrbracket$ of type τ from a *classical set* $\llbracket\tau\rrbracket^{\mathsf{c}}$ and a *quantum ground set* $\llbracket\tau\rrbracket^{\mathsf{q}}$ as $\llbracket\tau\rrbracket = \llbracket\tau\rrbracket^{\mathsf{c}} \times \mathcal{H}\left(\llbracket\tau\rrbracket^{\mathsf{q}}\right)$. Note that $\llbracket\tau\rrbracket$ 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, $\llbracket\tau\rrbracket^{\mathsf{q}}$ contains the ground set from which we build the Hilbert space $\mathcal{H}\left(\llbracket\tau\rrbracket^{\mathsf{q}}\right)$.

Classical Set and Quantum Ground Set. Fig. 13 defines both the classical set $\llbracket\tau\rrbracket^{\mathsf{c}}$ and the quantum ground set $\llbracket\tau\rrbracket^{\mathsf{q}}$ for all possible types τ. For type $\mathbb{1}$, both the classical set and the quantum ground set are the singleton set $\{()\}$. The (quantum) Boolean type $\mathbb{B}$ 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 $\mathcal{H}(\{0, 1\})$ contains all superpositions of $|0\rangle$ and $|1\rangle$. The sets associated with the product type are standard. Functions store no quantum information, and hence their quantum ground set is $\{()\}$. 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 $!\tau$ store no quantum information and hence their quantum ground set is $\{()\}$. In contrast, their classical set consists of (i) the classical set $\llbracket\tau\rrbracket^{\mathsf{c}}$ which remains classical and (ii) the quantum ground set $\llbracket\tau\rrbracket^{\mathsf{q}}$. As a straightforward consequence of our definition, duplicate classical annotations do not affect the semantics: $\llbracket!!\tau\rrbracket \simeq \llbracket!\tau\rrbracket$.

Type semantics

$$\begin{aligned}
[\![!\mathbb{B}]\!] &= [\![!\mathbb{B}]\!]^{c} \times \mathcal{H}\left([\![!\mathbb{B}]\!]^{q}\right) = \left([\![\mathbb{B}]\!]^{c} \times [\![\mathbb{B}]\!]^{q}\right) \times \mathcal{H}(\{()\}) = \left(\{()\} \times \{0,1\}\right) \times \mathcal{H}(\{()\}) \simeq \{0,1\} \times \mathcal{H}(\{()\})\\
[\![!\mathbb{B} \times \mathbb{B}]\!] &= [\![!\mathbb{B} \times \mathbb{B}]\!]^{c} \times \mathcal{H}\left([\![!\mathbb{B} \times \mathbb{B}]\!]^{q}\right) = \left([\![!\mathbb{B}]\!]^{c} \times [\![\mathbb{B}]\!]^{c}\right) \times \mathcal{H}\left([\![!\mathbb{B}]\!]^{q} \times [\![\mathbb{B}]\!]^{q}\right) = \left(\{()\} \times \{0,1\} \times \{()\}\right) \times \mathcal{H}(\{()\} \times \{0,1\}) \simeq \{0,1\} \times \mathcal{H}(\{0,1\})\\
[\![!\mathbb{B} \times \mathbb{B}]\!]^{+} &= \mathcal{H}\left([\![!\mathbb{B}]\!]^{c} \times [\![\mathbb{B}]\!]^{q}\right) \simeq \mathcal{H}(\{0,1\} \times \{0,1\}) = \mathcal{H}\left(\{0,1\}^2\right) = \left\{ \sum_{w \in \{0,1\}^2} \gamma_w \,|w\rangle \;\middle|\; \gamma_w \in \mathbb{C} \right\}
\end{aligned}$$

Context semantics

$$\begin{aligned}
[\![x\colon !\mathbb{B} \times \mathbb{B}]\!] &= [\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{c} \times \mathcal{H}\left([\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{q}\right) \simeq \{(v)_x \mid v \in \{0,1\}\} \times \mathcal{H}\left(\{(v')_x \mid v' \in \{0,1\}\}\right) = \left\{ \left((v)_x, \gamma_0\,|0\rangle_x + \gamma_1\,|1\rangle_x\right) \;\middle|\; \begin{matrix} v \in \{0,1\} \\ \gamma_0, \gamma_1 \in \mathbb{C} \end{matrix} \right\}\\
[\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{+} &= \mathcal{H}\left([\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{c} \times [\![x\colon !\mathbb{B} \times \mathbb{B}]\!]^{q}\right) \simeq \mathcal{H}\left(\left\{ \underbrace{\left((v)_x, (v')_x\right)}_{(v,v')_x} \;\middle|\; \begin{matrix} v \in \{0,1\} \\ v' \in \{0,1\} \end{matrix} \right\} \right) = \left\{ \sum_{w \in \{0,1\}^2} \gamma_w \,|w\rangle_x \;\middle|\; \gamma_w \in \mathbb{C} \right\}
\end{aligned}$$

Embedding

$$\iota\left([\![x\colon !\mathbb{B} \times \mathbb{B}]\!]\right) \simeq \left\{ \iota\left((v)_x, \gamma_0\,|0\rangle_x + \gamma_1\,|1\rangle_x\right) \;\middle|\; v \in \{0,1\}, \gamma_0, \gamma_1 \in \mathbb{C} \right\} \simeq \left\{ \gamma_0\,|v,0\rangle_x + \gamma_1\,|v,1\rangle_x \;\middle|\; v \in \{0,1\}, \gamma_0, \gamma_1 \in \mathbb{C} \right\}$$

Figure 14. Example semantics of type $!\mathbb{B}$, type $!\mathbb{B} \times \mathbb{B}$, and context $x\colon !\mathbb{B} \times \mathbb{B}$.

$$\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right] := \left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{c} \times \mathcal{H}\left(\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{q}\right) = \left\{(v_1)_{x_1}, \dots, (v_n)_{x_n} \mid v_i \in [\![\tau_i]\!]^{c}\right\} \times \mathcal{H}\left(\left\{(v_1')_{x_1}, \dots, (v_n')_{x_n} \mid v_i' \in [\![\tau_i]\!]^{q}\right\}\right)$$

$$\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{+} := \mathcal{H}\left(\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{c} \times \left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{q}\right) \simeq \left\{ \underbrace{\sum_{w_i \in [\![\tau_i]\!]^{c} \times [\![\tau_i]\!]^{q}} \gamma_{w_1,\dots,w_n} \,|w_1\rangle_{x_1} \otimes \dots \otimes |w_n\rangle_{x_n}}_{\text{standard representation}} \;\middle|\; \gamma_{w_1,\dots,w_n} \in \mathbb{C} \right\}$$

Figure 15. Semantics $\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]$ and extended semantics $\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{+}$ of context $\vec{\beta}\vec{x}\colon \vec{\tau}$.

To illustrate the semantics of types, Fig. 14 provides semantics for two example types. In particular, $[\![!\mathbb{B}]\!]$ is isomorphic to $\{0,1\} \times \mathcal{H}(\{()\})$ — note that it is not formally isomorphic to $\{0,1\}$ because $\mathcal{H}(\{()\}) = \{\gamma\,|()\rangle \mid \gamma \in \mathbb{C}\}$ tracks a (physically irrelevant) global phase $\gamma \in \mathbb{C}$.

Extended Semantic Space. Unfortunately, working with elements $(v, \varphi) \in [\![\tau]\!]$ for $v \in [\![\tau]\!]^{c}$ and $\varphi \in \mathcal{H}\left([\![\tau]\!]^{q}\right)$ is inconvenient because (i) every operation on (v, φ) needs to handle v and φ separately (as they are different mathematical objects) and (ii) some operations, like $(v, \varphi) + (v', \varphi')$ are not defined because $[\![\tau]\!]$ is not a vector space.

Therefore, we define the semantics of expressions (§7.2) and annotations (§7.4) on a more convenient, larger space that also allows superpositions of classical values:

$$[\![\tau]\!]^{+} := \mathcal{H}\left([\![\tau]\!]^{c} \times [\![\tau]\!]^{q}\right) = \mathcal{H}\left([\![\tau]\!]^{c}\right) \otimes \mathcal{H}\left([\![\tau]\!]^{q}\right).$$

Here, (i) the classical and quantum part of $\phi \in [\![\tau]\!]^{+}$ can be handled analogously and (ii) operation + is defined on $[\![\tau]\!]^{+}$. We provide an example of this extended semantics in Fig. 14.

While elements of $[\![\tau]\!]$ can be naturally lifted to $[\![\tau]\!]^{+}$ via embedding $\iota\colon [\![\tau]\!] \to [\![\tau]\!]^{+}$ defined by $\iota(v, \varphi) := |v\rangle \otimes \varphi$, the larger space $[\![\tau]\!]^{+}$ also contains *invalid elements*, namely those where classical values are in superposition. In contrast, *valid elements* do not put classical values in superposition, i.e., the classical part of every summand in their superposition coincides. Our semantics exclusively produces valid elements, as we formally prove in Thm. 7.1.

Semantics of Contexts. Fig. 15 provides the semantics of context $\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]$. Here, $(v_i)_{x_i}$ indicates that variable x_i stores value v_i. Fig. 14 provides semantics for an example context, where we write $|0\rangle_x$ as a short-hand for $|(0)_x\rangle$.

Analogously to $[\![\tau]\!]^{+}$, Fig. 15 also introduces the extended semantics $\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{+}$ 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 $\iota\colon \left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right] \to \left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{+}$ is

$$\iota\left((\vec{v})_{\vec{x}}, \sum_{\vec{v}'} \gamma_{\vec{v}'}\,|\vec{v}'\rangle\right) = \sum_{\vec{v}'} \gamma_{\vec{v}'}\,|\vec{v}, \vec{v}'\rangle_{\vec{x}},$$

for $(\vec{v})_{\vec{x}} \in \left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{c}$ and $\sum_{\vec{v}'} \gamma_{\vec{v}'}\,|\vec{v}'\rangle_{\vec{x}} \in \mathcal{H}\left(\left[\!\!\left[\vec{\beta}\vec{x}\colon \vec{\tau}\right]\!\!\right]^{q}\right)$. We illustrate this in Fig. 14 on an example context.

7.2 Semantics of Expressions

Our operational semantics evaluates an expression e in state ψ 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 $\left[\Gamma \vdash^{\alpha} e\colon \tau \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi_i'$ for each possible sequence of measurement results, indicating that evaluating e (typed as $\Gamma \vdash^{\alpha} e\colon \tau$), on state ψ yields state ψ_i' with probability $\left\|\psi_i'\right\|^2$,

$$\frac{\left[\texttt{const}\ x\colon \mathbb{B}, \texttt{const}\ y\colon \mathbb{B} \vdash^{\alpha} x \,||\, y\colon \mathbb{B} \,\middle|\, |0\rangle_x |0\rangle_y |1\rangle_z\right] \xrightarrow{\text{run}} |0\rangle_x |0\rangle_y |1\rangle_z |0\rangle_{\underline{x||y}} \qquad \left[\texttt{const}\ z\colon \mathbb{B} \vdash^{\alpha} z\colon \mathbb{B} \,\middle|\, |1\rangle_z |0\rangle_x |0\rangle_y |0\rangle_{\underline{x||y}}\right] \xrightarrow{\text{run}} |0\rangle_x |0\rangle_y |1\rangle_z |0\rangle_{\underline{x||y}} |1\rangle_{\underline{z}} \qquad \cdots}{\left[\texttt{const}\ x\colon \mathbb{B}, \texttt{const}\ y\colon \mathbb{B}, \texttt{const}\ z\colon \mathbb{B} \vdash^{\alpha} (x \,||\, y) \,||\, z\colon \mathbb{B} \,\middle|\, |0\rangle_x |0\rangle_y |1\rangle_z\right] \xrightarrow{\text{run}} |0\rangle_x |0\rangle_y |1\rangle_z |1\rangle_{\underline{x||y||z}}}$$

Figure 16. Part of derivation when evaluating expression $x \,||\, y \,||\, z$ on state $\psi = |0\rangle_x |0\rangle_y |1\rangle_z$. For a complete semantics derivation tree that leverages more rules, see App. F.3.

$$\left[x\colon !\mathbb{B} \vdash^{\texttt{mfree},\alpha} 1\colon \mathbb{B} \,\middle|\, |0\rangle_x\right] \xrightarrow{\text{run}} |1\rangle_{\underline{1}}$$
$$\langle\cdot|\cdot\rangle \neq \langle\cdot|\cdot\rangle$$
$$\left[x\colon !\mathbb{B} \vdash^{\texttt{mfree},\alpha} 1\colon \mathbb{B} \,\middle|\, |1\rangle_x\right] \xrightarrow{\text{run}} |1\rangle_{\underline{1}}$$

Figure 17. Non-isometry.

assuming $\|\psi\|^2 = 1$ (see §3). If e is undefined for a given input ψ (possible since **reverse** returns unsafe functions), we do not provide any evaluation.

***Domain of* ψ, ψ'.** When our semantics evaluates e according to $\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi'$, it requires that $\psi \in \iota(\llbracket\Gamma, \Delta\rrbracket)$. By construction, for context $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, output state ψ' lies in $\llbracket\texttt{const}\ \vec{x}\colon \vec{\tau}, \Delta, \underline{e}\colon \tau''\rrbracket^{+}$, i.e., we preserve constant variables $\vec{x}$ and the additional context Δ (discussed next), and store the value of e in a temporary variable $\underline{e}$.

Here, Δ is additional context containing the remainder of the state preserved while evaluating e. We illustrate the need for Δ in Fig. 16. Here, we cannot evaluate $(x \,||\, y)$ and z independently, as their values may be entangled in ψ. Hence, we must evaluate $x \,||\, y$ in a state that not only contains x, y (in the context when typing $x \,||\, y$, cp. blue box), but also z (in Δ, cp. red box). After this, we evaluate z in a state containing z (in the context when typing z), x, y (in Δ), and the value of $x \,||\, y$ (stored as $\underline{x \,||\, y}$ in Δ).

Formal Semantics. Here, we discuss the most important aspects of the formal semantics of Silq-core expressions (see App. F.1 for details, and App. F.3 for an example). Recall that the structure of semantic derivation trees follows the structure of the type derivation trees, and hence, every type rule corresponds to a semantic derivation rule.

The semantics of evaluating a variable x is to rename x to $\underline{x}$ in the new state if x is consumed, and to duplicate x to $\underline{x}$ if x is constant. Contraction of constant variable x duplicates x, according to $\sum_v \gamma_v |v\rangle_x \otimes \tilde{\psi}_v \mapsto \sum_v \gamma_v |v\rangle_x |v\rangle_x \otimes \tilde{\psi}_v$. Weakening of constant variables postpones uncomputing them until the end of the function body. When evaluating a function call $e'(e_1, \ldots, e_n)$, we uncompute the constant arguments e_i (i.e., preserved according to the signature of e') at the end of the function call. To reverse functions, we postpone the reversal until the reversed function is called. We handle control flow **if** e **then** e_1 **else** e_2 by separately evaluating e_1 (respectively e_2) in the part of the state where e is true (respectively false).

7.3 Type Preservation

Thm. 7.1 ensures that our semantics never produces invalid states ψ', meaning that the classical values in ψ' can never be in superposition (since ι only returns *valid* elements).

Theorem 7.1 (Type Preservation). *If* $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, $\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \psi\right] \xrightarrow{run} \psi'$, *and* $\psi \in \iota(\llbracket\Gamma, \Delta\rrbracket)$, *then* ψ' *lies in* $\iota\left(\llbracket\texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'', \Delta\rrbracket\right)$.

We provide a proof for Thm. 7.1 in App. G. Here, $\iota(\llbracket\Gamma, \Delta\rrbracket)$ contains all elements of $\llbracket\Gamma, \Delta\rrbracket^{+}$ where classical values are not in superposition.

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 ! 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* $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, $\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \psi\right] \xrightarrow{run} \psi'$, *and* $\psi = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}$, *then* $\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} |\vec{v}\rangle_{\vec{x}} \otimes \chi_{\vec{v},\vec{w}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}$ *for some* $\chi_{\vec{v},\vec{w}}$.

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: Fig. 17 shows an example where $\xrightarrow{\text{run}}$ is not isometric because we drop a classical value from the input. Thm. 7.3 side-steps this issue, intuitively by ensuring that our semantics is isometric when the classical components of its input are fixed.

Theorem 7.3 (Mfree Semantics). *If* $\texttt{mfree} \in \alpha$, $\sigma \in \llbracket\Gamma, \Delta\rrbracket^{c}$,

$$\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \iota(\sigma, \psi_1)\right] \xrightarrow{run} \psi_1' \quad \textit{for} \quad \psi_1 \in \mathcal{H}\left(\llbracket\Gamma, \Delta\rrbracket^{q}\right), \textit{and}$$
$$\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \iota(\sigma, \psi_2)\right] \xrightarrow{run} \psi_2' \quad \textit{for} \quad \psi_2 \in \mathcal{H}\left(\llbracket\Gamma, \Delta\rrbracket^{q}\right),$$

then $\langle\psi_1|\psi_2\rangle = \langle\psi_1'|\psi_2'\rangle$.

We provide a proof for Thm. 7.3 in App. G.

A useful interpretation of Thm. 7.3 states that $\xrightarrow{\text{run}}$ acts like an isometry on the subspace consistent with a fixed classical component $\sigma \in [\![\Gamma, \Delta]\!]^c$,

$$\{\iota(\sigma, \chi) \mid \chi \in \mathcal{H}\left([\![\Gamma, \Delta]\!]^q\right)\} \subseteq [\![\Gamma, \Delta]\!]^+ .$$

This corresponds to the intuition that in order to evaluate e on ψ_1, we can (i) extract the classical component σ from ψ_1, (ii) build a circuit C that realizes the linear isometry for this classical component and (iii) run C, yielding ψ_1'.

Qfree Expressions. Thm. 7.4 ensures that qfree expressions can be described by a function $\bar{f}$ on the ground sets.

Theorem 7.4 (Qfree Semantics)**.** *If* $\Gamma \vdash^{\alpha} e\colon \tau''$ *for* $\textbf{qfree} \in \alpha$ *and context* $\Gamma = \textbf{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, *then there exists a function* $\bar{f}\colon [\![\Gamma]\!]^s \to [\![\textbf{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'']\!]^s$ *on ground sets such that*

$$\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \sum_{\sigma \in [\![\Gamma]\!]^s} \gamma_\sigma \left|\sigma\right\rangle \otimes \tilde{\psi}_\sigma \right] \xrightarrow{run} \sum_{\sigma \in [\![\Gamma]\!]^s} \gamma_\sigma \left|\bar{f}(\sigma)\right\rangle \otimes \tilde{\psi}_\sigma,$$

where $[\![\Gamma]\!]^s$ *is a shorthand for the ground set* $[\![\Gamma]\!]^c \times [\![\Gamma]\!]^q$ *on which the Hilbert space* $[\![\Gamma]\!]^+ = \mathcal{H}\left([\![\Gamma]\!]^s\right)$ *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 uncompute 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#.

	Silq			Q#		
	S18	W19	Both	S18	W19	Both
Lines of code	**99**	**168**	**267**	251	242	493
Quantum primitives	**8**	**10**	**10**	12	19	22
Annotations	**2**	**3**	**3**	3	6	6
Low-level quantum gates	**14**	**23**	**37**	33	54	87

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. [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`, `adjoint self`, `Controlled`, . . .) and Silq (**`mfree`**, **`qfree`**, **`const`**, **`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

Table 2. Comparing Silq to other quantum languages. Parenthesized features are partially (but not fully) supported.

Language	Type system	Autom. Uncomp.	**const**	**mfree**	**qfree**
QPL [27]	linear[11]	✗	✗	✗	✗
Quantum λ-calc. [28]	affine	✗	✗	✗	✗
Quipper [7]	non-linear	(✓)	✗	✗	✗
ReVerC [2]	non-linear	(✓)	✗	✗	(✗)
QWire [22]	linear	✗	✗	✗	✗
Q# [30]	non-linear	✗	✗	✓	✗
ReQWire [23]	linear	(✗)	(✗)	✗	(✗)
Silq (this work)	linear+	✓	✓	✓	✓

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 $0 \leq j < k < 2^{\text{rbar}}$, and counts how often `ee[tau[j]][tau[k]] && eew[j] && eew[k]` is `true`, where we use quantum indexing into `ee`. In contrast, Quipper's code is cluttered by explicit uncomputation (e.g., of `eedd_k`), custom functions aiding uncomputation (e.g., `.&&.`), and separate initialization and assignment (e.g., `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 `gate_T` and `gate_T_at` or due to definition of inverse gates, e.g., `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.

[11] QPL (i) enforces no-cloning syntactically and (ii) disallows implicitly dropping variables (cp. rule `discard` in Fig. 12)

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 **`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.

References

[1] Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. Qiskit: An Open-source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562110

[2] Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified Compilation of Space-Efficient Reversible Circuits. In *CAV'17*. Vol. 10427. Cham, 3–21. https://doi.org/10.1007/978-3-319-63390-9_1

[3] Charles H Bennett. 1973. Logical Reversibility of Computation. *IBM Journal of Research and Development* 17, 6 (Nov. 1973), 525–532. https://doi.org/10.1147/rd.176.0525

[4] Andrew M. Childs and Robin Kothari. 2011. Quantum query complexity of minor-closed graph properties. In *STACS'11* (Dagstuhl, Germany, 2011) *(Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 9. 661–672. https://doi.org/10.4230/LIPIcs.STACS.2011.661

[5] Vedran Dunjko, Jacob M. Taylor, and Hans J. Briegel. 2016. Quantum-Enhanced Machine Learning. *Physical Review Letters* 117, 13 (Sept. 2016). https://doi.org/10.1103/physrevlett.117.130501

[6] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. Psi: Exact symbolic inference for probabilistic programs. In *CAV'16*. 62–83.

[7] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *PLDI'13*. ACM Press, Seattle, Washington, USA. https://doi.org/10.1145/2491956.2462177

[8] Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. ACM, 212–219.

[9] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. 2009. Quantum Algorithm for Linear Systems of Equations. *Phys. Rev. Lett.* 103 (Oct 2009), 150502. Issue 15. https://doi.org/10.1103/PhysRevLett.103.150502

[10] Steve Klabnik and Carol Nichols. 2018. *The Rust programming language*. No Starch Press, San Francisco.

[11] Emmanuel Knill. 1996. *Conventions for quantum pseudocode*. Technical Report. Citeseer.

[12] Seth Lloyd, Masoud Mohseni, and Patrick Rebentrost. 2014. Quantum principal component analysis. *Nature Physics* 10, 9 (July 2014), 631–633. https://doi.org/10.1038/nphys3029

[13] Guang Hao Low, Theodore J. Yoder, and Isaac L. Chuang. 2014. Quantum inference on Bayesian networks. *Phys. Rev. A* 89, 6 (June 2014), 062315. https://doi.org/10.1103/PhysRevA.89.062315

[14] F. Magniez, M. Santha, and M. Szegedy. 2007. Quantum Algorithms for the Triangle Problem. 37, 2 (2007), 413–424. https://doi.org/10.1137/050643684

[15] Microsoft. 2018. Public submissions of the Microsoft Q# Coding Contest - Summer 2018. (2018). https://codeforces.com/contest/1002/

[16] Microsoft. 2019. Public submissions of the Microsoft Q# Coding Contest - Winter 2019. (2019). https://codeforces.com/contest/1116/

[17] Mariia Mykhailova and Martin Roetteler. 2018. Microsoft Q# Coding Contest - Summer 2018 - Main Contest July 6-9, 2018. (2018). https://assets.codeforces.com/rounds/997-998/main-contest-editorial.pdf

[18] Mariia Mykhailova and Martin Roetteler. 2019. Microsoft Q# Coding Contest - Winter 2019 - Main Contest March 1-4, 2019. (2019). https://assets.codeforces.com/rounds/1116/contest-editorial.pdf

[19] Peter Müller and Arnd Poetzsch-Heffter. 1999. Universes: a type system for controlling representation exposure.

[20] Daniel Nagaj, Or Sattath, Aharon Brodutch, and Dominique Unruh. 2016. An Adaptive Attack on Wiesner's Quantum Money. *Quantum Info. Comput.* 16, 11-12 (Sept. 2016), 1048–1070. http://dl.acm.org/citation.cfm?id=3179330.3179337

[21] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum computation and quantum information* (10th anniversary ed ed.). Cambridge University Press, Cambridge ; New York.

[22] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: a core language for quantum circuits. In *POPL'17*. ACM Press, Paris, France. https://doi.org/10.1145/3009837.3009894

[23] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2019. ReQWIRE: Reasoning about Reversible Quantum Circuits. *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 2019), 299–312. https://doi.org/10.4204/EPTCS.287.17 arXiv: 1901.10118.

[24] Patrick Rebentrost, M Mohseni, and Seth Lloyd. 2013. Quantum Support Vector Machine for Big Data Classification. *Physical Review Letters* 113 (2013).

[25] Steven Roman. 2008. *Advanced Linear Algebra*. New York, NY. http://site.ebrary.com/id/10230315 OCLC: 730328666.

[26] Neil J. Ross and Peter LeFanu Lumsdaine. 2015. Algorithms.TF.Main. https://www.mathstat.dal.ca/~selinger/quipper/doc/Algorithms-TF-Main.html.

[27] Peter Selinger. 2004. Towards a Quantum Programming Language. *Mathematical. Structures in Comp. Sci.* 14, 4 (Aug. 2004), 527–586. https://doi.org/10.1017/S0960129504004256

[28] Peter Selinger and Benoit Valiron. 2006. A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science* 16, 03 (June 2006), 527. https://doi.org/10.1017/S0960129506005238

[29] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum* 2 (Jan. 2018), 49. https://doi.org/10.22331/q-2018-01-31-49

[30] Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. ACM Press, Vienna, Austria. https://doi.org/10.1145/3183895.3183901

[31] Google AI Quantum Team. 2017. Cirq. (2017). https://github.com/quantumlib/Cirq

[32] Dave Wecker and Krysta M Svore. 2014. LIQUi|>: A software design architecture and domain-specific language for quantum computing. *arXiv preprint arXiv:1402.4467* (2014).

[33] Nathan Wiebe, Daniel Braun, and Seth Lloyd. 2012. Quantum Algorithm for Data Fitting. *Physical Review Letters* 109, 5 (Aug. 2012). https://doi.org/10.1103/physrevlett.109.050505

[34] Stephen Wiesner. 1983. Conjugate Coding. *SIGACT News* 15, 1 (Jan. 1983), 78–88. https://doi.org/10.1145/1008908.1008920

```
cTri <- foldM (\cTri j -> do
  let tau_j = tau ! j
  eed <- qinit (intMap_replicate rr False)
  -- computing eed = ee[tau[j]]
  (taub,ee,eed) <- a11_FetchE tau_j ee eed
  cTri <- foldM (\cTri k -> do
    let tau_k = tau ! k
    eedd_k <- qinit False
    -- eedd_k=eed[tau[k]]=ee[tau[j]][tau[k]]
    (tauc, eed, eedd_k) <- qram_fetch qram tau_k eed eedd_k
    -- using eedd_k as ctrl
    cTri <- increment cTri `controlled` eedd_k .&&. (eew ! j) .&&. (eew ! k)
    -- uncomputing eedd_k
    (tauc, eed, eedd_k) <- qram_fetch qram tau_k eed eedd_k
    qterm False eedd_k
    return cTri)
   cTri [j+1..rrbar-1]
  -- uncomputing eed
  (taub,ee,eed) <- a11_FetchE tau_j ee eed
  qterm (intMap_replicate rr False) eed
  return cTri)
 cTri [0..rrbar-1]
```

Figure 18. Quipper code from Fig. 2.

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 Bennet'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.

```
index  : ∏(n:Nat,i:Nat) . CIRC(t[n]         ,t[n]⊗         t)
     = ...
qindex : ∏(n:Nat,m:Nat) . CIRC(t[n]⊗qubit[m],t[n]⊗qubit[m]⊗t)
     = ...
controlledInc : ∏(n:Nat). CIRC(qubit[n]⊗qubit,qubit[n]⊗qubit)
     = ...

EvalCondition : ∏(r:Nat,rrbar:Nat,j:Nat,k:Nat). CIRC(
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar],...⊗qubit
) = box(ee,tau,eew) =>
  (tau,tauj) <- unbox (index rrbar j) tau; -- tauj=tau[j]
  (tau,tauk) <- unbox (index rrbar k) tau; -- tauk=tau[k]
  (ee, tauj, eed) <- unbox (qindex rrbar r) ee tauj; -- eed=ee[tauj]
  (eed,tauk,eedd_k) <- unbox (qindex rrbar r) eed tauk; -- eedk=eed[tauk]
  (eew,eewj) <- unbox (index rrbar j) eew; -- eewj=eew[j]
  (eew,eewk) <- unbox (index rrbar k) eew; -- eewk=eew[k]
  (eedd_k,eewj,eewk,c) <- unbox and eedd_k eewj eewk; -- condition
  output (ee,tau,eew,tauj,tauk,eed,eedd_k,eewj,eewk,c) -- output

LoopBody : ∏(r:Nat,rrbar:Nat,j:Nat,k:Nat). CIRC(
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar]⊗qubit[rrbar],
  qubit[rrbar][rrbar]⊗qubit[rrbar][r]⊗qubit[rrbar]⊗qubit[rrbar]
) = box (ee,tau,eew,cTri) =>
  (ee,tau,eew,tauj,tauk,eed,eedd_k,eewj,eewk,c) <- unbox (EvalCondition r rrbar j k) ee tau eew; -- evaluate condition
  (cTri,c) <- unbox (controlledInc rrbar) cTri c; -- controlled increment
  (ee,tau,eew) <- unbox (reverseIsometric EvalCondition r rrbar j k) ee tau eew tauj tauk eed eedd_k eewj eewk c -- uncompute
  output (ee,tau,eew,cTri) -- output
```

Figure 19. QWire code from Fig. 2.

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.

```
def groverDiff[n:!ℕ](cand:uint[n]){
  for k in [0..n) { cand[k] := H(cand[k]); }
  if cand!=0 {
    phase(π);
  }
  for k in [0..n) { cand[k] := H(cand[k]); }
  return cand;
}

def grover[n:!ℕ](f:uint[n]! → lifted 𝔹){
  nIterations:= floor(π/4/asin(2^(-n/2)));
  cand:=0:uint[n];
  for k in [0..n) { cand[k] := H(cand[k]); }

  for k in [0..nIterations){
    if f(cand) { phase(π); }
    cand:=groverDiff(cand);
  }
  return measure(cand);
}
```

Figure 20. Grover's diffusion operator in Silq.

```
def PeriodFinding[n:!ℕ](f:!(const uint[n]→qfree uint[n])):!ℕ{
  cand := 0:uint[n];
  for k in [0..n) { cand[k] ~= H(cand[k]); }
  measure(f(cand));
  cand := reverse(QFT[n])(cand);
  return measure(cand);
}

def QFT[n:!ℕ](x: uint[n])mfree: uint[n]{
    for k in [0..n div 2){
        (x[k],x[n-k-1]) := (x[n-k-1],x[k]);
    }
    for k in [0..n){
        x[k] := H(x[k]);
        for l in [k+1..n){
            if x[l] && x[k]{
                phase(2*π*2^(k-l-1));
            }
        }
    }
    return x;
}
```

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

```
1 def nonQfree(const x:𝔹,y:𝔹){
2   t := dup(x);
3   t := H(t);
4   if t{
5     y := X(y);
6   }
7   // uncompute t
8 }
```

(a) Rewritten version of `nonQfree` that makes its individual operations more explicit.

$$\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|0\right\rangle_y+\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|1\right\rangle_y$$

$$\xrightarrow{\text{Line 2}}\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|0\right\rangle_y\left|1\right\rangle_t+\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|1\right\rangle_y\left|1\right\rangle_t$$

$$\xrightarrow{\text{Line 3}}\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|0\right\rangle_y\left|-\right\rangle_t+\tfrac{1}{\sqrt{2}}\left|1\right\rangle_x\left|1\right\rangle_y\left|-\right\rangle_t$$

$$=\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y\left|0\right\rangle_t-\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y\left|1\right\rangle_t+\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y\left|0\right\rangle_t-\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y\left|1\right\rangle_t$$

$$\xrightarrow{\text{Lines 4–6}}\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y\left|0\right\rangle_t-\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y\left|1\right\rangle_t+\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y\left|0\right\rangle_t-\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y\left|1\right\rangle_t$$

$$\xrightarrow{\text{Lines 7}}\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y-\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y+\frac{1}{2}\left|1\right\rangle_x\left|1\right\rangle_y-\frac{1}{2}\left|1\right\rangle_x\left|0\right\rangle_y=0$$

(b) Semantics of `nonQfree`.

Figure 22. Semantics of `nonQfree` on input $\frac{1}{\sqrt{2}}\left|1\right\rangle_\mathtt{x}\left|0\right\rangle_\mathtt{y}+\frac{1}{\sqrt{2}}\left|1\right\rangle_\mathtt{x}\left|1\right\rangle_\mathtt{y}$ when uncomputing the condition.

Expressions	e
Constants	c
Variables	$x, y, z, x_i, \ldots$

(a) Symbols used in grammar.

Type	$\mathbb{1}, \mathbb{B}, \ldots$
Type (meta variable)	$\tau, \tau', \tau_i, \tau_i'$
Annotation	!, **const**, **qfree**, **mfree**
Annotation (meta variable)	$\alpha, \alpha', \alpha_i, \alpha_i', \beta, \beta_i$
Context	Γ, Γ_i, Δ

(b) Symbols used in type system.

Indices	i, j, k
Numbers	n, m, l
Value	$v, v', \vec{v}, \vec{v}', b$
Storage (of form $(\vec{v})_{\vec{x}}$)	σ
Unnamed state (of form $\sum_{\vec{v}}\gamma_{\vec{v}}\left|\vec{v}\right\rangle$)	φ, ϕ
Named state (of form $\sum_{\vec{v}}\gamma_{\vec{v}}\left|\vec{v}\right\rangle_{\vec{x}}$)	ψ, χ
Remainder of state (as in $\sum_v \gamma_v'\left|v\right\rangle\otimes\tilde{\varphi}_v$)	$\tilde{\varphi}, \tilde{\phi}, \tilde{\psi}$
Imaginary unit	i
Coefficients	$\gamma_v, \gamma_{\vec{v}}, \gamma_\sigma, \ldots$
Identity	$\mathbb{I}$
Embedding (injection)	ι

(c) Symbols used in semantics.

Figure 23. Notational conventions used in this work.

H	$\mathbb{B}$	$!\xrightarrow{\texttt{mfree}}$	$\mathbb{B}$
phase	$!\texttt{float}$	$!\xrightarrow{\texttt{mfree}}$	$\mathbb{1}$
rotX, **rotY**, **rotZ**	$!\texttt{float} \times \mathbb{B}$	$!\xrightarrow{\texttt{mfree}}$	$\mathbb{B}$
X	$\mathbb{B}$	$!\xrightarrow{\texttt{qfree,mfree}}$	$\mathbb{B}$
Y, **Z**	$\mathbb{B}$	$!\xrightarrow{\texttt{mfree}}$	$\mathbb{B}$
dup	$\texttt{const}\ \tau$	$!\xrightarrow{\texttt{qfree,mfree}}$	τ
$(\cdot, \ldots, \cdot)$	$\bigtimes_{i=1}^{n} \tau_i$	$!\xrightarrow{\texttt{qfree,mfree}}$	$\bigtimes_{i=1}^{n} \tau_i$
forget$(\cdot = \cdot)$	$\tau \times \texttt{const}\ \tau$	$!\xrightarrow{\texttt{qfree,mfree}}$	$\mathbb{1}$
$\cdot \oplus \cdot$	$\texttt{const uint} \times \texttt{const uint}$	$!\xrightarrow{\texttt{qfree}}$	$\texttt{uint}$

Figure 24. Types of selected built-in functions.

$$\frac{\Gamma_i \vdash e_i : \tau_i \qquad \Gamma' \vdash e' : \bigtimes_{i=1}^{n} \tau_i \to \tau'}{\Gamma_1, \ldots, \Gamma_n, \Gamma' \vdash e'(e_1, \ldots, e_n) : \tau'}$$

$$\frac{\vec{x} : \vec{\tau}, \vec{y} : \vec{\tau}' \vdash e : \tau''}{\vec{y} : \vec{\tau}' \vdash \lambda(\vec{x} : \vec{\tau}). e : \bigtimes_{i=1}^{n} \tau_i \to \tau''}$$

$$\frac{\Gamma_c \vdash e : \mathbb{B} \qquad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma_c, \Gamma \vdash \textbf{if}\ e\ \textbf{then}\ e_1\ \textbf{else}\ e_2 : \tau}$$

Figure 25. The basic patterns of our typing rules (ignoring annotations) are standard for a linear type system.

Hadamard, Phase. The parameter of **H** is not **const**, meaning that evaluating **H** consumes its argument (the argument is not available after the call). In contrast, the parameters of $\oplus$ are **const**, meaning that adding two expressions preserves them. Further, **H** is only **mfree**, while $\oplus$ is **qfree** and **mfree**. The function **phase** requires a classical phase (**!float**), and does not return anything (indicated by $\mathbb{1}$).

Dup, Tupling. Function **dup** returns a copy of its argument, without changing the argument (indicated by **const** τ). For tupling $(\cdot, \ldots, \cdot)$, our typing rule relies on the implicit tupling by function calls (see Fig. 9). It consumes its arguments and is **qfree**. As an implicit consequence, $(e_1, \ldots, e_n)$ is classical if all e_i's are classical.

Forget. Function **forget**$(e_1 = e_2)$ leaves its second argument constant, but consumes its first. This allows us to *forget* e_1.

F Semantics

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

F.1 Semantics of Expressions

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, $\mathbb{I}_{x \to \underline{x}}$ renames x to $\underline{x}$ in ψ without affecting other variables in ψ (shortly discussed in more detail). In contrast, the rule for constant variables preserves x and introduces an explicit duplicate $\underline{x}$ by $[\![\textbf{dup}]\!]$, 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 "$x \to \underline{x}$" in Fig. 31, which shows how to apply $[\![\textbf{X}]\!]_{x \to y}$ to state $|b\rangle_{x} \otimes |\vec{w}\rangle_{\vec{z}}$, where the formal definition of $[\![\textbf{X}]\!]$ is $[\![\textbf{X}]\!]\,(|b\rangle) = |1 - b\rangle$ (see App. F.2).

Here, the subscript $x \to y$ of **X** ensures that we (i) preserve $|\vec{w}\rangle_{\vec{z}}$ (cp. Eq. (10–11)) and (ii) run **X** on $|b\rangle_{x}$ and name the output y (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 $|(v, v')\rangle_x$. As a consequence, we know that if $\vec{z}$ 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 $\text{drop}^{(x)}\left(|v\rangle_x\,|\vec{w}\rangle_{\vec{y}}\right) = |\vec{w}\rangle_{\vec{y}}$. 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 classical 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 $e'(\vec{e})$. First, the rule evaluates all arguments, resulting in state ψ_n. Second, the rule evaluates e', resulting in state ψ_{n+1} containing the function e'' to be evaluated, which may capture variables σ. 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 $[e''(\vec{e}) \mid \sigma \mid \psi_{n+1}] \xrightarrow{\text{eval}} \psi_{n+2}$. In contrast to run-transitions, eval-transitions assume that all arguments $\vec{e}$ are already evaluated in ψ_{n+1} (as guaranteed by run-transitions). Finally, the rule drops the **const** arguments of e'' by uncomputing them, and renames the output value from $\underline{\text{ret}}$ to $\underline{e'(\vec{e})}$.

$$\frac{}{\left[\emptyset \overset{\alpha}{\vdash} c\colon \tau \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi \otimes |c\rangle_{\underline{c}}}\ \text{const} \qquad \frac{}{\left[x\colon \tau \overset{\alpha}{\vdash} x\colon \tau \,\middle|\, \psi\right] \xrightarrow{\text{run}} \mathbb{I}_{x\to\underline{x}}(\psi)}\ \text{var} \qquad \frac{}{\left[\texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} x\colon \tau \,\middle|\, \psi\right] \xrightarrow{\text{run}} \texttt{dup}_{x\to x,\underline{x}}(\psi)}\ \text{var-const}$$

Figure 26. Semantics of constants and variables.

$$\frac{\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau' \,\middle|\, \text{drop}^{(x)}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau' \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi'}\ !\text{W} \qquad \frac{\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau' \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau' \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi'}\ \text{W} \qquad \frac{\left[\Gamma, x\colon !\tau, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau' \,\middle|\, \texttt{dup}_{x\to x,x}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau' \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi'}\ !\text{C}$$

$$\frac{\left[\Gamma, \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau' \,\middle|\, \texttt{dup}_{x\to x,x}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau' \,\middle|\, \psi\right] \xrightarrow{\text{run}} \text{drop}^{(x)}(\psi')}\ \text{C}$$

Figure 27. Semantics of contraction and weakening.

$$\frac{\overbrace{\left[\Gamma_i \overset{\alpha_i}{\vdash} e_i\colon \tau_i \,\middle|\, \psi_{i-1}\right] \xrightarrow{\text{run}} \psi_i' \quad \psi_i = \psi_i'}^{\text{evaluate arguments}} \qquad \overbrace{\left[\Gamma' \overset{\alpha'}{\vdash} e'\colon \bigtimes_{i=1}^{n} \beta_i\tau_i ! \xrightarrow{\alpha'_{\text{func}}} \tau' \,\middle|\, \psi_n\right] \xrightarrow{\text{run}} \psi_{n+1} \otimes |e'', \underbrace{\sigma}_{\text{captured in } e''}\rangle_{\underline{e'}}}^{\text{determine function}} \qquad \overbrace{\left[e''(\vec{e})\colon \bigtimes_{i=1}^{n} \beta_i\tau_i ! \xrightarrow{\alpha'_{\text{func}}} \tau' \,\middle|\, \sigma\colon \Gamma' \,\middle|\, \psi_{n+1}\right] \xrightarrow{\text{eval}} \psi_{n+2}}^{\text{evaluate function}}}{\left[\vec{\Gamma}, \Gamma' \overset{\alpha''}{\vdash} e'(\vec{e})\colon \tau' \,\middle|\, \psi_0\right] \xrightarrow{\text{run}} \underbrace{\mathbb{I}_{\underline{\text{ret}}\to\underline{e'(\vec{e})}}}_{\text{name result}} \circ \underbrace{\text{drop}^{(\{\underline{e_i} | \texttt{const}\in\beta_i\})}}_{\text{uncompute}}(\psi_{n+2})}\ \text{func-eval}$$

$$\frac{\left[\vec{\beta}\vec{x}\colon \vec{\tau}, \Gamma \overset{\alpha}{\vdash} e''\colon \tau' \,\middle|\, \overbrace{\mathbb{I}_{\underline{\vec{e}}\to\vec{x}}}^{\text{name arguments}}\big(\psi \otimes |\sigma\rangle\big)\right] \xrightarrow{\text{run}} \psi'}{\left[\big(\lambda(\vec{\beta}\vec{x}\colon \vec{\tau}).e''\big)(\vec{e})\colon \bigtimes_{i=1}^{n} \beta_i\tau_i\, ! \xrightarrow{\alpha} \tau' \,\middle|\, \sigma\colon \Gamma \,\middle|\, \psi\right] \xrightarrow{\text{eval}} \underbrace{\mathbb{I}_{\underline{e''}\to\underline{\text{ret}}}}_{\text{name return value}} \circ \underbrace{\mathbb{I}_{x_i\to\underline{e_i} | \texttt{const}\in\beta_i}}_{\text{reset variable names}}(\psi')}\ \text{eval-}\lambda\text{-abs}$$

$$\frac{\psi = \sum_{v} \gamma_v\, |v\rangle_{\underline{e}} \otimes \tilde{\psi}_v \qquad v' \in [\![\tau]\!]^{\text{c}} \times [\![\tau]\!]^{\text{q}}}{\left[\texttt{measure}(e)\colon \tau\, ! \xrightarrow{\alpha} !\tau \,\middle|\, \emptyset\colon \emptyset \,\middle|\, \psi\right] \xrightarrow{\text{eval}} \gamma_{v'}\, |v'\rangle_{\underline{\text{ret}}} \otimes \tilde{\psi}_{v'}}\ \text{measure} \qquad \frac{}{\left[c(\vec{e})\colon \bigtimes_{i=1}^{n} \beta_i\tau_i\, ! \xrightarrow{\alpha} \tau' \,\middle|\, \emptyset\colon \emptyset \,\middle|\, \psi\right] \xrightarrow{\text{eval}} [\![c]\!]_{\underline{\vec{e}}\to(\underline{e_i})_{i|\texttt{const}\in\beta_i}, \underline{\text{ret}}}(\psi)}\ \text{built-in-eval}$$

Figure 28. Semantics of function calls.

$$\frac{\psi = |e_{\text{func}}, \sigma\rangle_{\underline{e}} \otimes \tilde{\psi}}{\left[\texttt{reverse}(e)\colon \left(\bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{j=1}^{m} \tau_j' ! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{k=1}^{l} \tau_k''\right) ! \xrightarrow{\texttt{mfree},\texttt{qfree}} \left(\bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{k=1}^{l} \tau_k'' ! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{j=1}^{m} \tau_j'\right) \,\middle|\, \emptyset \,\middle|\, \psi\right] \xrightarrow{\text{eval}} |\texttt{reverse}(e_{\text{func}}), \sigma\rangle_{\underline{\text{ret}}} \otimes \tilde{\psi}}\ \text{rev}$$

$$\frac{\left[e_{\text{func}}(\vec{e}^{\text{c}}, \vec{t})\colon \bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{j=1}^{m} \tau_j' ! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{k=1}^{l} \tau_k'' \,\middle|\, \sigma\colon \Gamma \,\middle|\, \psi'\right] \xrightarrow{\text{eval}} \mathbb{I}_{\underline{\vec{e}^{\not{c}}}\to\underline{\text{ret}}}(\psi) \qquad \psi' \in \left[\!\!\left[\underline{\vec{e}^{\text{c}}}\colon \bigtimes_{i=1}^{n} \texttt{const}\ \tau_i, \vec{t}\colon \bigtimes_{j=1}^{m} \tau_j', \Delta\right]\!\!\right]^{+} \qquad \Delta \text{ according to } \psi}{\left[\texttt{reverse}(e_{\text{func}})(\vec{e}^{\text{c}}, \vec{e}^{\not{c}})\colon \bigtimes_{i=1}^{n} \texttt{const}\ \tau_i \times \bigtimes_{k=1}^{l} \tau_k'' ! \xrightarrow{\texttt{mfree},\alpha} \bigtimes_{j=1}^{m} \tau_j' \,\middle|\, \sigma\colon \Gamma \,\middle|\, \psi\right] \xrightarrow{\text{eval}} \mathbb{I}_{\underline{\vec{t}}\to\underline{\text{ret}}}(\psi')}\ \text{call-rev}$$

Figure 29. Semantics of reverse.

$$\frac{\left[\Gamma_c \overset{\alpha_c}{\vdash} e_c\colon \mathbb{B} \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi_t \otimes |1\rangle_{\underline{e_c}} + \psi_f \otimes |0\rangle_{\underline{e_c}} \qquad \left[\Gamma \overset{\alpha_t}{\vdash} e_t\colon \tau \,\middle|\, \psi_t\right] \xrightarrow{\text{run}} \psi_t' \qquad \left[\Gamma \overset{\alpha_f}{\vdash} e_f\colon \tau \,\middle|\, \psi_f\right] \xrightarrow{\text{run}} \psi_f'}{\left[\Gamma_c, \Gamma \overset{\alpha}{\vdash} \underbrace{\texttt{if}\ e_c\ \texttt{then}\ e_t\ \texttt{else}\ e_f}_{e}\colon \tau \,\middle|\, \psi\right] \xrightarrow{\text{run}} \mathbb{I}_{\underline{e_t}\to\underline{e}}(\psi_t') + \mathbb{I}_{\underline{e_f}\to\underline{e}}\left(\psi_f'\right)}\ \text{ite-q}$$

Figure 30. Semantics of control flow. The rule is analogous for $e_c\colon !\mathbb{B}$.

$$\left(\llbracket \mathbf{X} \rrbracket_{x \to y}\right)\left(|b\rangle_{x} \otimes |\vec{w}\rangle_{\vec{z}}\right) \tag{9}$$

$$= \llbracket \mathbf{X} \rrbracket_{x \to y}\left(|b\rangle_{x}\right) \otimes \mathbb{I}\left(|\vec{w}\rangle_{\vec{z}}\right) \tag{10}$$

$$= \llbracket \mathbf{X} \rrbracket_{x \to y}\left(|b\rangle_{x}\right) \otimes |\vec{w}\rangle_{\vec{z}} \qquad \mathbb{I} \tag{11}$$

$$= \left(\llbracket \mathbf{X} \rrbracket\left(|b\rangle\right)\right)_{y} \otimes |\vec{w}\rangle_{\vec{z}} \qquad x \to y \tag{12}$$

$$= |1-b\rangle_{y} \otimes |\vec{w}\rangle_{\vec{z}} \qquad \mathbf{X} \tag{13}$$

Figure 31. Operating on named states with context.

Eval-transitions. All remaining rules in Fig. 28 are eval-transitions. The rules modify their input state ψ according to the called function, and then store the result in $\underline{\mathrm{ret}}$.

Measurement. The rule for measurement selects one possible measurement v' and collapses the state to this value. Note that measurement allows multiple transitions, one for each possible measured value v'. 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 $\underline{e_1}, \ldots, \underline{e_n}$, and names the output values $\underline{e_i}$ (if the i^{th} argument of c is **const**) and $\underline{\mathrm{ret}}$ (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 ψ. 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 ψ'. Finally, it resets the variable names of constant parameters to e'' back to $\underline{e_i}$ and names the return value $\underline{\mathrm{ret}}$.

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

The actual reversal is performed upon a call to the reversed function, also shown in Fig. 29. Here, we explicitly split the arguments into $\vec{e}^{c}$ (the **const** arguments) and $\vec{e}^{\not c}$ (the non-**const** arguments), as assumed by Fig. 11. Intuitively, rule call-reversed maps ψ to ψ', if running e_{func} on ψ' yields ψ. However, it must also account for naming mismatches: Running e_{func} on ψ' yields $\underline{\mathrm{ret}}$ instead of $\underline{\vec{e}^{\not c}}$, and the name of the returned value must be $\underline{\mathrm{ret}}$.

We note that it is possible that there is no ψ' satisfying the premise of call-reversed, when e_{func} is not surjective. In this case, **reverse**(e_{func}) is undefined on ψ, which intuitively happends if ψ is not in the range of e_{func}).

For $f\colon \bigtimes_{i=1}^{n} \textbf{const}\ \tau_i \times \bigtimes_{i=1}^{m} \tau_i' \xrightarrow{\alpha} \tau''$, we have

$$\llbracket f \rrbracket : \left\llbracket \bigtimes_{i=1}^{n} \tau_i \times \bigtimes_{i=1}^{m} \tau_i' \right\rrbracket^{+} \overset{\text{lin.}}{\nrightarrow} \left\llbracket \bigtimes_{i=1}^{n} \tau_i \times \tau'' \right\rrbracket^{+}$$

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

$$\llbracket \mathbf{X} \rrbracket\, |b\rangle = |1-b\rangle \tag{14}$$

$$\llbracket \mathbf{X} \rrbracket \left(\sum_{b=0}^{1} \gamma_b\, |b\rangle\right) = \sum_{b=0}^{1} \gamma_b \llbracket \mathbf{X} \rrbracket\, |b\rangle = \sum_{b=0}^{1} \gamma_b\, |1-b\rangle \tag{15}$$

(b) Semantics of **X**.

Figure 32. Semantic of built-in functions.

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 **const** by modifying constant arguments and even (ii) violating the rules of quantum physics as in $\alpha\,|0\rangle + \beta\,|1\rangle \mapsto (\alpha+\beta)\,|0\rangle$. 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 $|b\rangle_{x} \otimes |\vec{w}\rangle_{\vec{z}}$. The subscript $x \to y$ of **X** ensures (i) we preserve $|\vec{w}\rangle_{\vec{z}}$ (cp. Eq. (10–11)) and (ii) we run **X** on $|b\rangle_{x}$ 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 **forget**($\cdot = \cdot$) 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.

$$\llbracket \texttt{H} \rrbracket\, |b\rangle = \frac{1}{\sqrt{2}}\Big(|0\rangle + (-1)^b\, |1\rangle \Big)$$
$$\llbracket \texttt{phase} \rrbracket\, |r\rangle = e^{\mathrm{i}r} \cdot |()\rangle$$
$$\llbracket \texttt{rotX} \rrbracket\, |r\rangle\, |b\rangle = \Big(\cos\frac{r}{2}\, |b\rangle - \mathrm{i}\sin\frac{r}{2}\, \llbracket \texttt{X} \rrbracket\, |b\rangle\Big)$$
$$\llbracket \texttt{X} \rrbracket\, |b\rangle = |1-b\rangle$$
$$\llbracket \texttt{rotY} \rrbracket\, |r\rangle\, |b\rangle = \Big(\cos\frac{r}{2}\, |b\rangle - \mathrm{i}\sin\frac{r}{2}\, \llbracket \texttt{Y} \rrbracket\, |b\rangle\Big)$$
$$\llbracket \texttt{Y} \rrbracket\, |b\rangle = \mathrm{i}\cdot(-1)^b\, |1-b\rangle$$
$$\llbracket \texttt{rotZ} \rrbracket\, |r\rangle\, |b\rangle = \Big(\cos\frac{r}{2}\, |b\rangle - \mathrm{i}\sin\frac{r}{2}\, \llbracket \texttt{Z} \rrbracket\, |b\rangle\Big)$$
$$\llbracket \texttt{Z} \rrbracket\, |b\rangle = (-1)^b\, |b\rangle$$
$$\llbracket \texttt{dup} \rrbracket\, |v\rangle = |v, v\rangle$$
$$\llbracket (\cdot, \ldots, \cdot) \rrbracket\, |v_1\rangle \cdots |v_n\rangle = |v_1, \ldots, v_n\rangle$$
$$\llbracket \texttt{forget}(\cdot = \cdot) \rrbracket\, |v\rangle\, |w\rangle = \begin{cases} |v\rangle & v = w \\ \text{undefined} & v \neq w \end{cases}$$
$$\llbracket \cdot \oplus \cdot \rrbracket\, |v_1, v_2\rangle = |v_1, v_2, v_1 \oplus v_2\rangle$$

Figure 33. Example semantics of built-in functions. Most definitions are taken from [21, §4.2]. All definitions can be linearly extended.

G Proofs

Here, we provide proofs for key results.

G.1 Theorems

We recall all theorems presented in §7 in the following.

Theorem 7.1 (Type Preservation)**.** *If $\Gamma = \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, $\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \psi\right] \xrightarrow{run} \psi'$, and $\psi \in \iota(\llbracket \Gamma, \Delta \rrbracket)$, then ψ' lies in $\iota\left(\llbracket \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'', \Delta \rrbracket\right)$.*

Theorem 7.2 (Const Semantics)**.** *If $\Gamma = \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, $\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \psi\right] \xrightarrow{run} \psi'$, and $\psi = \sum_{\vec{v}, \vec{w}} \gamma_{\vec{v},\vec{w}}\, |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}$, then $\psi' = \sum_{\vec{v}, \vec{w}} \gamma_{\vec{v},\vec{w}}\, |\vec{v}\rangle_{\vec{x}} \otimes \chi_{\vec{v},\vec{w}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}$ for some $\chi_{\vec{v},\vec{w}}$.*

Theorem 7.3 (Mfree Semantics)**.** *If $\textbf{\textit{mfree}} \in \alpha$, $\sigma \in \llbracket \Gamma, \Delta \rrbracket^c$,*

$$\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \iota(\sigma, \psi_1)\right] \xrightarrow{run} \psi_1' \quad \textit{for} \quad \psi_1 \in \mathcal{H}\left(\llbracket \Gamma, \Delta \rrbracket^q\right), \textit{and}$$
$$\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \iota(\sigma, \psi_2)\right] \xrightarrow{run} \psi_2' \quad \textit{for} \quad \psi_2 \in \mathcal{H}\left(\llbracket \Gamma, \Delta \rrbracket^q\right),$$

then $\langle \psi_1 | \psi_2 \rangle = \langle \psi_1' | \psi_2' \rangle$.

Theorem 7.4 (Qfree Semantics)**.** *If $\Gamma \vdash^{\alpha} e\colon \tau''$ for $\textbf{\textit{qfree}} \in \alpha$ and context $\Gamma = \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$, then there exists a function $\bar{f}\colon \llbracket \Gamma \rrbracket^s \to \llbracket \textbf{\textit{const}}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'' \rrbracket^s$ on ground sets such that*

$$\left[\Gamma \vdash^{\alpha} e\colon \tau'' \,\middle|\, \sum_{\sigma \in \llbracket \Gamma \rrbracket^s} \gamma_\sigma\, |\sigma\rangle \otimes \tilde{\psi}_\sigma\right] \xrightarrow{run} \sum_{\sigma \in \llbracket \Gamma \rrbracket^s} \gamma_\sigma\, \left|\bar{f}(\sigma)\right\rangle \otimes \tilde{\psi}_\sigma,$$

where $\llbracket \Gamma \rrbracket^s$ is a shorthand for the ground set $\llbracket \Gamma \rrbracket^c \times \llbracket \Gamma \rrbracket^q$ on which the Hilbert space $\llbracket \Gamma \rrbracket^+ = \mathcal{H}\left(\llbracket \Gamma \rrbracket^s\right)$ 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 Γ is preserved, hence it suffices to prove that there exists a function $\bar{f}\colon \llbracket \vec{\tau}, \vec{\tau}' \rrbracket^s \to \llbracket \tau'' \rrbracket^s$ such that

$$\psi = \sum_{\vec{v}, \vec{w}} \gamma_{\vec{v},\vec{w}}\, |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}$$

gets mapped to

$$\psi' = \sum_{\vec{v}, \vec{w}} \gamma_{\vec{v},\vec{w}}\, |\vec{v}\rangle_{\vec{x}} \otimes \left|\bar{f}(\vec{v}, \vec{w})\right\rangle_{\underline{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}.$$

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 $\texttt{mfree}$ expression e can be implemented on a QRAM which maps $\psi \in \iota(\llbracket \Gamma, \Delta \rrbracket)$ to ψ' if $\left[\Gamma \vdash^{\alpha, \texttt{mfree}} e\colon \tau' \,\middle|\, \psi\right] \xrightarrow{run} \psi'$.*

Proof. Let $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$ and $\sigma \in \llbracket \Gamma, \Delta \rrbracket^c$. From Thm. 7.3, we know that there exists a linear isometry M_σ

$$M_\sigma\colon \mathcal{A} \to \iota\left(\left\llbracket \texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau, \Delta \right\rrbracket\right),$$

where $\mathcal{A} := \left\{\iota(\sigma, \tilde{\psi}) \mid \tilde{\psi} \in \mathcal{H}\left(\llbracket \Gamma, \Delta \rrbracket^q\right)\right\}$. Hence, given ψ, a QRAM can (i) extract the classical components of ψ, (ii) determine M_σ based on those classical components σ, and (iii) run M_σ on ψ, yielding ψ'. □

G.2 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 $\texttt{mfree}$ expressions (Thm. 7.3) by [M], the part for $\texttt{qfree}$ expressions (Thm. 7.4) by [Q], and the part for physicality (Thm. 7.5) by [P].

G.2.1 [const].

The rule

$$\frac{}{\left[\emptyset \vdash^{\alpha} c\colon \tau \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi \otimes |c\rangle_{\underline{c}}}$$

maps ψ to $\psi \otimes |c\rangle_{\underline{c}}$.

- [T] Since $\Gamma = \emptyset$ we have that $\psi \in \iota(\llbracket \Delta \rrbracket)$. Hence we have immediately $\psi' = \psi \otimes |c\rangle_{\underline{c}} \in \iota\left(\left\llbracket \underline{c}\colon \tau, \Delta \right\rrbracket\right)$.
- [C] Since $\Gamma = \emptyset$ we have that $\psi = \tilde{\psi}$. Hence we have immediately $\psi' = \tilde{\psi} \otimes \chi = \psi \otimes |c\rangle_{\underline{c}}$, where $\chi = |c\rangle_{\underline{c}}$.
- [M] We have
$$(\psi_1^\dagger \otimes \langle c|_{\underline{c}})(\psi_2 \otimes |c\rangle_{\underline{c}}) = \psi_1^\dagger \psi_2,$$
where $\psi_1^\dagger \psi_2$ denotes the inner product $\langle \psi_1 | \psi_2 \rangle$.
- [Q] Function $\bar{f}(\cdot) = c$ has the correct behavior.
- [P] A QRAM can prepare prepare state c in variable $\underline{c}$.

$$\boxed{1} = \dfrac{}{\left[\mathtt{const}\ x\colon \mathbb{B} \vdash^{\alpha} x\colon \mathbb{B} \,\middle|\, |0\rangle_{\boldsymbol{x}}\, |0\rangle_x\, |1\rangle_y \right] \xrightarrow{\text{run}} |0\rangle_{\boldsymbol{x}}\, |0\rangle_x\, |0\rangle_{\underline{\boldsymbol{x}}}\, |1\rangle_y}\ \text{var}$$

$$\boxed{2} = \dfrac{}{\left[\mathtt{const}\ x\colon \mathbb{B} \vdash^{\alpha} x\colon \mathbb{B} \,\middle|\, |0\rangle_x\, |0\rangle_{\boldsymbol{x}}\, |0\rangle_{\underline{x}}\, |1\rangle_y \right] \xrightarrow{\text{run}} |0\rangle_x\, |0\rangle_{\boldsymbol{x}}\, |0\rangle_{\underline{x}}\, |0\rangle_{\underline{\boldsymbol{x}}}\, |1\rangle_y}\ \text{var}$$

$$\boxed{3} = \dfrac{}{\left[\emptyset \vdash^{\alpha} \cdot \,||\, \cdot\colon \mathtt{const}\ \mathbb{B} \times \mathtt{const}\ \mathbb{B} \overset{!}{\longrightarrow} \mathbb{B} \,\middle|\, |0\rangle_x\, |0\rangle_x\, |0\rangle_{\underline{x}}\, |0\rangle_{\underline{x}}\, |1\rangle_y \right] \xrightarrow{\text{run}} |0\rangle_x\, |0\rangle_x\, |0\rangle_{\underline{x}}\, |0\rangle_{\underline{x}}\, |1\rangle_y \otimes |\cdot \,||\, \cdot, \emptyset\rangle_{\underline{\cdot||\cdot}}}\ \text{const}$$

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

$$\dfrac{\boxed{1} \quad \boxed{2} \quad \boxed{3} \quad \dfrac{\text{Assuming: } [\![\cdot \,||\, \cdot]\!]\, |a\rangle\, |b\rangle = |a\rangle\, |b\rangle\, |a \,||\, b\rangle}{\left[(\cdot \,||\, \cdot)(x, x)\colon \mathtt{const}\ \mathbb{B} \times \mathtt{const}\ \mathbb{B} \overset{!}{\longrightarrow} \mathbb{B} \,\middle|\, \emptyset\colon \emptyset \,\middle|\, |0\rangle_x\, |0\rangle_x\, |0\rangle_{\underline{x}}\, |0\rangle_{\underline{x}}\, |1\rangle_y \right] \xrightarrow{\text{eval}} |0\rangle_x\, |0\rangle_x\, |0\rangle_{\underline{x}}\, |0\rangle_{\underline{x}}\, |1\rangle_y\, |0\rangle_{\underline{\text{ret}}}}\ \text{built-in-eval}}{\dfrac{\left[\mathtt{const}\ x\colon \mathbb{B}, \mathtt{const}\ x\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B} \,\middle|\, |0\rangle_x\, |0\rangle_x\, |1\rangle_y \right] \xrightarrow{\text{run}} |0\rangle_x\, |0\rangle_x\, |1\rangle_y\, |0\rangle_{\underline{x||x}}}{\dfrac{\left[\mathtt{const}\ x\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B} \,\middle|\, |0\rangle_x\, |1\rangle_y \right] \xrightarrow{\text{run}} |0\rangle_x\, |1\rangle_y\, |0\rangle_{\underline{x||x}}}{\left[\mathtt{const}\ x\colon \mathbb{B}, \mathtt{const}\ y\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B} \,\middle|\, |0\rangle_x\, |1\rangle_y \right] \xrightarrow{\text{run}} |0\rangle_x\, |1\rangle_y\, |0\rangle_{\underline{x||x}}}\ \text{W}}\ \text{C}}\ \text{func-eval}$$

(b) Full semantic derivation tree for $\mathtt{const}\ x\colon \mathbb{B}, \mathtt{const}\ y\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B}$.

$$\dfrac{\dfrac{}{\mathtt{const}\ x\colon \mathbb{B} \vdash^{\alpha} x\colon \mathbb{B}}\ \text{var} \quad \dfrac{}{\mathtt{const}\ x\colon \mathbb{B} \vdash^{\alpha} x\colon \mathbb{B}}\ \text{var} \quad \dfrac{\text{(assuming } || \text{ is built-in)}}{\emptyset \vdash^{\alpha} \cdot \,||\, \cdot\colon \mathtt{const}\ \mathbb{B} \times \mathtt{const}\ \mathbb{B} \overset{!}{\longrightarrow} \mathbb{B}}}{\dfrac{\mathtt{const}\ x\colon \mathbb{B}, \mathtt{const}\ x\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B}}{\dfrac{\mathtt{const}\ x\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B}}{\mathtt{const}\ x\colon \mathbb{B}, \mathtt{const}\ y\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B}}\ \text{W}}\ \text{C}}\ \text{func-eval}$$

(c) Type derivation tree for $\mathtt{const}\ x\colon \mathbb{B}, \mathtt{const}\ y\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B}$.

Figure 34. Semantics of $\mathtt{const}\ x\colon \mathbb{B}, \mathtt{const}\ y\colon \mathbb{B} \vdash^{\alpha} x \,||\, x\colon \mathbb{B}$ on input state $|0\rangle_x\, |1\rangle_y$. Here, $\alpha = \mathtt{qfree}, \mathtt{mfree}$ and gray parts of states correspond to the additional context Δ.

G.2.2 [var].

The rule

$$\dfrac{}{\left[x\colon \tau \vdash^{\alpha} x\colon \tau \,\middle|\, \psi \right] \xrightarrow{\text{run}} \mathbb{I}_{x \to \underline{x}}(\psi)}\ \text{var}$$

maps $\psi = \sum_w \gamma_w\, |w\rangle_x \otimes \tilde{\psi}_w$ to $\sum_w \gamma_w\, |w\rangle_{\underline{x}} \otimes \tilde{\psi}_w$.

- [T] Since $\Gamma = x\colon \tau$ and $\psi \in \iota([\![x : \tau, \Delta]\!])$, we have that $\psi' \in \iota\left(\left[\!\left[\underline{x} : \tau, \Delta\right]\!\right]\right)$.
- [C] We have that $\psi' = \sum_w \gamma_w\, |w\rangle_{\underline{x}} \otimes \tilde{\psi}_w$, hence the claim.
- [M] This is straightforward as renaming does not change the inner product.
- [Q] Function $\bar{f}(v) = v$ has the correct behavior.
- [P] A QRAM can simply rename variable x to $\underline{x}$.

G.2.3 [var-const].

The rule is

$$\dfrac{}{\left[\mathtt{const}\ x\colon \tau \vdash^{\alpha} x\colon \tau \,\middle|\, \psi \right] \xrightarrow{\text{run}} \mathtt{dup}_{x \to x, \underline{x}}(\psi)}\ \text{var-const}$$

mapping $\psi = \sum_v \gamma_v\, |v\rangle_x \otimes \tilde{\psi}_v$ to $\psi' = \sum_v \gamma_v\, |v\rangle_x\, |v\rangle_{\underline{x}} \otimes \tilde{\psi}_v$

- [T] Since $\psi \in \iota([\![\mathtt{const}\ x\colon, \Delta]\!])$, we have that the state $\psi' \in \iota\left(\left[\!\left[\mathtt{const}\ x\colon \tau, \underline{x}\colon \tau, \Delta\right]\!\right]\right)$.
- [C] The claim follows immediately.
- [M] We have

$$\begin{aligned} \psi_1'^{\dagger} \psi_2' &= \sum_v \gamma_v^{1*}\, \langle v|_x\, \langle v|_{\underline{x}} \otimes \tilde{\psi}_v^{1\dagger} \sum_w \gamma_w^2\, |w\rangle_x\, |w\rangle_{\underline{x}} \otimes \tilde{\psi}_w^2 \\ &= \sum_v \gamma_v^{1*} \gamma_v^2 \tilde{\psi}_v^{1\dagger} \tilde{\psi}_v^2 \\ &= \psi_1^{\dagger} \psi_2. \end{aligned}$$

- [Q] Function $\bar{f}(v) = v$ has the correct behavior.
- [P] A QRAM can run the linear isometry $\mathtt{dup}$.

G.2.4 [!W]. The rule is

$$\frac{\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \mathrm{drop}^{(x)}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}\ !\text{W}.$$

The general state for $\psi \in \iota(\llbracket \Gamma, x\colon !\tau, \Delta \rrbracket)$ is

$$\psi = |v\rangle_x \otimes \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},$$

where $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$. Note that

$$\mathrm{drop}^{(x)}(\psi) = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}.$$

- [T] As $\mathrm{drop}^{(x)}(\psi) \in \iota(\llbracket \Gamma, \Delta \rrbracket)$, the claim follows from the induction hypothesis.
- [C] The claim follows from the induction hypothesis.
- [M] By the induction hypothesis, we know that

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

  Further, because $x\colon !\tau$, $\psi_1 = |v\rangle_x \otimes \mathrm{drop}^{(x)}(\psi_1)$ and similarly $\psi_2 = |v\rangle_x \otimes \mathrm{drop}^{(x)}(\psi_2)$. Thus the claim

  $$\psi_1^\dagger \psi_2 = \mathrm{drop}^{(x)}(\psi_1)^\dagger\, \mathrm{drop}^{(x)}(\psi_2) = \psi_1'^\dagger \psi_2'.$$
- [Q] By the induction hypothesis, we know that there is an $\bar{f}'$ satisfying

  $$\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} |\vec{v}\rangle_{\vec{x}} \otimes \left|\bar{f}'(\vec{v},\vec{w})\right\rangle_{\underline{e}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},$$

  hence $\bar{f}(v,\vec{v},\vec{w}) := \bar{f}'(\vec{v},\vec{w})$ suffices.
- [P] A QRAM can remove x from consideration (which has the semantics of $\mathrm{drop}^{(x)}(\cdot)$ for classical values), and then compute ψ' by the induction hypothesis.

G.2.5 [W]. The rule is

$$\frac{\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}\ \text{W}.$$

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

$$\psi = \sum_{v,\vec{v},\vec{w}} \gamma_{v,\vec{v},\vec{w}} |v\rangle_x \otimes |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{v,\vec{v},\vec{w}}$$

- [T] We can apply the induction hypothesis by considering $\texttt{const}\ x\colon \tau$ as part of the remainder $\Delta' = \texttt{const}\ x\colon \tau, \Delta$, yielding $\psi' \in \iota\left(\texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'', \Delta'\right)$, hence the claim.
- [C] Similarly, this claim follows immediately by applying the induction hypothesis after grouping $|v\rangle_x$ with $\tilde{\psi}_{v,\vec{v},\vec{w}}$.
- [M] The induction hypothesis immediately yields the claim.
- [Q] Here $\bar{f}(v,\vec{v},\vec{w}) := \bar{f}'(\vec{v},\vec{w})$, where $\bar{f}'$ is the function from the induction hypothesis.
- [P] A QRAM can compute ψ' by the induction hypothesis.

G.2.6 [!C]. The rule is

$$\frac{\left[\Gamma, x\colon !\tau, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \mathbf{dup}_{x\to x,x}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, x\colon !\tau \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \psi'}\ !\text{C}.$$

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

$$\psi = |v\rangle_x \otimes \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},$$

- [T] It is clear that $\mathbf{dup}_{x\to x,x}(\psi) \in \iota(\llbracket \Gamma, x\colon !\tau, x\colon !\tau, \Delta \rrbracket)$. Applying the induction hypothesis to $\mathbf{dup}_{x\to x,x}(\psi)$, yields that $\psi' \in \iota(\llbracket \texttt{const}\ \vec{x}\colon \vec{\tau}, e\colon \tau'', \Delta \rrbracket)$, hence the claim.
- [C] The claim follows from the induction hypothesis.
- [M] The induction hypothesis yields the claim.
- [Q] Here $\bar{f}(v,\vec{v},\vec{w}) := \bar{f}'(v,v,\vec{v},\vec{w})$, where $\bar{f}'$ is the function from the induction hypothesis.
- [P] A QRAM can duplicate x (which has the semantics of $\mathbf{dup}$ for classical values) and then compute ψ' by the induction hypothesis.

G.2.7 [C]. The rule is

$$\frac{\left[\Gamma, \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \mathbf{dup}_{x\to x,x}(\psi)\right] \xrightarrow{\text{run}} \psi'}{\left[\Gamma, \texttt{const}\ x\colon \tau \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \psi\right] \xrightarrow{\text{run}} \mathrm{drop}^{(x)}(\psi')}\ \text{C}.$$

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

$$\psi = \sum_{v,\vec{v},\vec{w}} \gamma_{v,\vec{v},\vec{w}} |v\rangle_x \otimes |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \tilde{\psi}_{v,\vec{v},\vec{w}}.$$

- [T] It is clear that

  $$\mathbf{dup}_{x\to x,x}(\psi) \in \iota(\llbracket \Gamma, \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau, \Delta \rrbracket).$$

  Thus, the induction hypothesis yields

  $$\psi' \in \iota\left(\left\llbracket \texttt{const}\ x\colon \tau, \texttt{const}\ x\colon \tau, \texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau'', \Delta \right\rrbracket\right).$$

  The claim follows by applying $\mathrm{drop}^{(x)}(\cdot)$ to ψ'.
- [C] The induction hypothesis yields the claim.
- [M] A straightforward calculation and the induction hypothesis yield the claim.
- [Q] Similar to before, $\bar{f}(v,\vec{v},\vec{w}) := \bar{f}'(v,v,\vec{v},\vec{w})$, where $\bar{f}'$ is the function from the induction hypothesis.
- [P] A QRAM can duplicate x using the linear isometry $\llbracket \mathbf{dup} \rrbracket$, yielding a state of the form $\sum_v \gamma_v |v\rangle_x |v\rangle_x \otimes \tilde{\psi}_v$. By the induction hypothesis, the QRAM can then compute ψ' of the form $\psi' = \sum_v \gamma_v |v\rangle_x |v\rangle_x \otimes \chi_v$ (Thm. 7.2). Hence, reversing $\mathbf{dup}$ yields

  $$\llbracket \mathbf{dup} \rrbracket^{-1}_{x,x\to x}(\psi') = \sum_v \gamma_v |v\rangle_x \otimes \chi_v = \mathrm{drop}^{(x)}(\psi').$$

G.2.8 [ite]. The rule is depicted in Fig. 30.

[T] We first consider quantum control flow ($e_c\colon \mathbb{B}$). Using the induction hypothesis, we know that the state after evaluating the condition is

$$\psi' \in \iota\left(\left[\!\left[\Gamma_c, \underline{e_c}\colon \mathbb{B}, \Gamma, \Delta\right]\!\right]\right),$$

hence we can write

$$\psi' = \psi_t \otimes |1\rangle_{\underline{e_c}} + \psi_f \otimes |0\rangle_{\underline{e_c}}.$$

Next we show that $\mathbb{I}_{\underline{e_t}\to\underline{e}}\left(\psi'_t\right) + \mathbb{I}_{\underline{e_f}\to\underline{e}}\left(\psi'_f\right)$ is in

$$\iota\left(\left[\!\left[\texttt{const}\ \vec{x}_c\colon \vec{\tau}_c, \texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e}\colon \tau, \Delta\right]\!\right]\right).$$

The induction hypothesis yields that

$$\psi'_t \in \iota\left(\left[\!\left[\texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e_t}\colon \tau, \texttt{const}\ \vec{x}_c\colon \vec{\tau}_c, \Delta\right]\!\right]\right),$$
$$\psi'_f \in \iota\left(\left[\!\left[\texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e_f}\colon \tau, \texttt{const}\ \vec{x}_c\colon \vec{\tau}_c, \Delta\right]\!\right]\right).$$

Because τ does not have any classical components, the classical components of ψ'_t and ψ'_f coincide, hence they can be added. This yields, after renaming, the claim.

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

$$\left[\Gamma_c \vdash^{\alpha_c} e_c\colon !\mathbb{B} \,\middle|\, \psi\right] \xrightarrow{\text{run}} \psi',$$

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

$$\begin{aligned}\psi' &= \psi_t \otimes |1\rangle_{\underline{e_c}} + \psi_f \otimes |0\rangle_{\underline{e_c}}\\ &\in \iota\left(\left[\!\left[\texttt{const}\ \vec{x}_c\colon \vec{\tau}_c, \underline{e_c}\colon !\mathbb{B}, \Gamma, \Delta\right]\!\right]\right).\end{aligned}$$

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

The induction hypothesis yields that

$$\psi'_t \in \iota\left(\left[\!\left[\texttt{const}\ \vec{x}\colon \vec{\tau}, \underline{e_t}\colon \tau, \texttt{const}\ \vec{x}_c\colon \vec{\tau}_c, \Delta\right]\!\right]\right),$$

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

[C] From the induction hypothesis for e_c, we know that $\psi_t \otimes |1\rangle_{\underline{e_c}} + \psi_f \otimes |0\rangle_{\underline{e_c}}$ 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

$$\begin{aligned}\psi_1^\dagger \psi_2 &= \left(\psi_t^{1\dagger} \otimes \langle 1|_{\underline{e_c}} + \psi_f^{1\dagger} \otimes \langle 0|_{\underline{e_c}}\right)\left(\psi_t^2 \otimes |1\rangle_{\underline{e_c}} + \psi_f^2 \otimes |0\rangle_{\underline{e_c}}\right)\\ &= \psi_t^{1\dagger}\psi_t^2 + \psi_f^{1\dagger}\psi_f^2\end{aligned}$$

The induction hypothesis on e_t and e_f yields that

$$\begin{aligned}\psi_t^{1\dagger}\psi_t^2 &= \psi_t^{1\prime\dagger}\psi_t^{2\prime}\\ \psi_f^{1\dagger}\psi_f^2 &= \psi_f^{1\prime\dagger}\psi_f^{2\prime},\end{aligned}$$

thus

$$\psi_1^\dagger\psi_2 = \psi_t^{1\dagger}\psi_t^2 + \psi_f^{1\dagger}\psi_f^2 = \psi_t^{1\prime\dagger}\psi_t^{2\prime} + \psi_f^{1\prime\dagger}\psi_f^{2\prime}.$$

This proves the claim after renaming.

Next, we consider classical control flow. If the classical components of ψ_1 and ψ_2 coincide, then also $\underline{e_c}^1 = \underline{e_c}^2$. Using the induction hypothesis, we see that the term for e_c preserves the inner product between ψ_1 and ψ_2. Furthermore, because $\underline{e_c}\colon !\mathbb{B}$, we get $\psi_1^\dagger\psi_2 = \psi_t^{1\dagger}\psi_t^2$, w.l.o.g. assuming $\underline{e_c} = 1$. Thus with the induction hypothesis on the term for e_t, we get that $\psi_1^\dagger\psi_2 = \psi_t^{1\prime\dagger}\psi_t^{2\prime}$. Renaming does not change the inner product, hence the claim.

[Q] First, we consider quantum control flow. If $\texttt{qfree} \in \alpha$ then by the typing rule $\texttt{qfree} \in \alpha_c \cap \alpha_t \cap \alpha_f$.
Let $\Gamma_c = \texttt{const}\ \vec{x}_c\colon \vec{\tau}_c$ and $\Gamma = \texttt{const}\ \vec{x}\colon \vec{\tau}, \vec{y}\colon \vec{\tau}'$. The general form of ψ is

$$\psi = \sum_{\vec{v}_c, \vec{v}, \vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}}\, |\vec{v}_c\rangle_{\vec{x}_c} \otimes |\vec{v}\rangle_{\vec{x}} \otimes |\vec{w}\rangle_{\vec{y}} \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}$$

Using the induction hypothesis, we get the functions $\bar{f}_{e_c}, \bar{f}_{e_t}$ and $\bar{f}_{e_f}$. For the next step we only suppress variable names that are not immediately clear to lighten the notation. Thus evaluating e_c yields

$$\begin{aligned}\psi' &= \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}}\, |\vec{v}_c\rangle \otimes \left|\bar{f}_{e_c}(\vec{v}_c)\right\rangle_{\underline{e_c}} \otimes |\vec{v}\rangle \otimes |\vec{w}\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}\\ &= \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}}\, |\vec{v}_c\rangle \otimes |0\rangle_{\underline{e_c}} \otimes |\vec{v}\rangle \otimes |\vec{w}\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}\\ &+ \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}}\, |\vec{v}_c\rangle \otimes |1\rangle_{\underline{e_c}} \otimes |\vec{v}\rangle \otimes |\vec{w}\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}\end{aligned}$$

Further, evaluating e_t and e_f yields

$$\begin{aligned}\psi'' &= \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}}\, |\vec{v}_c\rangle \otimes |0\rangle_{\underline{e_c}} \otimes |\vec{v}\rangle \otimes \left|f_{e_f}(\vec{v},\vec{w})\right\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}\\ &+ \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}}\, |\vec{v}_c\rangle \otimes |1\rangle_{\underline{e_c}} \otimes |\vec{v}\rangle \otimes \left|f_{e_t}(\vec{v},\vec{w})\right\rangle \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}}.\end{aligned}$$

Thus the final state can be described by

$$\psi''' = \sum_{\vec{v}_c,\vec{v},\vec{w}} \gamma_{\vec{v}_c,\vec{v},\vec{w}}\, |\vec{v}_c\rangle \otimes |\vec{v}\rangle \otimes \left|\bar{f}(\vec{v}_c,\vec{v},\vec{w})\right\rangle_{\underline{e}} \otimes \psi_{\vec{v}_c,\vec{v},\vec{w}},$$

where $\bar{f}$ is defined by

$$\bar{f}(\vec{v}_c,\vec{v},\vec{w}) := \begin{cases}\bar{f}_{e_t}(\vec{v},\vec{w}) & \text{if } \bar{f}_{e_c}(\vec{v}_c) = 1\\ \bar{f}_{e_f}(\vec{v},\vec{w}) & \text{otherwise.}\end{cases}$$

The proof works analogously for the case where $e_c\colon !\mathbb{B}$.

[P] For quantum control flow ($e_c\colon \mathbb{B}$), expression

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

is **mfree** (ensured by our type system), hence Lem. G.1 applies.

For classical control flow ($e_c\colon !\mathbb{B}$), a QRAM can first evaluate e_c (by the induction hypothesis). Then, as $e_c\colon !\mathbb{B}$, 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 $\xrightarrow{\text{eval}}$, we strengthen our theorems to also cover the following:

For $\vec{e} = \vec{e}^{\text{c}}, \vec{e}^{\not c}$, split into constant and non-constant arguments, assume

$$\left[e'(\vec{e})\colon \bigtimes_{i=1}^{n} \alpha_i \tau_i ! \xrightarrow{\alpha} \tau' \;\middle|\; \sigma\colon \Gamma \;\middle|\; \psi \right] \xrightarrow{\text{eval}} \psi',$$

where the general form of ψ is

$$\psi = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \otimes |\vec{v}\rangle_{\underline{\vec{e}^{\text{c}}}} \otimes |\vec{w}\rangle_{\underline{\vec{e}^{\not c}}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}.$$

Then, we have the following:

[T] If $\psi \in \iota\left(\left[\!\!\left[\underline{\vec{e}}\colon \vec{\tau}, \Delta\right]\!\!\right]\right)$, then

$$\psi' \in \iota\left(\left[\!\!\left[\underline{\vec{e}^{\text{c}}}\colon \vec{\tau}^{\text{c}}, \underline{\text{ret}}\colon \tau', \Delta\right]\!\!\right]\right).$$

[C]

$$\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \otimes |\vec{v}\rangle_{\underline{\vec{e}^{\text{c}}}} \otimes \chi_{\vec{v},\vec{w}} \otimes \tilde{\psi}_{\vec{v},\vec{w}}.$$

[M] If **mfree** $\in \alpha$, $\rho \in \left[\!\!\left[\underline{\vec{e}}\colon \vec{\tau}, \Delta\right]\!\!\right]^{\text{c}}$,

$$\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \sigma\colon \Gamma \;\middle|\; \iota(\rho, \psi_1) \right] \xrightarrow{\text{eval}} \psi_1'$$

for $\psi_1 \in \mathcal{H}\left(\left[\!\left[\underline{\vec{e}}\colon \vec{\tau}, \Delta\right]\!\right]^{\text{q}}\right)$ and

$$\left[\Gamma \overset{\alpha}{\vdash} e\colon \tau'' \;\middle|\; \sigma\colon \Gamma \;\middle|\; \iota(\rho, \psi_2) \right] \xrightarrow{\text{eval}} \psi_2'$$

for $\psi_2 \in \mathcal{H}\left(\left[\!\left[\underline{\vec{e}}\colon \vec{\tau}, \Delta\right]\!\right]^{\text{q}}\right)$, then it holds that

$$\psi_1^\dagger \psi_2 = \psi_1'^\dagger \psi_2'.$$

[Q] If **qfree** $\in \alpha$, then there exists $\bar{f}\colon [\![\vec{\tau}]\!]^{\text{s}} \to [\![\tau']\!]^{\text{s}}$, such that

$$\psi' = \sum_{\vec{v},\vec{w}} \gamma_{\vec{v},\vec{w}} \otimes |\vec{v}\rangle_{\underline{\vec{e}^{\text{c}}}} \otimes \left|\bar{f}(\vec{v},\vec{w})\right\rangle_{\underline{\text{ret}}} \otimes \tilde{\psi}_{\vec{v},\vec{w}},$$

where $\bar{f}$ can depend on σ.

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

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 ψ is

$$\psi = \sum_{w} \gamma_w |w\rangle_{\underline{e}} \otimes \tilde{\psi}_w$$

[T] Let $w' \in [\![\tau]\!]^{\text{c}} \times [\![\tau]\!]^{\text{q}}$. Then immediately

$$\psi' = \gamma_{w'} |w'\rangle_{\underline{\text{ret}}} \otimes \tilde{\psi}_{w'} \in \iota\left([\![\underline{\text{ret}}\colon !\tau, \Delta]\!]\right).$$

[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

$$\textbf{reverse}(e_{\text{func}})\colon \bigtimes_{i=1}^{n} \textbf{const}\ \tau_i \times \bigtimes_{k=1}^{l} \tau_k''! \xrightarrow{\textbf{mfree},\alpha} \bigtimes_{j=1}^{m} \tau_j',$$

hence the claim.

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

[M] The classical components of ψ_1 and ψ_2 coincide, hence in particular the expression e and the captured values σ coincide. The non-classical part of ψ_1 and ψ_2 does not get modified, thus the inner product is preserved.

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

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

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

G.4 [call-rev]

[T] Using the induction hypothesis, we know that

$$\psi' \in \left[\!\!\left[\underline{\vec{e}^{\text{c}}}\colon \vec{\tau}^{\text{c}}, \underline{\vec{t}}\colon \vec{t}', \Delta\right]\!\!\right]^{+}.$$

We need to show $\psi' \in \iota\left(\left[\!\left[\underline{\vec{e}^{\text{c}}}\colon \vec{\tau}^{\text{c}}, \underline{\vec{t}}\colon \vec{\tau}', \Delta\right]\!\right]\right)$, then the claim follows immediately after renaming.

By contradiction: Let $\psi' \notin \iota\left(\left[\!\left[\underline{\vec{e}^{\text{c}}}\colon \vec{\tau}^{\text{c}}, \underline{\vec{t}}\colon \vec{\tau}', \Delta\right]\!\right]\right)$, then there exists a classical component of ψ' 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 Δ. By the induction hypothesis (specifically [C]), we know that evaluating $e_{\text{func}(\vec{e}^{\text{c}},\vec{t})}$ leaves Δ unchanged, hence the classical component in superposition is also a classical component in superposition of ψ, which is a contradiction to $\psi \in \iota\left(\left[\!\!\left[\underline{\vec{e}}\colon \vec{\tau}, \Delta\right]\!\!\right]\right)$.

[C] We know that $\psi' \in \left[\!\!\left[\underline{\vec{e}^{\text{c}}}\colon \vec{\tau}^{\text{c}}, \underline{\vec{t}}\colon \vec{\tau}', \Delta\right]\!\!\right]^{+}$. Further, the linear map sending ψ' to ψ is

$$\sum_{\vec{v}} |\vec{v}\rangle\langle\vec{v}| \otimes M_{\vec{v};\vec{t}\to\underline{\text{ret}}} \otimes \mathbb{I}_\Delta,$$

where $M_v\colon [\![\vec{\tau}']\!] \to [\![\vec{\tau}'']\!]$ is an isometry. The map becomes unitary by restricting the codomain to its image, which can be inverted resulting in

$$\sum_{\vec{v}} |\vec{v}\rangle\langle\vec{v}| \otimes M^{-1}_{\vec{v};\vec{t}\to\underline{\text{ret}}}|_{M_{v;t\to\underline{\text{ret}}}([\![\vec{\tau}']\!])} \otimes \mathbb{I}_\Delta,$$

which preserves $\vec{e}^{c}\colon \vec{\tau}^{c}$ and Δ, hence the claim.

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

[Q] Let **qfree** $\in \alpha$. By the induction hypothesis, we get that

$$\psi' = \sum_{\vec{v},\vec{w}'} \gamma_{\vec{v},\vec{w}'} |\vec{v}\rangle_{\vec{e}^{c}} \otimes |\vec{w}'\rangle_{\vec{t}} \otimes \tilde{\psi}_{\vec{v},\vec{w}'}$$

and that there exists an $\bar{f}'$ such that after renaming $\underline{\text{ret}}$ to $\vec{e}^{\not c}$ we have

$$\psi = \sum_{\vec{v},\vec{w}'} \gamma_{\vec{v},\vec{w}'} |\vec{v}\rangle_{\vec{e}^{c}} \otimes \left|\bar{f}'(\vec{v},\vec{w}')\right\rangle_{\vec{e}^{\not c}} \otimes \tilde{\psi}_{\vec{v},\vec{w}'}.$$

We note that $\bar{f}$ is injective by Thm. 7.3, since non-injectivity would violate the semantics of the **mfree** e_{func} being a linear isometry. Thus, there exists a function $\bar{f} = \bar{f}'^{-1}$ satisfying $\bar{f}'^{-1}(\vec{v},\bar{f}'(\vec{v},\vec{w}')) = \vec{w}'$, hence the claim.

[P] As e_{func} is **mfree**, a QRAM can implement it, according to Lem. G.1. As e_{func} has no classical components in its type, the implementation depends only on e_{func}, not on classical components of the input state. Then, applying the reverse of the implementation to ψ yields the correct result (up to renaming).

G.4.1 [eval-λ-abs].

[T] We know that $\psi \in \iota\left([\![\underline{\vec{e}}\colon \vec{\tau},\Delta]\!]\right)$, thus

$$\psi\otimes|\sigma\rangle \in \iota\left([\![\underline{\vec{e}}\colon \vec{\tau},\Gamma,\Delta]\!]\right),$$

which leads to $\mathbb{I}_{\underline{\vec{e}}\to\vec{x}}(\psi\otimes|\sigma\rangle) \in \iota([\![\vec{x}\colon\vec{\tau},\Gamma,\Delta]\!])$. The induction hypothesis yields now that

$$\psi' \in \iota\left([\![\vec{\alpha}^{c}\vec{x}^{c}\colon \vec{\tau}^{c}, \underline{e''}\colon \tau',\Delta)]\!\right],$$

thus after renaming, $\psi' \in \iota\left(\left[\!\!\left[\vec{e}^{c}\colon \vec{\tau}^{c}, \underline{\text{ret}}\colon \tau',\Delta\right]\!\!\right]\right)$.

[C] The claim follows from the induction hypothesis.

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

[Q] The $\bar{f}$ obtained from the induction hypothesis behaves correctly, up to adding σ to the state and renaming variables.

[P] Given input $\psi\otimes|e'',\sigma\rangle_{\underline{e'}}$, a QRAM can rename variables ($\underline{\vec{e}}\to\vec{x}$), run e'' (by induction hypothesis), and rename variables in the result again.

G.4.2 [func-eval].

[T] After applying the induction hypothesis from left to right on all terms on top, we get

$$\psi_{n+2} \in \iota\left([\![\underline{\vec{e}}^{c}\colon \vec{\tau}^{c}, \underline{\text{ret}}\colon \tau',\Delta_{n+2}]\!]\right),$$

where Δ_{n+2} accumulated additionally to the Δ of ψ_0 all constant parts of $\vec{\Gamma}$ and Γ'. Thus

$$\text{drop}^{(\underline{\vec{e}^{c}})}(\psi_{n+2}) \in \iota\left([\![\underline{\text{ret}}\colon \tau',\Delta_{n+2}]\!]\right),$$

which leads to

$$\mathbb{I}_{\underline{\text{ret}}\to\underline{e'(\vec{e})}} \circ \text{drop}^{(\underline{\vec{e}^{c}})}(\psi_{n+2}) \in \iota\left(\left[\!\!\left[\underline{e'(\vec{e})}\colon \tau',\vec{\Gamma}^{c},\Gamma'^{c},\Delta\right]\!\!\right]\right).$$

[C] The claim follows from the induction hypotheses.

[M] Using the induction hypothesis iteratively, we get that $\psi_{0,1}^\dagger\psi_{0,2} = \psi_{i,1}^\dagger\psi_{i,2}$ for all $1\le i\le n$. Using the induction hypothesis on the other parts yields $\psi_{0,1}^\dagger\psi_{0,2} = \psi_{n+2,1}^\dagger\psi_{n+2,2}$. The type system guarantees that subexpressions which are not consumed, that is **const** $\in \alpha_i'$ are **qfree**, and thus they can be uncomputed similarly to the case for ite where we uncomputed the expression $\underline{e_c}$. Thus $\text{drop}^{(\underline{\vec{e}^{c}})}(\psi_{n+2})$ preserves the inner product, and hence the claim.

[Q] An appropriate composition of all $\bar{f}$ from the induction hypotheses, $\text{drop}^{(\underline{e_i})}$, and variable renamings yields the claim.

[P] We can evaluate all arguments, and determine the function itself by the induction hypothesis, yielding $\psi_{n+1}\otimes|e'',\sigma\rangle_{\underline{e'}}$. By the strengthened induction hypothesis, we have

$$\psi_{n+2} = \sum_{\vec{v},\vec{w}_1,\ldots,\vec{w}_n} \gamma_{\vec{v}}\,|\vec{v}\rangle_{\vec{x}} \otimes \bigotimes_{\{i|\textbf{const}\in\alpha_i'\}} \left|\bar{f}_i(\vec{v},\vec{w}_i)\right\rangle_{\underline{e_i}} \otimes \tilde{\psi}_{\vec{v}},$$

where $\vec{x}$ consists of all constant variables in $\vec{\Gamma},\Gamma'$. This is due to Thm. 7.4 (which ensures this holds after evaluating all arguments) and Thm. 7.2 (which ensures this form is preserved).

Hence, a QRAM can reverse $\bar{f}_i$, to implement $\text{drop}^{(\underline{e_i})}(\cdot)$ for each i with **const** $\in \alpha_i'$. This yields the correct result up to renaming of variables.

H Evaluation

H.1 Evaluation against Q#

In this section, we provide a detailed evaluation of the Q# Summer 2018 [17] and Winter 2019 [18] coding contests.

Tab. 3 summarizes the comparison of our solutions written in Silq against the solutions written by (i) the Q# language designers and (ii) the respective top 10 contest participants. Our results demonstrate that Silq requires significantly less lines of code and only requires roughly half the built-in features and library functions. The remaining tables contain more detailed results.

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

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.

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.

	Summer 2018															Winter 2019													
	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	D1	D2	D3	E1	E2	A1	A2	B1	B2	C1	C2	C3	D1	D2	D3	D4	D5	D6	Sum
Quantum primitives																													
`ApplyToEach`								3						2			1								2				8
`ApplyToEachC`																										1			1
`ApplyToEachCA`																								1					1
`CCNOT`♠													3														1		4
`CNOT`♠		1	1	1							1	2							3		2				2		3	2	18
`ControlledOnBitString`																	2					2							4
`ControlledOnInt`																	1	1	1		2	1		1					7
`H`♠	1	1	1	1			2	1		1				3			1		2				1	1	1	1	2	1	21
`M`					1	1	2	2	1	2				1	1				1										12
`MResetZ`																			1										1
`MeasureInteger`																		1											1
`PrepareUniformSuperposition`																1													1
`R1`♠																		2											2
`ResetAll`														1	1														2
`ResultAsInt`							1	1																					2
`Ry`♠									1									1	1										3
`S`♠																			1										1
`SWAP`♠				1				1																				1	3
`With`								1																					1
`WithA`																						1							1
`X`♠			2	1				2				2		1	1		3	1	1	2	3	3				5	2	4	33
`Z`♠								1											1										2
Annotations																													
`Adjoint`																		1		1	1							1	4
`Controlled`				1				1											2	1				1		3	1	2	12
`adjoint self`								1													1								2
`adjoint auto`				1														2		2	1	2		2				1	11
`controlled auto`																		1						1		1			3
`controlled adjoint auto`																		1						1					2
Low-level quantum gates (marked by ♠)	1	2	4	4			2	5	1	1	1	4	3	4	1		4	4	9	2	5	3	1	1	3	6	8	8	87
Lines of code	9	12	32	24	12	16	9	19	11	28	11	15	9	23	21	3	20	21	30	18	27	19	3	12	5	21	10	53	493

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

	Summer 2018															Winter 2019													
	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	D1	D2	D3	E1	E2	A1	A2	B1	B2	C1	C2	C3	D1	D2	D3	D4	D5	D6	Sum
Quantum primitives																													
`dup`																												1	1
`forget`		1	1	1													1							2	1		2	1	10
`H`♠	1	1	1	1			2	2		2				2		2	1		2				1	1	1	1	2	1	24
`if`		1	1	1				1						1			3	3	5					1	2	2	2	4	27
`measure`					1	1	1	1	1	3				1	1	2		1	1										14
`phase`								1						1				2	2										6
`reverse`																		1											1
`rotY`♠									1									1	1										3
`X`♠				1														1	1						2	3	2		10
`[]`				1													1												2
Annotations																													
`mfree`																		1											1
`lifted`				1							1	1	1	1	1					1	1	1						1	10
`!` (classical)	1	2	2	1	1	1					2	2		3	3		2	2		1	1	1	1	1	1	1		16	45
Low-level quantum gates (marked by ♠)	1	1	1	2			2	2	1	2				2		2	1	2	4				1	1	3	4	4	1	37
Lines of code	5	6	12	12	3	9	4	5	3	7	7	7	7	7	5	10	10	17	15	7	11	7	4	15	18	17	15	22	267

Table 5. Evaluation of the Silq solutions for Q# Summer 2018 and Winter 2019 coding contest.

	1	2	3	4	5	6	7	8	9	10	average
Quantum primitives											
`ApplyToEach`			4				3	5	5	5	2.2
`BoolArrFromResultArr`									3		0.3
`BoolFromResult`									1		0.1
`CCNOT`♠	6	3		3		3		12	3	3	3.3
`CNOT`♠	7	8	9	7	7	9	8	9	6	6	7.6
`H`♠	14	11	12	12	13	13	12	14	13	12	12.6
`IsResultZero`									1		0.1
`M`	12	11	12	16	12	18	10	12	3	12	11.8
`MultiM`									6		0.6
`R`♠										1	0.1
`Reset`								3			0.3
`ResetAll`	2		2		5		3	2	2	2	1.8
`ResultAsInt`			5					1	2		0.8
`Ry`♠	1	1	1	2	3	3	1	1	2		1.5
`SWAP`♠			1							2	0.3
`X`♠	7	12	15	5	42	11	9	10	6	10	12.7
`Z`♠			1							1	0.2
Annotations											
`Controlled`		2	5	1	8		3		2	2	2.3
`controlled auto`		1							1	1	0.3
Quantum primitives (number of non-zero rows)	7	6	10	6	6	6	7	10	13	10	8.1
Annotations (number of non-zero rows)		2	1	1	1		1		2	2	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.

	1	2	3	4	5	6	7	8	9	10	average
Quantum primitives											
`ApplyPauliFromBitString`				4							0.4
`ApplyToEach`				2				7	6		1.5
`ApplyToEachA`								8	3		1.1
`ApplyToEachC`				1							0.1
`ApplyToEachCA`	1			5					7	6	1.9
`CCNOT`♠	6	10		1	2	9	31		1	4	6.4
`CNOT`♠	5	15	12	14	20	17	13	23	18	15	15.2
`ControlledOnBitString`	2			6						2	1.0
`ControlledOnInt`	4			6							1.0
`H`♠	13	11	13	14	13	10	13	12	13	15	12.7
`IntegerIncrementLE`				2						4	0.6
`M`		3	3		12	3	5	4		8	3.8
`Measure`								1			0.1
`MeasureInteger`				1					2		0.3
`MultiM`	2			1							0.3
`MultiX`	9			2							1.1
`QFT`										1	0.1
`R1`♠			2	2	2				5		1.1
`Reset`	1			1	2			1			0.5
`ResetAll`			1							3	0.4
`ResultAsInt`	2			1							0.3
`Rx`♠				1			1				0.2
`Ry`♠	2	3	3	2	3	4	4	8	6	2	3.7
`Rz`♠		4		1		5	1	2		2	1.5
`S`♠	2		1		1			1			0.5
`SWAP`♠		1	1	1		1		5	1	1	1.1
`StatePreparationComplexCoefficients`	2									1	0.3
`StatePreparationPositiveCoefficients`										1	0.1
`WithA`									1		0.1
`X`♠	18	62	50	36	98	65	119	64	58	27	59.7
`Z`♠	1	1	2		4						0.8
Annotations											
`Adjoint`	2	1	8	2		1		2		3	1.9
`Controlled`	14	26	27	25	37	27	30	57	36	12	29.1
`adjoint self`	1			3				1			0.5
`adjoint auto`	2	5	23	15	5	8	3	4	9	8	8.2
`controlled auto`		2	10	28		2		4	6		5.2
`controlled adjoint auto`				2		2			1	4	0.9
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.

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`							1	1			0.2
`H`	1	1	1	1	1	1	1	1	1	1	1.0
`ResetAll`					1						0.1
Lines of code	7	11	9	10	10	10	5	7	12	10	9.1

(a) Summer 18: A1

Rank	1	2	3	4	5	6	7	8	9	10	average
`CNOT`	1	1	1	1	1	1	1	1	1	1	1.0
`H`	1	1	1	1	1	1	1	2	1	1	1.1
`M`				1							0.1
`X`				1							0.1
Lines of code	11	17	12	24	12	23	11	16	24	20	17.0

(b) Summer 18: A2

Rank	1	2	3	4	5	6	7	8	9	10	average
`CNOT`	1	2	1	1	2	1	1	1	1	2	1.3
`H`	1	1	1	1	1	1	1	1	1	2	1.1
`M`				1							0.1
`X`	2	3	3	2	5	1	1	3	2	2	2.4
Lines of code	24	46	38	35	39	30	22	33	40	42	34.9

(c) Summer 18: A3

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`										1	0.1
`CCNOT`	3							2			0.5
`CNOT`	1	1	1	1		3	2	1		1	1.1
`H`	1	1	1	1			1	1		1	0.7
`M`				1							0.1
`Ry`					2	2			1		0.5
`SWAP`										2	0.2
`X`	1	3	1	1	2	1	3	1	1	3	1.7
`Controlled`		2	1	1	1		2		1	1	0.9
`controlled auto`		1							1	1	0.3
Lines of code	24	25	22	46	17	19	25	24	13	47	26.2

(d) Summer 18: A4

Rank	1	2	3	4	5	6	7	8	9	10	average
`BoolFromResult`									1		0.1
`M`	1	1	1	1	1	1	1	1		1	0.9
`MultiM`									1		0.1
`ResetAll`							1				0.1
`X`							1				0.1
Lines of code	8	16	12	12	13	16	25	13	9	13	13.7

(e) Summer 18: B1

Rank	1	2	3	4	5	6	7	8	9	10	average
`BoolArrFromResultArr`									1		0.1
`CNOT`								1			0.1
`H`								1			0.1
`M`	2	1	1	1	1	1	1	1		2	1.1
`MultiM`									1		0.1
Lines of code	9	22	18	17	16	24	15	17	19	14	17.1

(f) Summer 18: B2

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`			1								0.1
`H`	2	2	1	2	2	2	2	2	2	2	1.9
`M`	2	2	2	3	2	2	2	2		2	1.9
`MultiM`									1		0.1
`ResultAsInt`			2					1	1		0.4
Lines of code	10	20	10	20	16	34	10	9	15	15	15.9

(g) Summer 18: B3

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`										2	0.2
`CNOT`	1	1		1		1	1	1	1		0.7
`H`	4	2	2	3	3	2	2	2	2	2	2.4
`M`	2	2	2	3	2	2	2	2		2	1.9
`MultiM`									1		0.1
`ResetAll`					1						0.1
`ResultAsInt`			1						1		0.2
`SWAP`			1								0.1
`X`					6					2	0.8
`Z`			1							1	0.2
`Controlled`			1		1					1	0.3
Lines of code	13	21	12	24	26	35	14	17	13	18	19.3

(h) Summer 18: B4

Table 8. Evaluation of the submissions of the top 10 contestants of the Q# Summer 2018 coding contest.

Rank	1	2	3	4	5	6	7	8	9	10	average
`IsResultZero`									1		0.1
`M`	1	1	1	1	1	1	1	1	1	1	1.0
`R`										1	0.1
`ResultAsInt`			1								0.1
`Ry`	1	1	1	1	1	1	1	1	1		0.9
Lines of code	5	15	9	12	14	14	10	11	13	11	11.4

(a) Summer 18: C1

Rank	1	2	3	4	5	6	7	8	9	10	average
`H`	1	1	2		2	2	1	2	1	1	1.3
`M`	2	2	2	2	3	5	1	3	2	2	2.4
`Reset`								2			0.2
`ResetAll`			1		1						0.2
`ResultAsInt`			1								0.1
`Ry`				1							0.1
`X`						2					0.2
`Controlled`			1								0.1
Lines of code	10	26	22	20	29	40	13	23	24	18	22.5

(b) Summer 18: C2

Rank	1	2	3	4	5	6	7	8	9	10	average
`CNOT`	1	1	1	1	1	1	1	1	1	1	1.0
Lines of code	7	13	12	11	11	13	10	12	14	12	11.5

(c) Summer 18: D1

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`							2				0.2
`CNOT`	2	2	2	2	2	2	2	2	2	1	1.9
`X`	2	2	2	1	2	2	2	2	1	2	1.8
Lines of code	13	18	17	14	16	19	17	23	19	16	17.2

(d) Summer 18: D2

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`			2					2			0.4
`CCNOT`	3	3		3		3		10	3	3	2.8
`CNOT`			3		1			1			0.5
`X`			5		26		1	3			3.5
`Controlled`			2		6		1				0.9
Lines of code	7	9	14	9	33	9	9	22	10	9	13.1

(e) Summer 18: D3

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`								2	2	2	0.6
`BoolArrFromResultArr`									1		0.1
`H`	3	2	3	3	3	4	3	2	2	2	2.7
`M`	1	1	2	1	1	3	1	1		1	1.2
`MultiM`									1		0.1
`Reset`								1			0.1
`ResetAll`	1				1		1		1	1	0.5
`X`	1	3	3		1	3	1	1	1	1	1.5
Lines of code	19	30	33	32	30	45	20	25	27	21	28.2

(f) Summer 18: E1

Rank	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`			1						3		0.4
`BoolArrFromResultArr`									1		0.1
`H`									3		0.3
`M`	1	1	1	1	1	3	1	1		1	1.1
`MultiM`									1		0.1
`ResetAll`	1		1		1		1	2	1	1	0.8
`X`	1	1	1			2			1		0.6
`Controlled`									1		0.1
Lines of code	14	23	19	27	31	56	22	19	28	19	25.8

(g) Summer 18: E2

Table 9. Evaluation of the submissions of the top 10 contestants of the Q# Summer 2018 coding contest.

	1	2	3	4	5	6	7	8	9	10	average
ApplyToEach				1							0.1
ApplyToEachC				1							0.1
CNOT									1		0.1
H	1	1		2	2	1	1	1	1		1.0
M					2						0.2
Reset				1							0.1
Ry	1	1	1			1	1	1	1		0.7
StatePreparationPositiveCoefficients										1	0.1
X	1	2	8	1	2	1	2	2			1.9
Controlled	1	1	2	2	1	1	1	1	1		1.1
Lines of code	5	6	27	10	21	5	6	8	6	6	10.0

(a) Winter 19: A1

	1	2	3	4	5	6	7	8	9	10	average
ApplyPauliFromBitString				4							0.4
ApplyToEach				1					6		0.7
CCNOT		4				4	4			1	1.3
ControlledOnBitString	2			4							0.6
ControlledOnInt	1										0.1
H	2	2	2	1	2	2	2		2	4	1.9
M					1					3	0.4
ResetAll										2	0.2
Ry								2			0.2
X	3	18	3	8	23	18	18	5	20	5	12.1
Controlled		4	1	4	8	4	4	2	8		3.5
Lines of code	22	66	78	19	72	65	89	63	56	33	56.3

(b) Winter 19: A2

	1	2	3	4	5	6	7	8	9	10	average
CCNOT		1		1		1					0.3
CNOT				1	2		3	4	2		1.2
H				1	1		1		1		0.4
M		1	1		2	1	3	3		3	1.4
MeasureInteger									1		0.1
MultiM	1			1							0.2
MultiX	4										0.4
R1			2	2	2				2		0.8
ResultAsInt	1			1							0.2
Ry		1	1	1	1	1	1	2	1		0.9
Rz		2				2	1	2			0.7
S	1										0.1
StatePreparationComplexCoefficients	2									1	0.3
X		4	6	1	2	4		1			1.8
Z	1										0.1
Adjoint	2		1	1				1		1	0.6
Controlled	2	4	2	1	1	4	2	1	1		1.8
adjoint auto			1	1		2					0.4
controlled auto		1				1					0.2
controlled adjoint auto						1					0.1
Lines of code	16	24	86	18	11	33	13	28	11	18	25.8

(c) Winter 19: B1

	1	2	3	4	5	6	7	8	9	10	average
CNOT			4	5	4			3	2		1.8
H	2		2	2	2		1	2	2	4	1.7
M		2	2		6	2	2	1		2	1.7
Measure								1			0.1
MeasureInteger				1					1		0.2
MultiM	1										0.1
QFT										1	0.1
R1									3		0.3
Reset	1				2			1			0.4
ResetAll			1							1	0.2
ResultAsInt	1										0.1
Rx				1			1				0.2
Ry	1	1	1	1	1	1	2	1	1	1	1.1
Rz		2		1		3				2	0.8
S	1		1		1			1			0.4
SWAP									1		0.1
X		1	3	1	5	2	1	2	6		2.1
Z		1	2		4						0.7
Controlled	1	4	3	2	6	4	1	1	3	3	2.8
adjoint auto			4	2		2				1	0.9
controlled auto		1	3	1		1					0.6
controlled adjoint auto				1		1				1	0.3
Lines of code	12	34	44	28	50	48	27	31	30	40	34.4

(d) Winter 19: B2

	1	2	3	4	5	6	7	8	9	10	average
CNOT		2	2			4			2		1.0
ControlledOnBitString				2						1	0.3
ControlledOnInt	1										0.1
X	4	1	2	2	6	2	5	5	1	4	3.2
Adjoint			1								0.1
Controlled	1	1	1		2	1	2	2	1		1.1
adjoint self								1			0.1
adjoint auto	1	1	2	1	1	1	1		1	1	1.0
controlled auto				2							0.2
Lines of code	13	13	25	23	28	26	25	23	13	19	20.8

(e) Winter 19: C1

	1	2	3	4	5	6	7	8	9	10	average
ApplyToEachA								6	3		0.9
CCNOT							24				2.4
CNOT		4	2	2	2	4		4	2	2	2.2
ControlledOnBitString										1	0.1
ControlledOnInt	1			2							0.3
WithA									1		0.1
X	1	10	6	3	7	11	60	10	6	1	11.5
Adjoint			1								0.1
Controlled		3	2		2	3	5	3	2		2.0
adjoint self	1			2							0.3
adjoint auto		1	2		2	1	1	1	3	1	1.2
controlled auto				4							0.4
Lines of code	19	46	22	33	31	48	116	46	34	17	41.2

(f) Winter 19: C2

Table 10. Evaluation of the submissions of the top 10 contestants of the Q# Winter 2018 coding contest.

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachA`								2			0.2
`ApplyToEachCA`										2	0.2
`CCNOT`	5	5				4				1	1.5
`CNOT`							2		4		0.6
`ControlledOnInt`				4							0.4
`IntegerIncrementLE`										1	0.1
`M`					1						0.1
`SWAP`								4		1	0.5
`X`	4	12	6	5	10	13	4	6	7	4	7.1
`Adjoint`			1	1				1		1	0.4
`Controlled`			5	3	3	1	2	3	3	1	2.1
`adjoint self`				1							0.1
`adjoint auto`	1	1	3	2	2	1	1	2	3	3	1.9
`controlled auto`			2	3				2	4		1.1
`controlled adjoint auto`				1						1	0.2
Lines of code	20	27	40	32	35	31	33	36	39	31	32.4

(a) Winter 19: C3

	1	2	3	4	5	6	7	8	9	10	average
`H`	1	1	1	1	1		1	1	1	1	0.9
`Ry`						1					0.1
`adjoint auto`			1	1							0.2
`controlled auto`				2							0.2
Lines of code	3	3	7	8	3	3	3	4	3	3	4.0

(b) Winter 19: D1

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachCA`	1			1					1	2	0.5
`ControlledOnInt`	1										0.1
`H`	1	1	1	1	1	1	1		1	1	0.9
`Ry`								1			0.1
`X`		2	2	2	2	2	2		2	2	1.6
`Controlled`		1	2	2	1	1	1	1	1	1	1.1
`adjoint auto`			1	1							0.2
`controlled auto`			1	2							0.3
Lines of code	5	14	16	13	14	17	14	9	10	9	12.1

(c) Winter 19: D2

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachCA`				2					2		0.4
`CNOT`		2	2		2	2	2	2		2	1.4
`H`	1	1	1	1	1	1	1	1	1	1	1.0
`MultiX`	2										0.2
`X`				2					2		0.4
`Controlled`	2			2					2		0.6
`adjoint auto`			1	1							0.2
`controlled auto`				2							0.2
Lines of code	5	10	14	12	10	11	11	11	6	9	9.9

(d) Winter 19: D3

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`								6			0.6
`ApplyToEachCA`				2					4		0.6
`CNOT`		2	1		3	2	2	8		4	2.2
`H`	1	1	1	1	1	1	1	4	1	1	1.3
`IntegerIncrementLE`				2						2	0.4
`MultiX`	3			2							0.5
`X`	1	7	5	4	10	7	7	15	8	3	6.7
`Adjoint`			2								0.2
`Controlled`	3	2	3	4	4	2	4	7	7	2	3.8
`adjoint auto`			4	3							0.7
`controlled auto`			2	6					1		0.9
Lines of code	10	29	44	38	35	28	51	153	25	22	43.5

(e) Winter 19: D4

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEach`								1			0.1
`CCNOT`	1				2		3		1	2	0.9
`CNOT`	3	5	1	4	6	5	2	2	4	4	3.6
`H`	3	3	3	3	2	3	2	3	3	3	2.8
`IntegerIncrementLE`										1	0.1
`SWAP`		1	1	1		1		1			0.5
`X`	2	2	7	5	22	2	5	5	2	2	5.4
`Adjoint`			1								0.1
`Controlled`	3	4	3	4	5	4	1	3	3	4	3.4
`adjoint auto`			2	3							0.5
`controlled auto`				6							0.6
Lines of code	11	13	25	38	17	13	14	19	13	15	17.8

(f) Winter 19: D5

	1	2	3	4	5	6	7	8	9	10	average
`ApplyToEachCA`										2	0.2
`CNOT`	2			2	1		2		1	3	1.1
`H`	1	1	2	1		1	2				0.8
`Ry`					1			1	3	1	0.6
`X`	2	3	2	2	9	3	15	13	4	6	5.9
`Adjoint`		1	1			1				1	0.4
`Controlled`	1	2	3	1	4	2	7	33	4	1	5.8
`adjoint auto`		2	2			1		1	2	2	1.0
`controlled auto`			2					2	1		0.5
`controlled adjoint auto`									1	2	0.3
Lines of code	22	37	33	26	40	30	141	179	77	60	64.5

(g) Winter 19: D6

Table 11. Evaluation of the submissions of the top 10 contestants of the Q# Winter 2018 coding contest.

	funcs	data	type	class
The Circ monad		1		
Basic types		2	2	
Basic gates	76		1	
Other circuit-building functions	17			
Notation for controls	4	1		1
Signed items	2	1		
Comments and labelling	4			1
Hierarchical circuits	4			
Block structure	17			
Operations on circuits	17	2		
Circuit transformers	3	2	5	
Circuit generation from classical code	2			
Extended quantum data types	8			8
Sum	154	9	8	10

Table 12. The number of functions, data types, types and classes provided by Quippers core library in the respective category.

H.2 Evaluation against Quipper

In order to compare the amount of features, we counted the definitions provided in Quipper's core library [12] 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

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);
}

def verify[n:!N](bill:uint[n],secret:!ℕ^n):uint[n]×!𝔹{
    // 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);
    }
}

// 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;
}

// SIMPLE TEST

def verify_new_test[n:!ℕ](){
    // verify a new bill twice

    // create new bill
    (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);
}
def main(){
    verify_new_test[3]();
}

// HELPER FUNCTIONS

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

12 https://www.mathstat.dal.ca/~selinger/quipper/doc/Quipper.html

```
    // returns (x1,...,x_length) with xi~{0,...range-1}
    n:=round(log(range)/log(2)) coerce !ℕ;
    assert(2^n==range);
    r:=vector(length,0:!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:!ℕ](bill:uint[n]){
    forged:=dup(bill);
    return (bill,forged);
}

// SIMPLE TEST

def forge_primitive_test[n:!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]();
}
```

```
// An adaptive attack on Wiesner's quantum money, https://
    arxiv.org/abs/1404.1507

import quantum_money;

def forge_nagaj[n:!ℕ](bill:uint[n],verifier:uint[n]! → uint[n]
    ×!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{
            (bill,is_minus):=determine(bill,verifier,k,false);
            if is_minus{
                secret[k]=3;
            }else{
                secret[k]=measure(bill[k]);
            }
        }
    }
    return (bill, encode(secret)(0:uint[n]));
}

def determine[n:!ℕ](bill:uint[n],verifier:uint[n]! → uint[n]×!
    𝔹,k:!ℕ,check_plus:!𝔹):uint[n]×!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(π^2*n/(2*fail_prob)); // choose N
    if N%2==1{ N+=1; } // ensure N is even
    // choose δ
    δ:=π/(2*N);

    probe:=0:𝔹;
    repeat N{
        probe:=rotY(δ*2,probe); // rotate slightly towards 1
        if probe{ // entangle
            bill[k]:=X(bill[k]);
            if !check_plus{ phase(π); }
        }
        (bill,ok):=verifier(bill); // project back by
            verification
        assert(ok==1); // we should not be caught
    }
    return (bill, measure(probe));
}

// SIMPLE TEST
def forge_nagaj_test[n:!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);
    assert(ok_original);
    (forged,ok_forged):=verifier(forged);
    assert(ok_forged);
    // discard both bills
    measure(bill);
    measure(forged);
}

def main(){
    forge_nagaj_test[2]();
}
```