Static Analysis of Quantum Programs

Nicola Assolini[1][0000−0002−6754−6206], Alessandra Di Pierro[1][0000−0003−4173−7941], and Isabella Mastroeni[1][0000−0003−1213−536X]

Dipartimento di Informatica
Università di Verona, Verona, Italy
{nicola.assolini,alessandra.dipierro,
isabella.mastroeni}@univr.it

Abstract. In principle, the design and implementation of quantum programming languages are the same essential tasks as for conventional (classical) programming languages. High-level programming constructs and compilation tools are structurally similar in both cases. The difference is mainly in the hardware machine executing the final code, which in the case of quantum programming languages is a quantum processor, i.e. a physical object obeying the laws of quantum mechanics. Therefore, special technical solutions are required to comply with such laws. In this paper, we show how static analysis can guarantee the correct implementation of quantum programs by introducing two data-flow analyses for detecting some 'wrong' uses of quantum variables. A compiler including such analyses would allow for a higher level of abstraction in the quantum language, relieving the programmer of low-level tasks such as the safe removal of temporary variables.

1 Introduction

With the rapid progress of quantum technology thanks to the effort of many companies that have been working for years with the objective of building a large-scale quantum computer, and the growing number of applications where quantum computers are employed to work on practical use cases in industry, the study of quantum programming languages is becoming more and more valuable. In fact, we are witnessing an increasing number of quantum computing language proposals based mostly on open-source projects with the objective of accelerating progress by sharing knowledge and resources. Current research and implementation efforts have mainly produced languages with a low level of abstraction focused on circuit description and optimizations, with only a few attempts at the design and implementation of high-level quantum programming languages (see e.g. Silq[6], Quipper[14], QWire[33], Qunity[43] and Guppy[37]). This task is not a trivial one and cannot rely completely on the experience and the knowledge acquired in the development of classical programming languages due to the difference in the fundamental elements that are the basis of the system to be 'programmed', which is classical in one case and quantum in the other.

For example, almost all high-level programming languages provide variables as some information holders. However, while in classical languages, variables are

abstractions of memory regions where information can be stored and retrieved, in quantum languages, they are rather abstractions of the information contained in the physical space of quantum states (i.e. vectors in a Hilbert space). Although the advantages offered by these abstractions (in terms of easy writing for the programmer) are the same in both classical and quantum contexts, quantum variables introduce new challenges strictly related to the specific features of quantum computation. In fact, the implementation of a simple operation such as duplicating a quantum variable is not trivial due to the *no-cloning* theorem [25, Chapter 12]. For the same reason, a simple statement like the assignment requires a more complicated treatment than in the classical setting to avoid overwriting the assigned variable.

It is also very important (contrary to the classical setting) to take care of unused quantum variables, which, if not appropriately dealt with, could have a negative impact on the correctness of the result of the whole program due to the *implicit measurement* principle [25, Section 4.4]). In fact, while unused variables require no action in classical computation, unused quantum variables correspond to portions of the quantum circuit that would be measured even if no explicit measurement operator is applied. This may have unexpected side effects when it involves entangled states. In fact, such states are so strongly correlated that measuring one part also affects the other. Thus, a quantum program with unused quantum variables requires careful implementation, ensuring that at the circuit level, they are appropriately 'reset' before the end of the execution of the program to eliminate the presence of entanglement. This process is commonly referred to as *uncomputation*. To overcome these problems, the most common solution adopted in the literature of quantum programming languages is *linear typing*, which forces the programmer to use each quantum variable exactly once. This prevents copying variables (i.e., using them twice) and their implicit discarding (i.e., never using them). Such languages typically provide specific functions (like the `forget` function in Silq and the `discard` function in Guppy and QWIRE), leaving to the programmer's responsibility the task of triggering uncomputation where needed.

In this paper, we introduce an approach based on static analysis instead of a type checker. This approach introduces more automation in the compilation process, reducing the number of error messages with respect to a type system based approach and disburdening the programmer of explicitly deciding when uncomputation is to be performed. In particular, we relax the *exactly once* constraint of linear type systems, simply requiring that variables are used *at most once*. To this purpose, we define a preliminary data flow analysis to determine if variables are used *at most once*. We then use a second data flow analysis to detect, for each program point, which variables are not used in all execution paths. The result of this analysis gives the compiler useful information on the variables to be uncomputed and allows transforming a program that uses variables 'at most once' into a program that uses variables 'exactly once'. With this information, uncomputing variables can be automatically done by the compiler rather than manually by the programmer. Furthermore, adopting a static analysis approach

instead of the most commonly used type of system introduces greater flexibility in language design, as it is independent of the language itself and can potentially be integrated with other analyses to identify infeasible paths, thereby increasing accuracy. Any quantum language that relies on linear typing could benefit from our procedure, e.g. QWIRE[33], Silq[6], and Guppy[23], etc. We can apply our approach also to languages that do not use linear types, like Quipper [14], to ensure their correctness. In this paper, we will demonstrate our analysis on Guppy, an embedded quantum programming language in Python that employs quantum variables instead of wires and is provided with a control flow based on classical guards and measurements.

For the reader convenience we introduce the necessary notions of quantum computation in section 2. We then address the challenges faced in quantum programming languages (section 3) and introduce the syntax of the language that we use to present our analyses (section 4), which we define in section 5. In section 6, we explain how to use the information collected by the analyses, and we demonstrate them by an example in section 7. Finally, in section 8, we discuss some related work and in section 9, we conclude with a summary of contributions and some future research directions.

2 Quantum Computation

Programs in a quantum programming language are designed to run on quantum computers and are very different from classical computing programs. To understand these languages and work with them effectively, a sound knowledge of the principles of quantum mechanics and the underlying mathematics is often essential. In this section, we briefly recall the main aspects of quantum computation that make this computational model different from the classical one. We will refer to the circuit model of computation and highlight such differences in terms of the meaning of wires and gates in a classical and a quantum circuit. In a quantum circuit, wires represent quantum bits, or qubits, rather than bits. This means that the classical unit of information (the bit) generates, with its two values 0 and 1, a complex vector space (a quantum system), where each complex unitary vector is the state of a qubit. This is, therefore, a linear combination of the form $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, where α and β are complex numbers from which we can infer the probability of the state resulting in 1 or 0, respectively. Such probabilities are obtained as $|\alpha|^2$ and $|\beta|^2$, which explains why quantum states must be unitary, i.e. $|\alpha|^2 + |\beta|^2 = 1$ must hold[1].

The behaviour of quantum circuits is determined by two important quantum phenomena which have no classical counterparts, namely entanglement and measurement. The measurement of a quantum state $|\psi\rangle$ is an operation that allows us to extract a classical result from the quantum superposition (or linear combination) by making the quantum state *collapse* to 0 or 1 with the associated probability. Measurement is typically the last operation in a quantum

[1] The *ket* notation $|\psi\rangle$ is due to Dirac and represents the vector $(\alpha, \beta)^T$ in linear algebraic notation.

circuit and is applied to get the final (classical) result of the coherent (i.e. in superposition) evolution of the quantum system represented by the circuit. The entanglement phenomenon occurs when, as a result of the application of specific quantum gates, two or more qubits become so strongly correlated that they are no longer separable. As a consequence, measuring one of the component qubits affects the probability distribution on the whole state by changing the probability associated with the other qubits, even if these are not (explicitly) measured.

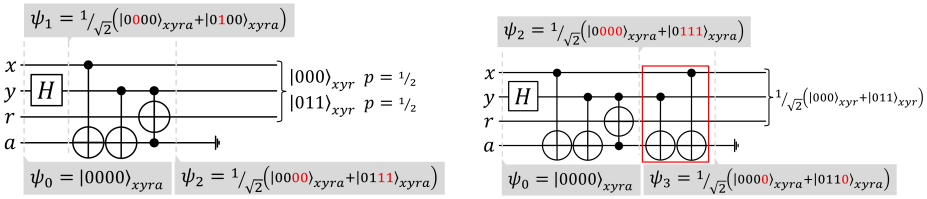
The state of a quantum circuit on n wires (qubits) corresponds to a unitary vector in the 2^n-dimensional Hilbert space ($\mathcal{H}^{2^n}$) obtained by composing by tensor product the unitary states corresponding to each wire (qubit), i.e. a vector in a 2-dimensional complex Hilbert space ($\mathcal{H}^2$) [25, Chapter 2]. A state like $|01\rangle$, in $\mathcal{H}^2 \otimes \mathcal{H}^2$ is not entangled as it can be expressed as the tensor product $|0\rangle \otimes |1\rangle$, of the states of the two-component qubits. On the other hand, the state $1/\sqrt{2}(|00\rangle + |11\rangle)$ in the same Hilbert space $\mathcal{H}^2 \otimes \mathcal{H}^2$ is entangled because it cannot be expressed as a tensor product of the states of the two-component qubits.

Throughout the paper, we will adopt the following convention to distinguish between qubits (or qubit registers) as wires of a physical quantum circuit and their high-level representation as a variable in a quantum program. For a variable q, we will write $|\psi\rangle_q$ to indicate that q represents the state $|\psi\rangle$ of a qubit register. For entangled states, such as for example $1/\sqrt{2}(|01\rangle+|10\rangle$, we will write $1/\sqrt{2}(|01\rangle+|10\rangle)_{p,q}$ to indicate that variable p represents the first qubit and q represents the second qubit of the entangled pair.

2.1 The Need for Uncomputation

Entanglement is a powerful feature of quantum computation; it is at the base of important quantum communication protocols, which cannot be realized by classical means (see, for example, quantum teleportation [25, Chapter 1.3]). However, this feature introduces some complications when it comes to the implementation of quantum programming languages due to the inevitable presence of temporary variables in a program. At the circuit level, the problem arises in the computation of a classical irreversible function on a quantum computer, as this requires additional (auxiliary) qubits to obtain reversibility and then unitarity[22, Chapter 7]. It turns out that when feeding the circuit with a superposition of input states, the auxiliary qubits become entangled with the output qubits, and their elimination (which induces an implicit measurement) affects the state of the output qubits. The solution to this problem is to 'uncompute' these auxiliary qubits before their elimination, returning them to their initial unentangled state. Therefore, at the program level, temporary quantum variables cannot easily be dropped as we do with temporary values in classical programs, and a careful compilation is necessary to avoid the generation of an incorrect circuit. Moreover, quantum variables cannot be overwritten because this would imply copying a quantum state at the physical level, which, as we know, is impossible [20, Chapter 5].

We illustrate the problem of uncomputation with the circuit in Figure 1a. This circuit computes $(x \oplus y) \wedge y$, i.e. the conjunction of y and the exclusive-



(a) Circuit without uncomputation (b) Circuit with uncomputation

Fig. 1

or between x and y, using a temporary qubit a and storing the result in r. Firstly, we apply the Hadamard operation (H) to y, thus obtaining the state $\psi_1 = {}^1/\sqrt{2}(|0000\rangle_{xyra} + |0100\rangle_{xyra})$; secondly, we compute $x \oplus y$ through the two controlled-NOT gates obtaining the state $\psi_1' = {}^1/\sqrt{2}(|0000\rangle_{xyra} + |0101\rangle_{xyra})$; finally, we calculate $a \wedge y$ in r using a Toffoli gate[2] getting the state $\psi_2 = {}^1/\sqrt{2}(|0000\rangle_{xyra} + |0111\rangle_{xyra})$. Now, if we decided to delete the qubit a (since it is no longer needed), the state of the first three qubits would collapse with probability ${}^1/_2$ to $|000\rangle_{xyr}$ or $|011\rangle_{xyr}$, depending on the result of the implicit measurement. This incorrect result is due to the principle of implicit measurement [25, Chapter 4], (deleting a qubit means measuring it) and the entanglement introduced by the controlled-NOT gate.

The example clearly shows the need of uncomputing all temporary variables by resetting their initial state to ensure their (implicit or explicit) removal without side effects. The simplest way to uncompute a qubit is to take all previously applied operations and re-apply their inverses in reverse order, returning the qubit to its initial unentangled state. In Figure 1b, we show that if we uncompute a before the final measurement, introducing additional gates to bring the value of a back to $|0\rangle$ before deleting it (causing an implicit measurement), we do not lose information about the other three qubits.

3 Challenges in Quantum Programming Languages

In this section, we discuss in more detail the challenges that one needs to tackle when designing a quantum programming language. As already pointed out, avoiding the copy and the implicit discarding of quantum variables is crucial for the correctness of the program results. A typically adopted solution is linear typing, which forces the programmer to use variables exactly once so that when a variable x is used the first time, it is 'consumed' and no longer available. We will exemplify the effects of a linear type system by means of programs written in the Guppy language [37], a Python-embedded language, which we take as a model of a quantum language with linear typing. A program like the one in

[2] The Toffoli gate corresponds to a double controlled not; in particular we negate the target if both controllers are 1.

```
def used_consume(qubit: a):
    b = h(a)
    c = t(a) # Error, a is not defined
    return b,c
```

Fig. 2: Simple function that uses a consumed variable

```
def xor_and(x: qubit,y: qubit):
    a,r = qubit(), qubit()
    x,a = cx(x,a)
    y,a = cx(y,a)
    a,y,r = toff(a,y,r)
    # Error: a is not 'consumed'

    return x,y,r
```

(a) Guppy program corresponding to the circuit in Figure 1a

```
def xor_and(x: qubit,y: qubit):
    a,r = qubit(), qubit()
    x,a = cx(x,a)
    y,a = cx(y,a)
    a,y,r = toff(a,y,r)
    discard(a)

    return x,y,r
```

(b) Guppy program with safe variable discarding

Fig. 3

Figure 2 does not pass the check of a linear type checker since the variable `a` is used twice.

Linear typing also ensures that no unused variables occur in a program, i.e. all program variables must be consumed before the end of the execution. For example, consider the implementation of the circuits in Figure 1 in Guppy, given in Figure 3. The Guppy compiler would reject the program in Figure 3a since `a` is not consumed. Instead, the code in Figure 3b would be assessed as well-typed since the Guppy primitive `discard`(a) consumes `a`. In general, we cannot say if this program will be compiled in the circuit in Figure 1b since it depends on the implementation of the `discard` primitive; nevertheless, the final result, after the return, will always be the same correct result. Implicit discarding also occurs when we redefine a non-consumed variable as in the code of Figure 4a. This program is ill-typed since the variable `b` is not consumed when redefined. In this case, the type checker returns an error indicating that the first `b` is not consumed. Instead, the code in Figure 4b is well-typed since we properly discard (and thus consume) `b` before redefining it.

With the aim of relaxing the constraints imposed by linear type systems, we propose a different approach, which is based on static analysis and consists of two steps: first, we check that variables are used *at most* once, filtering out programs that violate this rule; second, we identify unused and overwritten variables and automatically insert the discard function. Two key analyses are necessary for our approach: a forward data-flow analysis (Consuming Analysis) that collects information about the availability of variables at each program point and a backward data-flow analysis (Uncomputation Analysis) that collects information about the usage of variables. The information resulting from the first analysis allows the compiler to verify, in place of the type checker, that the programs use

```
b = h(a)
# error: b not consumed
b = qubit()
```

(a) Wrong re-definition

```
b = h(a)
discard(b) #'drop' b
b = qubit()
```

(b) Proper re-definition

Fig. 4: Two simple programs where we assume that `a` is already defined

variables at most once. Then, the second analysis gives the necessary information on the program points where to insert the discard function, transforming a program that uses variables *at most once* into a program that uses variables *exactly once*. This procedure introduces greater flexibility than the type system approach, enhancing the language usability. In fact, with our analysis, we still reject programs that use consumed variables, but we automatically insert the `discard` when needed, relieving the programmer from this task. As an example, the function in Figure 2 is still rejected, but functions in Figure 3a and Figure 4a are automatically transformed at compile time, respectively, into the ones in Figure 3b and Figure 4b.

4 Control Flow Graph

We define our static analyses as data-flow analyses based on control flow graphs. Following [27,7], we consider the Control Flow Graphs (CFG) language defined by the syntax:

$$\begin{aligned} \mathtt{c} &::= \mathsf{label} \mid \mathtt{c};\mathtt{c} \mid \mathtt{c} \oplus \mathtt{c} \mid \mathtt{c}^* \\ \mathsf{label} &::= \mathtt{NonZero(b)} \mid \mathtt{Zero(b)} \mid \mathsf{stm} \end{aligned} \tag{1}$$

where the term $\mathtt{c};\mathtt{c}$ represents sequential composition; the term $\mathtt{c} \oplus \mathtt{c}$ is the choice command that corresponds to the execution of one of the two possible branches; the term $\mathtt{c}^*$ is the Kleene closure of $\mathtt{c}^n$, $n \in \mathbb{N}$, where $\mathtt{c}^n$ is the composition $\mathtt{c};\ldots\mathtt{c};$, n times. For a Boolean expression `b` and a statement stm (both defined by the grammar of a specific language), $\mathtt{NonZero(b)}$ is a special label that indicates that in its path `b` is true. In contrast, $\mathtt{Zero(b)}$ indicates that the Boolean condition `b` is false in its path. This syntax is general enough to cover deterministic imperative languages [44, Chapter 14, Exercise 14.4] whenever stm contains at least an assignment statement and a null operator, such as skip. In fact, in this language, we can write both the `while` and `if then else` statements as follows:

$$\begin{aligned} \mathtt{if\ b\ then\ c_1\ else\ c_2} &\equiv (\mathtt{NonZero(b)};\mathtt{c_1}) \oplus (\mathtt{Zero(b)};\mathtt{c_2}) \\ \mathtt{while\ b\ do\ c} &\equiv (\mathtt{NonZero(b)};\mathtt{c})^*;\mathtt{Zero(b)} \end{aligned}$$

The language defined by (1) is interesting because it corresponds precisely to the control flow graph representation of programs, on which standard data-flow analysis is usually performed [41]. The CFG associated with a program is a graph with a start node corresponding to the program entry point, an end node

(a) label (b) $\mathtt{c}_1;\mathtt{c}_2$ (c) $\mathtt{c}_1 \oplus \mathtt{c}_2$ (d) $\mathtt{c}^*$

Fig. 5: Graphical representation of language c, the CFG.

corresponding to the exit point, and all other nodes corresponding to intermediate points in the execution of the program; each edge of the graph has a label that represents the change produced by the execution of an instruction of the language that is analysed. Hence, label defines the language of CFG edge labels displayed in Figure 5a where $l \in \mathsf{label}$, while the other elements of c determine how we compose the edges depending on their labels as displayed in Figure 5b for the CFG corresponding to the sequential composition $\mathtt{c};\mathtt{c}$, in Figure 5c for the nondeterministic choice $\mathtt{c} \oplus \mathtt{c}$ and in Figure 5d for the iteration $\mathtt{c}^*$. In this way, we can define a new analysis simply by providing abstract semantics for the instructions defining stm and for the truth evaluation of b, namely for the language of labels label.

In the following, if V is the set of program points, the CFG will be defined as sets of edges, labelled in label, between nodes in V, i.e., as subsets of $V \times \mathsf{label} \times V$.

CFG for Guppy Programs We will demonstrate our approach on the Guppy language [37]. This is a Python-embedded language whose compiler runs within the Python interpreter, but the compiled program is independent of the Python runtime. Guppy adopts Python's control flow (`if`, `for`, `while`, and `return` statements), allowing measurement outcomes as a guard.

When we analyse a concrete language, we need to specify the label category of the syntax in (1) by defining stm, i.e. the syntax of the language statements that will label the CFG. This section extends the label syntax to represent a control flow graph Guppy language. Like Python, a Guppy program is a set of function declarations. We can represent each function by a CFG and a program (i.e., a set of functions) as a set of CFGs.

Let $\mathbb{V}_q$ be the set of quantum variables, $\mathbb{V}_c$ the set of classical variables and $\mathbb{V} = \mathbb{V}_q \cup \mathbb{V}_c$ the set of variables of a program. We use $\vec{x} \in \mathbb{V}_c^n$, $\vec{q} \in \mathbb{V}_q^n$ and $\vec{v} \in \mathbb{V}^n$, where $n \in \mathbb{N}$, for denoting, respectively, a list of classical, quantum or both types of variables. Note that the list could be empty.

We denote by b a generic Boolean expression composed of classical variables or measurement of quantum variables (e.g. $\neg x \wedge \mathtt{measure}(q)$) and e to indicate a generic classical expression (that does not contain any quantum variable). We can extend the syntax in Equation 1, with Guppy statements as follows:

$$\mathsf{stm} ::= \mathtt{pass} \mid \mathtt{A}\colon \vec{v} \mid \vec{v} = \mathsf{fun}(\vec{v}) \mid \vec{x} = \mathtt{e} \mid \mathtt{return}\ \vec{v} \mid \mathtt{discard}(\vec{q}) \tag{2}$$

where pass is the Python statement that corresponds to *skip*, $\mathtt{A}\colon \vec{v}$ declares which variables are the function arguments and fun indicates both built-in function

(`measure`, quantum gates and initialisation function `qubit`()) and user-defined functions.

5 Data-Flow Analyses

In this section, we introduce the two key analyses needed to implement our approach: a forward data flow analysis, which we call *consuming analysis*, gathering information about the availability of variables at each program point, and a backward data flow analysis, which we call *uncomputation analysis*, gathering information about the usage of variables.

5.1 Consuming Analysis

This analysis aims to detect the available variables at each node of the program's CFG, i.e. the variables that are defined and not yet consumed. To this aim, we must check that, in all paths, each used variable is defined and not yet consumed. This means that the analysis must be definite [41], i.e., the information propagates among nodes by intersection.

Let $\mathbb{V}_q$ be the set of quantum variables. We define the lattice, $\langle \wp(\mathbb{V}_q), \supseteq \rangle$, of the powerset of $\mathbb{V}_q$ ordered by inverse inclusion. Thus, the top element is $\varnothing$, while the bottom is $\mathbb{V}_q$. For expressions $\mathtt{e}$, we denote by $Q(\mathtt{e}) \subseteq \mathbb{V}_q$ the set of the quantum variables in $\mathtt{e}$. We define the (abstract) consuming semantics $(\!| l |\!) : \wp(\mathbb{V}_q) \to \wp(\mathbb{V}_q)$ as the abstract edge effects defined for each label $l \in \mathsf{label}$, as follows:

$$\begin{aligned}
(\!|\, \mathtt{NonZero(b)} \,|\!)\mathcal{D} &= (\!|\, \mathtt{Zero(b)} \,|\!)\mathcal{D} = \mathcal{D} \setminus Q(\mathtt{b}) \\
(\!|\, \mathtt{pass} \,|\!)\mathcal{D} &= (\!|\, \vec{x} = \mathtt{e} \,|\!)\mathcal{D} = \mathcal{D} \\
(\!|\, \mathtt{return}\ \vec{v} \,|\!)\mathcal{D} &= \mathcal{D} \setminus Q(\vec{v}) \\
(\!|\, \vec{v}_1 = \mathsf{fun}(\vec{v}_2) \,|\!)\mathcal{D} &= (\mathcal{D} \setminus Q(\vec{v}_2)) \cup Q(\vec{v}_1) \\
(\!|\, \mathtt{A:}\ \vec{v} \,|\!)\mathcal{D} &= \mathcal{D} \cup Q(\vec{v}) \\
(\!|\, \mathtt{discard}(\vec{q}) \,|\!)\mathcal{D} &= \mathcal{D} \setminus \{\vec{q}\}
\end{aligned} \tag{3}$$

As a general rule, since any use consumes variables, the abstract semantics is simple: when a variable is used, it is removed from $\mathcal{D}$, and it is added when defined. In particular, for Boolean expressions, since in `b` the only quantum variables that can be used are the ones that are measured, we remove $Q(\mathtt{b})$ from $\mathcal{D}$.

Proposition 1 (Consuming semantics distributivity). *The abstract semantics $(\!| \cdot |\!)$ defined in Eq. 3 is distributive w.r.t. intersection, i.e., for all labels l and subsets $\mathbf{X} \subseteq \wp(\mathbb{V}_q)$, we have*

$$(\!| l |\!)(\cap \mathbf{X}) = \bigcap \{ (\!| l |\!)\mathcal{D} | \mathcal{D} \in \mathbf{X} \}$$

Proof (Sketch). This follows easily from the definition of $(\!| l |\!)$ and the properties of the set-theoretic operations (see e.g. [26,1,41]).

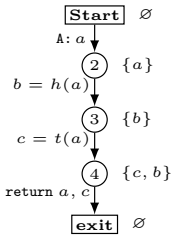


Fig. 6: CFG of program in Figure 2 with the computed $\mathcal{D}$ for each node

Computing the analysis. To compute the CFG analysis, we need to compute the set $\mathcal{D}$ for each node of the CFG [41], namely at each program point of the analysed program. Since the analysis is forward, the set $\mathcal{D}$ at node v, denoted $\mathcal{D}[v]$, depends on the sets $\mathcal{D}[u]$ of its predecessors u, and the label semantics of the edges entering v. Let **start** (**end**) be the starting (exit) node of a a CFG G. For each $v \in G$, we define

$$\mathcal{D}[v] = \begin{cases} \varnothing & \text{if } v = \textbf{start} \\ \bigcap \{ (\!| l |\!)(\mathcal{D}[u]) \mid (u, l, v) \in G \} & \text{otherwise} \end{cases}$$

thus obtaining a system of $n = |V|$ equations in n unknowns, which can be solved by fix-point, achieving the so-called Maximum Fixed Point (MFP) solution [41]. As shown in [41], this solution provides the best solution we can compute on a CFG. In fact, Proposition 1 implies that, for each $v \in V$, the computed available variables set $\mathcal{D}[v]$ corresponds to the MOP (Meet Over all Paths), namely

$$\mathcal{D}^*[v] = \bigcap \left\{ (\!| \pi |\!) \;\middle|\; \begin{array}{l} \pi = k_1, \ldots, k_n \text{ is a path from } \textbf{start} \text{ to } v, \\ (\!| \pi |\!) \stackrel{\text{def}}{=} (\!| k_n |\!) \circ \cdots \circ (\!| k_1 |\!) \end{array} \right\}$$

Soundness of the Analysis. By using the equality with the MOP solution, we can prove that, for each program point, the computed set of available variables is indeed an under-approximation of the available variables.

Theorem 1. *Given an edge (u, l, v), if the command represented by l uses a variable q which has not been defined or has been already consumed in at least one path, then $q \notin \mathcal{D}[u]$.*

Proof (Sketch). This follows directly from how we combine the paths' semantics. In fact, if a variable is not defined or is consumed in at least one path, it will not be included in $\mathcal{D}[u]$ thanks to the intersection operator.

As an example, consider the simple program in Figure 2.

This analysis shows that in node 3, the variable a is not available; in fact, the edge $(3, 4)$ leads to an error.

5.2 Uncomputation Analysis

The uncomputation analysis determines the appropriate locations where to insert the discard function. To this purpose, we must identify those unused variables which lead to implicit discarding. For this analysis, the notion of *live* variable comes in handy. We recall that a variable x is called *live* at point u if u is in a path between a previous definition of x and a following use of x (without interleaving further definitions of x) [1,26]. However, to figure out where it is necessary to uncompute, we need to know not only where a variable is live but also where the last use of the variable consumes it. Hence, we extend the notion of liveness to include this additional information.

Definition 1. *A quantum variable $q \in \mathbb{V}_q$ is* unsafe live *at point u if it is live and it is not consumed in at least one path starting from u; q is said to be* safe live *if it is consumed or returned in all paths from u.*

To simultaneously compute these two types of liveness, we define the abstract domain as a pair of sets of quantum variables. The analysis computes $(\mathcal{S}, \mathcal{U}) \in \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q)$, where $\mathcal{U} \subseteq \mathbb{V}_q$ is the set of *unsafe* live variables and $\mathcal{S} \subseteq \mathbb{V}_q$ is the set of *safe* live variables. Note that while *unsafety* will be over-approximated as it requires that the property of being non-consumed holds at least in one exiting path, *safety* requires the property holding on all the exiting paths, leading therefore to an under-approximation. A computational ordering $\sqsubseteq$, allowing for simultaneously over-approximating *unsafe live* variables and under-approximating *safe live* variables, can be defined on the abstract domain $\wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q)$ as follows.

Definition 2. *Let $(\mathcal{S}_1, \mathcal{U}_1), (\mathcal{S}_2, \mathcal{U}_2) \in \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q)$, we define*

$$(\mathcal{S}_1, \mathcal{U}_1) \sqsubseteq (\mathcal{S}_2, \mathcal{U}_2) \text{ iff } \mathcal{S}_2 \subseteq \mathcal{S}_1 \text{ and } \mathcal{U}_2 \supseteq \mathcal{U}_1 \cup (\mathcal{S}_1 \setminus \mathcal{S}_2).$$

Let $\sqcup$ be the least upper bound (lub) induced by $\sqsubseteq$. It is

$$(\mathcal{S}_1, \mathcal{U}_1) \sqcup (\mathcal{S}_2, \mathcal{U}_2) = (\mathcal{S}_3, \mathcal{U}_3), \text{ with}$$
$$\mathcal{S}_3 \stackrel{def}{=} \mathcal{S}_1 \cap \mathcal{S}_2 \text{ and } \mathcal{U}_3 \stackrel{def}{=} \mathcal{U}_1 \cup \mathcal{U}_2 \cup (\mathcal{S}_1 \Delta\, \mathcal{S}_2),$$

where Δ is the set-theoretic symmetric difference[3]*.*

This definition guarantees that the lub operator adds to $\mathcal{S}$ the variables that are safe in all paths and to $\mathcal{U}$ the variables that are unsafe in at least one path. Given that the operators of union, intersection, and symmetric difference are all both associative and commutative, it follows that the least upper bound operator is also associative and commutative. Moreover, as the union and intersection operators are idempotent and $A \Delta A = \emptyset$, it follows that the least upper bound operator is also idempotent.

[3] Given two sets A and B, $A \Delta B \stackrel{\text{def}}{=} (A \cup B) \setminus (A \cap B)$

Abstract Uncomputation Semantics. We define the abstract semantics of edge labels $(\!|\, l \,|\!) : \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q) \to \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q)$, for each $l \in \mathsf{label}$, as follows:

$$
\begin{aligned}
(\!|\, \mathtt{pass} \,|\!)(\mathcal{S}, \mathcal{U}) &= (\!|\, \vec{x} = \mathtt{e} \,|\!)(\mathcal{S}, \mathcal{U}) = (\mathcal{S}, \mathcal{U}) \\
(\!|\, \mathtt{return}\ \vec{v} \,|\!)(\mathcal{S}, \mathcal{U}) &= (\mathcal{S} \cup Q(\vec{v}), \mathcal{U}) \\
(\!|\, \vec{v}_1 = \mathsf{fun}(\vec{v}_2) \,|\!)(\mathcal{S}, \mathcal{U}) = (\mathcal{S}_1, \mathcal{U}_1) \text{ where } &\begin{cases} \mathcal{S}_1 \stackrel{\text{def}}{=} (\mathcal{S} \setminus Q(\vec{v}_1)) \cup Q(\vec{v}_2) \\ \mathcal{U}_1 \stackrel{\text{def}}{=} (\mathcal{U} \setminus Q(\vec{v}_1)) \end{cases} \\
(\!|\, \mathtt{NonZero(b)} \,|\!)(\mathcal{S}, \mathcal{U}) &= (\!|\, \mathtt{Zero(b)} \,|\!)(\mathcal{S}, \mathcal{U}) = (\mathcal{S} \cup Q(\mathtt{b}), \mathcal{U}) \\
(\!|\, \mathtt{A}\colon \vec{v} \,|\!)(\mathcal{S}, \mathcal{U}) &= (\mathcal{S} \setminus Q(\vec{v}), \mathcal{U} \setminus Q(\vec{v})) \\
(\!|\, \mathtt{discard}(\vec{q}) \,|\!)(\mathcal{S}, \mathcal{U}) &= (\mathcal{S} \cup \{q\}, \mathcal{U})
\end{aligned}
$$

The first rule deals with a classical instruction and does not change the analysis. In the other rules, the quantum variables which are used are added to $\mathcal{S}$ since all the uses consume variables. The variables $\vec{v}_1$ and $\vec{v}$ are removed from both $\mathcal{S}$ and $\mathcal{U}$ in the third and fifth rule, since these instructions define the variables, making them no more live. Since each statement consumes the variable, variables are never added to $\mathcal{U}$ by the abstract semantics. Nevertheless, this set will be updated by the join operator between paths.

Proposition 2. *The abstract semantics $(\!|\, \cdot \,|\!) : \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q) \to \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q)$ is monotonic w.r.t. $\sqsubseteq$.*

Proof. We have to prove that if $(\mathcal{S}_1, \mathcal{U}_1) \sqsubseteq (\mathcal{S}_2, \mathcal{U}_2)$ then $(\!|\, l \,|\!)(\mathcal{S}_1, \mathcal{U}_1) \sqsubseteq (\!|\, l \,|\!)(\mathcal{S}_2, \mathcal{U}_2)$. Since for all possible l, $(\!|\, l \,|\!)$ changes $(\mathcal{S}_1, \mathcal{U}_1)$ and $(\mathcal{S}_2, \mathcal{U}_2)$ in the same ways, the proof follows trivially by the definition of the abstract semantics.

Since the abstract semantics is monotonic and $\langle \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q), \sqsubseteq \rangle$ is a finite domain (as $\mathbb{V}_q$ is finite for all programs), we are guaranteed that by iteratively applying the equations, we reach a fix-point.

Proposition 3. *The abstract semantics $(\!|\, \cdot \,|\!) : \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q) \to \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q)$ is distributive, i.e., for all labels l and sets $\mathbf{X} \subseteq \wp(\mathbb{V}_q) \times \wp(\mathbb{V}_q)$, we have*

$$(\!|\, l \,|\!)(\sqcup \mathbf{X}) = \bigsqcup \{ (\!|\, l \,|\!)(\mathcal{S}, \mathcal{U}) | (\mathcal{S}, \mathcal{U}) \in \mathbf{X} \}.$$

Proof. Thanks to the associativity of the lub operator, we need only to prove that, given $X = (\mathcal{S}_1, \mathcal{U}_1)$ and $Y = (\mathcal{S}_2, \mathcal{U}_2)$, then $(\mathcal{S}_3, \mathcal{U}_3) \stackrel{\text{def}}{=} (\!|\, \cdot \,|\!)(X \sqcup Y) = (\!|\, \cdot \,|\!)(X) \sqcup (\!|\, \cdot \,|\!)(Y) \stackrel{\text{def}}{=} (\mathcal{S}_4, \mathcal{U}_4)$. The generic abstract semantics, for any $l \in \mathsf{label}$, on a pair $(\mathcal{S}, \mathcal{U})$ can be defined as $(\!|\, l \,|\!)(\mathcal{S}, \mathcal{U}) = ((\mathcal{S} \setminus K) \cup G, (\mathcal{U} \setminus K') \cup G')$ where $K, G, K', G' \in \wp(\mathsf{Vars})$ depend on l. Firstly, we consider $\mathcal{S}_1$ and $\mathcal{S}_2$, i.e., the safe variable sets:

$$\mathcal{S}_3 = ((\mathcal{S}_1 \cap \mathcal{S}_2) \setminus K) \cup G = ((\mathcal{S}_1 \setminus K) \cup G) \cap ((\mathcal{S}_2 \setminus K) \cup G) = \mathcal{S}_4$$

Now we consider $\mathcal{U}_1$ and $\mathcal{U}_2$, i.e., the unsafe variables are:

$$
\begin{aligned}
\mathcal{U}_3 &= (((\mathcal{U}_1 \cup \mathcal{U}_2) \setminus K) \cup G) \cup (\mathcal{S}_1 \Delta\, \mathcal{S}_2) \\
&= ((\mathcal{U}_1 \setminus K) \cup G) \cup ((\mathcal{U}_1 \setminus K) \cup G) \cup (\mathcal{S}_1 \Delta\, \mathcal{S}_2) = \mathcal{U}_4
\end{aligned}
$$

Computing the analysis. To compute the analysis on the CFG, we need to compute the pair $(\mathcal{S}, \mathcal{U})$ for each node of the CFG [41]. Similarly to standard liveness, our analysis is backward, i.e., the pair $(\mathcal{S}, \mathcal{U})$ at node u depends on the pairs $(\mathcal{S}', \mathcal{U}')$ of its successors and the label semantics of the edges exiting from u. Given a CFG G, for all node v in G, we define the following system of equations:

$$(\mathcal{S}, \mathcal{U})[u] \begin{cases} (\varnothing, \varnothing) & \text{if } u = \textbf{end} \\ \bigsqcup \left\{ \; (\!| l |\!)((\mathcal{S}, \mathcal{U})[v]) \;\middle|\; (u, l, v) \in G \right\} & \text{otherwise} \end{cases} \tag{4}$$

As we did for consuming analysis, this system can be solved by fix-point, also obtaining, in this case, the so-called MFP solution [41]. Due to Th. 3 [1, Chapter 9][26, Chapter 2], this solution provides the best solution we can compute on CFG, which is the MOP solution computed on the graph nodes, i.e.,

$$(\mathcal{S}, \mathcal{U})^*[v] = \bigsqcup \left\{ \; (\!| \pi |\!)(\mathcal{S}_{\mathbf{e}}, \mathcal{U}_{\mathbf{e}}) \;\middle|\; \begin{array}{l} \pi = k_1, \ldots, k_n \text{ is a path from } v \text{ to } \textbf{end}, \\ (\!| \pi |\!) \stackrel{\text{def}}{=} (\!| k_1 |\!) \circ \cdots \circ (\!| k_n |\!) \end{array} \right\},$$

Where **end** is the final node of the control flow graph, i.e., the program exit point, and $(\mathcal{S}_{\mathbf{e}}, \mathcal{U}_{\mathbf{e}}) = \bot$ is the pair $(\mathcal{S}, \mathcal{U})$ holding the **end** point.

Soundness. By construction, $\mathcal{U} \cap \mathcal{S} = \varnothing$, and if we join $\mathcal{U}$ and $\mathcal{S}$, we obtain the set of live variables. Hence, being the liveness analysis sound, if a variable x is live in a point u, then $x \in \mathcal{U} \cup \mathcal{S}$, with $(\mathcal{S}, \mathcal{U}) = (\mathcal{S}, \mathcal{U})[u]$.

Proposition 4. *For each program point u, if x is unsafe live in u, then $x \in \mathcal{U}$, with $(\mathcal{S}, \mathcal{U}) = (\mathcal{S}, \mathcal{U})[u]$.*

Proof. Consider a node u and a live variable x at u. By definition, x is unsafe live if it is not consumed in at least one path from node u to the end node. Therefore, in at least one path, the abstract semantics does not add x in $\mathcal{S}$, so when applying the join on paths, x will be added to $\mathcal{U}$.

The analysis is incomplete because the MOP solution considers all feasible paths, including those that may never be executed. Nonetheless, this is not an issue since uncomputation is placed only in unsafe paths. If unfeasible paths make the variable unsafe, the uncomputation is also placed in those paths and thus will never be performed.

Examples. First, we consider the code in Figure 4a that we represent as a CFG accompanied by the analysis results depicting $(\mathcal{S}, \mathcal{U})$ for each node in Figure 7a. In Figure 7a, where the `discard` is needed, the variable b is not live. Instead, if we analyse the correct versions of the program (Figure 4b and Figure 7b), we see that after the definition, the variable b is live, thanks to the discard function.

Consider the code in Figure 3a. Figure 8 shows the CFG corresponding to the program and the analysis results. After node 4, the variable a is no longer

(a) CFG corresponding to Figure 4a (b) CFG corresponding to Figure 4b

Fig. 7: In both CFGs,on the rights the pairs $(\mathcal{S}, \mathcal{U})$

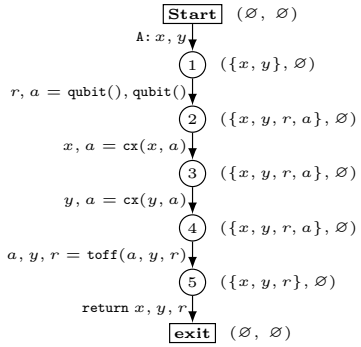


Fig. 8: CFG corresponding to Figure 3a, on the rights the pairs $(\mathcal{S}, \mathcal{U})$

live; thus, in Figure 3b, we insert the uncomputation just after edge $(4, a, y, r = \texttt{toff}(a, y, r), 5)$. The uncomputation analysis is very useful when a variable needs to be discarded only in one branch or when an overwriting occurs in a loop. As an example, consider the program in Figure 9a. Since `a` and `b` are used only in one branch, they are unsafe. In fact, when we execute the if-branch, we implicitly discard `b`, and when we execute the else-branch, we implicitly discard `a`. In this case, a simple analysis that detects only the live variable is not enough. Instead, as we see in Figure 9b, due to the lub operator, our analysis inserts both a and b in the set $\mathcal{U}[1]$.

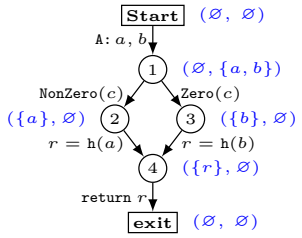
6 Applying the Analyses to Quantum Programs

It should be clear that the main motivation for the design of the analyses presented above lies in exploiting the information they provide to transform the program automatically without requiring programmer actions. Hence, after formally defining the analyses, it becomes fundamental to define the appropriate pipeline in which they should be performed. This is represented in Figure 10 for a given program CFG and consists of four stages:

1. *Consuming Analysis.* We first perform this analysis to obtain the sets $\mathcal{D}[u]$ for each node u, over-approximating the sets of variables available in u;

```
def fun(a: qubit,b: qubit,c: bool)
    -> qubit:
  # c is classical
  if c:
      r = h(a)
  else:
      r = h(b)
  return r
```

(a)



(b)

Fig. 9: The function (a) consumes different variables in different branches. In particular, the `if-branch` consumes only `a`, thus implicitly discarding `b`, while the `else-branch` consumes `b`, thus implicitly discarding `a`. In (b), we show the results of the uncomputation analysis on the CFG of the function (a) and how it detects `a,b` as unsafe at node 1.

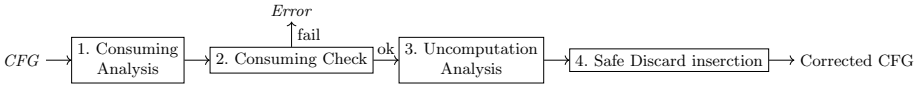


Fig. 10: Analysis pipeline

2. *Consuming check*. We use $\mathcal{D}$ to check whether the program is correct, i.e., whether it does not use consumed or undefined variables;
3. *Uncomputation Analysis*. On correct programs, we perform the uncomputation analysis;
4. *Discard insertion*. The results of such analysis are then used to decide where it is recommended to insert a discard, avoiding implicit discard operations.

The first and the third steps are precisely the analyses described in the previous section. What we need now to define precisely are the procedures for performing the second and fourth steps of the pipeline.

Consuming Check. After performing the Consuming Analysis, we obtain the sets $\mathcal{D}[u]$ for each program point. First, we check if every used variable is defined and not consumed. Moreover, we also have to check that a variable is not passed more than once as an argument (e.g., the statement $\mathsf{fun}(q, q)$ is discarded since it introduces implicit copies). As shown in Algorithm 1, to check this property, for all edges (u, l, v), i.e., for all labels l in the CFG, we get the set of all used variable ($usedVars$), and if this set is not included in $\mathcal{D}[u]$ this means that there are some variables that are used without being defined or after being consumed, so we return an error. In Algorithm 1, we simplified the returned message in case of an error. In the implemented version, we return a detailed error specifying which variables generate errors and at which point the programs. For instance, the example in Figure 6 shows that the check fails when considering the edge $(2, b = \mathtt{h}(a), 3)$ since a is not in $\mathcal{D}[2]$.

Algorithm 1 Check for Variable Usage

Require: *edges*: control flow graph edges, $\mathcal{D}$
1: **for** u, v **in** *edges* **do**
2: $label \leftarrow$ getLabel(u, v)
3: **if** *label* is a function call **then**
4: **if** $\neg$ checkLegitUse(*label*) **then**
5: **return** 'Error: implicit copy'
6: **end if**
7: **end if**
8: $usedVars \leftarrow$ getUsedVars(*label*)
9: **if** $usedVars \nsubseteq \mathcal{D}[u]$) **then**
10: **return** 'Error, not defined variable used.'
11: **end if**
12: **end for**

Discard Insertion. The last step consists of inserting the `discard` function. We must discard all variables which were not consumed or returned in at least one path, i.e., defined and never used (not live) or used but not in all paths. Hence, in the first case, we can identify all non-live variables and insert the uncomputation just after their definition. In the second case, the variables, at some program point, are in the set $\mathcal{U}$ determined by the Uncomputation analysis, and we have to insert the discard only in those paths that do not consume it. This means that, in the second case, we must check if (and where) a variable is in $\mathcal{U}$ to understand where we have to insert discard. We show the algorithm in detail in Algorithm 2.

The procedure receives as input the set of arcs and nodes from the CFG, the list of sets pair ($\mathcal{S}$, $\mathcal{U}$) from the previous analysis, and the program quantum variables set $\mathbb{V}_q$. First, for each quantum variable $var \in \mathbb{V}_q$, we check if the variable is defined and not used, checking if, after the definition, it is live. If not, we must insert the discard after the definition. Then, we consider all variables that, for some nodes u, are in the sets $\mathcal{U}[u]$. If var is in some $\mathcal{U}[u]$, there is some node in which, in some of the paths that start in that node, the variable is safe live, and others in which it is not live. So, for each node u of the CFG, if x is in the set $\mathcal{U}[u]$, we check the successors of u. For each successor v of u, if x is not in the set $\mathcal{S}[v]$ and is not in the set $\mathcal{U}[v]$, the node v is the head of the 'unsafe' path, so a discard operation is added between nodes u and v.

In Algorithm 2, we apply our algorithm to the simple examples introduced so far. Let us consider the example in Figure 7a. Here the assignment in edge $(1, b = \mathtt{h}(a), 2)$ defines a non-live variable (b). So we insert the discard at the end of that edge (just like we did in Figure 4b and in Figure 7b). Similarly, for the example in Figure 8, when we apply the discard insertion, the only assignment that defines a non-live variable is the one in edge $(4, a, y, r = \mathtt{toff}(a, y, r), 5)$, so we insert the discard at the end of that edge (line 6 in Figure 3a, just like the function in Figure 3b shows). Now consider the function analysed in Figure 9b. In this case, there are no non-living variables, but two variables are in $\mathcal{U}$. Applying

Algorithm 2 Insert discard

Require: $edges, nodes, (\mathcal{S}, \mathcal{U}), \mathbb{V}_q$
for var **in** $\mathbb{V}_q$ **do**
  for u, v **in** getAllDefinition(cfg, var) **do**
    if $var \notin (\mathcal{S}[v] \cup \mathcal{U}[v])$ **then**
      addDiscard$((u, v), var)$
    end if
  end for
end for
for var **in** $\bigcup_u \mathcal{U}[u]$ **do**
  for $node$ **in** $nodes$ **do**
    if $var \in \mathcal{U}[node]$ **then**
      for v **in** successor$(node)$ **do**
        if $var \notin (\mathcal{S}[v] \cup \mathcal{U}[v])$ **then**
          addDiscard$((node, v), var)$
        end if
      end for
    end if
  end for
end for

the discard insertion algorithm, we select the edge $(1, \texttt{NonZero}(c), 2)$ to insert the discard of b and $(1, \texttt{Zero}(c), 3)$ to insert the discard of a.

Evaluation. We have implemented a prototype of our procedure in Python 3. The code is available at Github repository. We tested both the consuming check and insertion of the discard operation, paying more attention to the discard insertion. In particular, the discard insertion has been tested on about ten simple programs that present redefinition of unconsumed variables inside loops, redefinition in different computation branches and unused variables. Moreover, all the examples in this work have been tested.

Our analysis introduces the same level of approximation as a type checker. Both the analysis and the type system consider all possible paths, even those that are not executable. One crucial distinction between the two approaches is that the type system will generate an error, forcing the programmer to modify an infeasible path. In contrast, our method will automatically modify only the impossible path so that the program semantics will not be affected.

Moreover, our static analysis is more informative than the type system, as the example in Figure 11 highlights. A type checker based on linear types would return an error at line 1, indicating that `a` is not consumed but giving no information on the point where the discard must be inserted, namely the `else-branch`. Instead, our approach would identify that the problem lies in the `else-branch` and would pass the information to the next step of the pipeline without rejecting the program. Consequently, even when using our approach only as a linearity checker, it would provide more informative results than the type system. This

```
a = qubit()
if _:
    c = f(a)
else:
    c = qubit()
return c
```

Fig. 11

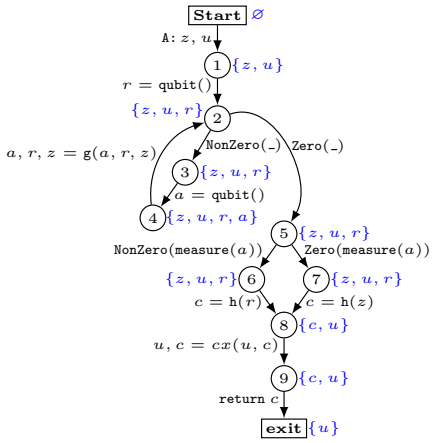
```
def f(z:qubit,u:qubit,n:int):
    r = qubit()
    for _ in range(n):
        a = qubit()
        a,r,z = g(a,r,z)

    if measure(a):
        c = h(r)
    else:
        c = h(z)

    u,c = cx(u,c)

    return c
```

(a)



(b)

Fig. 12: The example function

is particularly beneficial when the definition and the path that does not use the variable are far apart in the code and the control flow is complicated.

7 Putting all toghether

In this section, we apply the entire pipeline to a more complex program, detailing the system evolution that leads to the MFP solution. Consider the function f in Figure 12a and the corresponding CFG in Figure 12b. We show the system of equations derived from the CFG in Figure 13 and the system's computation in Figure 13b. In Figure 12b, we also indicate the sets $\mathcal{D}$ for each program point computed by Consuming Analysis. Then, applying the algorithm in Algorithm 1, we raise an error in edges $(5, \texttt{NonZero}(\texttt{measure}(a)), 6)$ and $(5, \texttt{Zero}(\texttt{measure}(a)), 7)$ since we use a, which is not defined in all paths.

Figure 14a and Figure 14b show the code and the CFG of the function after correcting the errors highlighted by the previous step. Now, this function passes the correctness check, and we can analyse it to see if it needs some discard function. We show the system of equations derived from the CFG in Figure 15a and the system's computation in Figure 15b. In Figure 14b, we show on each node the sets $(\mathcal{S}, \mathcal{U})$ corresponding to the MFP solution computed in Figure 15b. We

$$
\begin{aligned}
&\mathcal{D}[\textbf{Start}] = \varnothing\\
&\mathcal{D}[1] = \mathcal{D}[\textbf{Start}] \cup \{z, u\}\\
&\mathcal{D}[2] = (\mathcal{D}[1] \cup \{r\}) \cap (\mathcal{D}[4] \setminus \{a, r, z\} \cup \{a, r, z\})\\
&\mathcal{D}[3] = \mathcal{D}[2]\\
&\mathcal{D}[4] = \mathcal{D}[3] \cup \{a\}\\
&\mathcal{D}[5] = \mathcal{D}[2]\\
&\mathcal{D}[6] = \mathcal{D}[5] \setminus \{a\}\\
&\mathcal{D}[7] = \mathcal{D}[5] \setminus \{a\}\\
&\mathcal{D}[8] = (\mathcal{D}[6] \setminus \{r\} \cup \{c\}) \cap (\mathcal{D}[6] \setminus \{z\} \cup \{c\})\\
&\mathcal{D}[9] = (\mathcal{D}[8] \setminus \{u, c\} \cup \{u, c\})\\
&\mathcal{D}[\textbf{End}] = \mathcal{D}[9] \setminus \{c\}
\end{aligned}
$$

(a)

	0	1	F.P.
Start	$\varnothing$	$\varnothing$	$\varnothing$
1	$\mathbb{V}_q$	$\{z, u\}$	$\{z, u\}$
2	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
3	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
4	$\mathbb{V}_q$	$\{z, u, r, a\}$	$\{z, u, r, a\}$
5	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
6	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
7	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
8	$\mathbb{V}_q$	$\{u, c\}$	$\{u, c\}$
9	$\mathbb{V}_q$	$\{u, c\}$	$\{u, c\}$
End	$\mathbb{V}_q$	$\{u\}$	$\{u\}$

(b)

Fig. 13: We show in (a) the system of equations derived from the CFG in Figure 12b and in (b) MFP solution of the system

now apply the algorithm in Algorithm 2. The variable u is not live after its last definition in the edge $(8, u, c = \texttt{cx}(u, c), 9)$, so we insert the discard there. The variable a is in $\mathcal{U}[2]$, being in $\mathcal{S}[5]$ and not being live in 3, we insert `discard`(a) in $(2, \texttt{NonZero}(\_), 3)$. The variable r is Unsafe in multiple nodes, in particular:

- $r \in \mathcal{U}[2]$ but is live in both 3 and 5, so I don't need to do anything
- $r \in \mathcal{U}[5]$ and being in $\mathcal{S}[6]$ and not being live in 7, we insert `discard`(r) in $(5, \texttt{Zero}(\texttt{measure}(a)), 7)$.

Similarly, z is in $\mathcal{U}[2]$ and live in the successors, so it does not need to be discarded, whereas it is in $\mathcal{U}[5]$ and is in $\mathcal{S}[7]$ and not live in 6, we insert `discard`(z) in $(5, \texttt{NonZero}(\texttt{measure}(a)), 6)$. The procedure employed in this example successfully resolved the implicit discarding, which would have resulted in the type system rejecting the program. Finally, we present the output programs with the discard insertions in Figure 16. We note that by using a type-based approach, the type checker rejects the program in Figure 14a unless the programmer enters all the discard functions shown in Figure 16.

8 Related Works

Safe Discarding and Uncomputation. Most of the literature has focused on synthesising uncomputation and optimising the efficiency of the resulting procedure instead of detecting which variables or qubits need to be uncomputed (i.e. discarded). Square [11] and `staq` [3] utilize the uncomputation to reduce the number of ancilla qubits. To manage ancillae Quipper [14] provides a function to convert classical Haskell programs in the quantum circuits automatically and, during this process, introduces the uncomputation. Other works like REVS [32,31], Amy Matthew et al. work [5], ReQWIRE [38], Paris A. et al. works [29,30], Q# [42] and Silq [6] provide functions that synthesise the quantum circuit for uncomputation. Still, the synthesis is limited to circuits representing a classical function

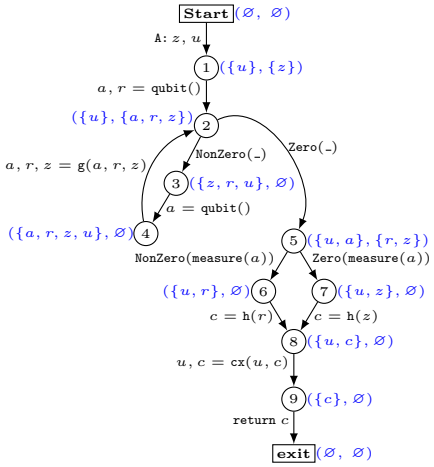
```
def r_f(z:qubit,u:qubit,n:int):
    a, r = qubit()
    for _ in range(n):
        a = qubit()
        a,r,z = g(a,r,z)

    if measure(a):
        c = h(r)
    else:
        c = h(z)

    u,c = cx(u,c)

    return c
```

(a)



(b)

Fig. 14: The correct version of function `f` in Figure 12

(Qfree circuit.)[4]. In contrast, ReQWIRE [38] only verifies the user-defined uncomputation. Silq [6] and Qunity [43] provide automatic uncomputation but only for backend variables and do not consider user-defined variables. Finally, Qrisp[39,40] introduces automatic uncomputation in functions which are marked with a `@auto_uncompute` decorator and provides manual uncomputation via an `uncompute` function. However, Qrisps is at a slightly higher level than a circuit language with an abstraction provided by a `QuantumVariables` class (with various subclasses), which still represents a set of qubits and works in a positional way (e.g. `q = QuantumVariable(1); x(q)`).

Program safety. Most of the quantum languages based on variables like Silq [6] and QWIRE [33] use linear typing to prevent copy and implicit discarding. Quipper does not use a linear type system and is prone to error. Instead, Quipper typed evolution, Proto-Quipper [13], uses a linear type system. We note that Silq provides more sophisticated solutions introducing an annotation (`cost`) that permits not to consume some variables and use it multiple times (implementing a sort of copy using CNOT gate).

Abstract interpretation. Feng et al. [12] explore the application of abstract interpretation [9] in the context of quantum programs. Perdrix's [35,36] and Honda's [17] entanglement analyses use abstract semantics based on data flow rather than circuits. Finally, using the formalism of Abstract Interpretation, Yu et al. [46] propose an abstraction of quantum domains based on the quantum circuits model of quantum computation.

[4] Functions that contain only classical gates, such as the NOT and the CNOT gates.

$(\mathcal{S}, \mathcal{U})[\textbf{End}] = (\varnothing, \varnothing)$
$(\mathcal{S}, \mathcal{U})[9] = (\mathcal{S}[\textbf{End}] \cup \{c\}, \mathcal{U}[\textbf{End}])$
$(\mathcal{S}, \mathcal{U})[8] = (\mathcal{S}[9] \setminus \{u, c\} \cup \{u, c\}, \mathcal{U}[9] \setminus \{u, c\})$
$(\mathcal{S}, \mathcal{U})[7] = (\mathcal{S}[8] \setminus \{c\} \cup \{z\}, \mathcal{U}[8] \setminus \{c\})$
$(\mathcal{S}, \mathcal{U})[6] = (\mathcal{S}[8] \setminus \{c\} \cup \{r\}, \mathcal{U}[8] \setminus \{c\})$
$(\mathcal{S}, \mathcal{U})[5] = (\mathcal{S}[6] \cup \{a\}, \mathcal{U}[6]) \sqcup (\mathcal{S}[7] \cup \{a\}, \mathcal{U}[7])$
$(\mathcal{S}, \mathcal{U})[4] = (\mathcal{S}[2] \setminus \{a, r, z\} \cup \{a, r, z\}, \mathcal{U}[2] \setminus \{a, r, z\})$
$(\mathcal{S}, \mathcal{U})[3] = (\mathcal{S}[4] \setminus \{a\}, \mathcal{U}[4] \setminus \{a\})$
$(\mathcal{S}, \mathcal{U})[2] = (\mathcal{S}, \mathcal{U})[3] \sqcup (\mathcal{S}, \mathcal{U})[5]$
$(\mathcal{S}, \mathcal{U})[1] = (\mathcal{S}[2] \setminus \{a, r\}, \mathcal{U}[2] \setminus \{a, r\})$
$(\mathcal{S}, \mathcal{U})[\textbf{Start}] = (\mathcal{S}[1] \setminus \{z, u\}, \mathcal{U}[1] \setminus \{z, u\})$.

(a) System of equations derived from the CFG in Figure 14b

	0	1	2	F.P.
End	$(\varnothing, \varnothing)$	$(\varnothing, \varnothing)$	$(\varnothing, \varnothing)$	
9	$\bot$	$(\{c\}, \varnothing)$	$(\{c\}, \varnothing)$	
8	$\bot$	$(\{u, c\}, \varnothing)$	$(\{u, c\}, \varnothing)$	
7	$\bot$	$(\{u, z\}, \varnothing)$	$(\{u, z\}, \varnothing)$	
6	$\bot$	$(\{u, r\}, \varnothing)$	$(\{u, r\}, \varnothing)$	
5	$\bot$	$(\{u, a\}, \{r, z\})$	$(\{u, a\}, \{r, z\})$	
4	$\bot$	$\bot$	$(\{a, r, z, u\}, \varnothing)$	
3	$\bot$	$\bot$	$(\{z, r, u\}, \varnothing)$	
2	$\bot$	$(\{u, a\}, \{r, z\})$	$(\{u\}, \{a, r, z\})$	
1	$\bot$	$(\{u\}, \{z\})$	$(\{u\}, \{z\})$	
Start	$\bot$	$(\varnothing, \varnothing)$	$(\varnothing, \varnothing)$	

(b) MFP solution of the system

Fig. 15

```
def rr_f(z:qubit,u:qubit,n:int):
    a, r = qubit()
    for _ in range(n):
        discard(a)
        a = qubit()
        a,r,z = g(a,r,z)
    if measure(a):
        discard(z)
        c = h(r)
    else:
        discard(r)
        c = h(z)
    u,c = cx(u,c)
    discard(u)

    return c
```

Fig. 16: The function in Figure 14 after the discard insertion

Static Analysis. Static analysis has been applied to quantum optimisation. Some works use static analysis on Intermediate Representation (IR) based on the Static-Single-Assignment (SSA). QIRO [18] introduces an MLIR dialect to represent data flow with quantum computation and use static analysis techniques to optimise the resulting circuit. QSSA [34] introduces an analysis to verify that the IR based on SSA qubits are used at most once. However, since their IR is based on SSA, their analysis is less flexible than our approach (e.g. writing $x = g(x)$ is not allowed). The isQ compiler [16,15] introduces optimisations applying static analysis on their IR in the compiling pipeline. Amy et al. [4,2] use static analysis for phase folding optimisation (i.e. merging phase gates) and integrating classical data flow into a circuit language. Chen et al. [8] adapt classical constant propagation techniques to optimise quantum circuits. Kaul et al. [21] extend Code Property Graphs [45] to represent quantum circuits. Additionally, Javadi-Abhari et al. [19] introduce a compiler, ScaffCC, which includes various analyses such as entanglement, timing, resource analysis, and instruction reordering on a modified version of QASM[10]. QChecker [47] and LintQ [28] utilise static

techniques to identify potential bugs such as resource allocation problems or incorrect measurements. QChecker operates as a static analysis tool, leveraging AST information on Qiskit code. LintQ also considers Python code with Qiskit and incorporates control flow modelling. All these approaches focus on quantum circuits or IR, in contrast with our work, which operates at the highest level of the stack.

9 Conclusion

We have introduced an analysis of uncomputation for high-level quantum programming languages. This analysis is intended to facilitate programming tasks in such languages by reducing the errors generated during compilation and relieving the programmer of low-level tasks.

As we have shown in Figure 10, our approach is based on two separate analyses in sequence, with the aim of overcoming some limitations of the type systems-based approach. The first analysis ensures that the variables are used at most once. After that, a second analysis additionally tells the compilers the points of the program where a variable should be discarded (uncomputed), thus unburdening the programmer from this task. The replacement of the type system with a static analysis adds flexibility in two ways: it is language-independent, and it can be integrated with other analyses to potentially identify infeasible paths, thus increasing precision. In particular, we can use well-known classical analyses, such as interval analysis and constant propagation, to obtain more precise classical control flows. Additionally, we could consider supplementary quantum analyses such as entanglement analysis to avoid unnecessary uncomputation. Various approaches have been proposed for such an analysis, e.g. [35,36], [17] and [19], which we believe can be further improved for our purposes.

Our analyses can be easily adapted to any language using a linear type system or even to improve languages that do not use it, such as Quipper [14]. Furthermore, languages like Silq [6], which rely on annotations to determine if a variable is consumed, can also be analysed with minimal modifications. Our analysis enables the definition of commands that do not consume resources while still identifying variables for uncomputation, thanks to the modified liveness information. This versatility allows for a broader applicability across various quantum programming languages.

As a future work, it would be interesting to integrate the analysis into the Guppy compiler. This would allow the definition of a new version of Guppy with a lighter type checker, making the language easier to use. This integration should be straightforward since we have implemented our analyses in Python, which is the language of Guppy's compiler. Finally, our analysis could be used to detect unused resources in classical languages such as Rust[24], where a notion of resource consumption is used to handle pointers.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
2. Amy, M.: Formal Methods in Quantum Circuit Design. Ph.D. thesis, University of Waterloo (2019)
3. Amy, M., Gheorghiu, V.: staq—A full-stack quantum processing toolkit. Quantum Science and Technology **5**(3), 034016 (2020)
4. Amy, M., Maslov, D., Mosca, M.: Polynomial-time T-depth optimization of Clifford+ T circuits via matroid partitioning. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **33**(10), 1476–1489 (2014)
5. Amy, M., Roetteler, M., Svore, K.M.: Verified compilation of space-efficient reversible circuits. In: Majumdar, R., Kunčak, V. (eds.) Computer Aided Verification. pp. 3–21. Springer International Publishing, Cham (2017)
6. Bichsel, B., Baader, M., Gehr, T., Vechev, M.: Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 286–300. PLDI 2020, Association for Computing Machinery, New York, NY, USA (2020)
7. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A correctness and incorrectness program logic. Journal of the ACM **70**(2), 1–45 (2023)
8. Chen, Y., Stade, Y.: Quantum Constant Propagation. In: International Static Analysis Symposium. pp. 164–189. Springer (2023)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. pp. 238–252 (1977)
10. Cross, A., Javadi-Abhari, A., Alexander, T., De Beaudrap, N., Bishop, L.S., Heidel, S., Ryan, C.A., Sivarajah, P., Smolin, J., Gambetta, J.M., et al.: OpenQASM 3: A broader and deeper quantum assembly language. ACM Transactions on Quantum Computing **3**(3), 1–50 (2022)
11. Ding, Y., Wu, X.C., Holmes, A., Wiseth, A., Franklin, D., Martonosi, M., Chong, F.T.: SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA). pp. 570–583 (2020)
12. Feng, Y., Li, S.: Abstract interpretation, Hoare logic, and incorrectness logic for quantum programs. Information and Computation **294**, 105077 (2023)
13. Fu, P., Kishida, K., Ross, N.J., Selinger, P.: A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper. In: Reversible Computation: 12th International Conference, RC 2020, Oslo, Norway, July 9-10, 2020, Proceedings 12. pp. 153–168. Springer (2020)
14. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: A Scalable Quantum Programming Language. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 333–342. PLDI '13, Association for Computing Machinery, New York, NY, USA (2013)
15. Guo, J., Lou, H., Li, R., Fang, W., Liu, J., Long, P., Ying, S., Ying, M.: isQ: Towards a practical software stack for quantum programming. arXiv preprint arXiv:2205.03866 (2022)

16. Guo, J., Lou, H., Yu, J., Li, R., Fang, W., Liu, J., Long, P., Ying, S., Ying, M.: isQ: An Integrated Software Stack for Quantum Programming. IEEE Transactions on Quantum Engineering (2023)
17. Honda, K.: Analysis of quantum entanglement in quantum programs using stabilizer formalism. arXiv preprint arXiv:1511.01572 (2015)
18. Ittah, D., Häner, T., Kliuchnikov, V., Hoefler, T.: Enabling dataflow optimization for quantum programs. arXiv preprint arXiv:2101.11030 (2021)
19. JavadiAbhari, A., Patil, S., Kudrow, D., Heckey, J., Lvov, A., Chong, F.T., Martonosi, M.: ScaffCC: A framework for compilation and analysis of quantum computing programs. In: Proceedings of the 11th ACM Conference on Computing Frontiers. pp. 1–10 (2014)
20. Johnston, E., Harrigan, N., Gimeno-Segovia, M.: Programming Quantum Computers: Essential Algorithms and Code Samples. O'Reilly Media, Incorporated (2019)
21. Kaul, M., Küchler, A., Banse, C.: A Uniform Representation of Classical and Quantum Source Code for Static Code Analysis. In: 2023 IEEE International Conference on Quantum Computing and Engineering (QCE). vol. 1, pp. 1013–1019. IEEE (2023)
22. Kitaev, A.Y., Shen, A.H., Vyalyi, M.N.: Classical and Quantum Computation. American Mathematical Society (2002)
23. Koch, M., Lawrence, A., Singhal, K., Sivarajah, S., Duncan, R.: GUPPY: Pythonic Quantum-Classical Programming. https://popl24.sigplan.org/details/planqc-2024-papers/8/GUPPY-Pythonic-Quantum-Classical-Programming
24. Matsakis, N.D., Klock, F.S.: The rust language. ACM SIGAda Ada Letters **34**(3), 103–104 (2014)
25. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press (2010)
26. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (2015)
27. O'Hearn, P.W.: Incorrectness logic. Proceedings of the ACM on Programming Languages **4**(POPL), 1–32 (2019)
28. Paltenghi, M., Pradel, M.: LintQ: A Static Analysis Framework for Qiskit Quantum Programs. arXiv preprint arXiv:2310.00718 (2023)
29. Paradis, A., Bichsel, B., Steffen, S., Vechev, M.: Unqomp: Synthesizing Uncomputation in Quantum Circuits. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 222–236. Association for Computing Machinery (2021)
30. Paradis, A., Bichsel, B., Vechev, M.: Reqomp: Space-constrained Uncomputation for Quantum Circuits (2022)
31. Parent, A., Roetteler, M., Svore, K.M.: Reversible circuit compilation with space constraints. arXiv preprint arXiv:1510.00377 (2015)
32. Parent, A., Roetteler, M., Svore, K.M.: REVS: A Tool for Space-Optimized Reversible Circuit Synthesis. In: Phillips, I., Rahaman, H. (eds.) Reversible Computation. pp. 90–101. Springer International Publishing (2017)
33. Paykin, J., Rand, R., Zdancewic, S.: QWIRE: a core language for quantum circuits. ACM SIGPLAN Notices **52**(1), 846–858 (2017)
34. Peduri, A., Bhat, S., Grosser, T.: QSSA: an SSA-based IR for Quantum computing. In: Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction. pp. 2–14 (2022)
35. Perdrix, S.: Quantum patterns and types for entanglement and separability. Electronic Notes in Theoretical Computer Science **170**, 125–138 (2007)

36. Perdrix, S.: Quantum entanglement analysis based on abstract interpretation. In: International Static Analysis Symposium. pp. 270–282. Springer (2008)
37. Quantinuum: guppylang (2024), https://github.com/CQCL/guppylang
38. Rand, R., Paykin, J., Lee, D.H., Zdancewic, S.: ReQWIRE: Reasoning about Reversible Quantum Circuits. Electronic Proceedings in Theoretical Computer Science **287**, 299–312 (jan 2019)
39. Seidel, R., Bock, S., Tcholtchev, N., Hauswirth, M.: Qrisp: A framework for compilable high-level programming of gate-based quantum computers. PlanQC-Programming Languages for Quantum Computing (2022)
40. Seidel, R., Tcholtchev, N., Bock, S., Hauswirth, M.: Uncomputation in the qrisp high-level quantum programming framework. In: International Conference on Reversible Computation. pp. 150–165. Springer (2023)
41. Seidl, H., Wilhelm, R., Hack, S.: Compiler Design: Analysis and Transformation. Springer (2012)
42. Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M.: Q#: Enabling Scalable Quantum Computing and Development with a High-Level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop 2018. RWDSL2018, Association for Computing Machinery, New York, NY, USA (2018)
43. Voichick, F., Li, L., Rand, R., Hicks, M.: Qunity: A Unified Language for Quantum and Classical Computing. Proceedings of the ACM on Programming Languages **7**(POPL), 921–951 (2023)
44. Winskel, G.: The formal semantics of programming languages: an introduction. MIT press (1993)
45. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE symposium on security and privacy. pp. 590–604. IEEE (2014)
46. Yu, N., Palsberg, J.: Quantum abstract interpretation. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. pp. 542–558 (2021)
47. Zhao, P., Wu, X., Li, Z., Zhao, J.: Qchecker: Detecting bugs in quantum programs via static analysis. In: 2023 IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE). pp. 50–57. IEEE (2023)