International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-026-00845-1

GENERAL

Special Issue CSV 2025

Challenges in Quantum Programs Analysis

Nicola Assolini[1] · Alessandra Di Pierro[1] · Isabella Mastroeni[1]

Accepted: 20 February 2026
© The Author(s) 2026

Abstract
The rapid progress of quantum technologies, fostered by the efforts of both academia and industry, has stimulated the design of quantum programming languages and the development of methods to support their verification and optimization. As in the classical setting, static analysis plays a fundamental role in such an endeavour. In this paper, we provide a survey on static analysis approaches for quantum programs, which have been proposed in the literature, distinguishing between dataflow-oriented approaches, which are based on a graph representation of the program information flow, and domain-oriented approaches, which essentially consist of the definition of some appropriate abstract domains representing the program property to be analysed. To illustrate these two perspectives concretely, we also present in detail two specific analyses: a dataflow analysis for managing quantum variables and uncomputation, and a static analysis based on abstract interpretation for detecting state entanglement.

Keywords Quantum Programming Languages · Survey · Abstract Interpretation · Static Analysis · Formal Verification

1 Introduction

After the initial explorations back in 1997 [56], the field of quantum technologies has known a rapid progress recently boosted by several investments in the public (national governments, the European Union, etc.) as well as the private sector (large companies such as DWave, IBM, and Google). The intensive research activities and the efforts in building an effective quantum computer justify the common feeling that we are now living in a quantum era. In this day and age, learning how to program quantum computers is the next critical skill for software developers. Despite this, current quantum programming tools are still at an early stage of the development process, which limits the possibility of creating real complex quantum software systems. Besides the high-level definition of programming languages via a machine-independent specification of their syntax and semantics, and a structured methodology for building quantum compilers, one of the most urgent needs for quantum software is the development of formal methods for the analysis of program properties and for compiler optimisation. These aspects are all very well developed in the area of classical programming languages and implementation, and we can take advantage of the skills acquired in the process of developing such advanced classical tools for devising an equally advanced quantum software. However, quantum-specific features, such as superposition, entanglement, the no-cloning theorem [58, Chap. 12] and implicit measurement [58, Chap. 4] pose unique challenges for reasoning about quantum programs, which makes a straightforward application of classical approaches not viable.

Abstract interpretation [34], originally introduced for the static analysis of classical programs, provides a powerful and general framework to approximate program semantics. In the context of quantum programming, this methodology has recently attracted growing interest [10, 46, 69, 73] because of its dual applicability. On one hand, we can analyse and infer information on the use of quantum variables in analogy with classical settings; this perspective highlights issues such as variable usage, duplication, and discarding, which are central in quantum languages [16, 54, 66]. We call these approaches flow-based analysis, since they are based on a graph representation of the program information flow (or control flow graph). On the other hand, static analysis can be leveraged to approximate semantic properties of quantum programs, by defining abstractions of quantum states and their correlations; this kind of approach, which we call domain-based

✉ N. Assolini
nicola.assolini@univr.it

A. Di Pierro
alessandra.dipierro@univr.it

I. Mastroeni
isabella.mastroeni@univr.it

[1] Dipartimento di Informatica, Università di Verona, Verona, Italy

Published online: 10 March 2026

anaysis, allows the investigation of non-trivial features of program execution, such as entanglement [10, 69], while coping with the exponential growth of the underlying Hilbert space. Even in the case of terminating programs, such as quantum circuits, approximating exponentially large states remains a fundamental challenge.

In this article, we provide a survey of the literature on static analyses for quantum programs with the aim of systematising the existing landscape, highlighting commonalities, differences, and open challenges (Sect. 3). In addition, to concretely illustrate the two classes of flow-based analysis and domain-based approaches, we present in detail two applications: in Sect. 4 we show a flow-based analysis to automatically manage the usage of quantum variables and uncomputation, firstly introduced in [11], and in Sect. 5 we illustrate a domain-based analysis for the entanglement property of quantum states, introduced in [10, 12].

2 Quantum computation

In this section, we briefly recall the main aspects of quantum computation with particular attention to the entanglement phenomenon, which will be the subject of our case studies. In doing so, we will refer to the circuit model of computation. In a quantum circuit, wires represent quantum bits, or qubits, rather than bits. Thus, a qubit replaces the classical unit of information (the bit) in the quantum computation model, generalising the two only possible values 0 and 1 of a bit to any vector in a complex Hilbert space (the quantum system), with 0 and 1 as basis vectors.[1] The typical notation of such vectors (or states of a qubit) is the Dirac *ket* notation, according to which $|0\rangle = (1,0)^T$ and $|1\rangle = (0,1)^T$ indicate the basis states 0 and 1; in general, $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ denotes a linear combination of $|0\rangle$ and $|1\rangle$ or *superposition* state. The numbers α and β are complex numbers called probability amplitudes since, from them, we can infer the probability of the state resulting in 0 or 1 after measuring the system. 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.

Implementing significant and powerful quantum algorithms requires performing quantum computation on circuits that are more complex than a single qubit operation and involve n qubit states with $n > 1$. A n qubit state corresponds to a unitary vector in the 2^n-dimensional Hilbert space ($\mathcal{H}^{2^n}$), obtained by composing by tensor product ($\otimes$) the vector space of the single qubits, each living in a 2-dimensional complex Hilbert space ($\mathcal{H}^2$) [58, Chap. 2]. For instance, the space of two qubits is $\mathcal{H}^4 = \mathcal{H}^2 \otimes \mathcal{H}^2$ and a generic state $|\psi\rangle$ in $\mathcal{H}^4$ can be written as $|\psi\rangle = \alpha_0\,|00\rangle + \alpha_1\,|01\rangle + \alpha_2\,|10\rangle + \alpha_3\,|11\rangle$ where all α_i are complex numbers.

Measurement Quantum measurement is an operation that allows us to extract a classical result from a quantum superposition $|\psi\rangle$. This operation transforms the quantum state into a classical one by breaking the quantum coherence (and so the quantum nature) of the state. Therefore, measurement is typically applied as the last operation in a quantum circuit to get the final (classical and probabilistic) result of the coherent (i.e., in superposition) evolution of the quantum system represented by the circuit. Formally, quantum measurement on the state space of the quantum system is represented using measurement operators $\{M_m\}_m$, where m corresponds to each of the possible outcomes of the measurement. If the system is in the quantum state $|\psi\rangle$ before the measurement, the probability of obtaining outcome m is given by $p(m) = \|M_m\,|\psi\rangle\,\|^2$, where $\|\cdot\|$ is the vector norm,[2] and the system state after the measurement is $\frac{M_m|\psi\rangle}{\sqrt{p(m)}}$. For instance, given one qubit, the measurement operators are $M_0 = |0\rangle\langle 0|$ and $M_1 = |1\rangle\langle 1|$, corresponding to the outcomes 0 and 1. If the state of the qubit before the measurement is $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, the probability of measuring 0 is: $p(0) = \|M_0\psi\|^2 = |\alpha|^2$ and the probability of measuring 1 is $p(1) = \|M_1\psi\|^2 = |\beta|^2$. After the measurement, if outcome 0 is observed, the state collapses to $\frac{M_0|\psi\rangle}{|\alpha|} = |0\rangle$ while if outcome 1 is observed, the state collapses to $\frac{M_1|\psi\rangle}{|\beta|} = |1\rangle$.

Entanglement The behaviour of quantum circuits is determined by the laws of quantum mechanics and undergoes the effect of an important quantum phenomenon with no classical counterparts, namely *entanglement*. This can be intuitively described as an application of the superposition principle to a system composed of two or more subsystems. Concretely, the term entanglement describes a situation in which two particles, designated as x and y, which form a composite system, become strongly correlated. This occurs as a result of a computational process that generates a superposition of product states for both particles. This superposition implies that the state of the composite system cannot be described without considering the other particle's state. Consequently, if measurements are made on an entangled state $ab + cd$, where a and c are two possible states of x and b and d are two possible states of y, then if x is found in state a, y must be in state b; similarly if x is found in state c, y must be in state d. As an example, the state $1/\sqrt{2}(|00\rangle + |11\rangle)$ in the Hilbert space $\mathcal{H}^2 \otimes \mathcal{H}^2$ is entangled because it cannot be expressed as a tensor product of the

[1] Any pair of orthonormal vectors can be assumed as a basis for the space of 1 qubit states.

[2] We refer here to the Hilbert space vector norm defined as $\|\,|\psi\rangle\,\| = \sqrt{\langle\psi|\psi\rangle}$, where $\langle\psi|$ is the conjugate transpose of $|\psi\rangle$ and $\langle x|y\rangle$ is the inner product between vector $|x\rangle$ and vector $|y\rangle$.

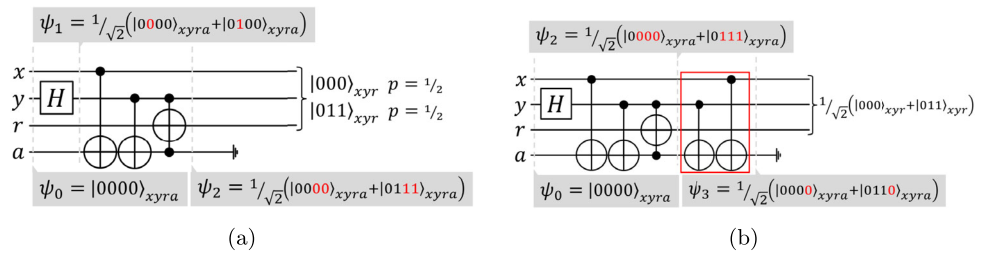


Fig. 1 A circuit implementing $(x \oplus y) \wedge y$ without uncomputation (a) and the same circuit with uncomputation (b). The result of circuit (a) may be incorrect with half probability, while in (b) we always obtain the correct result

individual states of the two-component qubits. In this state, if one qubit is measured and found to be in the state $|0\rangle$, the other qubit will instantaneously collapse to the state $|0\rangle$ as well, and similarly for the state $|1\rangle$. In some cases, measuring a qubit of an entangled pair alters the other, keeping it in a quantum state. For instance, consider the entangled state $1/2(|00\rangle + |01\rangle + |10\rangle - |11\rangle)$. If the first qubit is measured and found in the state $|0\rangle$, the other qubit will instantaneously collapse to the state $1/\sqrt{2}(|0\rangle + |1\rangle)$ and, similarly, if the first qubit after the measurement collapses to $|1\rangle$, then the other one will be in state $1/\sqrt{2}(|0\rangle - |1\rangle)$.

2.1 Quantum variables

A quantum variable is the high-level abstraction of the state of a quantum register. Its type is the Hilbert space to which those states belong, namely, a 2^n-dimensional Hilbert space for a n-qubit quantum register. Thus, the abstraction of a qubit is a quantum variable q of type the two-dimensional complex Hilbert space $\mathcal{H}_q$. Following [91], we construct the space of values for a set $\mathbb{V} = \{q_i\}_i$ of quantum variables as the Hilbert space $\mathcal{H}_{\mathbb{V}} = \bigotimes_i \mathcal{H}_{q_i}$ obtained by composing via tensor product the Hilbert space associated to each variable q_i. Given a quantum program characterised by a set Q of quantum variables, we say that the Hilbert space $\mathcal{H}_Q$ is the program space, and the semantics of the program can be described using operators on the space $\mathcal{H}_Q$.

We write $|\psi\rangle_{q_i}$ to indicate that q_i represents the state $|\psi\rangle$ in $\mathcal{H}_{q_i}$. For entangled states, such as for example $1/\sqrt{2}(|01\rangle + |10\rangle)$, we write $(1/\sqrt{2}(|01\rangle + |10\rangle))_{p,q}$ to indicate that p and q represent, respectively, the first and the second variable of the entangled pair. In this case, the vector represents an inseparable state in the space $\mathcal{H}_p \otimes \mathcal{H}_q$. Given an operator U on the Hilbert space $\mathcal{H}_Q$ and a set of variables $\mathbb{V} \subseteq Q$, we write $U_{\mathbb{V}}$ to indicate that the operator acts as U on the variables in $\mathbb{V}$ and as the identity on the other variables of Q. For instance, H_q is the unitary operator on $\mathcal{H}_Q$ that acts as the Hadamard gate on (the type of) q and as the identity on the other variables. In the same way, $CX_{p,q}$ is the operator corresponding to the control-NOT (CNOT) operator on p and q and the identity on the other variables.

2.2 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 [58, Chap. 1.3]). However, this feature introduces some complications when it comes to the implementation of quantum programming languages due to the 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 [52, Chap. 7]. It turns out that when feeding the circuit with a superposition of input states, the auxiliary qubits may 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 'deleting', 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 [49, Chap. 5].

We illustrate the problem of uncomputation with the circuit in Fig. 1a. This circuit computes $(x \oplus y) \wedge y$, i.e., the conjunction of y and the exclusive-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 CNOT 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[3] 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 [58, Chap. 4] (deleting a qubit means

[3] The Toffoli gate corresponds to a double controlled NOT: the target is negated if both controllers are 1.

measuring it) and the entanglement introduced by the CNOT gate. This example clearly shows why it is necessary to reset all temporary variables, ensuring they are removed without introducing side effects. The simplest way to uncompute a qubit is to take all previously applied operations and reapply their inverses in reverse order. In Fig. 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 Static analysis of quantum programs

As in the classical case, analysing a quantum program requires different approaches depending on the property we want to analyse. Specifically, one can be interested in properties of the quantum program variables, which are detectable by statically analyzing the flow of execution within the program. Classically, this is the case of the typical dataflow analyses used for compiler optimization (e.g., constant propagation, reaching definitions, copy elimination, register allocation, etc.), which aim to check whether variables are properly used, to verify that operations are applied in the correct order, and to ensure that no unintended duplication or discarding of variables occurs. The quantum methodology in this case remains essentially the same as in the classical setting and consists of defining algorithms based on the execution flow structure of the program to extract information about variable properties. The key difference lies in the properties of interest, which are shaped by the features of quantum computation and are not shared by the classical paradigm. For example, duplication or discarding of variables is non-trivial in quantum languages, as it conflicts with principles such as the already mentioned no-cloning theorem [58, Chap. 12] and implicit measurement [58, Sect. 4.4]. It is therefore essential to develop analyses that can prevent or detect such incorrect behaviours. While the infrastructure of these analyses is classical, their necessity stems from quantum-specific constraints. In this paper, we refer to such approaches as *Flow-Based Analyses*.

On the other hand, one can also be interested in properties of the computational states, which are inherently quantum, such as entanglement. In this case, classical techniques cannot be directly reused, since reasoning requires the construction of abstract domains specifically designed to capture such phenomena. The difficulty is amplified by the exponential growth of the Hilbert space with the number of qubits, which makes exact state representations infeasible. Abstract interpretation [34] offers a suitable framework to address this issue: abstract domains can approximate quantum states, enabling program reasoning or abstract simulation with a controlled degree of precision. Designing such approaches, which we refer to as *Domain-Based Analyses*, is therefore a central challenge in quantum program analysis.

In this section, we first review the literature on flow-based analyses, highlighting how traditional techniques have been adapted to the quantum setting. We then turn to domain-based analyses, where abstract interpretation is employed to approximate quantum properties.

3.1 Flow-based analyses

These approaches are characterized by the application of classical techniques, such as data-flow analysis or type-checkers, to quantum programs.

3.1.1 Pattern based approaches

Pattern-based analyses have been developed to detect potential bugs in a quantum program. They focus on identifying problematic patterns or inappropriate uses of variables that could lead to unexpected behaviours. An example is the QChecker tool by Zhao et al. [96, 98], which analyses the abstract syntax tree (AST) of quantum programs in Qiskit, to detect common bug patterns. Inspired by the Bugs4Q dataset [97], QChecker identifies syntax-level issues such as the use of gates not supported by the backend, incorrect parameters in gate applications, misuse of measurements, inconsistencies between the number of qubits and registers, as well as various API and compilation errors. The tool focuses on local, syntax-driven problems, such as undefined or deprecated methods, invalid object calls, or redundant register declarations.

LintQ [61] takes a different approach by translating quantum programs into the CodeQL intermediate representation, thus allowing reasoning not only about syntax through the AST, but also about data-flow and control-flow. In this setting, the program is modeled as a database of facts encoding elements and their relationships, enabling both classical and quantum data-flow analyses. The latter is used to track the ordering of gate applications on qubits, a key aspect in detecting subtle bugs. Compared to other tools, LintQ recognizes the largest variety of patterns, drawing from both empirical studies and prior work. Moreover, it is the only approach that explicitly models complex circuits built from multiple subcircuits, which allows the detection of additional errors such as incorrect usage of the composed API. LintQ also captures data-flow–related issues (e.g., applying gates to qubits after measurement, or measuring the same qubit multiple times), problems with resource allocation (such as insufficient classical registers or unused qubits in a circuit), and violations of implicit API constraints (for example, composing circuits without using the composed result, or adding gates after transpilation).

Kaul et al. [50] propose Quantum-CPG, an extension of Code Property Graphs [88] to the quantum setting. Their approach models both data-flow and control-flow of programs written in Qiskit and OpenQASM, and—similarly to LintQ—captures the execution order of quantum gates through a graph-based representation of quantum data flow. Beyond detecting programming errors, Quantum-CPG also identifies classical code smells in hybrid quantum-classical programs. A distinctive feature of this work is the explicit modelling of the connection between qubits and classical registers, which makes it possible to spot unused or redundant measurement results. For instance, the tool can detect when a measurement result bit is produced but never consumed, when result bits remain constant due to missing transformations in the quantum circuit, or when conditional statements depend on constant classical values. Moreover, it highlights superfluous quantum operations, such as gates that have no observable effect on measurements.

3.1.2 Ensuring correctness

In quantum programming languages, variables represent information encoded in quantum states rather than in memory locations, as is the case for classical languages. While they offer similar abstraction benefits, quantum variables introduce unique challenges due to the physical principles they obey, i.e., the laws of quantum mechanics. Operations such as copying or assignment are of non-trivial realisation due to the no-cloning theorem, and unused quantum variables must be carefully handled to avoid implicit measurements that can affect entangled states.

High-level quantum languages such as QWIRE [66] and Guppy [54] are equipped with linear typing systems ensuring that values are used exactly once, preventing unintended duplication or deletion of resources.

Silq [16] introduces a more fine-grained type system that checks whether a variable is in superposition (for instance, after the application of quantum gates such as the Hadamard) and reduces errors related to implicit discarding. In particular, it can automatically infer safe discards when no uncomputation is required, or when the discarded state does not affect the computation. However, this approach relies on annotations and is effective mostly in the presence of local variables and simple control flow. Similar to Silq, Qurts [45] extends the Rust type system to achieve lifetime information about variables to support automatic uncomputation.

Building on this line of work, ReQWIRE [71] introduces a series of syntactic checks to verify the correctness of user-defined uncomputation, based solely on circuit-level information.

In [11], we propose a static analysis approach to automate uncomputation in quantum programs. Unlike type system-based approaches, our analysis reduces programmer workload and minimizes error messages by relaxing the requirement for variables to be used "exactly once" and allowing them to be used at most once. The analysis is based on two data-flow analyses: one to check variable usage constraints and another to detect unused variables across execution paths. This information enables the compiler to automatically insert discard functions, increasing flexibility and allowing integration with other analyses for improved accuracy. We present in detail this analysis in Sect. 4.

Rovara et al. [75] introduce a comprehensive debugging framework that integrates classical simulation with lightweight static analyses to assist developers in diagnosing faults in quantum programs. The framework employs a *cone-of-influence analysis* to identify the subset of instructions that may affect the qubits involved in a failed assertion, thereby reducing the search space for potential sources of error. Complementarily, an *interaction analysis* constructs an interaction graph for the qubits. This analysis is employed to check the entanglement assertion in the program. In fact, when two or more qubits appear in entanglement assertions without having interacted previously, it is a typical indication of misconfigured controlled operations.

3.1.3 Code optimization

Static analysis is also a useful tool for optimizing quantum code, as it helps identify substitution patterns or eliminate redundant gates.

Peduri et al. [67] introduce Quantum-SSA (QSSA), an SSA-based intermediate representation for quantum computing that positions itself between the works on optimization, as discussed in this section, and those on correctness checks, presented in the previous one. They formalize the semantics of QSSA for hybrid quantum–classical programs and propose a static no-cloning verification algorithm, quadratic on general control-flow graphs and linear on structured ones. Classical SSA analyses and transformations are adapted to preserve quantum invariants, enabling both redundancy elimination and peephole optimizations (e.g., CNOT elimination, unitary merging, and Pauli/Hadamard simplifications). QSSA can also leverage standard optimizations such as loop unrolling and dead-code elimination.

Ittah et al. [47] present QIRO, a quantum intermediate representation based on MLIR and static single assignment (SSA), designed to explicitly model the dataflow of quantum states. QIRO represents the flow of quantum states between gates, allowing the reuse of both classical optimization techniques (e.g., constant folding, common subexpression elimination, inlining, loop unrolling, and dead-code elimination) and quantum-specific ones (e.g., unitary gate cancellation, rotation merging, and loop-boundary optimizations). The SSA structure enables optimization of side-effect-free quantum operations in the same way as classical ones, while

preserving quantum invariants. QIRO also supports partial lowering and decomposition of high-level quantum operations, as well as resource estimation by lowering gates to classical counters.

Chen et al. in [28] introduce a circuit optimization technique that exploits context information from the classical part of hybrid quantum algorithms. The central observation is that many measurement outcomes in hybrid workflows are never used in the subsequent classical computation. By formally distinguishing between valid qubits (those whose measurement outcomes are actually consumed) and dead qubits (those whose outcomes are discarded), they design an elimination algorithm that iteratively scans the circuit and removes dead gates without altering the probability distribution of valid outcomes. This approach, inspired by classical liveness analysis, adapts the idea to the setting of hybrid quantum–classical computation.

Behler et al. [14] present QStatic, a tool for static analysis and refactoring quantum programs. The work exploits the srcML [30, 31] infrastructure to support both classical and quantum constructs in Abstract Syntax Trees (ASTs). QStatic implements automated refactoring rules for common quantum patterns, including iteration restructuring, Hadamard gate reduction, and code encapsulation into custom gates, improving both code readability and efficiency.

3.1.4 Resource estimation

A static analysis of quantum code can also be useful for estimating the resources required by a quantum program or circuit.

ScaffCC [48] introduces timing and resource analysis. Resource analysis estimates the number of qubits and gates required, providing early feedback on algorithm efficiency before execution on hardware. Timing analysis estimates the circuit's critical path length under the assumption of unbounded resources, respecting quantum data dependencies enforced by the no-cloning theorem.

Colledan and Dal Lago [32] introduce a flexible type-based approach for quantum resource estimation in Quipper [43]. They define a family of type systems, Proto-Quipper-RA, to derivie upper bounds on circuit size according to various metrics (e.g., width, depth, gate count). The system combines refinement types for local wire-level metrics and effects for global circuit metrics, using arithmetic index terms to express bounds parametrically. The framework is provably correct with respect to the chosen metric and is implemented in the QuRA tool.

3.2 Domain-based analyses

We now turn our attention to the problem of analysing properties of quantum states. Broadly speaking, we can classify the approaches to this problem into three categories. The first category targets a specific property of quantum computation, namely entanglement, and develops abstract domains. The second category introduces general-purpose abstractions of quantum computation, aiming to represent quantum states (or families of states) efficiently, without explicitly dealing with their exponentially growing size. Finally, the third category exploits abstraction to optimize quantum circuits.

3.2.1 Entanglement analyses

The focus of such analyses is on providing a sound representation of entanglement in quantum programs: their goal is to determine, at a given program point, whether two variables are entangled or not. To ensure soundness, these approaches over-approximate the entanglement property: if the analysis reports that two variables are not entangled, this is guaranteed to be correct; however, some variables may be conservatively considered entangled even if they are not.

ScaffCC [48] introduced the first static analysis of entanglement. The approach assumes that any two-qubit gate may create entanglement and records pairs of control and target qubits, together with a timestamp. When the same gate is encountered again, the analysis checks if control qubits have changed state in the meantime (i.e., served as targets of other gates): if not, the second gate is recognized as the inverse of the first, and disentanglement is recorded; otherwise, both entanglements are preserved. As a result, the tool can identify when ancilla qubits remain entangled at the end of a module, issuing a *disentangled qubit check* warning whenever qubits are not properly uncomputed, which could otherwise lead to incorrect outputs.

In [69], Perdrix introduces the first entanglement analysis based on abstract interpretation. The analysis provides a sound approximation of the property. The domain combines state value information and entanglement. It is composed of a map from the qubit to one of four values: $\bot$ (both bases), s (standard), d (diagonal), or $\top$ (none), where s states that the qubit is in a standard basis, d that it is in a diagonal basis. Entanglement is represented by a partition of qubits: qubits in the same set may be entangled, and qubits in different sets are separable. The target is a small imperative language with qubits as memory, supporting sequential composition, conditionals, loops, and a universal set of gates (Pauli, Hadamard, Phase, CNot). Conditional and loop constructs are typically treated by using measurements. Single-qubit gates update basis info, while two-qubit gates may merge blocks to approximate entanglement. This approach is efficient, but it lacks the ability to detect when entanglement is nullified.

Honda [46] extends the stabiliser formalism to handle the uncertainty introduced by non-Clifford gates such as the T gate. The key idea is that the effect of a non-Clifford gate can be approximated locally and, when possible, neutralised later

in the computation. In general, abstract analyses of quantum programs use stabiliser-based representations [1] to track entanglement and basis information. When a non-Clifford gate is applied, the exact stabiliser information is lost, but the effect can be conservatively approximated, allowing the analysis to continue soundly while retaining as much information as possible. Honda's work targets a similar language to Perdrix's previous work. Conditional and Loop operations are performed via join operations, which combine abstract states from different branches, ensuring a safe approximation of entanglement evolution. This approach provides a sound method to track entanglement, which is more precise than the previous one. However, it deals with non-stabilizer states simply by treating them as unknown, which could propagate imprecision throughout the program. This could be useful when a non-stabilizer state is quickly discarded, but generally meant the system could not meaningfully speak about non-Clifford circuits.

In [72, 73], Rand et al. introduce a type system based on Gottesman's [41] representation of Clifford gates (H, S, CX). The key idea is to type states during simulation—represented using stabilizers—so that separability can be checked efficiently. This approach proposes an entanglement analysis, although working at the circuit level (i.e., no control flow) and limited to Clifford gates and measurements.

The design of the language Twist [94] includes the verification of the separability of quantum states, based on combining a type system with manual annotations and runtime dynamic checking based on classical simulation. Twist is a functional quantum language including classical Booleans, pairs, functions, qubits, measurement `let` and `if`-expressions. The key idea of Twist is the notion of a *pure expression*, that is, an expression whose evaluation is unaffected by the measurement outcomes of any qubit that does not belong to it, ensuring that its qubits remain separable from the rest of the program. Twist introduces a sound purity type system that conservatively tracks entanglement, allowing programmers to explicitly assert the separability of expressions or components of entangled pairs, along with a hybrid verification approach that combines static analysis with runtime checks based on Schmidt decomposition. In Twist, the static analysis is based on a type system, and it performs simple checks. Its precision is similar to the one introduced in [69]. The dynamic check is primarily used to verify assertions where the entanglement is nullified during computation.

In [87], Xia et al. present the first static entanglement analysis method for the quantum programming language Q# [79]. The analysis is built on an interprocedural control flow graph and tracks the operations applied to each variable in order to determine whether it represents a classical state or a superposition. By doing so, it identifies both the creation and the cancellation of entanglement. The approach is illustrated, focusing on Hadamard, single-qubit rotation, and CNOT operations.

Finally, our work [10, 12] introduces a static analysis based on a new abstract domain for entanglement. This domain not only determines the sets of entangled variables, but also distinguishes a particular relation, which we call *direct inseparability*, corresponding to entangled states of the form $\alpha\,|00\ldots0\rangle + \beta\,|11\ldots1\rangle$. These states are particularly interesting because they can be disentangled in a simple way. Moreover, the domain incorporates labels that abstract the quantum states of variables. These labels enable a static analysis that not only identifies entangled variables but also approximates their quantum state. Our approach improves the accuracy of existing methods that rely on a more imprecise domain, such as the analysis in [69], while targeting a basic imperative quantum language equipped with a universal gate set (CNOT, Hadamard, and Phase gates) and measurement. We will discuss the details of our analysis in Sect. 5. An ML implementation of our domain is presented in [70].

3.2.2 Abstracting quantum states

The approaches described here do not introduce an abstract domain with the goal of capturing a specific property (such as entanglement, as discussed before). Instead, they focus on representing the overall state of the computation for verification or simulation purposes.

Yu et al. [92] introduce a method, called quantum abstract interpretation (QAI), to simplify the analysis of quantum programs by reducing the dimensionality of the quantum state. Instead of considering all n qubits at once, their approach partitions the system into smaller groups of qubits, each represented by a lower-dimensional abstraction. This allows for efficient static verification of certain properties and assertions in quantum circuits without simulating the entire state. The approach uses algebraic operations to combine and compare these smaller abstractions, ensuring that the results still soundly approximate the behavior of the full system. It can determine whether specific states belong to a subspace, which is described as a span of standard bases. However, the abstraction loses precision because it breaks global correlations among qubits into simpler relations between subsets of them. Building on this abstraction, recent work [93] introduced the logical framework *SAQR-QC*, which combines *Quantum Hoare Logic* [90] with the framework of QAI. In particular, it enables performing QHL-style reasoning about program behavior, utilizing QAI approximation to ensure scalability.

In [2, 23, 24, 26], the authors introduce a verification methodology for quantum circuits and programs based primarily on tree automata. Tree automata (TAs) provide a formal model for representing sets of trees and, in the context of quantum verification, are used to compactly encode sets

of quantum states. In the original AutoQ framework [23], TAs were employed to efficiently represent pre- and post-conditions of quantum circuits. The semantics of quantum gates are implemented in the TA domain, and verification is reduced to checking whether the output-state TA is included in the post-condition TA. In [24], the framework was extended to use symbolic TAs, introducing symbolic variables in the leaves of the automata to enable reasoning about relational properties between quantum states. In [2], the authors introduced Level-Synchronized Tree Automata (LSTAs) to offer a more compact symbolic representation that allows verification of quantum circuits with at most quadratic complexity per gate. Moreover, LSTAs support parameterized verification, making it possible to analyze entire families of circuits with arbitrary numbers of qubits. Finally, AutoQ 2.0 [26] generalizes the framework from circuits to full quantum programs, incorporating classical control structures such as branches and loops, while retaining the symbolic and scalable verification capabilities enabled by TAs and LSTAs.

The path-sum introduced in [5], provides a symbolic and compositional representation of quantum circuits, expressing their behavior as a sum over all computational paths. Formally, a circuit acting on input variables x can be represented as $|x\rangle \mapsto \frac{1}{\sqrt{2^m}} \sum_{y \in 0,1^m} e^{i\pi P(x,y)} \, |f(x,y)\rangle$, where y enumerates the path variables, $P(x, y)$ encodes the phase accumulated along each path, and $f(x, y)$ describes the corresponding output transformation. This representation enables compositional reasoning, allowing equivalence proofs of large circuits to be constructed modularly, without explicitly manipulating exponentially large state vectors or matrices. Path-sums have been effectively employed for the automated equivalence checking of Clifford+T circuits. However, the original formalism is limited to fixed-size circuits and cannot directly capture parametrized circuit-generating programs. The Qbricks tool [21] extends the path-sum semantics to support verification of parametric quantum programs, i.e. quantum programs with a parametric number of qubits.

Hietala et al. [44] introduce *VOQC*, the first fully verified optimizer for quantum circuits. Their work relies on a symbolic matrix semantics that avoids the exponential blow-up of concrete $2^n \times 2^n$ matrices by manipulating algebraic expressions denoting the underlying linear operators. This approach enables correctness proofs for optimizations that apply to circuits with a parametric number of qubits. Circuit rewrites are formalized as equivalence proofs between symbolic matrix expressions, supported by Coq tactics such as `gridify` [57] for normalizing tensor products and matrix multiplications.

Bichsel et al. [17] present *Abstraqt*, an abstract stabilizer simulator for efficiently analysing quantum circuits, including those with non-Clifford gates. It overcomes the exponential growth of summands in exact simulation by using abstract interpretation to compress multiple concrete states into a single abstract representation, trading some precision for efficiency. Quantum states are represented with Pauli operators and complex intervals, and Abstraqt provides abstract domains and transformers to handle gates and measurements, enabling static analysis and verification of circuits that are infeasible to simulate exactly.

In [13], we have applied abstract interpretation to analyze Variational Quantum Circuits (VQCs). VQCs are hybrid quantum-classical models in which the circuit is characterized by a set of trainable parameters, optimized during the learning process, similarly to neural networks. In this work, we extend the classical interval abstract domain [35] to the quantum setting, representing the quantum state as a vector of intervals. This representation allows us to model noisy inputs to the VQC and analyze the circuit robustness against noise and adversarial perturbations.

3.2.3 Abstraction for optimization

Finally, we review a set of works where abstraction is employed to optimise quantum programs. Unlike the approaches discussed in Sect. 3.1.3, which primarily rely on data-flow information (such as liveness), these methods exploit semantic properties—namely, approximations of the program state—to guide the optimisation process.

In [55], Liu et al. introduce the Relaxed Peephole Optimization (RPO). This approach tries to statically determine the state of the qubits to reduce the number of gates. They use labels to indicate that a qubit is in one of the X-, Y-, or Z-basis states together with a $\top$ label and compute the approximation of the execution on these labels. Based on this information, they replace expensive operations with equivalent but less costly ones.

In [22], Chen et al. introduce an optimization technique, called Quantum Constant Propagation, that exploits partial circuit execution and removes superfluous controlled gates, leveraging the common assumption that all qubits are initially in the $|0\rangle$ state. The key idea is to keep non-entangled groups of qubits separated for as long as possible and to describe their state in a compact form. To avoid excessive complexity, a bound is imposed on the size of the entangled state that can be represented; once this bound is exceeded, the state of the group is abstracted into a form that signals a loss of information. In [27] based on the description of mid-circuit measurement presented in [25] the Quantum Constant Propagation is extended to such circuits.

Amy et al. [7] generalise the phase folding optimisation [8], traditionally used to reduce costly T-gates in straight-line quantum circuits, so that it can also be applied to quantum programs with classical control flow. The key idea is to reinterpret phase folding as a form of relational analysis, in particular, affine relation analysis. They also incorporate a non-linear analysis, which makes it possible to optimise

programs involving operations such as Toffoli gates, whose effect corresponds to non-linear transformations on classical states. To address precision losses caused by interference when composing transition relations, the authors introduce a sum-over-paths technique [5]. This work also shows that classical program analysis can be used on classical data in superposition, making classical techniques feasible for quantum programs.

4 A flow based analysis: preventing implicit discarding

Almost all high-level programming languages provide variables as 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. 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 [58, Chap. 12]. 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 [58, Sect. 4.4]). In fact, as explained in Sect. 2.2, unused quantum variables correspond to portions of the quantum circuit that would be measured even if no explicit measurement operator is applied, and this may have unexpected side effects when it involves entangled states. 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. Thus, 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 [54], a Python-embedded language, which we take as a model of a quantum language with linear typing. A program like the one in Fig. 2 does not pass the check of a linear type checker since the variable a is used twice.

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

Fig. 2 Simple function that uses a consumed variable

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 Fig. 1 in Guppy, given in Fig. 3. The Guppy compiler would reject the program in Fig. 3a since a is not consumed. Instead, the code in Fig. 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 Fig. 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 Fig. 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 Fig. 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. In Fig. 5, we represent the complete pipeline for a given program represented as a Control Flow Graph (CFG). The information resulting from the first analysis allows the compiler to verify, in place of the type checker, that the programs use 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 Fig. 2 is still rejected, but functions in Fig. 3a and Fig. 4a are automatically transformed at compile time, respectively, into the ones in Fig. 3b and Fig. 4b.

Fig. 3 Two simple guppy programs with (a) and without (b) safe discarding

```python
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

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

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

(a) Wrong re-definition

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

(b) Proper re-definition

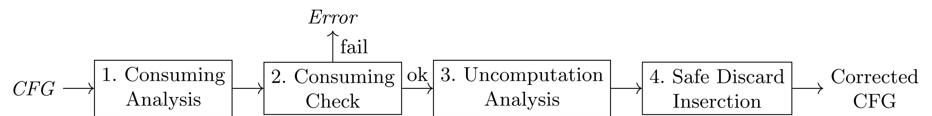


Fig. 5 Analysis pipeline

4.1 Control flow graph language

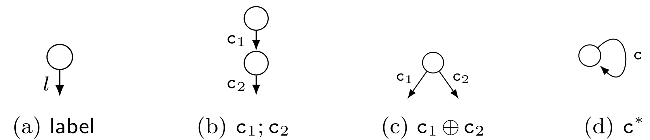
Static analysis is usually performed using the Control Flow Graph (CFG) representation of programs [78]. We define our static analyses as data-flow analyses based on control flow graphs. Following [19, 60], we consider the CFG language defined by the syntax:

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

where the term $\mathsf{c};\mathsf{c}$ represents sequential composition; the term $\mathsf{c} \oplus \mathsf{c}$ is the choice command that corresponds to the execution of one of the two possible branches; the term c^* is the Kleene closure of c^n, $n \in \mathbb{N}$, where c^n is the composition $\mathsf{c};\ldots\mathsf{c};$, n times. For a Boolean expression b and a statement stm (both defined by the grammar of a specific language), NonZero(b) is a special label that indicates that in its path b is true. In contrast, Zero(b) indicates that the Boolean condition b is false in its path. This syntax is general enough to cover deterministic imperative languages [85, Chap. 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 `ifthenelse` statements as follows:

$$\begin{aligned} \mathtt{if}\,\mathsf{b}\,\mathtt{then}\,\mathsf{c}_1\,\mathtt{else}\,\mathsf{c}_2 &\equiv (\mathtt{NonZero}(\mathsf{b});\mathsf{c}_1) \oplus (\mathtt{Zero}(\mathsf{b});\mathsf{c}_2) \\ \mathtt{while}\,\mathsf{b}\,\mathtt{do}\,\mathsf{c} &\equiv (\mathtt{NonZero}(\mathsf{b});\mathsf{c})^*;\mathtt{Zero}(\mathsf{b}) \end{aligned} \tag{2}$$

The language defined by Equation (1) is interesting because it corresponds precisely to the control flow graph representation of programs, on which standard data-flow analysis is



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

Fig. 6 Graphical representation of language c, the CFG

usually performed [78]. The CFG associated with a program is a graph with a start node corresponding to the program entry point, an end node 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 Fig. 6a where $l \in$ label, while the other elements of c determine how we compose the edges depending on their labels as displayed in Fig. 6b for the CFG corresponding to the sequential composition $\mathsf{c};\mathsf{c}$, in Fig. 6c for the nondeterministic choice $\mathsf{c} \oplus \mathsf{c}$ and in Fig. 6d for the iteration 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 $V \times \mathsf{label} \times V$.

We will demonstrate our approach on the Guppy language [54]. 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. We can extend the label syntax to represent a control flow graph of the 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:

$$\begin{aligned} \mathsf{stm} \ ::= &\ \mathtt{pass} \mid \mathtt{A}\!:\vec{v} \mid \vec{v} \\ &= \mathsf{fun}(\vec{v}) \mid \vec{x} = \mathtt{e} \mid \mathtt{return}\,\vec{v} \mid \mathtt{discard}(\vec{q}) \end{aligned} \tag{3}$$

where pass is the Python statement that corresponds to skip, $\mathtt{A}\!:\vec{v}$ declares which variables are the function arguments and fun indicates both built-in function (measure, quantum gates and initialization function qubit()) and user-defined functions.

4.2 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.

4.2.1 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 [78], 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 e, we denote by $Q(\mathtt{e}) \subseteq \mathbb{V}_q$ the set of quantum variables in 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 $l \in$ label, as follows:

$$\begin{aligned} (\!|\mathtt{NonZero(b)}|\!)\mathcal{D} &= (\!|\mathtt{Zero(b)}|\!)\mathcal{D} = \mathcal{D} \smallsetminus Q(\mathtt{b}) \\ (\!|\mathtt{pass}|\!)\mathcal{D} &= (\!|\vec{x} = \mathtt{e}|\!)\mathcal{D} = \mathcal{D} \\ (\!|\mathtt{return}\,\vec{v}|\!)\mathcal{D} &= \mathcal{D} \smallsetminus Q(\vec{v}) \\ (\!|\vec{v}_1 = \mathsf{fun}(\vec{v}_2)|\!)\mathcal{D} &= (\mathcal{D} \smallsetminus Q(\vec{v}_2)) \cup Q(\vec{v}_1) \\ (\!|\mathtt{A}\!:\vec{v}|\!)\mathcal{D} &= \mathcal{D} \cup Q(\vec{v}) \\ (\!|\mathtt{discard}(\vec{q})|\!)\mathcal{D} &= \mathcal{D} \smallsetminus \{\vec{q}\} \end{aligned} \tag{4}$$

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 Equation (4) 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. [3, 59, 78]). □

To compute the CFG analysis, we need to compute the set $\mathcal{D}$ for each node of the CFG [78], 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 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 [78]. As shown in [78], 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

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

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; in other words, our analysis is sound.

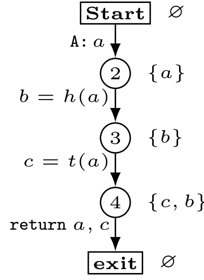


Fig. 7 CFG of program in Fig. 2 with the computed $\mathcal{D}$ for each node

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 already been 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 Fig. 2.
This analysis shows that in node 3, the variable a is not available; in fact, the edge $(3, 4)$ leads to an error.

4.2.2 Uncomputation analysis

The uncomputation analysis determines the appropriate locations where to insert the discard function. To this purpose, we must identify those unused variables that 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) [3, 59]. 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

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

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

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

where Δ is the set-theoretic symmetric difference.[4]

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.

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:

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

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 $\mathcal{S}$

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

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:

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

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

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

□

To compute the analysis on the CFG, we need to compute the pair $(\mathcal{S}, \mathcal{U})$ for each node of the CFG [78]. 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:

$$(\mathcal{S}, \mathcal{U})[u] \begin{cases} (\varnothing, \varnothing) & \text{if } u = \textbf{end} \\ \bigsqcup \left\{ (\!|l|\!)((\mathcal{S}, \mathcal{U})[v]) \,\middle|\, (u, l, v) \in G \right\} & \text{otherwise} \end{cases} \tag{5}$$

As we did for the consuming analysis, this system can be solved by fix-point, also obtaining, in this case, the so-called MFP solution [78]. Due to Proposition 3 [3, Chap. 9] [59, Chap. 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 $(\mathcal{S}, \mathcal{U}) = (\mathcal{S}, \mathcal{U})[u]$.

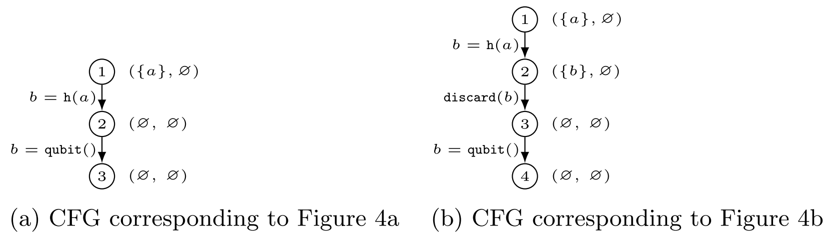
Proposition 4
For each program point u, *if* x *is unsafe live in* u, *then* $x \in \mathcal{U}$, *with* $(\mathcal{S}, \mathcal{U}) = (\mathcal{S}, \mathcal{U})[u]$.

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 Fig. 4a that we represent as a CFG accompanied by the analysis results depicting $(\mathcal{S}, \mathcal{U})$ for each node in Fig. 8a. In Fig. 8a, where the `discard` is needed, the variable b is not live. Instead, if we analyse the correct versions of the program (Fig. 4b and Fig. 8b), we see that after the definition, the variable b is live, thanks to the discard function.

Fig. 8 In both CFGs, on the right, the pairs $(\mathcal{S}, \mathcal{U})$



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

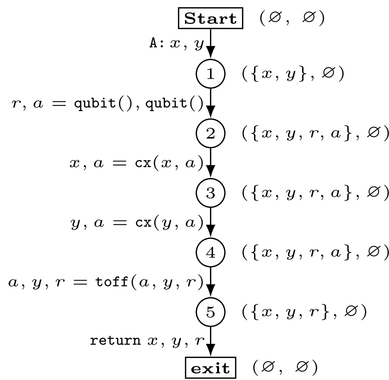


Fig. 9 CFG corresponding to Fig. 3a, on the rights the pairs $(\mathcal{S}, \mathcal{U})$

Consider the code in Fig. 3a. Fig. 9 shows the CFG corresponding to the program and the analysis results. After node 4, the variable a is no longer live; thus, in Fig. 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 Fig. 10a. 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 Fig. 10b, due to the lub operator, our analysis inserts both a and b in the set $\mathcal{U}[1]$.

4.3 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 Fig. 5 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.
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.

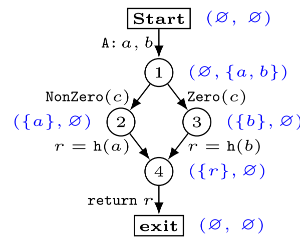
The first and the third steps are precisely the analyses described in the previous section. What we need now to define precisely are the procedures for performing the second and fourth steps of the pipeline.

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 in the program. For instance, the example in Fig. 7 shows that the check fails when considering the edge $(2, b = \mathsf{h}(a), 3)$ since a is not in $\mathcal{D}[2]$.

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

```python
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. 10 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

Algorithm 1 Check for Variable Usage

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

Algorithm 2 Insert discard

Require: $edges, nodes, (\mathcal{S}, \mathcal{U}), \mathbb{V}_q$
1: **for** var **in** $\mathbb{V}_q$ **do**
2: $\quad$ **for** u, v **in** GETALLDEFINITION(cfg, var) **do**
3: $\quad\quad$ **if** $var \notin (\mathcal{S}[v] \cup \mathcal{U}[v])$ **then**
4: $\quad\quad\quad$ ADDDISCARD$((u, v), var)$
5: $\quad\quad$ **end if**
6: $\quad$ **end for**
7: **end for**
8: **for** var **in** $\bigcup_u \mathcal{U}[u]$ **do**
9: $\quad$ **for** $node$ **in** $nodes$ **do**
10: $\quad\quad$ **if** $var \in \mathcal{U}[node]$ **then**
11: $\quad\quad\quad$ **for** v **in** SUCCESSOR$(node)$ **do**
12: $\quad\quad\quad\quad$ **if** $var \notin (\mathcal{S}[v] \cup \mathcal{U}[v])$ **then**
13: $\quad\quad\quad\quad\quad$ ADDDISCARD$((node, v), var)$
14: $\quad\quad\quad\quad$ **end if**
15: $\quad\quad\quad$ **end for**
16: $\quad\quad$ **end if**
17: $\quad$ **end for**
18: **end for**

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 pairs $(\mathcal{S}, \mathcal{U})$ 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 Fig. 8a. 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 Fig. 4b and in Fig. 8b). Similarly, for the example in Fig. 9, 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 Fig. 3a, just like the function in Fig. 3b shows). Now consider the function analysed in Fig. 10b. In this case, there are no non-living variables, but two variables are in $\mathcal{U}$. Applying the discard insertion algorithm, we select the edge $(1, \mathtt{NonZero}(c), 2)$ to insert the discard of b and $(1, \mathtt{Zero}(c), 3)$ to insert the discard of a.

Evaluation We have implemented a prototype of our procedure in Python 3.[5] We tested both the consuming check and the 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

[5] Code at https://github.com/NicolaAssolini98/StaticAnalysisQuantumPrograms.

Fig. 11 Failing program: a is not consumed in the else branch

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

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 Fig. 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 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.

4.4 A complete example

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 Fig. 12a and the corresponding CFG in Fig. 12b. We show the system of equations derived from the CFG in Fig. 13 and the system's computation in Fig. 13b. In Fig. 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.

Fig. 14a and Fig. 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 functions. We show the system of equations derived from the CFG in Fig. 15a and the system's computation in Fig. 15b. In Fig. 14b, we show on each node the sets $(\mathcal{S}, \mathcal{U})$ corresponding to the MFP solution computed in Fig. 15b. We 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 $\texttt{discard}(a)$ 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 $\texttt{discard}(\mathrm{r})$ 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 $\texttt{discard}(\mathrm{z})$ 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 Fig. 16. We note that by using a type-based approach, the type checker rejects the program in Fig. 14a unless the programmer enters all the discard functions shown in Fig. 16.

4.5 The pipeline in action

The goal of the pipeline presented in this section is to ensure the correctness of high-level quantum programs by informing the compiler about the points in the program where a variable must be discarded (uncomputed), thus relieving the programmer from having to handle this task manually. As with a type checker, the safety and semantics of the discard operation depend on the compiler implementation beneath the analyzer. This is also true in languages with linear types, where variables typically need to be discarded manually, and the exact discard behavior remains compiler-dependent. Our analysis is designed as a preliminary compilation step to gather additional information related to linear types, thereby avoiding the need for explicit discard operations.

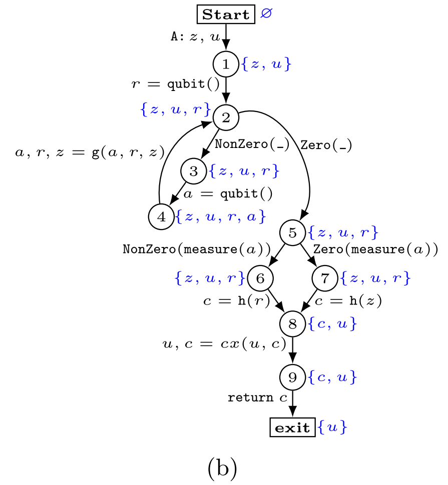
Replacing the type system with a static analysis introduces flexibility in two ways: it is language-independent, and it can be combined with other analyses to identify infeasible paths, thereby improving precision. In particular, classical analyses such as interval analysis and constant propagation can be employed to obtain more accurate classical control flows.

The pipeline operates intra-procedurally and, like a linear type system, assumes that all variables are local. However, our analysis allows for commands that do not consume resources while still identifying variables requiring uncomputation, thanks to the refined liveness information. This versatility makes the approach broadly applicable across different quantum programming languages. In fact, our analyses can be easily adapted to any language that employs a

Fig. 12 The example function

```
1  def f(z:qubit,u:qubit,n:int):
2      r = qubit()
3      for _ in range(n):
4          a = qubit()
5          a,r,z = g(a,r,z)
6
7      if measure(a):
8          c = h(r)
9      else:
10         c = h(z)
11
12     u,c = cx(u,c)
13
14     return c
```

(a)



(b)

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

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

(a)

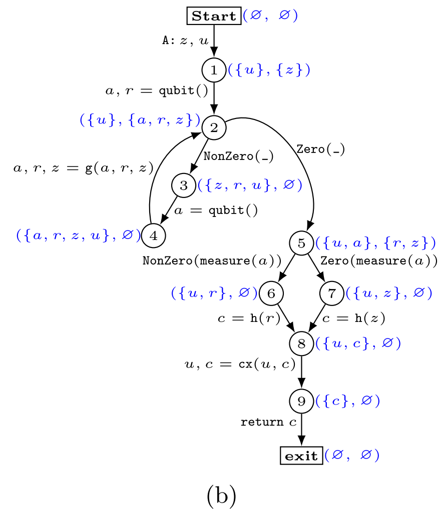
	0	1	F.P.
Start	$\varnothing$	$\varnothing$	$\varnothing$
1	$\mathbb{V}_q$	$\{z, u\}$	$\{z, u\}$
2	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
3	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
4	$\mathbb{V}_q$	$\{z, u, r, a\}$	$\{z, u, r, a\}$
5	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
6	$\mathbb{V}_q$	$\{z, u, r\}$	$\{z, u, r\}$
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. 14 The correct version of function `f` in Fig. 12

```
1  def r_f(z:qubit,u:qubit,n:int):
2      a, r = qubit()
3      for _ in range(n):
4          a = qubit()
5          a,r,z = g(a,r,z)
6
7      if measure(a):
8          c = h(r)
9      else:
10         c = h(z)
11
12     u,c = cx(u,c)
13
14     return c
```

(a)



(b)

linear type system, or even used to enhance those that do not, such as Quipper [42]. Furthermore, languages like Silq [16], which rely on annotations to determine whether a variable is consumed, can also be analyzed with minimal modifications. The analysis can also be extended to support global variables—i.e., variables that must not be discarded for intentional reuse—by introducing a special `global` definition in the language.

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

(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 MFP equations (a) and solutions (b) of the analysis applied to the CFG in Fig. 14

```
1 def rr_f(z:qubit,u:qubit,n:int):
2     a, r = qubit()
3     for _ in range(n):
4         discard(a)
5         a = qubit()
6         a,r,z = g(a,r,z)
7     if measure(a):
8         discard(z)
9         c = h(r)
10    else:
11        discard(r)
12        c = h(z)
13    u,c = cx(u,c)
14    discard(u)
15
16    return c
```

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

5 Entanglement analysis: a domain-based approach

In the previous section, we described how to identify temporary variables that may require uncomputation. Since uncomputing these variables incurs overhead for the quantum compiler, it is crucial to avoid performing this operation when it is not necessary. To reduce the number of calls to the uncomputation routine, we extend our analysis with a procedure that determines which of the detected temporary variables may cause side effects. Such side effects arise only when the variables in question are entangled with other program variables. A key aspect of this analysis is the identification of entanglement sets, that is, groups of variables such that operations on one variable may influence the others. Using these sets, we refine the initial collection of variables marked as 'to be uncomputed' and retain only those that are actually entangled, thereby limiting uncomputation to the necessary cases. This analysis is best performed at a relatively late stage of compilation, after the program has been lowered to its circuit-level representation. At this point, the structure of the circuit, including the precise placement of gates and the wires they operate on, is fully determined. Having this detailed information is essential for accurately identifying entanglement patterns and, consequently, for deciding which temporary variables may cause side effects if discarded. Earlier in the pipeline, such information is either incomplete or still subject to optimization passes, which would risk invalidating or weakening the conclusions of the analysis. Thus, to define our analysis, we refer to a minimal quantum language characterised by quantum statements on quantum variables with a control flow based on measurement.

Similarly to what we have done in Sect. 4.1, we introduce our minimal quantum language as a CFG language. This means that the branching in the CFG is guided by the probabilistic result of a quantum measurement. Given a finite set of quantum variables $\mathbb{V}_q$ and $q, p \in \mathbb{V}_q$ we define the language label as follows:

$$
\begin{aligned}
\mathsf{c} &::= \mathsf{label} \mid \mathsf{c};\mathsf{c} \mid \mathsf{c} \oplus \mathsf{c} \mid \mathsf{c}^* \\
\mathsf{label} &::= \mathtt{M}_1(q) \mid \mathtt{M}_0(q) \mid \mathtt{skip} \mid \mathtt{h}(q) \mid \mathtt{t}(q) \mid \mathtt{cx}(p,q),
\end{aligned}
\tag{6}
$$

where $\mathtt{M}_0(q)$ ($\mathtt{M}_1(q)$) indicates that the measurement of q returns 0 (1); the statements, $\mathtt{h}$, $\mathtt{t}$, and $\mathtt{cx}$ indicate, respectively, the Hadamard gate, the T gate and the control-not gate [58, Chap. 4]. Considering only these three quantum operations is not a limitation because they allow us to cover all possible quantum computations [18]. Moreover, the presence of measurement allows us to describe both circuit and QRAM [40, 95] programs.

Moreover, without loss of generality, we can assume that every variable $q_i \in \mathbb{V}_q$ corresponds to a 1-qubit register initialised to the state $|0\rangle$. As we show in Equation (2), our language is equivalent to the quantum while language used in [68, 90, 91], and to the one that is used to define the other two entanglement analyses based on abstract semantics [46, 69].

5.1 Collecting semantics

Let $Q = \{q_i\}_n$ be the set of variables, we call $\mathcal{H}_{\mathbb{V}_q} = \bigotimes_i^n \mathcal{H}_{q_i}$ the n-qubit Hilbert space, i.e., a space of dimension 2^n. Let $V_{\mathbb{V}_q}$ be the set of all vectors $|\psi\rangle \in \mathcal{H}_{\mathbb{V}_q}$, we define the collecting semantics as a function $[\![\cdot]\!] : \wp(V_{\mathbb{V}_q}) \to \wp(V_{\mathbb{V}_q})$. First, given a set $v \in \wp(V_Q)$, we define the collecting semantics for

each instruction of the language label as follows:

$$\begin{aligned}
[\![\, \texttt{skip} \,]\!] v &= v \\
[\![\, \texttt{h}(q) \,]\!] v &= \{\, H_q \, |\psi\rangle \,\big|\, |\psi\rangle \in v \,\} \\
[\![\, \texttt{t}(q) \,]\!] v &= \{\, T_q \, |\psi\rangle \,\big|\, |\psi\rangle \in v \,\} \\
[\![\, \texttt{cx}(p,q) \,]\!] v &= \{\, CX_{p,q} \, |\psi\rangle \,\big|\, |\psi\rangle \in v \,\} \\
[\![\, \texttt{M}_1(q) \,]\!] v &= \left\{ \frac{M_{1q} \, |\psi\rangle}{\| M_{1q} \, |\psi\rangle \, \|} \,\Big|\, \| M_{1q} \, |\psi\rangle \, \|^2 > 0, |\psi\rangle \in v \right\} \\
[\![\, \texttt{M}_0(q) \,]\!] v &= \left\{ \frac{M_{0q} \, |\psi\rangle}{\| M_{0q} \, |\psi\rangle \, \|} \,\Big|\, \| M_{0q} \, |\psi\rangle \, \|^2 > 0, |\psi\rangle \in v \right\}
\end{aligned}$$

where H_q and T_q are the unitary operators in $\mathcal{H}_{\mathbb{V}_q}$ that correspond to the gate Hadamard and T applied to q, $CX_{p,q}$ is the unitary that corresponds to control-not on p and q (where p is the controller and q the target) and the identity on the other variables while $\texttt{M}_1(q)$ and $\texttt{M}_0(q)$ corresponds to measurement 1 and 0 on q. For instance $[\![\, \texttt{M}_1(q) \,]\!] \{|1\rangle_q\} = \{|1\rangle_q\}$ while $[\![\, \texttt{M}_0(q) \,]\!] \{|1\rangle_q\} = \varnothing$. Finally, we can define the collecting semantics for the whole language:

$$\begin{aligned}
[\![\, \texttt{c}_1; \texttt{c}_2 \,]\!] v &= [\![\, \texttt{c}_2 \,]\!] ([\![\, \texttt{c}_1 \,]\!] v) \\
[\![\, \texttt{c}_1 \oplus \texttt{c}_2 \,]\!] v &= [\![\, \texttt{c}_1 \,]\!] v \cup [\![\, \texttt{c}_2 \,]\!] v \\
[\![\, \texttt{c}^* \,]\!] v &= \bigcup_n [\![\, \texttt{c}^n \,]\!] v.
\end{aligned} \tag{7}$$

With this collecting semantics, we only want to represent the set of all states that a program can return as a result. For this reason, when we consider the measurement, we follow a conservative approach by collecting all possible results, ignoring the probability with which these results occur.

5.2 Characterising entanglement

In this section, we define the properties of *separability* and of *direct inseparability*, and the abstract domain proposed to represent them. Here, the idea is to refine this domain to make it suitable for performing a static analysis for soundly detecting entanglement. To define an abstract domain which is able to capture the entanglement property of quantum variables, we introduce a characterisation of this property by means of an equivalence relation. For bipartite systems (e.g., two qubits), entanglement and separability are dual concepts. In fact, a composite quantum state $|\psi\rangle_{q_1,q_2} \in \mathcal{H}_{q_1} \otimes \mathcal{H}_{q_2}$ is separable if and only if it can be written as a tensor product $|\psi\rangle_{q_1,q_2} = |\phi_{q_1}\rangle \otimes |\phi_{q_2}\rangle$ for some states $|\phi_{q_1}\rangle \in \mathcal{H}_{q_1}$ and $|\phi_{q_2}\rangle \in \mathcal{H}_{q_2}$. A state $|\psi\rangle_{q_1,q_2}$ is entangled if and only if it is *not* separable. However, the scenario becomes more complex when considering systems consisting of three or more subsystems. Entanglement in such systems corresponds to inseparability across the entire system, but various degrees of entanglement can occur within subsystems.

Some metrics have been introduced to analyse the entanglement of these systems, such as *entanglement monotones* and *entanglement measures* [82], which quantify entanglement between subsystems. In [37], the entanglement of two subsystems S_1, S_2 is measured in terms of an entanglement monotone function $E_{|\psi\rangle}(S_1, S_2)$, such that $E_{|\psi\rangle}(S_1, S_2) = 0$ if and only if S_1 and S_1 taken in isolation are not entangled in the global system state $|\psi\rangle$. For instance, the 3-qubits state $|\psi\rangle_{q_1,q_2,q_3} = {}^1\!/\!_2(|000\rangle + |001\rangle + |011\rangle + |111\rangle)_{q_1,q_2,q_3}$ is *fully inseparable* since it cannot be decomposed via the tensor product ($|\psi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle$ for any $|\phi_1\rangle$ and $|\phi_2\rangle$). In fact, the entanglement monotone $E_{|\psi\rangle}(q_i, \{q_j, q_k\})$ always differs from zero for all $i, j, k \in \{1, 2, 3\}$. However, if we measure the entanglement between pairs of qubits, we have $E_{|\psi\rangle}(q_1, q_2) > 0$, $E_{|\psi\rangle}(q_2, q_3) > 0$ but $E_{|\psi\rangle}(q_1, q_3) = 0$, that is the qubits q_1 and q_3 taken in isolation are a separable subsystem. This occurs because tracing out q_2 yields, with equal probability, either ${}^1\!/\!\sqrt{2}(|00\rangle + |01\rangle)_{q_1,q_3}$ or ${}^1\!/\!\sqrt{2}(|01\rangle + |11\rangle)_{q_1,q_3}$, both corresponding to separable states of q_1 and q_3. This example shows that the entanglement is not transitive, i.e., the fact that q_1 is entangled with q_2 and q_2 with q_3 does not imply that q_1 is entangled with q_3.

In our analysis, we are interested in understanding when a set of variables is fully inseparable or whether the variables are separable in some way. When two variables are separable (and thus not entangled), we know we can measure one without altering the other. Thus, in our analysis, we speak about *separability* and we consider its dual notion *inseparability* instead of entanglement. Let us formally define the separability of two variables in a multi-variable state.

Definition 3 (Separability)
Let $\mathbb{V}_q$ be the set of variables in a state $|\psi\rangle_{\mathbb{V}_q}$. Two variables $q_1, q_2 \in \mathbb{V}_q$ are *separable* if the state $|\psi\rangle_{\mathbb{V}_q}$ can be written as $|\psi\rangle_{\mathbb{V}_q} = |\phi_1\rangle_{Q_1} \otimes |\phi_2\rangle_{Q_2}$, where $Q_1, Q_2 \subset \mathbb{V}_q$, $q_1 \in Q_1$ and $q_2 \in Q_2$. Otherwise, we say that q_1, q_2 are *inseparable*. Given a set $v \in \wp(V_{\mathbb{V}_q})$, two variables q_1, q_2 are inseparable in v if they are inseparable in *at least one state* $|\psi\rangle_{\mathbb{V}_q} \in v$.

There exists a particular relation between inseparable variables. For instance, let us consider the state $|\psi\rangle_{a,b,c} = (|000\rangle + |110\rangle + |001\rangle - |111\rangle)_{a,b,c}$, where a, b, c are inseparable. On an intuitive level, it can be seen that the three variables are not related in the same way. In fact, a and b are more closely related to each other than either a with c or b with c: if we measure a we obtain one of the two states: $(|00\rangle + |01\rangle)_{b,c}$ or $(|10\rangle - |11\rangle)_{b,c}$, where b has collapsed to a base state (0 or 1) in both states while c is still in superposition. Instead, if we measure c we obtain: $(|00\rangle + |11\rangle)_{a,b}$ or $(|00\rangle - |11\rangle)_{a,b}$ where a and b are in a entangled and superpose state. When two variables are related as a and b in this example, we say that they are *directly inseparable*.

Definition 4 (Direct Inseparability)
Given $a, b \in \mathbb{V}_q$ in a state $|\psi\rangle_{\mathbb{V}_q}$, we say that a and b are *directly inseparable* (d-inseparable) if, upon measuring one of them, the other collapses to a basis state in all possible measurement outcomes. Two variables $q_1, q_2 \in \mathbb{V}_q$ are d-inseparable in $v \in \wp(V_{\mathbb{V}_q})$ if they are d-inseparable in *all states* $|\psi\rangle_{\mathbb{V}_q} \in v$.

Being d-inseparable is a useful property when reasoning about entanglement. In fact, if we apply a controlled not (CX) between two d-inseparable variables, we will always 'disentangle' the target variable. For instance, if we apply $CX_{a,b}$, where a is the controller and b is the target, the state $|\psi\rangle_{a,b,c}$ defined above, we obtain:

$$\begin{aligned} &CX_{a,b}(|\psi\rangle_{a,b,c}) \\ &\quad = (|000\rangle + |100\rangle + (|001\rangle - |101\rangle)_{a,b,c} = \\ &\quad = ((|0\rangle + |1\rangle)\,|0\rangle + (|0\rangle - |1\rangle)\,|1\rangle)_{a,c} \otimes |0\rangle_b\,. \end{aligned} \tag{8}$$

In other words, we have separated b from the other variables. Instead if we apply $CX_{a,c}$ we obtain:

$$CX_{a,c}(|\psi\rangle_{a,b,c}) = (|000\rangle + |111\rangle + |001\rangle - |110\rangle)_{a,b,c}, \tag{9}$$

and we do not 'disentangle' a since c and a are not d-inseparable.

We now show that the properties of separability and direct inseparability are transitive.

Proposition 5
Given a set of variables $\mathbb{V}_q$ in a state $|\psi\rangle_{\mathbb{V}_q}$ and three variables q_1, q_2 and q_3 in $\mathbb{V}_q$, if q_1 and q_2 are non-separable and q_2 and q_3 are non-separable, then q_1 and q_3 are non-separable.

Proof
Let $|\psi\rangle_{\mathbb{V}_q} = |\phi_1\rangle_{Q_1} \otimes |\phi_2\rangle_{Q_2}$. Without loss of generality, we can assume that $q_1 \in Q_1$. Since q_1 and q_2 are non-separable q_2 must be in Q_1. But, by hypothesis, also q_2 and q_3 are non-separable, so q_3 must also be in Q_1. This means that q_1 and q_3 must be in the same set (in this case, Q_1), and so they are non-separable. □

Proposition 6
If q_1 and q_2 are d-inseparable and q_2 and q_3 are d-inseparable, then q_1 and q_3 are d-inseparable.

Proof
If q_1 and q_2 are d-inseparable, then if we measure q_1, then q_2 collapses to a base state, but since q_2 and q_3 are d-inseparable, then also q_3 will collapse. Thus, q_1 and q_3 are d-inseparable. □

Since the non-separability and d-inseparability are transitive, trivially symmetric, and we can easily assume that a variable is non-separable and d-inseparable from itself, i.e., these two properties are reflexive. Thus, inseparability and d-inseparability are equivalence relations.

5.3 An abstract domain for entanglement

The generation of entanglement depends on the values of the variables, which, therefore, must be taken into account when defining the elements of our abstract domain. Thus, we define an abstract state as consisting of two parts: the first is based on sets of variables representing inseparability and d-inseparability. In contrast, the second consists of a function that associates each variable with a specific label indicating the variable's state.

Inseparability and D-inseparability domain Inseparability and d-inseparability are both equivalence relations; thus, given a set of variables $\mathbb{V}_q$, we can represent both properties on $\mathbb{V}_q$ by partitions of $\mathbb{V}_q$. For instance, consider the state $|\psi\rangle_{a,b,c,d} = ((|00\rangle + |11\rangle)\,|0\rangle + (|00\rangle - |11\rangle)\,|1\rangle)_{a,b,c} \otimes |1\rangle_d$. We can build the partition $(\{a, b, c\}\{d\})$ that represents the inseparable variables in $|\psi\rangle_{a,b,c,d}$ and another partition $(\{a, b\}, \{c\}, \{d\})$ that identifies the d-inseparable variables. Since being d-inseparable implies being inseparable, the d-inseparable partition is always included in the inseparable one. We encode this information in our abstract states by representing them as a list of numbered sets (e.g., $[(\{a, b\}, 0), (\{c\}, 0), (\{d\}, 1)]$), where the smaller partition $((\{a, b\}, \{c\}, \{d\}))$ represents which variables are d-inseparable, while by merging sets with the same number, we obtain the partition representing inseparability $((\{a, b, c\}\{d\}))$.

Definition 5
Given a set of quantum variables $\mathbb{V}_q$, we define the abstract state $\mathcal{E}^{\mathbb{V}_q}$ as the set of tuples

$$\mathcal{E}^{\mathbb{V}_q} = \left\{ (e, k) \,\middle|\, e \in \wp(\mathbb{V}_q) \text{ and } k \in \mathbb{N} \right\},$$

where $\forall\,(e, k), (e', k') \in \mathcal{E}^{\mathbb{V}_q}, e \cap e' = \varnothing$ and $\bigcup_{(e,k)\in\mathcal{E}^{\mathbb{V}_q}} e = \mathbb{V}_q$. In other words, the sets e form a partition of $\mathbb{V}_q$.

We call $\mathbb{E}^{\mathbb{V}_q} \subset \wp(\wp(\mathbb{V}_q) \times \mathbb{N})$ the abstract domain of all possible $\mathcal{E}^{\mathbb{V}_q}$.

To better refer to the abstract state, we introduce the following notation. Given an abstract state $\mathcal{E}^{\mathbb{V}_q}$, we write E_k, using a capital letter, to refer to the set $E_k = \bigcup_{(e,k)\in\mathcal{E}^{\mathbb{V}_q}} e$, i.e., the union of all e with the same index k. For instance, if $\mathcal{E}^{\{a,b,c,d,e\}} = [(\{a, b\}, 0), (\{c\}, 0), (\{d, e\}, 1)]$, $E_0 = \{a, b, c\}$ while $E_1 = \{d, e\}$. Using this notation, we introduce a partial order in $\mathbb{E}^{\mathbb{V}_q}$.

Definition 6 ($\mathbb{E}^{\mathbb{V}_q}, \leqslant_{\mathbb{E}}$)
Given $\mathcal{E}_1^{\mathbb{V}_q}, \mathcal{E}_2^{\mathbb{V}_q} \in \mathbb{E}^{\mathbb{V}_q}$. $\mathcal{E}_1^{\mathbb{V}_q} \leqslant_{\mathbb{E}} \mathcal{E}_2^{\mathbb{V}_q}$ iff $\forall (e_2, k) \in \mathcal{E}_2^{\mathbb{V}_q}$, $\exists (e_1, k') \in \mathcal{E}_1^{\mathbb{V}_q}$ such that $e_2 \subseteq e_1$ and $\forall E_k \in \mathcal{E}_1^{\mathbb{V}_q}$, $\exists E_h \in \mathcal{E}_2^{\mathbb{V}_q}$ such that $E_k \subseteq E_h$.

We write $\vee_{\mathbb{E}}$ and $\wedge_{\mathbb{E}}$ to refer to the least upper bound (lub) and the greatest lower bound (glb) induced by $\leqslant_{\mathbb{E}}$, and the resulting domain is a complete lattice. For instance, $[(\{a,b,c\},0),(\{d\},1)] \leqslant_{\mathbb{E}} [(\{a,b\},0),(\{c\},0),(\{d\},1)] \leqslant_{\mathbb{E}} [(\{a\},0),(\{b\},0),(\{c\},0),(\{d\},1)] \leqslant_{\mathbb{E}} [(\{a\},0),(\{b\},0),(\{c\},0),(\{d\},0)]$, and $[(\{a,b\},0),(\{c\},1)] \vee_{\mathbb{E}} [(\{a\},0),(\{b,c\},1)] = [(\{a\},0),(\{b\},0),(\{c\},0)]$.

To ensure soundness, we overestimate the non-separabilities and determine which variables are potentially inseparable. On the other hand, since being d-inseparable implies special effects in relation to control-not and measurement, we underestimate the d-inseparability property to make sure we do not introduce errors in the abstract semantics.

We have defined an abstract domain that allows us to represent inseparability and d-inseparability. However, we need a final ingredient to define the abstract semantics: some elements abstracting the variables' state.

Labels We introduce some labels that represent some specific states that are relevant to entanglement abstraction. The CX gate does not introduce entanglement if the controller is in a classical state ($|0\rangle$ or $|1\rangle$) or the target is in a uniform superposition ($1/\sqrt{2}\,|0\rangle \pm |1\rangle$). To track these two states, we introduce two labels that represent two sets of states: $Z = \{\phi\,|b\rangle\}$ (the set of classical values), and $X = \{\phi(1/\sqrt{2}\,|b\rangle \pm 1/\sqrt{2}\,|\overline{b}\rangle)\}$ (the set of values in uniform superposition), where ϕ represents a global phase, b is a binary string and $\overline{b}$ is the negation of b (e.g. if $b=0$ then $\overline{b}=1$ and if $b=010$ then $\overline{b}=101$). Moreover, we introduce three other labels:

$$P = \{\phi(1/\sqrt{2}\,|b\rangle \pm {}^{i+1}/\sqrt{2}\,|\overline{b}\rangle)\} \qquad Y = \{\phi(1/\sqrt{2}\,|b\rangle \pm {}^{i}/\sqrt{2}\,|\overline{b}\rangle)\}$$

$$R = \{\phi(1/\sqrt{2}\,|b\rangle \pm {}^{i-1}/\sqrt{2}\,|\overline{b}\rangle)\}.$$

We need these labels to represent the semantics of the gate $\mathtt{t}$. In particular, $\mathtt{t}X = P$, $\mathtt{t}P = Y$, $\mathtt{t}Y = R$ and $\mathtt{t}R = X$ while $\mathtt{t}Z = Z$. Then, we add a final label to represent states that are not classical, i.e., those that are definitely in superposition: $S = \{\phi(\alpha\,|b\rangle \pm \beta\,|\overline{b}\rangle) \mid |\alpha|^2 + |\beta|^2 = 1 \text{ and } \alpha \neq 0 \wedge \beta \neq 0\}$. Finally, we define $\bot_{\mathcal{L}}$ as the empty set and $\top_{\mathcal{L}}$ as the set of all possible vectors. We can order these labels by inclusion, constructing a lattice $(\mathcal{L}, \leqslant_{\mathcal{L}})$, represented in Fig. 17.

5.3.1 Put all together: the abstract domain

Now, we include the labels in the definition of the abstract domain. When variables are inseparable, their state cannot be described as a combination of individual states. Recall that, in an abstract state $\mathcal{E}^{\mathbb{V}_q}$, a set of inseparable variables is the union of all sets e with the same index k. We introduce a labelling function to associate each set of inseparable variables with a label. Formally, we define the labelling function as $\lambda : \mathbb{N} \to \mathcal{L}$, and we call $\Lambda = \mathbb{N} \times \mathcal{L}$ the domain of the labelling function. Consequently, we reformulate the definition of the abstract state, including the labelling function, as follows.

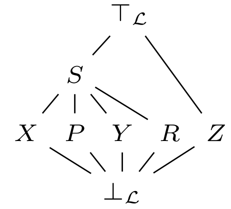


Fig. 17 Lattice $(\mathcal{L}, \leqslant_{\mathcal{L}})$

Definition 7
Given a set of quantum variables $\mathbb{V}_q$, let be $\mathcal{E}^{\mathbb{V}_q} \in \mathbb{E}^{\mathbb{V}_q}$ and $\lambda \in \Lambda$, we define an abstract state as the pair $(\mathcal{E}^{\mathbb{V}_q}, \lambda)$.

We define $\mathbb{A}^{\mathbb{V}_q} = \mathbb{E}^{\mathbb{V}_q} \times \Lambda$ as the abstract domains. For instance, let us consider a set of variables $Q = \{a,b,c,d\}$ in the state

$$|\psi\rangle_Q = (1/2(|0\rangle + |1\rangle)\,|0\rangle + {}^{i}/2(|0\rangle - |1\rangle)\,|1\rangle)_{a,b} \otimes 1/\sqrt{2}(|10\rangle + |01\rangle)_{c,d}.$$

First, we construct the partition corresponding to the sets of inseparable variables, i.e., the sets $\{a,b\}$ and $\{c,d\}$. Once these two sets have been identified, we construct the abstract state by identifying which variables are d-inseparable, obtaining the abstract state: $[(\{a\},0)(\{b\},0)(\{d,c\},1)]$. The last step is to label the sets of inseparable variables. In particular, given a set of variables in a state $|\phi\rangle$, we choose the smallest label L such that $|\phi\rangle \in L$. We see that the state of the variables a, b is contained only in $\top_{\mathcal{L}}$ while the state of c, d is contained in $X, S, \top_{\mathcal{L}}$, consequently the final state will be equal to: $(\mathcal{E}^{\mathbb{V}_q}, \lambda) = ([(\{a\},0)(\{b\},0)(\{c,d\},1)], \{0 : \top_{\mathcal{L}}, 1 : X\}$.

At this point, if we want to compute the set of concrete states represented by an abstract state, starting from the obtain $(\mathcal{E}^{\mathbb{V}_q}, \lambda)$, we consider $\mathcal{E}^{\mathbb{V}_q}$. In this case, we know that there are two sets $\{a,b\}$, $\{c,d\}$ of inseparable variables, so the abstract state corresponds to a set of concrete states in the form $|\phi_1\rangle_{a,b} \otimes |\phi_2\rangle_{c,d}$. Then, we can get more precise information about $|\phi_1\rangle$ and $|\phi_2\rangle$ by checking which variables are d-inseparable, that is, c and d. In particular, we know that the concrete states can be expressed by the set $\{|\Psi\rangle \mid |\Psi\rangle = (|\phi_1\rangle_{a,b} \otimes (|b\rangle + |\overline{b}\rangle)_{c,d}\}$. Now, we complete the concretisation by checking the information contained in the labels. By the labels we know that a, b can be in any quantum state

while c, d are in the form of $1/\sqrt{2}(|b\rangle + |\overline{b}\rangle)_{c,d}$. Finally, by intersecting this information with the previous one, we obtain the set: $\{|\psi\rangle \mid |\psi\rangle = (|\phi_1\rangle_{a,b} \otimes 1/\sqrt{2}(\alpha |b\rangle + \beta |\overline{b}\rangle)_{c,d}\}$, which represents the concretisation $\gamma((\mathcal{E}^{\mathbb{V}_q}, \lambda))$ of $(\mathcal{E}^{\mathbb{V}_q}, \lambda)$ and, of course, $|\psi\rangle \in \gamma((\mathcal{E}^{\mathbb{V}_q}, \lambda))$. Note that labels represent the state of a single variable or a state of n d-inseparable variables. So, if a set of inseparable variables is not labelled as $\top_{\mathcal{L}}$, Z, or $\bot_{\mathcal{L}}$, it means that it corresponds either to a single variable or to all d-inseparable variables.

Based on the partial order $\leqslant_{\mathbb{E}}$ defined in Definition 6, we can define an ordering in $\mathbb{A}^{\mathbb{V}_q}$.

Definition 8 ($\mathbb{A}^{\mathbb{V}_q}, \sqsubseteq$)
Given $(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1), (\mathcal{E}_2^{\mathbb{V}_q}, \lambda_2) \in \mathbb{A}^{\mathbb{V}_q}$, $(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1) \sqsubseteq (\mathcal{E}_2^{\mathbb{V}_q}, \lambda_2)$ if and only if

$$\mathcal{E}_1^{\mathbb{V}_q} \leqslant_{\mathbb{E}} \mathcal{E}_2^{\mathbb{V}_q} \ \wedge\ \forall E_h \in \mathcal{E}_2,$$

$$\begin{cases} \lambda(k) \leqslant_{\mathcal{L}} \lambda(h) & \text{if } \exists E_k \in \mathcal{E}_1 \text{such that } E_h = E_k \\ \lambda(h) = \top_{\mathcal{L}} & \text{otherwise} \end{cases}$$

We call $\sqcup$ and $\sqcap$ the lub and glb induced by the order. Since the lattices $(\mathbb{E}^{\mathbb{V}_q}, \leqslant_{\mathbb{E}})$ and $(\mathcal{L}, \leqslant_{\mathcal{L}})$ are finite and defined by inclusion operators, they are complete lattices. Thereby, also the lattice $((\mathcal{E}^{\mathbb{V}_q}, \lambda), \sqsubseteq)$ is complete.

For instance, consider two abstract states

$$(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1) = ([(\{p,q\},0)], 0 : X) \text{ and}$$
$$(\mathcal{E}_2^{\mathbb{V}_q}, \lambda_2) = ([(\{p,q\},0)], 0 : Y).$$

The lub between them is $(\mathcal{E}_3^{\mathbb{V}_q}, \lambda_3) = ([(\{p,q\},0)], 0 : S)$. In this case, since p, q are d-inseparable in both states, they are also in the lub, and we label the partition by the lub between the labels (we are in the 'if' case of the Definition 8). In fact, $(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1)$ represent the set of states

$$\{|\psi\rangle \mid \phi(1/\sqrt{2}\,|b\rangle \pm 1/\sqrt{2}\,|\overline{b}\rangle)\}$$

and $(\mathcal{E}_2^{\mathbb{V}_q}, \lambda_2)$ represent the set of states

$$\{|\psi\rangle \mid \phi(1/\sqrt{2}\,|b\rangle \pm i/\sqrt{2}\,|\overline{b}\rangle)\},$$

where $b \in \{00, 01, 10, 11\}$, and the abstraction of the union of these two sets is $(\mathcal{E}_3^{\mathbb{V}_q}, \lambda_3)$.

On the other hand, if we consider the states

$$(\mathcal{E}_4^{\mathbb{V}_q}, \lambda_4) = ([(\{p,q\},0), (\{t\},1)], \{0 : X, 1 : X\})$$

and

$$(\mathcal{E}_5^{\mathbb{V}_q}, \lambda_5) = ([(\{p\},0), (\{q,t\},1)], \{0 : X, 1 : X\},$$

the lub between them is

$$(\mathcal{E}_6^{\mathbb{V}_q}, \lambda_6) = ([(\{p\},0), (\{q\},0), (\{t\},0)], \{0 : \top_{\mathcal{L}}\}.$$

In fact, $(\mathcal{E}_4^{\mathbb{V}_q}, \lambda_4)$ represent the set of states

$$\{|\psi\rangle \mid \phi(1/\sqrt{2}\,|b\rangle \pm 1/\sqrt{2}\,|\overline{b}\rangle)_{p,q} \otimes 1/\sqrt{2}\,|0\rangle \pm 1/\sqrt{2}\,|1\rangle)_t\}$$

and $(\mathcal{E}_5^{\mathbb{V}_q}, \lambda_5)$ represent the set of states

$$\{|\psi\rangle \mid \phi(1/\sqrt{2}\,|0\rangle \pm 1/\sqrt{2}\,|1\rangle)_p \otimes 1/\sqrt{2}\,|b\rangle \pm 1/\sqrt{2}\,|\overline{b}\rangle)_{q,t}\}.$$

If we join these sets, we obtain a set of states where p, q, t are inseparable. However, p and q are only d-inseparable in some states, while q and t are d-inseparable in others. So, in general, we can only say that p, q, t are inseparable and not d-inseparable, and the only possible label for these states is $\top_{\mathcal{L}}$.

Concretisation function Now, we can introduce some formalism to define the concretisation function $\gamma : \mathbb{A} \rightarrow \wp(V_{\mathbb{V}_q})$.

Definition 9
Let $\mathbb{V}_q$ be a set of variables, and $\pi \subset \wp(\mathbb{V}_q)$ a partition of $\mathbb{V}_q$. Given a state $|\psi\rangle_{\mathbb{V}_q}$, we say that the variables in $\mathbb{V}_q$ are π-separable if $|\psi\rangle_{\mathbb{V}_q} = \bigotimes_{p \in \pi} |\phi\rangle_p$, i.e., their joint state can be decomposed into a product of states across a partition π of the variables.

For instance, given a state $|\psi\rangle_{q_1,q_2,q_3}$ and a partition $\pi = \{\{q_1, q_2\}\{q_3\}\}$, q_1, q_2, q_3 are π-separable if and only if we can write $|\psi\rangle_{q_1,q_2,q_3}$ as $|\phi_1\rangle_{q_1,q_2} \otimes |\phi_2\rangle_{q_3}$.

Given an abstract state $(\mathcal{E}^{\mathbb{V}_q}, \lambda)$, we write $\{E_k\}$ to indicate the sets that are obtained by $\mathcal{E}^{\mathbb{V}_q} = \{(e,k)\}$ joining the sets e with the same k. Recall that $\{E_k\}$ is a partition that represents the sets of inseparable variables.

Definition 10
Given a set of variables $\mathbb{V}_q$, we say that $|\psi\rangle_{\mathbb{V}_q} \triangleright (\mathcal{E}^{\mathbb{V}_q}, \lambda)$ (that is, $|\psi\rangle_{\mathbb{V}_q}$ is abstracted by $(\mathcal{E}^{\mathbb{V}_q}, \lambda)$) if and only if

- $|\psi\rangle_{\mathbb{V}_q}$ is $\{E_k\}$-separable;
- $\forall (e,k) \in \mathcal{E}^{\mathbb{V}_q}$, $q_i, q_j \in e \Rightarrow q_j$ and q_i are d-inseparable in $|\psi\rangle_{\mathbb{V}_q}$;
- given $|\psi\rangle_{\mathbb{V}_q} = \bigotimes_k |\psi\rangle_{E_k}$, for all k, $|\psi\rangle_{E_k} \in \lambda(k)$.

In other words, we say that an abstract state $\mathcal{E}^{\mathbb{V}_q}$ abstracts a concrete state if and only if the abstract state over-approximates the set of inseparable variables, under-approximates the d-inseparability, and all separable sub-states that compose the state are represented by labels. Now

we have all the elements to define the concretisation function $\gamma : \mathbb{A} \to \wp(V_{\mathbb{V}_q})$:

$$\gamma(\mathcal{E}^{\mathbb{V}_q}, \lambda) = \{|\psi\rangle_{\mathbb{V}_q} \mid |\psi\rangle_{\mathbb{V}_q} \triangleright (\mathcal{E}^{\mathbb{V}_q}, \lambda)\}.$$

Theorem 2
Given a set of abstract states $\{(\mathcal{E}_j^{\mathbb{V}_q}, \lambda_j)\}_j$, let $I = \bigcap_j \gamma(\mathcal{E}_j^{\mathbb{V}_q}, \lambda_j)$, exists an abstract state $(\mathcal{E}_I^{\mathbb{V}_q}, \lambda_I)$ such that $\gamma((\mathcal{E}_I^{\mathbb{V}_q}, \lambda_I)) = I$.

Proof (Sketch)
$\gamma(\mathcal{E}_j^{\mathbb{V}_q}, \lambda_j)$ is the set of state in $\mathcal{H}_{\mathbb{V}_q}$ that are abstracted by $(\mathcal{E}_j^{\mathbb{V}_q}, \lambda_j)$, so I is the set of states that are abstracted by all $(\mathcal{E}_j^{\mathbb{V}_q}, \lambda_j)$. Consequently, if $(\mathcal{E}_I^{\mathbb{V}_q}, \lambda_I) = \bigsqcap_j (\mathcal{E}_j^{\mathbb{V}_q}, \lambda_j)$ then $\gamma((\mathcal{E}_I^{\mathbb{V}_q}, \lambda_I)) = I$. □

Theorem 2 proves that $\mathbb{A}$ is isomorphic to a Moore family of $\wp(V_Q)$. This means that exists a function $\alpha_I : \wp(V_{\mathbb{V}_q}) \to \mathbb{A}$ such that $\langle \mathbb{A}, \alpha_I, \gamma_I, \wp(V_{\mathbb{V}_q})\rangle$ forms a Galois Insertion [34, 35, 84].

5.4 An abstract semantics

In this section, we define the abstract semantics for our analysis. We first need to introduce some additional notation to better represent the operations on abstract states. Let us consider a generic abstract state $(\mathcal{E}^{\mathbb{V}_q}, \lambda)$, where $\mathcal{E}^{\mathbb{V}_q} = \{(e_i, k_i)\}_i$. We write $\mathcal{E}^{\mathbb{V}_q}(q)$ to refer to the pair $(e, k) \in \mathcal{E}^{\mathbb{V}_q}$ such that $q \in e$, while we have $\mathcal{E}^{\mathbb{V}_q}\{q\}$ to refer to set $E_k \in \mathcal{E}^{\mathbb{V}_q}$ that contains q. For instance, if $\mathcal{E}_1 = [(\{q\},0), (\{t\},0), (\{r\},1)]$ then $\mathcal{E}_1(q) = (\{q\},0)$ and $\mathcal{E}_1\{q\} = \{q,t\}$. We write $\mathcal{E}^{\mathbb{V}_q}[q_1 + q_2]$ to mark q_1 and q_2 inseparable in $\mathcal{E}^{\mathbb{V}_q}$ (e.g., $\mathcal{E}_1[r+q]$ is equal to $[(\{q\},1), (\{t\},1), (\{r\},1)]$). Note that if we mark q and r as inseparable, this also affects t due to the transitivity of the inseparability. $\mathcal{E}^{\mathbb{V}_q}[q_1 \uplus q_2]$ means that we marked q_1 and q_2 as d-inseparable, so given $\mathcal{E}_1$ from above, $\mathcal{E}_1[q \uplus t] = [(\{q,t\},0), (\{r\},1)]$. Note that $\mathcal{E}[q_1 \uplus q_2]$ implies applying also $\mathcal{E}[q_1 + q_2]$. $\mathcal{E}^{\mathbb{V}_q}[\sim q]$ denotes the removal of q from the set of variables d-inseparable from itself, and $\mathcal{E}^{\mathbb{V}_q}[\neg q]$ denotes that we make q separable from the rest of the variables. For instance, given $\mathcal{E}_2 = [(\{q,r\},0), (\{t\},0)]$, $\mathcal{E}_2[\sim q]$ is equal to $[(\{r\},0), (\{q\},0), (\{t\},0)]$ and $\mathcal{E}_2[\neg t]$ correspond to $[(\{q,r\},0), (\{t\},1)]$. Of course, $\mathcal{E}_2[\neg q_1]$ implies $\mathcal{E}_2[\sim q_1]$. Finally, to ease the use of the labelling function, given a variable q, let $\mathcal{E}^{\mathbb{V}_q}(q) = (e,k)$, we write $\lambda(q)$ to refer to $\lambda(k)$. Additionally, we write $\lambda[q \leftarrow L]$ to state that we change the label referred to the index k associated with q, setting it equal to L. For instance, given $(\mathcal{E}^{q,p,t}, \lambda) = ([(\{q\},0)(\{p\},0), (\{t\},1)], \{0 : \top_{\mathcal{L}}, 1 : X\})$, $\lambda(q) = \lambda(p)$ correspond to $\lambda(0) = \top_{\mathcal{L}}$, $\lambda(t) = X$, since it

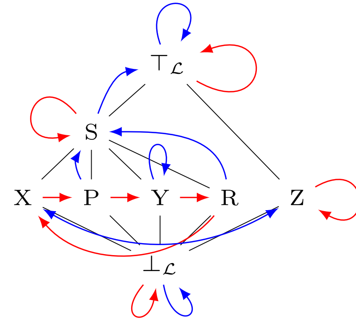


Fig. 18 The red arrow indicates the semantics of the abstract operation $T_q^\sharp$ while the blue one indicates the semantics of the abstract operation $H_q^\sharp$

corresponds to $\lambda(1)$, and $(\mathcal{E}^{q,p,t}, \lambda[q \leftarrow Y])$ is equal to $([(\{q\},0)(\{p\},0), (\{t\},1)], \{0 : Y, 1 : X\})$. In general, when we speak about a label associated with a variable q, we implicitly refer to the label associated with the index associated with q.

Before defining the abstract semantics of the language, we define four abstract operations $H_q^\sharp, T_q^\sharp, CX_{p,q}^\sharp, M_q^\sharp : \mathbb{A}^{\mathbb{V}_q} \to \mathbb{A}^{\mathbb{V}_q}$, that correspond to the abstraction of the unitary operation H, T, CX and the measurement, respectively.

For all the abstract operations it holds that if $(\mathcal{E}^{\mathbb{V}_q}, \lambda) = \bot$, then $G_v^\sharp(\mathcal{E}^{\mathbb{V}_q}, \lambda) = (\mathcal{E}^{\mathbb{V}_q}, \lambda)$. If fact $\gamma(\bot) = \emptyset$ and $G_v\emptyset = \emptyset$.

***T* gate ($T_q^\sharp$)** This gate does not make a difference if we apply it to a single or a group of d-inseparable variables. Formally,

$$T_q^\sharp(\mathcal{E}^{\mathbb{V}_q}, \lambda) = (\mathcal{E}^{\mathbb{V}_q}, \lambda[q \leftarrow V]),$$

where V can be derived from the red arrows in Fig. 18.

Hadamard gate ($H_q^\sharp$) The Hadamard gate distinguishes whether q is entangled with other variables. Formally, the abstract semantics is defined as follows:

$$H_q^\sharp(\mathcal{E}^{\mathbb{V}_q}, \lambda) = \begin{cases} (\mathcal{E}^{\mathbb{V}_q}, \lambda[q \leftarrow V]) & |\mathcal{E}^Q\{q\}| = 1, \\ (\mathcal{E}^{\mathbb{V}_q}(\sim q), \lambda[q \leftarrow \top_{\mathcal{L}}]) & \text{otherwise.} \end{cases}$$

In particular, if q is separable from the other variables (i.e., $|\mathcal{E}^Q\{q\}| = 1$), V can be derived from the blue arrows in Fig. 18. If q is inseparable (i.e., $|\mathcal{E}^Q\{q\}| > 1$), applying Hadamard to it produces a state that we can only label with $\top_{\mathcal{L}}$, and the variable q is no longer d-inseparable. For instance, given $|\psi\rangle_{p,q,t} = (1/\sqrt{2}(|000\rangle + |111\rangle)_{p,q,t}$, p,q,t are d-inseparable and their state can be labelled as X. Then, applying $H_q|\psi\rangle_{p,q,t} = (|000\rangle + |010\rangle + |101\rangle + |111\rangle)_{p,q,t}$, q is no longer d-inseparable from p,t, and the state is only labelable by $\top_{\mathcal{L}}$.

Controlled-not gate ($CX^\sharp_{c,t}$) The CX gate can introduce or nullify entanglement, so we need to consider different cases according to the state of c and t. Given an abstract state $(\mathcal{E}^{\mathbb{V}_q}, \lambda)$, we can define the abstract semantics as follows:

- if $\lambda(c) = Z$, i.e., the controller is in a base value, the CX corresponds to a classical controlled not, so it does not introduce or nullify entanglement and it does not change labels, thus:

$$CX^\sharp_{c,t}(\mathcal{E}^{\mathbb{V}_q}, \lambda) = (\mathcal{E}^{\mathbb{V}_q}, \lambda);$$

- if t is separable from other variables (i.e. $|\mathcal{E}^{\mathbb{V}_q}\{t\}| = 1$) we check the state of c and t:

$$CX^\sharp_{c,t}(\mathcal{E}^{\mathbb{V}_q}, \lambda) = \begin{cases} (\mathcal{E}^{\mathbb{V}_q}, \lambda) & \text{if } \lambda(t) = X, \\ (\mathcal{E}[c \uplus t], \lambda) & \text{if } \lambda(c) \neq \top_{\mathcal{L}} \wedge \lambda(t) = Z, \\ ((\mathcal{E}[\sim t])[c + t], \lambda[t \leftarrow \top_{\mathcal{L}}]) & \text{otherwise.} \end{cases}$$

  In particular, if the target is in uniform superposition, the CX gate corresponds to the identity.

  If the controller is surely in superposition (i.e., $\lambda(c) \neq \top_{\mathcal{L}}$ and $\lambda(c) \neq Z$) and the target is a classical value, the CX gate makes t d-inseparable from c. Moreover, when we set t d-inseparable from c, t automatically gets the label of c. For instance, if we have three variables, q, c, t in the state $|\psi_1\rangle = {}^1\!/\!\sqrt{2}(|00\rangle + |11\rangle)_{q,c} \otimes |1\rangle_t$, applying $CX_{c,t}|\psi_1\rangle$ we obtain ${}^1\!/\!\sqrt{2}(|001\rangle + |110\rangle)_{q,c,t}$ in which q, c, t are d-inseparable. Then, given $Q = \{q, c, t\}$ and $(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1) = \{(\{q, c\}, 0), (\{t\}, 1)\}, \{0 : X, 1 : Z\})$ such that $|\psi_1\rangle \in \gamma(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1)$,

$$CX^\sharp_{c,t}(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1) = (\{(\{q, c, t\}, 0)\}, \{0 : X\})$$

  and

$$CX_{c,t}|\psi_1\rangle \in \gamma(CX^\sharp_{c,t}(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1)).$$

  Finally, if none of the above conditions is fulfilled, we need to approximate the relation between c and t. To maintain the soundness, we mark c and t as inseparable without setting c and t d-inseparable. Moreover, we mark c and t equal to $\top_{\mathcal{L}}$ (recall that, since t and c are inseparable, they are related to the same k, so writing $\lambda[t \leftarrow \top_{\mathcal{L}}]$ or $\lambda[c \leftarrow \top_{\mathcal{L}}]$ produce the same effects).
- if t is inseparable from other variables, we need to check if the CX gate nullifies some entanglements:

$$CX^\sharp_{c,t}(\mathcal{E}^{\mathbb{V}_q}, \lambda) = \begin{cases} (\mathcal{E}[\neg t], \lambda[t \leftarrow Z]) & \mathcal{E}^{\mathbb{V}_q}(c) = \mathcal{E}^{\mathbb{V}_q}(t) \\ (\mathcal{E}[\sim t], \lambda[t \leftarrow \top_{\mathcal{L}}]) & \text{otherwise} \end{cases}.$$

  In particular, if c and t are d-inseparable ($\mathcal{E}^{\mathbb{V}_q}(c) = \mathcal{E}^{\mathbb{V}_q}(t)$), we 'disentangle' t, as we see in Equation (8), and, we set the disentangled variable separable from the rest, labelling it as Z. Otherwise, we do not know the exact effect of the gate since, as we have shown in Equation (9), the CX alters the entangled state. Thus, to maintain soundness, we mark t as not d-inseparable from the other variables and label it as $\top_{\mathcal{L}}$. For instance, if

$$(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1) = ([(\{a, b\}, 0), (\{c\}, 0)], \{0 : \top_{\mathcal{L}}\}),$$

  we have that

$$CX^\sharp_{a,b}(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1) = ([(\{a\}, 0), (\{c\}, 0), (\{b\}, 1)], \{0 : \top_{\mathcal{L}}, 1 : Z\});$$

  while

$$CX^\sharp_{c,a}(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1) = ([(\{a\}, 0), (\{c\}, 0), (\{b\}, 0)], \{0 : \top_{\mathcal{L}}\}).$$

  Given $|\psi\rangle_{a,b,c}$, $CX_{a,b}|\psi\rangle_{a,b,c}$ and $CX_{c,a}|\psi\rangle_{a,b,c}$ from Equation (8) and Equation (9), note that:

$$|\psi\rangle_{a,b,c} \in \gamma((\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1)),$$

$$CX_{a,b}|\psi\rangle_{a,b,c} \in \gamma(CX^\sharp_{c,a}(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1))$$

  and

$$CX_{c,a}|\psi\rangle_{a,b,c} \in \gamma(CX^\sharp_{a,b}(\mathcal{E}_1^{\mathbb{V}_q}, \lambda_1)).$$

Measurement ($M^\sharp_q$) In this case, we need to approximate which variables may be affected by the measurement. The semantics of measurement is formally defined as:

$$M^\sharp_q(\mathcal{E}^{\mathbb{V}_q}, \lambda) = (\mathcal{E}[\neg Q], L[Q \leftarrow Z, T \leftarrow \top_{\mathcal{L}}]),$$

where $Q = \mathcal{E}^{\mathbb{V}_q}(q)$ and $T = (\mathcal{E}^{\mathbb{V}_q}\{q\} \setminus Q)$. In particular, when a variable q is separable, $Q = \{q\}$ and $T = \emptyset$, and the measurement simply makes q collapse to a base state. Instead, if q is inseparable from other variables, all variables d-inseparable from q collapse to Z (making them separable), while all other variables inseparable and not d-inseparable from q are altered in a way that cannot be modelled given an abstract state. For this reason, we can only label these variables with $\top_{\mathcal{L}}$.

Language abstract semantics Now we have all the ingredients to define the abstract semantics of the language, which we define as a function $(\!|\cdot|\!) : \mathbb{A}^{\mathbb{V}_q} \to \mathbb{A}^{\mathbb{V}_q}$, for each

instruction of our language:

$$\begin{aligned}
(\!|skip|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &= (\mathcal{E}^{\mathbb{V}_q},\lambda)\\
(\!|\mathtt{h}(q)|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &= H_q^{\sharp}(\mathcal{E}^{\mathbb{V}_q},\lambda)\\
(\!|\mathtt{t}(q)|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &= T_q^{\sharp}(\mathcal{E}^{\mathbb{V}_q},\lambda)\\
(\!|\mathtt{cx}(p,q)|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &= CX_{p,q}^{\sharp}(\mathcal{E}^{\mathbb{V}_q},\lambda)\\
(\!|\mathtt{M}_1(b)|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &= M_q^{\sharp}(\mathcal{E}^{\mathbb{V}_q},\lambda)\\
(\!|\mathtt{M}_0(b)|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &= M_q^{\sharp}(\mathcal{E}^{\mathbb{V}_q},\lambda)\\
(\!|\mathtt{c}_1;\mathtt{c}_1|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &=(\!|\mathtt{c}_2|\!)((\!|\mathtt{c}_1|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda))\\
(\!|\mathtt{c}_1\oplus\mathtt{c}_2|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &=(\!|\mathtt{c}_1|\!)\sqcup(\!|\mathtt{c}_2|\!)\\
(\!|\mathtt{c}^*|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda) &= \bigsqcup_n (\!|\mathtt{c}^n|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda).
\end{aligned}\tag{10}$$

where $H_q^{\sharp}, T_q^{\sharp}, CX_{p,q}^{\sharp}, M_q^{\sharp} : \mathbb{A}^{\mathbb{V}_q} \to \mathbb{A}^{\mathbb{V}_q}$ represent the abstract semantics of gates and measurement.

Finally, we formulate the soundness of our abstraction in terms of the concretisation function γ [35].

Proposition 7 (Soundness)
Let $\mathbb{V}_q$ be a set of variables, $\forall l \in$ label,

$$\forall(\mathcal{E}^{\mathbb{V}_q},\lambda)\in\mathbb{A},\ [\![l]\!]\circ\gamma(\mathcal{E}^{\mathbb{V}_q},\lambda)\subseteq\gamma\circ(\!|l|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda).$$

Evidence in support of this proposition is shown by the examples in Sect. 5.5.1. Since every label is sound, by induction on $\mathtt{c}$ we can prove that $[\![\mathtt{c}]\!]\circ\gamma(\mathcal{E}^{\mathbb{V}_q},\lambda)\subseteq\gamma\circ(\!|\mathtt{c}|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda)$.

5.5 Computing the analysis

To compute the analysis on the CFG, we need to compute the abstract state $(\mathcal{E}^{\mathbb{V}_q},\lambda)$ for each node of the CFG, namely at each program point of the analysed program [78]. The analysis we propose is forward; namely, the state $(\mathcal{E}^{\mathbb{V}_q},\lambda)$ at node u, denoted $(\mathcal{E}^{\mathbb{V}_q},\lambda)[u]$, depends on the pairs $\{(\mathcal{E}_i^{\mathbb{V}_q},\lambda_i)\}$ of its predecessors and the label semantics of the edges entering u. Given a CFG G, for all node u in G (program points of the represented program), we define the following system of equations:

$$(\mathcal{E}^{\mathbb{V}_q},\lambda)[u] = \begin{cases}(\mathcal{E}_0^{\mathbb{V}_q},\lambda_0) & \text{if } u = \textbf{start}\\ \bigsqcup\left\{ (\!|l|\!)(\mathcal{E}^{\mathbb{V}_q},\lambda)[u] \;\middle|\; (u,l,v)\in G\right\} & \text{otherwise}\end{cases}$$

where $(\mathcal{E}_0^{\mathbb{V}_q},\lambda_0)$ is the initial state. Since we assume that all variables are initialised to $|0\rangle$, the initial state is the state where all variables are separable and labelled as Z. So if $\mathbb{V}_q = \{a,b,c\}$ then

$$(\mathcal{E}_0^{\mathbb{V}_q},\lambda_0) = ([(\{a\},0),(\{b\},1),(\{c\},2)],\{0,1,2,:Z\}).$$

This system can be solved by the least fixed point obtaining the best solution for each program point [51].

Proposition 8
For all statement $l \in$ label, the abstract semantics $(\!|l|\!) : \mathbb{A} \to \mathbb{A}$ is monotonic w.r.t. $\sqsubseteq$.

Since the semantics is monotonic, it is granted that we reach the fix-point.

We provide a prototype of our procedure[6] that analyses the quantum language used to present the analysis. Together with the prototype, we provide examples showing how our analysis works in various scenarios. In particular, we analyse the examples contained in [72] (superdense coding [15], Deutsch algorithm [36], and the Creation and disentanglement of the GHZ state) and in [69] (teleportation circuit and GHZ), obtaining the same results as [72] and improving [69]. We also provide some examples showing how we lose precision in the presence of control flow, showing when our analysis is sound but incomplete.

5.5.1 Showing the analysis

Consider the example in Fig. 19. In Fig. 19, we show the CFG to the program

$$dGHZ ::= \mathtt{h}(a);\mathtt{cx}(a,b);\mathtt{cx}(a,c);\mathtt{cx}(c,b);\mathtt{t}(b);\mathtt{cx}(c,a);$$

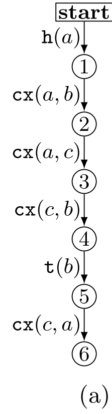
displaying for each program point the concrete state in blue and the abstract state in black. This program creates the GHZ states up to node **3**, then 'disentangles' b with the first cx, then changes the relative phase with the t gate, and then disentangles c with the last cx. In the abstract domain up to node **3**, we construct the state in which $\{a,b,c\}$ are d-inseparable. Then we are able to keep track in the abstract state the entanglement cancellation made by the cx gate in edges $(\mathbf{3},\mathtt{cx}(a,b),\mathbf{4})$ and $(\mathbf{5},\mathtt{cx}(c,a),\mathbf{6})$ and the phase change made by the t gate in edge $(\mathbf{4},\mathtt{t}(b),\mathbf{5})$. We show how our analysis works with control flow in Fig. 20. We consider the program

$$prog ::= \mathtt{h}(a);\mathtt{cx}(a,b);\mathtt{h}(c);\mathtt{if}\,(b)\,\mathtt{then}\,\{\mathtt{cx}(a,c)\}\ \mathtt{else}\,\{\mathtt{cx}(c,b)\};$$

writing $|\phi\rangle$ to indicate the state $1/\sqrt{2}(|0\rangle+|1\rangle)$ and $|\varphi\rangle$ to indicate $1/\sqrt{2}(|0\rangle-|1\rangle)$. In this example, we start in node **1** with a state where a and b form a Bell state and are therefore abstracted as being entangled d-inseparable. In nodes **2** and **3**, due to the measurement of b, both a and b collapse to a basis state. Since a and b are d-inseparable, they are both

[6] The following link https://github.com/NicolaAssolini98/EntaglementAnalysis contains our prototype implemented in Python.

Fig. 19 The $dGHZ$ CFG (a), and a table representing concrete and abstract states for each node (b)

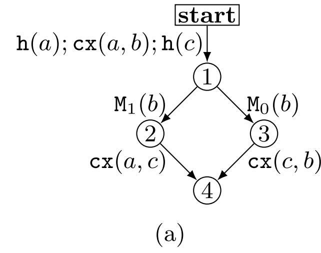


(a)

	Concrete state	Abstract state
Start	$\{\lvert 000\rangle_{a,b,c}\}$	$[(\{a\},0),(\{b\},1),(\{c\},2)],\{(0,1,2):Z\})$
1	$\{{}^1/\sqrt{2}(\lvert 0\rangle + \lvert 1\rangle)_a \lvert 00\rangle_{b,c}\}$	$[(\{a\},0),(\{b\},1),(\{c\},2)],\{0:X,(1,2):Z\})$
2	$\{{}^1/\sqrt{2}(\lvert 00\rangle + \lvert 11\rangle)_{a,b} \lvert 0\rangle_c\}$	$[(\{a,b\},0),(\{c\},2)],\{0:X,2:Z\})$
3	$\{{}^1/\sqrt{2}(\lvert 000\rangle + \lvert 111\rangle)_{a,b,c}\}$	$[(\{a,b,c\},0)],\{0:X\})$
4	$\{{}^1/\sqrt{2}(\lvert 00\rangle + \lvert 11\rangle)_{a,c} \lvert 0\rangle_b\}$	$[(\{a,c\},0),(\{b\},1)],\{0:X,1:Z\})$
5	$\{{}^{i+1}/\sqrt{2}(\lvert 00\rangle + \lvert 11\rangle)_{a,c} \lvert 0\rangle_b\}$	$[(\{a,c\},0),(\{b\},1)],\{0:P,1:Z\})$
6	$\{{}^{i+1}/\sqrt{2}(\lvert 0\rangle + \lvert 1\rangle)_c \lvert 0\rangle_b \lvert 0\rangle_c\}$	$[(\{c\},0),(\{b\},1),(\{a\},2)],\{0:P,(1,2):Z\})$

(b)

Fig. 20 The $prog$ CFG (a), and a table representing concrete and abstract states for each node (b), where $\lvert\phi\rangle = {}^1/\sqrt{2}(\lvert 0\rangle + \lvert 1\rangle)$ and $\lvert\varphi\rangle = {}^1/\sqrt{2}(\lvert 0\rangle - \lvert 1\rangle)$



(a)

	Concrete state	Abstract state
Start	$\{\lvert 000\rangle_{a,b,c}\}$	$[(\{a\},0),(\{b\},1),(\{c\},2)],\{(0,1,2):Z\})$
1	$\{{}^1/\sqrt{2}(\lvert 00\rangle+\lvert 11\rangle)_{a,b} \lvert\phi\rangle_c\}$	$[(\{a, b\},0),(\{c\},1)],\{0:X,1:Z\})$
2	$\{\lvert 11\phi\rangle_{a,b,c}\}$	$[(\{a\},0),(\{b\},1),(\{c\},2)],\{(0,1):Z,2:X\})$
3	$\{\lvert 00\phi\rangle_{a,b,c}\}$	$[(\{a\},0),(\{b\},1),(\{c\},2)],\{(0,1):Z,2:X\})$
4	$\{\lvert 11\varphi\rangle_{a,b,c}, (\lvert 0\rangle_a\, {}^1/\sqrt{2}\, \lvert 00\rangle+\lvert 11\rangle)_{a,b,c}\}$	$[(\{a\},0),(\{c\},1),(\{b\},1)],\{0:Z,1:\top_{\mathcal{L}}\})$

(b)

labelled with Z. In node **4**, the concrete state is obtained by merging the semantics of the two paths, while the abstract state is the lub between

$$CX^{\sharp}_{a,c}([(\{a\},0),(\{b\},1),(\{c\},2)],\{(0,1):Z,2:X\})$$

and

$$CX^{\sharp}_{c,b}([(\{a\},0),(\{b\},1),(\{c\},2)],\{(0,1):Z,2:X\}),$$

which correspond to:

$$[(\{a\},0),(\{b\},1),(\{c\},2)],$$
$$\{(0,1):Z,2:X\}) \sqcup ([(\{a\},0),(\{b,c\},1)],\{(0):Z,1:X\},$$

that is:

$$[(\{a\},0),(\{c\},1),(\{b\},1)],\{0:Z,1:\top_{\mathcal{L}}\}.$$

In both examples, using the labels, we can approximate the variable's state during the execution of the program.

6 Other related works

Here we highlight some work related to the analyses discussed in Sect. 4 that is not covered in Sect. 3. These works focus on synthesizing uncomputation and improving the efficiency of the resulting procedures, rather than on identifying which variables or qubits actually need to be uncomputed (i.e., discarded). Examples include [9, 62–65, 74, 80, 81]. Square [38] and `staq` [6] use uncomputation to reduce the number of ancilla qubits. Quipper [42], instead, automatically converts classical Haskell programs into quantum circuits and introduces uncomputation during this process. Similarly, Qunity [83] offers automatic uncomputation when compiling high-level controlled operations in circuits. Qrisp [76, 77] allows automatic uncomputation through an `@auto_uncompute` decorator, and manual uncomputation with an explicit `uncompute` function. Qrisp operates at a slightly higher level than circuit languages, offering an abstraction via a `QuantumVariable` class (and subclasses), which still represents sets of qubits and works positionally (e.g., `q = QuantumVariable(1); x(q)`). Finally, the recent language Quff [86], published at the same time as our

work [11], addresses a similar goal. Quff automatically triggers uncomputation whenever a quantum variable goes out of scope at runtime. Unlike our approach, however, it does not provide a formal definition of the method.

7 Conclusion

In this survey, we have examined how the non-classical characteristics of quantum computation shape the design of static analyses for resource optimization and property detection. We reviewed the main families of approaches proposed in the literature, distinguishing between flow-based analyses, which reason about the program structure (e.g., variable usage, linearity, uncomputation), and domain-based analyses, which rely on suitable abstract domains to approximate quantum phenomena such as entanglement. To illustrate these two perspectives in practice, we have presented two representative analyses: a flow-based technique for automatically managing temporary variables [11], and a domain-based approach for detecting entanglement [10, 12].

Overall, our survey highlights that static analysis can support quantum programming at multiple levels of abstraction and in different phases of the compilation pipeline [20, 89]. Moreover, the landscape of quantum programming languages remains highly heterogeneous [39], ranging from circuit-level languages such as Qiskit [4] and Cirq [29] to higher-level languages like Qrisp [77], Q# [79], Guppy [53, 54], or Quipper [43], which provide richer abstractions for circuit construction. This diversity reflects an ongoing uncertainty regarding the most suitable abstraction level for programming quantum computers [33]. In this context, static analysis can play a double role. First, it can provide formal guarantees that help make higher-level abstractions reliable, bridging the gap between expressive programming constructs and low-level quantum operations. Second, by revealing which abstractions admit effective analysis and optimization, it can inform the design of future quantum languages, contributing to the development of more ergonomic, analysable, and semantically principled programming models. As quantum software systems grow in scale and complexity, we expect static analysis to become increasingly influential in shaping both compiler and language design.

We identify domain-based reasoning as a major open challenge. Static verification of quantum programs is increasingly important given the high cost of execution and simulation, yet it remains difficult due to the exponential nature of quantum state spaces. While significant progress has been made, further advances in quantum-specific abstract domains will be essential to scale analyses to larger and more complex programs.

Funding information Open access funding provided by Università degli Studi di Verona within the CRUI-CARE Agreement.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Aaronson, S., Gottesman, D.: Improved simulation of stabilizer circuits. CoRR (2004). http://arxiv.org/abs/quant-ph/0406196. arXiv:quant-ph/0406196
2. Abdulla, P.A., Chen, Y.G., Chen, Y.F., Holík, L., Lengál, O., Lin, J.A., Lo, F.Y., Tsai, W.L.: Verifying quantum circuits with level-synchronized tree automata. Proc. ACM Program. Lang. **9**(POPL) (2025). https://doi.org/10.1145/3704868
3. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley, Reading (2006)
4. Aleksandrowicz, G., et al.: Qiskit: An Open-Source Framework for Quantum Computing (2019). https://doi.org/10.5281/zenodo.2562111
5. Amy, M.: Formal Methods in Quantum Circuit Design (2019)
6. Amy, M., Gheorghiu, V.: staq—a full-stack quantum processing toolkit. Quantum Sci. Technol. **5**(3), 034016 (2020)
7. Amy, M., Lunderville, J.: Linear and non-linear relational analyses for quantum program optimization. Proc. ACM Program. Lang. **9**(POPL) (2025). https://doi.org/10.1145/3704873
8. Amy, M., Maslov, D., Mosca, M.: Polynomial-time t-depth optimization of clifford+ t circuits via matroid partitioning. IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. **33**(10), 1476–1489 (2014)
9. Amy, M., Roetteler, M., Svore, K.M.: Verified compilation of space-efficient reversible circuits. In: Majumdar, R., Kuncak, V. (eds.) Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10427, pp. 3–21. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-63390-9_1
10. Assolini, N., Di Pierro, A., Mastroeni, I.: Abstracting entanglement. In: Arceri, V., Pasqua, M. (eds.) Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2024, Pasadena, CA, USA, 22 October 2024, pp. 34–41. ACM, New York (2024). https://doi.org/10.1145/3689609.3689998
11. Assolini, N., Di Pierro, A., Mastroeni, I.: Static analysis of quantum programs. In: Giacobazzi, R., Gorla, A. (eds.) Static Analysis - 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20-22, 2024, Proceedings. Lecture Notes in Computer Science, vol. 14995, pp. 1–25. Springer, Berlin (2024). https://doi.org/10.1007/978-3-031-74776-2_1
12. Assolini, N., Di Pierro, A., Mastroeni, I.: A static analysis of entanglement. In: Krishna, S., Sankaranarayanan, S., Trivedi, A. (eds.) Verification, Model Checking, and Abstract Interpretation - 26th International Conference, VMCAI 2025, Denver, CO, USA, January 20-21, 2025, Proceedings, Part II. Lecture Notes in Computer Science, vol. 15530, pp. 50–71. Springer, Berlin (2025). https://doi.org/10.1007/978-3-031-82703-7_3

13. Assolini, N., Marzari, L., Di Pierro, A., Mastroeni, I.: Formal verification of variational quantum circuits. CoRR (2025) https://doi.org/10.48550/arXiv.2507.10635. arXiv:2507.10635
14. Behler, J.A.C., Al-Ramadan, A.F., Baheri, B., Collard, M.L., Guan, Q., Maletic, J.I.: Static analysis and transformation for quantum programming languages. IEEE Softw. **42**(5), 58–64 (2025). https://doi.org/10.1109/MS.2025.3566634
15. Bennett, C.H., Wiesner, S.J.: Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. Phys. Rev. Lett. **69**, 2881–2884 (1992). https://doi.org/10.1103/PhysRevLett.69.2881
16. 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. PLDI 2020, pp. 286–300. Assoc. Comput. Mach., New York (2020). https://doi.org/10.1145/3385412.3386007
17. Bichsel, B., Paradis, A., Baader, M., Vechev, M.T.: Abstraqt: analysis of quantum circuits via abstract stabilizer simulation. Quantum **7**, 1185 (2023). https://doi.org/10.22331/Q-2023-11-20-1185
18. Boykin, P., Mor, T., Pulver, M., Roychowdhury, V., Vatan, F.: A new universal and fault-tolerant quantum basis. Inf. Process. Lett. **75**(3), 101–107 (2000). https://doi.org/10.1016/S0020-0190(00)00084-3. https://www.sciencedirect.com/science/article/pii/S0020019000000843
19. Bruni, R., Giacobazzi, R., Gori, R., Ranzato, F.: A correctness and incorrectness program logic. J. ACM **70**(2) (2023). https://doi.org/10.1145/3582267
20. Cardama, F.J., Vázquez-Pérez, J., Pena, T.F., Pichel, J.C., Gómez, A.: Quantum compilation process: a survey. In: Caíno-Lores, S., Zeinalipour, D., Doudali, T.D., Singh, D.E., Garzón, G.E.M., Sousa, L., Andrade, D., Cucinotta, T., D'Ambrosio, D., Diehl, P., Dolz, M.F., Jukan, A., Montella, R., Nardelli, M., Garcia-Gasulla, M., Neuwirth, S. (eds.) Euro-Par 2024: Parallel Processing Workshops - Euro-Par 2024 International Workshops, Madrid, Spain, August 26-30, 2024, Proceedings, Part I. Lecture Notes in Computer Science, vol. 15385, pp. 100–112. Springer, Berlin (2024). https://doi.org/10.1007/978-3-031-90200-0_9
21. Chareton, C., Bardin, S., Bobot, F., Perrelle, V., Valiron, B.: An automated deductive verification framework for circuit-building quantum programs. In: Yoshida, N. (ed.) Programming Languages and Systems. ESOP 2021. Lecture Notes in Computer Science, vol. 12648, pp. 156–182. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-72019-3_6
22. Chen, Y., Stade, Y.: Quantum constant propagation. In: Hermenegildo, M.V., Morales, J.F. (eds.) Static Analysis - 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14284, pp. 164–189. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-44245-2_9
23. Chen, Y., Chung, K., Lengál, O., Lin, J., Tsai, W.: Autoq: an automata-based quantum circuit verifier. In: Enea, C., Lal, A. (eds.) Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III. Lecture Notes in Computer Science, vol. 13966, pp. 139–153. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-37709-9_7
24. Chen, Y.F., Chung, K.M., Lengál, O., Lin, J.A., Tsai, W.L., Yen, D.D.: An automata-based framework for verification and bug hunting in quantum circuits. Proc. ACM Program. Lang. **7**(PLDI) (2023). https://doi.org/10.1145/3591270
25. Chen, Y., Fulginiti, I., Mendl, C.B.: Probabilistic circuit model. In: 2024 IEEE International Conference on Quantum Computing and Engineering (QCE), vol. 02, pp. 508–509 (2024). https://doi.org/10.1109/QCE60285.2024.10379
26. Chen, Y., Chung, K., Hsieh, M., Huang, W., Lengál, O., Lin, J., Tsai, W.: Autoq 2.0: from verification of quantum circuits to verification of quantum programs. In: Gurfinkel, A., Heule, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 31st International Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part III. Lecture Notes in Computer Science, vol. 15698, pp. 87–108. Springer, Berlin (2025). https://doi.org/10.1007/978-3-031-90660-2_5
27. Chen, Y., Fulginiti, I., Mendl, C.B.: Optimization framework for reducing mid-circuit measurements and resets. In: Paszynski, M., Barnard, A.S., Zhang, Y.J. (eds.) Proceedings, Part V, Computational Science - ICCS 2025 Workshops - 25th International Conference, Singapore, Singapore, July 7–9, 2025. Lecture Notes in Computer Science, vol. 15911, pp. 150–164. Springer, Berlin (2025). https://doi.org/10.1007/978-3-031-97570-7_13
28. Chen, Y., Mendl, C.B., Seidl, H.: Dead gate elimination. In: Lees, M.H., Cai, W., Cheong, S.A., Su, Y., Abramson, D., Dongarra, J.J., Sloot, P.M.A. (eds.) Proceedings, Part III, Computational Science - ICCS 2025 - 25th International Conference, Singapore, July 7–9, 2025. Lecture Notes in Computer Science, vol. 15905, pp. 135–150. Springer, Berlin (2025). https://doi.org/10.1007/978-3-031-97632-2_10
29. Cirq Developers: Cirq. https://doi.org/10.5281/zenodo.4062499. https://quantumai.google/cirq
30. Collard, M.L., Decker, M.J., Maletic, J.I.: Lightweight transformation and fact extraction with the srcml toolkit. In: 11th IEEE Working Conference on Source Code Analysis and Manipulation, SCAM 2011, Williamsburg, VA, USA, September 25-26, 2011, pp. 173–184. IEEE Comput. Soc., Los Alamitos (2011). https://doi.org/10.1109/SCAM.2011.19
31. Collard, M.L., Decker, M.J., Maletic, J.I.: srcML: an infrastructure for the exploration, analysis, and manipulation of source code: a tool demonstration. In: 2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, pp. 516–519. IEEE Comput. Soc., Los Alamitos (2013). https://doi.org/10.1109/ICSM.2013.85
32. Colledan, A., Dal Lago, U.: Flexible type-based resource estimation in quantum circuit description languages. Proc. ACM Program. Lang. **9**(POPL) (2025). https://doi.org/10.1145/3704883
33. Corrales-Garro, F., Valerio-Ramírez, D., Núñez-Corrales, S.: Is productivity in quantum programming equivalent to expressiveness? (2025). arXiv preprint. arXiv:2504.08876
34. 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. POPL '77, pp. 238–252. Assoc. Comput. Mach., New York (1977). https://doi.org/10.1145/512950.512973
35. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL '79, pp. 269–282. Assoc. Comput. Mach., New York (1979). https://doi.org/10.1145/567752.567778
36. Deutsch, D., Penrose, R.: Quantum theory, the church–Turing principle and the universal quantum computer. Proc. R. Soc. Lond. Ser. A, Math. Phys. Sci. **400**(1818), 97–117 (1985). https://doi.org/10.1098/rspa.1985.0070. https://royalsocietypublishing.org/doi/abs/10.1098/rspa.1985.0070
37. Di Pierro, A., Mancini, S., Memarzadeh, L., Mengoni, R.: Homological analysis of multi-qubit entanglement. Europhys. Lett. **123**(3), 30006 (2018). https://doi.org/10.1209/0295-5075/123/30006
38. Ding, Y., Wu, X., 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: 47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30

- June 3, 2020, pp. 570–583. IEEE, New York (2020). https://doi.org/10.1109/ISCA45697.2020.00054
39. Fürntratt, H., Schnabl, P., Krebs, F., Unterberger, R., Zeiner, H.: Towards higher abstraction levels in quantum computing. In: Monti, F., Plebani, P., Moha, N., Paik, H.Y., Barzen, J., Ramachandran, G., Bianchini, D., Tamburri, D.A., Mecella, M. (eds.) Service-Oriented Computing – ICSOC 2023 Workshops, pp. 162–173. Springer, Singapore (2024). https://doi.org/10.1007/978-981-97-0989-2_13
40. Giovannetti, V., Lloyd, S., Maccone, L.: Quantum random access memory. Phys. Rev. Lett. **100**, 160501 (2008). https://doi.org/10.1103/PhysRevLett.100.160501
41. Gottesman, D.: The Heisenberg representation of quantum computers (1998). arXiv preprint arXiv:quant-ph/9807006
42. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: An introduction to quantum programming in quipper. In: Dueck, G.W., Miller, D.M. (eds.) Reversible Computation - 5th International Conference, RC 2013, Victoria, BC, Canada, July 4-5, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7948, pp. 110–124. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-38986-3_10
43. 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. PLDI '13, pp. 333–342. Assoc. Comput. Mach., New York (2013). https://doi.org/10.1145/2491956.2462177
44. Hietala, K., Rand, R., Hung, S.H., Wu, X., Hicks, M.: A verified optimizer for quantum circuits. Proc. ACM Program. Lang. **5**(POPL) (2021). https://doi.org/10.1145/3434318
45. Hirata, K., Heunen, C.: Qurts: Automatic quantum uncomputation by affine types with lifetime. Proc. ACM Program. Lang. **9**(POPL) (2025). https://doi.org/10.1145/3704842
46. Honda, K.: Analysis of quantum entanglement in quantum programs using stabilizer formalism. In: Heunen, C., Selinger, P., Vicary, J. (eds.) Proceedings 12th International Workshop on Quantum Physics and Logic, QPL 2015, Oxford, UK, July 15-17, 2015. EPTCS, vol. 195, pp. 262–272 (2015). https://doi.org/10.4204/EPTCS.195.19
47. Ittah, D., Häner, T., Kliuchnikov, V., Hoefler, T.: Enabling dataflow optimization for quantum programs. CoRR (2021). https://arxiv.org/abs/2101.11030. arXiv:2101.11030
48. 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. CF '14. Assoc. Comput. Mach., New York (2014). https://doi.org/10.1145/2597917.2597939
49. Johnston, E., Harrigan, N., Gimeno-Segovia, M.: Programming Quantum Computers: Essential Algorithms and Code Samples. O'Reilly Media, Incorporated (2019)
50. Kaul, M., Küchler, A., Banse, C.: A uniform representation of classical and quantum source code for static code analysis. In: Cour, B.L., Yeh, L., Osinski, M. (eds.) IEEE International Conference on Quantum Computing and Engineering, QCE 2023, Bellevue, WA, USA, September 17-22, 2023, pp. 1013–1019. IEEE, Los Alamitos (2023). https://doi.org/10.1109/QCE57702.2023.00115
51. Khedker, U.P., Sanyal, A., Sathe, B.: Data Flow Analysis - Theory and Practice. CRC Press, Boca Raton (2009). http://www.crcpress.com/product/isbn/9780849328800
52. Kitaev, A.Y., Shen, A.H., Vyalyi, M.N.: Classical and Quantum Computation. Graduate Studies in Mathematics, vol. 47. Am. Math. Soc., Providence (2002). https://bookstore.ams.org/gsm-47/
53. Koch, M., Borgna, A., Roy, C., Lawrence, A., Singhal, K., Sivarajah, S., Duncan, R.: Imperative quantum programming with ownership and borrowing in guppy (2025). arXiv preprint arXiv:2510.13082
54. 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
55. Liu, J., Bello, L., Zhou, H.: Relaxed peephole optimization: a novel compiler optimization for quantum circuits. In: 2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 301–314 (2021). https://doi.org/10.1109/CGO51591.2021.9370310
56. Milburn, G.: Schrodinger's Machines: The Quantum Technology Reshaping Everyday Life. Holt, New York (1997)
57. Nasinski, M., Wall, A., Robson, S., Dash, P., Winick-Ng, J.: gridify: Enrich Figures and Tables with Custom Headers and Footers and More (2025). https://pharmaverse.github.io/gridify/, R package version 0.7.5
58. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information, 10th Anniversary edn. Cambridge University Press, Cambridge (2016). https://www.cambridge.org/de/academic/subjects/physics/quantum-physics-quantum-information-and-quantum-computation/quantum-computation-and-quantum-information-10th-anniversary-edition?format=HB
59. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Berlin (1999). https://doi.org/10.1007/978-3-662-03811-6
60. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (2019). https://doi.org/10.1145/3371078
61. Paltenghi, M., Pradel, M.: Analyzing quantum programs with lintq: a static analysis framework for qiskit. Proc. ACM Softw. Eng. **1**(FSE), 2144–2166 (2024). https://doi.org/10.1145/3660802
62. 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. PLDI 2021, pp. 222–236. Assoc. Comput. Mach., New York (2021). https://doi.org/10.1145/3453483.3454040
63. Paradis, A., Bichsel, B., Vechev, M.T.: Reqomp: space-constrained uncomputation for quantum circuits. Quantum **8**, 1258 (2024). https://doi.org/10.22331/Q-2024-02-19-1258
64. Parent, A., Roetteler, M., Svore, K.M.: Reversible circuit compilation with space constraints. CoRR (2015). http://arxiv.org/abs/1510.00377. arXiv:1510.00377
65. 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, Cham (2017)
66. Paykin, J., Rand, R., Zdancewic, S.: Qwire: a core language for quantum circuits. SIGPLAN Not. **52**(1), 846–858 (2017). https://doi.org/10.1145/3093333.3009894
67. 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. Assoc. Comput. Mach., New York (2022). https://doi.org/10.1145/3497776.3517772
68. Peng, Y., Ying, M., Wu, X.: Algebraic reasoning of quantum programs via non-idempotent Kleene algebra. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2022, pp. 657–670. Assoc. Comput. Mach., New York (2022). https://doi.org/10.1145/3519939.3523713
69. Perdrix, S.: Quantum entanglement analysis based on abstract interpretation. In: Alpuente, M., Vidal, G. (eds.) Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings. Lecture Notes in Computer Science, vol. 5079, pp. 270–282. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-69166-2_18

70. Raahauge, A.N., Marchioro, M.B., Nylandsted, R.R.: Approximating entanglement based on abstract interpretation (2025). arXiv preprint arXiv:2508.10056
71. Rand, R., Paykin, J., Lee, D., Zdancewic, S.: Reqwire: Reasoning about reversible quantum circuits, vol. 287 pp. 299–312 (2018). https://doi.org/10.4204/EPTCS.287.17
72. Rand, R., Sundaram, A., Singhal, K., Lackey, B.: Gottesman types for quantum programs, vol. 340 pp. 279–290 (2020). https://doi.org/10.4204/EPTCS.340.14
73. Rand, R., Sundaram, A., Singhal, K., Lackey, B.: Extending gottesman types beyond the Clifford group the Second International Workshop on Programming Languages for Quantum Computing. (PLanQC 2021) (2021)
74. Reichental, I., Alon, R., Preminger, L., Vax, M., Naveh, A.: Scalable memory recycling for large quantum programs (2025). arXiv preprint arXiv:2503.00822
75. Rovara, D., Burgholzer, L., Wille, R.: A framework for debugging quantum programs. In: 2025 IEEE International Conference on Quantum Software (QSW), pp. 130–136 (2025). https://doi.org/10.1109/QSW67625.2025.00024
76. Seidel, R., Tcholtchev, N., Bock, S., Hauswirth, M.: Uncomputation in the qrisp high-level quantum programming framework. In: Kutrib, M., Meyer, U. (eds.) Reversible Computation - 15th International Conference, RC 2023, Giessen, Germany, July 18-19, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13960, pp. 150–165. Springer, Berlin (2023). https://doi.org/10.1007/978-3-031-38100-3_11
77. Seidel, R., Bock, S., Zander, R., Petric, M., Steinmann, N., Tcholtchev, N., Hauswirth, M.: Qrisp: a framework for compilable high-level programming of gate-based quantum computers (2024). https://doi.org/10.48550/arXiv.2406.14792
78. Seidl, H., Wilhelm, R., Hack, S.: Compiler Design - Analysis and Transformation. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-17548-0
79. 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. Assoc. Comput. Mach., New York (2018). https://doi.org/10.1145/3183895.3183901
80. Tiwari, A., Sundaram, R.G., Gupta, H., Ramakrishnan, C., Yu, N.: Uncomputing ancilla qubits in quantum circuits. In: 2025 International Conference on Quantum Communications, Networking, and Computing (QCNC), pp. 379–387 (2025). https://doi.org/10.1109/QCNC64685.2025.00065
81. Vasconcelos, F., Gilyén, A.: Methods for reducing ancilla-overhead in block encodings (2025). arXiv preprint arXiv:2507.07900
82. Vidal, G.: Entanglement monotones. J. Mod. Opt. **47**(2–3), 355–376 (2000). https://doi.org/10.1080/09500340008244048. https://www.tandfonline.com/doi/abs/10.1080/09500340008244048
83. Voichick, F., Li, L., Rand, R., Hicks, M.: Qunity: a unified language for quantum and classical computing. Proc. ACM Program. Lang. **7**(POPL) (2023). https://doi.org/10.1145/3571225
84. Ward, M.: The closure operators of a lattice. Ann. Math. **43**(2), 191–196 (1942). https://doi.org/10.2307/1968865. http://www.jstor.org/stable/1968865
85. Winskel, G.: The Formal Semantics of Programming Languages - an Introduction. Foundation of Computing Series. MIT Press, Cambridge (1993)
86. Wright, C.J., Luján, M., Petoumenos, P., Goodacre, J.: Quff: a dynamically typed hybrid quantum-classical programming language. In: Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes. MPLR 2024, pp. 65–81. Assoc. Comput. Mach., New York (2024). https://doi.org/10.1145/3679007.3685063
87. Xia, S., Zhao, J.: Static entanglement analysis of quantum programs. In: 4th IEEE/ACM International Workshop on Quantum Software Engineering, Q-SE@ICSE 2023, Melbourne, Australia, May 17, 2023, pp. 42–49. IEEE, Los Alamitos (2023). https://doi.org/10.1109/Q-SE59154.2023.00013
88. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014, pp. 590–604. IEEE Comput. Soc., Los Alamitos (2014). https://doi.org/10.1109/SP.2014.44
89. Yan, G., Wu, W., Yuheng, C., Pan, K., Lu, X., Zixiang, Z., Yuhan, W., Wang, R., Yan, J.: Quantum circuit synthesis and compilation optimization: Overview and prospects. CoRR (2024) https://doi.org/10.48550/arXiv.2407.00736. arXiv:2407.00736
90. Ying, M.: Floyd–hoare logic for quantum programs. ACM Trans. Program. Lang. Syst. **33**(6) (2012). https://doi.org/10.1145/2049706.2049708
91. Ying, M.: Foundations of Quantum Programming. Morgan Kaufmann, San Mateo (2016)
92. Yu, N., Palsberg, J.: Quantum abstract interpretation. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. PLDI 2021, pp. 542–558. Assoc. Comput. Mach., New York (2021). https://doi.org/10.1145/3453483.3454061
93. Yu, N., Palsberg, J., Reps, T.: SAQR-QC: a logic for scalable but approximate quantitative reasoning about quantum circuits. CoRR (2025) https://doi.org/10.48550/arXiv.2507.13635. arXiv:2507.13635
94. Yuan, C., McNally, C., Carbin, M.: Twist: sound reasoning for purity and entanglement in quantum programs. Proc. ACM Program. Lang. **6**(POPL) (2022). https://doi.org/10.1145/3498691
95. Zhang, Z., Ying, M.: Quantum register machine: Efficient implementation of quantum recursive programs. Proc. ACM Program. Lang. **9**(PLDI) (2025). https://doi.org/10.1145/3729283
96. Zhao, P., Zhao, J., Ma, L.: Identifying bug patterns in quantum programs. In: 2nd IEEE/ACM International Workshop on Quantum Software Engineering, Q-SE@ICSE 2021, Madrid, Spain, June 1-2, 2021, pp. 16–21. IEEE, New York (2021). https://doi.org/10.1109/Q-SE52541.2021.00011
97. Zhao, P., Zhao, J., Miao, Z., Lan, S.: Bugs4q: a benchmark of real bugs for quantum programs. In: 36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021, pp. 1373–1376. IEEE, Los Alamitos (2021). https://doi.org/10.1109/ASE51524.2021.9678908
98. Zhao, P., Wu, X., Li, Z., Zhao, J.: Qchecker: detecting bugs in quantum programs via static analysis. In: 4th IEEE/ACM International Workshop on Quantum Software Engineering, Q-SE@ICSE 2023, Melbourne, Australia, May 17, 2023, pp. 50–57. IEEE, Los Alamitos (2023). https://doi.org/10.1109/Q-SE59154.2023.00014

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.