HUGR: A Quantum-Classical Intermediate Representation

Mark Koch
Agustín Borgna
Seyon Sivarajah
Alan Lawrence
Alec Edgington
Douglas Wilson
Ross Duncan
Craig Roy
Luca Mondada
Lukas Heidemann
Quantinuum
Cambridge, United Kingdom

Abstract

We introduce the Hierarchical Unified Graph Representation (HUGR): a novel graph based intermediate representation for mixed quantum-classical programs. HUGR's design features high expressivity and extensibility to capture the capabilities of near-term and forthcoming quantum computing devices, as well as new and evolving abstractions from novel quantum programming paradigms. The graph based structure is machine-friendly and supports powerful pattern matching based compilation techniques. Inspired by MLIR, HUGR's extensibility further allows compilation tooling to reason about programs at multiple levels of abstraction, lowering smoothly between them. Safety guarantees in the structure including strict, static typing and linear quantum types allow rapid development of compilation tooling without fear of program invalidation. A full specification of HUGR and reference implementation are open-source and available online.

1 Introduction

Modern applications of quantum computers usually involve both quantum and classical processors interacting with each other. In particular, there is an increasing interest in algorithms that require classical decision making *during* the execution of a circuit (i.e. within the coherence time of its qubits). For example, recently demonstrated repeat-until-success protocols [3, 22] and other algorithms use classical control-flow conditioned on mid-circuit measurements to determine which quantum operations should be applied next [9, 14, 21]. Going beyond that, quantum error correction algorithms, for example, might require even more complex classical logic to decode errors in real-time and apply corrections to the quantum state [24, 26].

Enabling this tight integration between the quantum and classical processor requires dedicated support throughout the entire quantum software stack. In particular, as previously argued in [3, 10, 19], there is a need for an intermediate representation (IR) for quantum programs that natively captures these classical operations, going beyond the traditional circuit picture. To this end, we introduce the Hierarchical Unified Graph Representation (HUGR), a novel quantum IR that can efficiently express, reason about, and optimise these hybrid quantum-classical programs in a unified graph structure. Its design is guided by the following main principles:

Expressivity. HUGR captures the various computational requirements of quantum algorithms in one unified framework. It can express everything from traditional, static (possibly parameterised) circuits and hybrid quantum-classical optimisation loops up to the real-time quantum classical logic described above.

Machine-friendliness. As an intermediate representation, HUGR is designed to be efficiently consumable and manipulable by software. We do not expect end users to read or write HUGR directly. Instead, they should rely on frontends like higher-level programming languages or libraries that compile to HUGR (see Section 4.2 for a more detailed explanation of this workflow).

Abstraction. Quantum algorithms are usually built up from multiple layers of abstraction, starting from some problem domain (say, a Hamiltonian describing a chemical system) that is then continuously lowered (for example by synthesising oracle circuits, inserting error mitigation steps, etc.). HUGR is designed to faithfully capture this staged lowering of abstraction levels, allowing compilers to exploit the unique opportunities for optimisation available at each step.

Extensibility. HUGR follows a modular design where new operations and data types can be added on the fly, comparable to the dialect system in the MLIR compiler tool chain [16]. This allows third parties to define their own bespoke abstractions and lowering routines that seamlessly compose with other components and passes.

Optimisability. HUGR is designed to enable efficient optimisation of quantum programs, both within and across the quantum-classical boundary. On top of that, we provide efficient routines for matching and rewriting of patterns within HUGR programs that third parties can hook into to define their own domain-specific optimisation routines (see Section 3 for details).

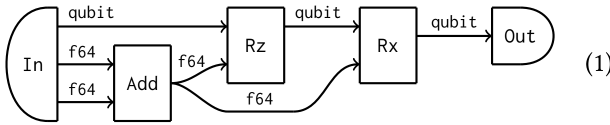
The full specification and reference implementation of HUGR are open-source and available at `github.com/CQCL/hugr`.

Authors' contact email: {name}.{surname}@quantinuum.com

2 The Hierarchical Unified Graph Representation

2.1 Quantum Programs as Dataflow Graphs

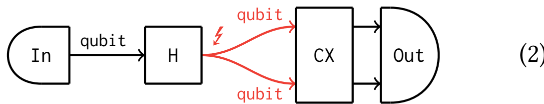
Quantum compiler architectures usually represent quantum circuits in the form of directed acyclic graphs (DAGs) where nodes are quantum gates and edges describe the qubit dependencies between them. HUGR generalises this model by encoding both quantum and classical operations in the same DAG structure. Concretely, HUGR represents programs via dataflow graphs spanned between an input and output node, where edges can carry either qubits or arbitrary classical data. The nodes correspond to quantum or classical processes that act on these values and produce some outputs that can be fed to the following nodes in the graph:



(1)

The graph above describes a program that applies an R_Z and an R_X rotation gate to an input qubit, where the rotation angle is dynamically computed as the sum of two floats that are given as additional inputs. The edges in HUGR are statically typed and node operations have a static signature (for example, the `Rz` operation above has the signature `qubit`, `f64` $\to$ `qubit`). We ensure that programs are well-typed by only allowing edges that match up with the operation signatures. Note that we leave out type annotations in the following examples if they can be inferred from context.

The inputs and outputs of HUGR nodes are explicitly ordered, corresponding to numbered *ports* on the nodes. For example, in Graph 1 the `qubit` edge is connected to the first input port of the `Rz` node, whereas the `f64` edge is connected to its second port. The `Add` node only has a single output port that is wired to both the `Rz` and `Rx` node. This is valid since classical values can be copied and thus used multiple times. However, the same is not true for quantum values such as qubits:



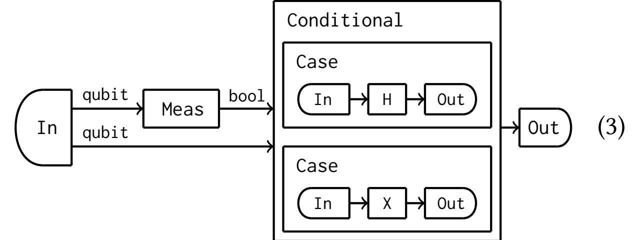
(2)

This program would not be physically realisable since the control and target of the `CX` gate act on the same qubit. To rule out mistakes like these, HUGR ensures that ports corresponding to qubits have *exactly one* connected edge, so Graph 2 is rejected. This corresponds to treating `qubit` as a *linear type* [27].

HUGR continuously enforces these typing and linearity constraints in between optimisation steps, thus preventing optimisation routines from erroneously invalidating programs.

2.2 Control Flow & Hierarchy

The dataflow representation described above requires additional primitives to represent control flow in a program. HUGR defines a set of node operations to express structured control flow. Given multiple execution graphs, a `Conditional` operation is able to branch between them based on a control input:



(3)

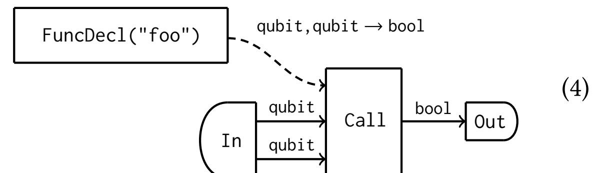
Here, the first qubit is measured and depending on the outcome either an `H` or an `X` gate is applied to the second qubit, which is then outputted. Note that the `Conditional` node has two `Case` *child nodes* that themselves contain children forming a nested dataflow graph. This highlights another core feature of HUGR: Graphs are *hierarchical* in the sense that each node can itself contain a nested child graph. This allows hierarchical nodes like the ones shown in Graph 3 to be nested arbitrarily deeply. Besides `Conditional`, HUGR also offers a `TailLoop` primitive to describe structured looping of a child dataflow graph (see Appendix A.1 for an example).

In the spirit of supporting varying levels of abstraction, HUGR also allows users to specify control flow in non-structured ways via arbitrary control-flow graphs. Control-flow graphs are expressed via the same hierarchical structure: `BasicBlock` nodes contain child graphs specifying the logic of each block and are then wired together inside a parent `CFG` node (see Appendix A.2 for an example). These graphs can be converted to the aforementioned structured primitives [1], or used in lowering stages when targeting CFG based representations like LLVM.

2.3 Functions and higher order types

Classical programs are often structured as collections of functions in a namespace, with a defined entry point and internal calls between them. HUGR provides operations for defining and calling functions, supported by the hierarchical structure presented in Section 2.2.

Functions can either be defined as a dataflow graph inside a `FuncDef` node, or be declared as an external reference with a `FuncDecl`. In the latter case, it is assumed that program will be linked with a definition of the function at a later stage.



(4)

In the example above, we declare an external function `foo` with signature `qubit`, `qubit` $\longrightarrow$ `bool` and pass it as an argument to a `Call` node, which executes it on some input values. The wire connecting the declaration to the `Call` is a special *constant edge* (represented by a dashed line in Graph 4) that denotes compile-time static values. A `LoadFunction` node may be used to turn such a static function value into a dynamic runtime value that can be passed around in the dataflow graph. Combined with the control flow primitives, this allows for the definition of higher-order functions that take functions as arguments or return functions as results.

While runtime function values must have a fixed signature, the static function definitions may have polymorphic signatures. That is, the input and output type definitions may include type variables that can be instantiated with user-defined types. The concrete signature for the function is only determined at the call site.

2.4 Extensibility

The HUGR representation is designed to be extensible, allowing users to define new operations and data types specific to their needs. In the examples presented so far, we have used a set of quantum and classical operations that are included as part of a standard library for the HUGR representation. Since the definitions are not hard-coded into the representation, users are free to mix and replace them with specialised operations relevant to their domain. For example, quantum abstractions like quantum control, multiplexed unitaries, uncomputation, etc. can all be captured inside the extension system and do not need to be baked into HUGR.

This design choice allows the core HUGR representation to remain agnostic to the operations being modelled. A program may be defined using the instruction set of a specific quantum device, and tooling that does not have access to its definition will still be able to reason about the program structure and perform optimisations that are agnostic to the operation semantics. It is the responsibility of the user to implement lowering routines or rewrite rules that define the behaviour of the new operations, but these are not required to be shared with the core HUGR implementation.

3 Optimisation

The HUGR representation is particularly well suited for pattern-matching based optimisation. This is a common technique in classical compiler design where small subgraphs are identified and replaced with more efficient or simpler ones. In contrast to pass-based optimisation, where the entire program is traversed and transformed in multiple iterations, pattern matching allows for efficient composition of rewrite rules and facilitates parallelisation.

The port labels on the nodes of a HUGR provide extra structure to the graph which enables much more efficient matching than generic subgraph isomorphism checks. Additionally, the incorporation of linear types, which are prevalent in most quantum operations represented in HUGR, guarantees that the majority of ports have a single connected edge. Finally, the structured control-flow primitives reduce the complexity when defining patterns on branching operations. These properties combined allow us to compile sets of patterns into matching structures which are able to efficiently search for tens of thousands of patterns simultaneously [20].

The operation and type extension framework described in Section 2.4 ensures that optimisation routines must always be aware of potentially unknown operations within the graph. This guarantees that all rewrite implementations are robust against new operations being added to the graph and that user-defined routines capable of reasoning about their domain-specifics can be safely composed with extension-agnostic ones.

4 Discussion

4.1 Related work

Traditional frameworks. Most traditional quantum compiler frameworks like Cirq [7], Pennylane [2], TKET [25], and Qiskit [12] internally represent quantum circuits as lists or graphs of gates and use OpenQASM 2 [6] as a common low-level assembly format for circuits. Support for dynamic quantum-classical programs tends to be fairly limited in these frameworks and usually relies on unrolling of all control-flow. OpenQASM 3 [5] was introduced to naively handle these classical operations, however it mainly serves as a high-level programming language rather than an intermediate representation.

QIR. The Quantum Intermediate Representation (QIR) [23] is arguably the most well-know standalone IR for quantum programs. It is based on the LLVM IR [15] and leverages the existing mature compiler infrastructure of the LLVM project. QIR is designed to be hardware-agnostic and as such offers a notion of *profiles* to specify the capabilities offered by a given device. In particular, there is ongoing work [4] to define a profile for QIR programs that captures the real-time classical operations and branching demonstrated in [3, 17].

Compared to HUGR, QIR is a more low-level representation where qubits are treated like opaque pointers and quantum operations are side-effectful opaque functions. This means that optimisers like that in [18] need to rely on global dataflow analyses to track qubits as opposed to the simpler graph-based matching available in HUGR (see Section 3). Furthermore, QIR is not easily extensible: while it provides some built-in quantum features like controlled and adjoint operations, there is currently no way to add custom higher-level abstractions.

MLIR. QIR's lack of customisability is at least in part due to the rigidness of LLVM's IR which was mainly designed

for C-like languages. The MLIR project [16] aims to address LLVM's drawbacks by introducing an IR with a dialect system that allows users to define their own domain-specific abstractions. This design served as a major inspiration for the extension system in HUGR.

While MLIR was initially used in the context of heterogeneous computing and machine learning, it has since also been applied to the quantum domain. MLIR based quantum dialects are used in QIRO [10], QCor [19], and the Catalyst compiler [11]. In fact, MLIR is expressive enough such that HUGR itself can be implemented as one of its dialects, which would make it compatible with the broader MLIR ecosystem. A prototype of such a dialect with conversions to and from the reference implementation has already been developed.[1]

However, we decided to keep the reference implementation of HUGR independent of MLIR for a few key reasons. First, MLIR is still under rapid development, and as such not fully stable and mature. Furthermore, as part of our mission-critical stack, we want to leverage the memory safety provided by Rust and avoid being tied to MLIR's C++ implementation. Finally, maintaining our own implementation allows us to focus on features that are particularly relevant to the quantum domain. For example, linear types are a first-class concept in HUGR whereas representing them in MLIR would require a more complicated setup and checks that would feel less natural.

4.2 Ecosystem

As a flexible intermediate representation, HUGR is suitable for generation, transformation, and consumption by a range of third-party front-ends, compilers, and back-ends respectively. A number of sister projects highlight some of the design choices made in HUGR, and demonstrate full workflows that employ it:

Guppy. Workflows involving quantum processors or simulators typically include Python at some level, especially at the program definition stage. Guppy[2] [13] is an embedded domain specific language (EDSL) within Python as a host language, with the goal of offering users powerful quantum programming abstractions with the familiarity of a Python working environment. Crucially, it allows existing Python code-bases to add Guppy for quantum programming without having to port any classical logic.

Guppy features Pythonic syntax, but is not an interpreted language; it compiles to HUGR. Quantum programs defined in Guppy can include arbitrary classical control flow and logic, which HUGR can represent natively. HUGR's expressive extension and type systems can capture abstract structures in user-programs, allowing powerful reasoning over these structures at compilation time.

[1]Available at `github.com/CQCL/hugr-mlir`.

[2]Available at `github.com/CQCL/guppylang`.

BRAT. Functional programming offers an alternative paradigm for quantum programming. The experimental BRAT language[3] [8] offers pure-functional quantum-classical programs, dependent typing for quantum resource management and unique compositional syntax for circuit building. BRAT also compiles to HUGR, allowing a shared tool-chain for quantum compilation despite offering a very different programming experience to Guppy.

TKET2. Version 2 of the TKET compiler[4] [25] is designed to optimise quantum programs represented as HUGRs. It includes quantum-specific HUGR extensions: an example of non-core tooling defining extension semantics. The compiler also includes implementations of the powerful optimisation techniques discussed in Section 3, enabled by the HUGR representation. TKET2 performs HUGR to HUGR transformations, making the optimisations highly re-usable across front-ends and back-ends.

HUGR-LLVM. Next-generation quantum targets are increasingly using LLVM based tool-chains, QIR compatible targets being good examples. To this end, the HUGR-LLVM project[5] enables lowering of HUGR to LLVM, in an extensible manner to allow HUGR extension operations to also be lowered. This will be used to compile HUGR to execute on Quantinuum ion-trap quantum processors.

References

[1] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. Perfect reconstructability of control flow from demand dependence graphs. *ACM Trans. Archit. Code Optim.*, 11(4), 2015. doi:10.1145/2693261.

[2] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M. Sohaib Alam, Guillermo Alonso-Linaje, B. AkashNarayanan, Ali Asadi, Juan Miguel Arrazola, Utkarsh Azad, Sam Banning, Carsten Blank, Thomas R Bromley, Benjamin A. Cordier, Jack Ceroni, Alain Delgado, Olivia Di Matteo, Amintor Dusko, Tanya Garg, Diego Guala, Anthony Hayes, Ryan Hill, Aroosa Ijaz, Theodor Isacsson, David Ittah, Soran Jahangiri, Prateek Jain, Edward Jiang, Ankit Khandelwal, Korbinian Kottmann, Robert A. Lang, Christina Lee, Thomas Loke, Angus Lowe, Keri McKiernan, Johannes Jakob Meyer, J. A. Montañez-Barrera, Romain Moyard, Zeyue Niu, Lee James O'Riordan, Steven Oud, Ashish Panigrahi, Chae-Yeun Park, Daniel Polatajko, Nicolás Quesada, Chase Roberts, Nahum Sá, Isidor Schoch, Borun Shi, Shuli Shu, Sukin Sim, Arshpreet Singh, Ingrid Strandberg, Jay Soni, Antal Száva, Slimane Thabet, Rodrigo A. Vargas-Hernández, Trevor Vincent, Nicola Vitucci, Maurice Weber, David Wierichs, Roeland Wiersema, Moritz Willmann, Vincent Wong, Shaoming Zhang, and Nathan Killoran. Pennylane: Automatic differentiation of hybrid quantum-classical computations, 2022. URL: https://arxiv.org/abs/1811.04968, arXiv:1811.04968.

[3] Natalie C. Brown, John Peter Campora III, Cassandra Granade, Bettina Heim, Stefan Wernli, Ciaran Ryan-Anderson, Dominic Lucchetti,

[3]Available at `github.com/CQCL/brat`.

[4]Available at `github.com/CQCL/tket2`.

[5]Available at `github.com/CQCL/hugr-llvm`.

Adam Paetznick, Martin Roetteler, Krysta Svore, and Alex Chernoguzov. Advances in compilation for quantum hardware – a demonstration of magic state distillation and repeat-until-success protocols, 2023. URL: https://arxiv.org/abs/2310.12106, arXiv:2310.12106.

[4] J. P. Campora III. *Specification for the QIR Adaptive Profile*. QIR Alliance, 2023. Work in progress. URL: https://github.com/qir-alliance/qir-spec/pull/35.

[5] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. Openqasm3: A broader and deeper quantum assembly language. *ACM Transactions on Quantum Computing*, 3(3), sep 2022. doi:10.1145/3505636.

[6] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017. URL: https://arxiv.org/abs/1707.03429, arXiv:1707.03429.

[7] Cirq Developers. Cirq, May 2024. doi:10.5281/zenodo.11398048.

[8] Ross Duncan, Mark Koch, Alan Lawrence, Connor McBridge, and Craig Roy. Introducing Brat. In *Fourth International Workshop on Programming Languages for Quantum Computing (PLanQC '24)*, January 2024. URL: https://popl24.sigplan.org/details/planqc-2024-papers/9/Introducing-BRAT.

[9] Robert B. Griffiths and Chi-Sheng Niu. Semiclassical fourier transform for quantum computation. *Phys. Rev. Lett.*, 76:3228–3231, Apr 1996. URL: https://link.aps.org/doi/10.1103/PhysRevLett.76.3228, doi:10.1103/PhysRevLett.76.3228.

[10] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. Qiro: A static single assignment-based quantum program representation for optimization. *ACM Transactions on Quantum Computing*, 3(3), jun 2022. doi:10.1145/3491247.

[11] Josh Izaac. Introducing catalyst: quantum just-in-time compilation, 2023. URL: https://pennylane.ai/blog/2023/03/introducing-catalyst-quantum-just-in-time-compilation/.

[12] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024. arXiv:2405.08810, doi:10.48550/arXiv.2405.08810.

[13] Mark Koch, Alan Lawrence, Kartik Singhal, Seyon Sivarajah, and Ross Duncan. GUPPY: Pythonic quantum-classical programming. In *Fourth International Workshop on Programming Languages for Quantum Computing (PLanQC '24)*, January 2024. URL: https://ks.cs.uchicago.edu/publication/guppy-planqc/.

[14] Greg Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM Journal on Computing*, 35(1):170–188, 2005. arXiv:https://doi.org/10.1137/S0097539703436345, doi:10.1137/S0097539703436345.

[15] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, page 75, USA, 2004. IEEE Computer Society.

[16] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, page 2–14. IEEE Press, 2021. doi:10.1109/CGO51591.2021.9370308.

[17] Thomas Lubinski, Cassandra Granade, Amos Anderson, Alan Geller, Martin Roetteler, Andrei Petrenko, and Bettina Heim. Advancing hybrid quantum–classical computation with real-time execution. *Frontiers in Physics*, 10, 2022. URL: https://www.frontiersin.org/journals/physics/articles/10.3389/fphy.2022.940293, doi:10.3389/fphy.2022.940293.

[18] Junjie Luo, Haoyu Zhang, and Jianjun Zhao. Dataflow-based optimization for quantum intermediate representation programs, 2024. URL: https://arxiv.org/abs/2406.19592, arXiv:2406.19592.

[19] Alexander McCaskey and Thien Nguyen. A mlir dialect for quantum assembly languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pages 255–264, 2021. doi:10.1109/QCE52317.2021.00043.

[20] Luca Mondada and Pablo Andrés-Martínez. Scalable pattern matching in computation graphs, 2024. URL: https://arxiv.org/abs/2402.13065, arXiv:2402.13065.

[21] Maris Ozols, Martin Roetteler, and Jérémie Roland. Quantum rejection sampling. *ACM Trans. Comput. Theory*, 5(3), aug 2013. doi:10.1145/2493252.2493256.

[22] Adam Paetznick and Krysta Svore. Repeat-until-success: Non-deterministic decomposition of single-qubit unitaries. *Quantum Information and Computation*, 14, 11 2013. doi:10.26421/QIC14.15-16-2.

[23] QIR Alliance. *QIR Specification*. QIR Alliance, 2021. https://qir-alliance.org. URL: https://github.com/qir-alliance/qir-spec.

[24] Peter W. Shor. Scheme for reducing decoherence in quantum computer memory. *Phys. Rev. A*, 52:R2493–R2496, Oct 1995. URL: https://link.aps.org/doi/10.1103/PhysRevA.52.R2493, doi:10.1103/PhysRevA.52.R2493.

[25] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: a retargetable compiler for nisq devices. *Quantum Science and Technology*, 6(1):014003, November 2020. URL: http://dx.doi.org/10.1088/2058-9565/ab8e92, doi:10.1088/2058-9565/ab8e92.

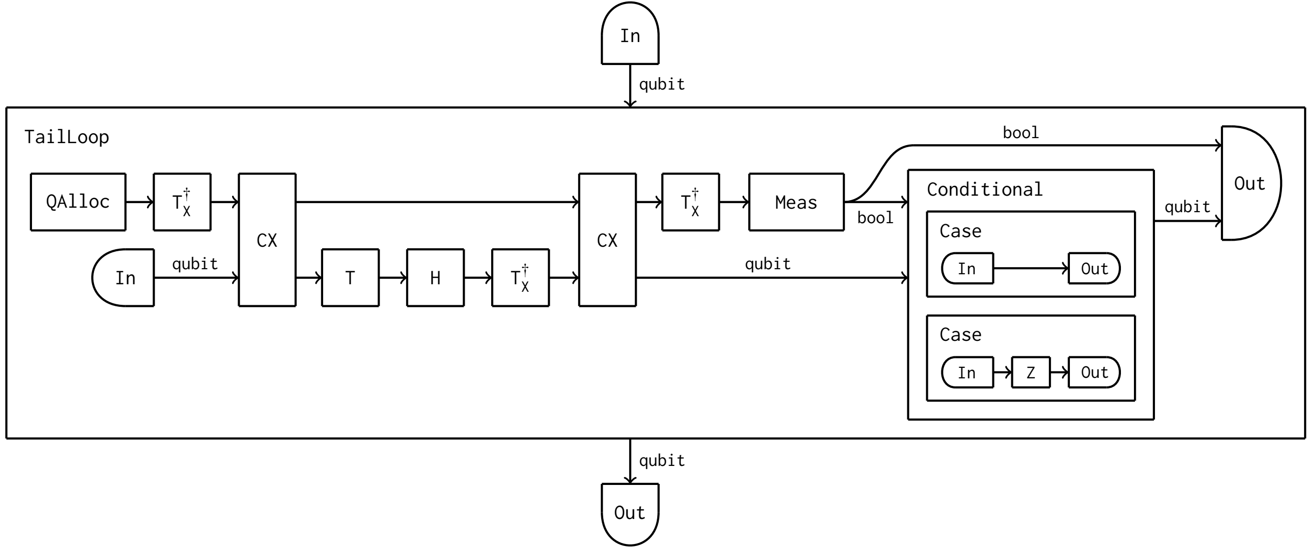
[26] A. M. Steane. Simple quantum error-correcting codes. *Phys. Rev. A*, 54:4741–4751, Dec 1996. URL: https://link.aps.org/doi/10.1103/PhysRevA.54.4741, doi:10.1103/PhysRevA.54.4741.

[27] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*, 1990. URL: https://api.semanticscholar.org/CorpusID:58535510.

A Additional Examples

A.1 Tail Loops

Below is an example HUGR program that uses a `TailLoop` node to implement a $(I + i\sqrt{2}X)/\sqrt{3}$ operation on its input qubit using the repeat-until-success scheme from [22, Fig. 8]. The first `bool` output of the loop body controls whether another loop iteration should be performed, which is the case if the measurement returned `false`. The `Conditional` node is used to apply an additional `Z` correction in that case. Note that the $\mathtt{T}_\mathtt{X}^\dagger$ nodes stand for the $HT^\dagger H \approx R_X(-\pi/4)$ gate.



A.2 Control Flow Graphs

The graph below implements the same program as in Appendix A.1 using a control-flow graph instead of a `TailLoop` node. The dotted edges between the basic blocks describe control flow instead of dataflow and thus are not required to be acyclic.