

Scaling Quantum Compilation with Graph Rewriting



Luca Mondada
Lady Margaret Hall
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Hilary 2025

Abstract

As the capabilities of quantum hardware advance and their architectures become more complex, quantum compilers must evolve to optimise increasingly large programs and support a growing range of computational primitives. In particular, the emergence of hybrid quantum-classical computations, required for instance by implementations of quantum error correcting protocols, poses significant challenges to established quantum compilation techniques.

This thesis argues that graph transformation systems (GTS) offer a principled foundation for compiler platforms that can express arbitrary hardware primitives and support computations over both quantum and classical data. To this end, it introduces a graph-based intermediate representation (IR), with support for linear types to model quantum data. This unifies graphical formalisms used for reasoning over quantum computations (such as the ZX calculus) with IR-based program transformation techniques from classical compiler design.

Building upon this foundation, the thesis tackles two critical scaling problems hindering the adoption of GTS in quantum compilers. First, it presents an efficient pattern matching algorithm based on a precomputed data structure that achieves query times independent of the number of transformation rules in the system. This removes a key bottleneck in quantum superoptimisers, which optimise quantum programs using tens to hundreds of thousands of rules.

Second, the thesis introduces a confluent persistent data structure that enables efficient exploration of the GTS state space of possible graph transformations. This factorised search space offers an exponential complexity advantage in search space size and traversal time compared to naively exploring all reachable graphs. The thesis also discusses the problem of extracting optimal programs from this data structure, relating it to Boolean satisfiability (SAT) problems.

Together, these contributions lay the groundwork for scalable, GTS-based quantum compilers and advance the integration of quantum and classical compilation techniques. The persistent data structure for graph rewriting also opens the door for concurrent graph rewriting in distributed systems, a technique that may have applications within the graph transformations field more broadly and merits further research.

To my family who have supported me throughout these years. I couldn't have done it without you.

To my friends and colleagues that made this adventure an unforgettable journey. Thank you to Aleks and the entire group in Oxford.

A special mention goes to Ross and the amazing crew at Quantinuum: Dan, Pablo and Thomas – thank you for proofreading – as well as Alan, Agustin, Alec, Callum, Craig, Doug, Ian, Lukas, Mark, Richie, Silas, Seyon, Will, Yao, and so many more! I look forward to continuing this incredible undertaking with you all.

This document is also available as HTML [here](#). If you are reading this document on a screen, you might find the experience much more pleasant in the browser (you can hover on citations, resize the window to your liking, search the document and more).

Contents

Chapter 1. Introduction	7
1. A new compilation regime	9
2. Contributions and thesis outline	12
Chapter 2. Quantum Computing: a Computer Scientist's Perspective	17
1. Foundations of quantum computing	17
2. Quantum circuit optimisation: a review	22
3. Rise of hybrid quantum-classical computation	29
4. Quantum compilers cannot do it alone	38
5. Summary and further reading	42
Chapter 3. Quantum Compilation as a Graph Transformation Problem	45
1. Related work	46
2. Computation graphs and linearity	48
3. A graph representation for quantum programs	49
4. Graph transformation in minIR	65
5. MinIR rewriting, operationally	70
Chapter 4. Pattern Matching in large Graph Transformation Systems	83
1. Related work	84
2. Preliminaries and simplifying assumptions	85
3. Tree reduction	90
4. Canonicalising the tree reduction	93
5. Enumerating all path-split graphs	97
6. An automaton for multi-pattern matching	105
7. Benchmarks	109
Chapter 5. Fully and Confluently Persistent Graph Rewriting	113
1. Related work	115
2. A closer look at equality saturation	117
3. The data structure	123
4. Exploration and extraction	131
5. Bounding the search space size	137
Chapter 6. Future Work and Conclusions	147
1. More expressive pattern matching	149
2. Massively parallel graph rewriting	150
Appendix	
1. Prefix trees	153
2. Lower bound on the number of patterns	154
Bibliography	156

Introduction

Quantum computing is the computational model that arises from the quantum mechanical manipulation of finite dimensional physical systems. Realising this new computing paradigm requires an entirely new technology stack: most obviously, new dedicated hardware, but also an extensive collection of software tools that transform the intents of a human user into a symphony of electric pulses that operate all components of the hardware installation (lasers, magnetic fields, currents, photodetectors, etc.).

Turning human-readable code into machine instructions is the realm of *compilers*, a problem as old as classical¹ computer science itself. By analogy, the same problem in the quantum world was named **quantum compilation**.

Interestingly, whereas the term quantum compilation has been in use for the longest part of the existence of quantum computing as a field, it is only recently that the quantum compilation community has started to adopt tools, ideas and results from our classical counterparts.

Meanwhile, quantum computing has a long history of adopting diagrammatic and graph-based representations to model and reason about computations and their quantum mechanical properties. The most famous example of this is undoubtedly the *quantum circuit*, a quantum analogue to boolean circuits that visualises how data flows from one operation to the next (cf. section 2.1).

Going beyond circuits, the field of categorical quantum mechanics has embraced and extended diagrammatic formalisms to model a variety of quantum processes and computations. Particularly noteworthy in this line of work are the numerous advances in quantum circuit optimisation (e.g. [119, 152]), quantum simulations (e.g. [186, 303]), error correction (e.g. [43, 97]) and many more related subjects (e.g. [293, 133]) that the family of ZX-like calculi have enabled in the last five years alone.

A challenge in quantum compilation has been to combine the principled and abstract graph-based transformation semantics of diagrammatic reasoning with the feature set and performance requirements of practical compilation tools. General purpose tools graph rewriting tools such as Qantomatic [129] proved too slow

1. To distinguish traditional computing from *quantum* computing, the field refers to the former as *classical* computing. We will adopt this term throughout, for lack of a better word. ↩

for quantum circuit optimisation and other tools from the graph transformation community such as GROOVE [267] and GrGen.NET [148] have not been adopted.

Instead, successful graph-based tools such as PyZX [187] and its faster re-implementation QuiZX [188] focused on performant rewriting for a restricted subdomain (in this case, the ZX calculus). This specialisation makes it difficult to expand these approaches to new primitives and constraints that are emerging from hardware advances within quantum computing. It also limits the interaction and sharing across field boundaries and impedes the development of tools applicable to a broader range of graph transformation domains.

The ambitious aim of this thesis is to advocate for graph transformation as a robust basis for a *scalable* and *modular* compiler platform for quantum computations – and hope that in the process, our contributions will strengthen the bridge between research in classical compilation, quantum computing and graph transformations. The key desired properties of our compilation framework can be summarised as follows:

Scalable. The compiler should handle quantum computations of the kind we realistically expect to execute within the coming decade: thousands of logical qubits, relying on possibly millions of physical qubits. Just as importantly, the compiler architecture should scale to take advantage of large classical computational resources, in order to maximise the optimisation potential when available.

Modular. The computational primitives available on present quantum hardware are wide-ranging and evolving rapidly, the programming models for end users are adapting, and hardware constraints and characteristics change from device to device. A future-proof compiler platform must therefore imperatively be extensible in its supported instruction set, its optimising cost function and the program transformation strategies.

Why is now the time for such a compiler and why are these qualities so important? We develop some arguments in section 1.1. Our concrete contributions to this goal are then summarised in section 1.2, along with an outline of the thesis.



This thesis hopes to strengthen the bridge between the fields of classical compilation, quantum computing and graph transformations. Three-legged bridges also exist in the real world – here the Butterfly Bridge in Copenhagen. Image credits: Christian Lindgren, [archdaily.com](https://www.archdaily.com).

1.1. A new compilation regime

We have introduced quantum compilation by drawing an analogy with the well-developed field of classical compilers. The novel directions in which quantum compilation is taking the field make for exciting new challenges. Three new quantum-specific properties of compilation form the core motivation for this work.

Large variations in architecture

The vast differences between proposed hardware architectures are a first distinguishing characteristic of current quantum computing developments. Unlike classical computing, where silicon-based transistors have become the definitive physical foundation for all electronic chips, the search for the most scalable and reliable technology for quantum computing is ongoing – and doubtless one of the most burning questions for the nascent industry. This introduces an incredible variety of compilation problems.

Quantum hardware designs differ both in the types of quantum particles used to implement qubits and in the control systems employed to manipulate these particles. Suggestions for the former include charged ions [185, 305], neutral atoms [178, 112], photons [194], transmons [54] and even Majorana [279] particles. The equipment that drives the desired operations on these particles is then drawn from a jolly mixture of lasers, magnetic fields, microwaves, dilution fridges, etc. Each combination results in different trade-offs: some will render a specific computation particularly easy; others promise to scale well to large systems but are very error-prone and unreliable; others still achieve high fidelities at the expense of slow operations.

From the perspective of a compiler engineer, this means we must equip quantum compilers to handle a wide variety of hardware primitives, multiple optimisation goals, and hardware-specific program constraints. Traditional compilation is ill-equipped to handle this considerable challenge.

A comparison of machine code for different architectures illustrates the difference between the quantum and classical worlds. Classical CPUs are dominated by two architectures, *x86*, used mainly by Intel and AMD, and *ARM*, used by a wide range of desktop and mobile chip manufacturers¹.

1. There are other architectures, such as RISC-V [316] and MIPS [165], but as of 2025 the quasi totality of consumer and professional CPUs run on *x86* or *ARM* from mobile phones to laptops, desktops, and data centres. See [311] for a detailed hardware market share analysis, albeit focused on gaming. Details on mobile market share can be found in [this survey](#) – all of the listed manufacturers use the *ARM* architecture. ↩

x86 CPU (e.g. Intel and AMD)

```
mov eax, 5      ; Load 5 into EAX
add eax, 3      ; Add 3 to EAX
mov [result], eax ; Store the result in memory
```

ARM CPU (e.g. mobile, Apple M-series)

```
ldr r0, =5      ; Load 5 into R0
add r0, r0, #3   ; Add 3 to R0
ldr r1, =result  ; Load address of result
str r0, [r1]     ; Store the result in memory
```

There are noticeable differences between the two architectures, mostly around variable naming conventions, as well as explicit memory loads `ldr` and stores `str` instructions in the case of *ARM*, which in *x86* are handled implicitly by `mov`. This simplistic example naturally ignores some of the more fine-grained considerations that can make translations hard in certain edge cases. A discussion of these can be found in [141]. However, overall, the instructions and capabilities of the

two platforms are broadly equivalent, as is confirmed by the existence of emulation tools such as [Apple Rosetta](#).

Let us contrast this with the difference between two quantum architectures. Consider, on the one hand, an architecture that can natively perform CX and H gates on qubits (e.g. superconducting qubits, ion traps, etc.) and, on the other hand, a platform based on photons and optical components.

Quantum circuit (qubits)

```
h q[0];
cz q[0],q[1];
```

Linear circuit (photons)

```
bs.h(5*pi/2, pi, pi, 2*pi) m[0], m[1];
bs.h m[2], m[3];
perm([2, 1, 3, 0]) m[1], m[2], m[3], m[4];
barrier m[0], m[1], m[2], m[3], m[4], m[5];
bs.h(1.910633) m[0], m[1];
bs.h(1.910633) m[2], m[3];
bs.h(1.910633) m[4], m[5];
perm([1, 0]) m[2], m[3];
bs.h m[3], m[4];
perm([3, 0, 1, 2]) m[1], m[2], m[3], m[4];
```

On the left is a quantum circuit expressed in the OpenQASM2 standard [100]. The right-hand side is the equivalent linear optics circuit computed by Perceval [166], expressed in a custom, OpenQASM2-like, format. The conversion is by no means straightforward! Some of the challenges include encoding qubits into multiple photon modes and mapping quantum operations to an optically realisable procedure made of optical components and measurements [133].

Other architectures, such as neutral atoms, may broadly support qubit-based operations but might not offer control over individual qubits and, instead require any operations to be applied in parallel to large groups of qubits [55]. Finally, it is to be expected that error-correcting codes that individual platforms will introduce to reduce error rates at the hardware level will introduce further constraints and new instruction sets yet again.

It is noteworthy that current trends in the classical world are also pushing compilers towards more heterogenous architectures that may include GPUs, FPGAs and other accelerators. This has led to significant changes in the design of current compilers, which we will touch upon later. Nonetheless, this shift has, so far, mostly “limited” itself to new forms of parallelism and the introduction of more specialised instruction sets rather than a fundamental redesign of existing tools and computing paradigms. The breadth of technologies and trade-offs that quantum compilers must face have no equivalent in the classical world – at least for the time being.

Asymmetric computational resources

A second exciting paradigm shift in compilation that quantum is driving forward is cross-compilation. A common assumption in compilation is that the program is executed on the same machine (or at least the same architecture) on which it was compiled. By contrast, in cross-compilation, the compiler and the compiled binary program run on different machines, possibly with different architectures. An instance of this would be using a recent ARM system-on-chip machine to create a

binary program for a traditional Windows PC with an Intel CPU. This is a supported feature of most modern compilers (and made easier by the relative similarities between processor architectures, as seen above), but such tasks are by no means trivial and can be [laborious to get to work well in practice](#)².

The situation is very different for quantum computing. Quantum computational resources are so limited that native compilation, in which the program is compiled and run on the same machine, is unfeasible – and will remain so for the foreseeable future³. When we put the possibility of pure-quantum compilation aside, we are left with a cross-compilation problem that is entirely the realm of classical computer science; the output of which happens to be destined to run on a quantum computer. This is similar to how in classical computing, GPU programs are typically compiled on CPUs before being uploaded and executed on the GPU.

Cross-compilation presents significant challenges. As quantum programs grow in size and complexity, debugging and verifying their correctness without access to the target hardware becomes increasingly difficult [272], as we hit the limits of what can be simulated classically. Quantum simulation is a vibrant research area that is the subject of theses (e.g. [140, 18]) in its own right.

On the flip side, using classical hardware for quantum program compilation comes with a giant opportunity for compilers: the classical computational resources available to the compiler, measured in the size of the memory and the number of operations that can be handled, are many orders of magnitude larger (and cheaper!) than what the quantum hardware that will execute the program is capable of. We can today execute tens to hundreds of billions of operations per second (GFLOPS) on desktop computers, up to the “exascale”, i.e. 10^{15} FLOPS, for the largest supercomputers [114]. Quantum hardware, on the other hand, will not be executing programs with sizes beyond 1000 error-corrected gates, or 10,000 physical gates, for another three years – that is believing the most optimistic roadmaps in the industry [172, 258].

It is expected that even a few thousand quantum gates will suffice to solve problems that our largest supercomputers struggle with. Meanwhile, every gate that must be performed comes at a high cost: it may fail, introduce errors, or take a long time to complete. It therefore behooves us to use all the classical resources at our disposal to reduce quantum operations to a minimum.

Given the strict hardware limitations, all near-term architectures are expected to face, quantum compilation must evolve into cross-compilers that are able to utilise the full power of classical hardware available to them; doing so will push the boundaries of what is possible with quantum computing just a bit further – in a field where every marginal gain may unlock new applications.

The confluence of classical and quantum compilation

Finally, quantum compilation also stands in front of some momentous engineering challenges. As we will see in section 2.2, significant research efforts have focused on the compilation and optimisation of quantum programs expressed as quantum circuits (cf. section 2.1). This formalism has its roots in quantum information theory, the field that gave birth to quantum computing and makes for an ideal framework to develop the theory and optimisation techniques. However, it does not include any of the fundamentals of compiler and programming language design that make classical software as composable and scalable as it is today.

2. There are new tools promising to make cross-compilation easier, such as [Zig](#). This only proves our point, though: classical cross-compilation has long been a neglected edge case. ↪
3. First valiant efforts at defining optimisation problems relevant to quantum compilation that could be run on quantum hardware have been recently presented in [264]. However, this concerns only specific optimisation subroutines of the overall compilation problem. It is hard to imagine today that deploying an entire compilation stack such as LLVM on quantum hardware would ever be sensible. Why tooling so close to the classical compiler frameworks will be required for quantum compilation is a topic we will return to in section 2.4. ↪

For example, there is no concept of subroutine or function calls; neither can a program execution be branching or looping based on runtime values. This makes code reuse impossible, resulting in huge program sizes and unsurmountable challenges for scaling up compilation to problems of real-world interest [176]. The absence of code abstractions is being felt even more acutely with the emergence of *hybrid* quantum-classical computations, as we discuss in section 2.3.

With applications of quantum computing that cannot be expressed as quantum circuits proliferating, a move away from circuit-based representations is becoming unavoidable [169, 256]. This is also an opportunity to incorporate learnings from the decades of experience that have been gathered in classical computer science. Many of the tools and software that were originally developed for classical computations are thus being adopted and adapted to the specificities of quantum. This convergence of quantum computing and classical compiler technologies is heralding new opportunities – but also pose important questions around how to represent quantum programs and optimise them.

1.2. Contributions and thesis outline

Preliminaries

The thesis starts in chapter 2 with a review of the main concepts on which the rest of the thesis is built. Aside from a short introduction to quantum computations (section 2.1) and a survey of the major quantum circuit optimisation techniques (section 2.2), this chapter makes two observations that impart a research direction to the rest of the thesis:

1. The emergence of hybrid quantum-classical computations is rendering the quantum circuit obsolete as the main representation of quantum computations within compilers (section 2.3).
2. The best optimisation outcomes will combine classical and quantum compiler optimisations. This can be achieved by adopting abstractions that are interoperable with classical compiler infrastructure (section 2.4).

A graph transformation formalism for quantum computations

Chapters 3, 4, and 5 form the core of this thesis and present our main contributions. The results in chapter 3 are crucial stepping stones for the rest of the thesis. Chapters 4 and 5 meanwhile present our most significant contributions to the state of the art.

In chapter 3, we propose minIR, a new graph-based intermediate representation (IR) for quantum computations. MinIR is a minimal subset of the *Hierarchical Unified Graph Representation* (HUGR), recently presented in joint work [216] and the subject of [ongoing development](#). It is to our knowledge the first compiler IR with support for linear types – required to model the restrictions that quantum mechanics imposes on quantum computations.

Unlike quantum circuits, minIR (and HUGR) programs can model computations that act on arbitrary combinations of classical (bits) and quantum data (qubits) within a single, unified representation. It represents the best of two worlds: it combines the safety guarantees of quantum-specific representations such as

quantum circuits (i.e. it is impossible to declare physically unrealisable computations), whilst at the same time being interoperable with classical compiler IRs.

Graph-based representations of computations, known as *computation graphs* in deep learning and *dataflow graphs* within the compiler community, are common in these fields. Our original contribution is in the formalisation of the IR transformation semantics: whereas classical compilers typically define IR transformations in terms of the values that they depend on and the values that they overwrite, this approach implicitly relies on value copying and discarding and thus does not generalise to linear values. Instead, we define graph rewriting semantics on minIR and show sufficient conditions for which minIR transformations preserve the validity of the program, and in particular the linearity conditions.

The encoding of quantum computations as graphs sets the stage for quantum compilation and optimisation using *graph transformation systems* (GTS), in which the set of transformations that the compiler is allowed to perform is expressed by a set of graph transformation rules. This is in effect a generalisation of an approach first proposed in [332] in the context of quantum circuits. We improve on this work with two major contributions that resolve critical issues that concerning the scaling of the technique to large numbers of transformation rules and large inputs respectively.

Pattern matching

Our first major contribution is a pattern matching algorithm, presented in **chapter 4**. The main result is a runtime complexity bound independent of the number of patterns being matched, achieved using a one-off pre-computation. This is to our knowledge the first pattern matching algorithm for quantum circuits that does not depend on the number of patterns. Whilst similar multi-pattern matching techniques have been explored in other domains such as RETE networks [142, 313, 171] and computational biology [104, 63], no algorithm is known with provable sub-exponential worst-case complexity. These results were published in [232].

The proved complexity bound applies to computations with only linear values¹, of which quantum circuits are a special case. The result is expressed in terms of maximal pattern *width* w and *depth* d , two measures of pattern size defined in section 4.2. The main result, presented in Proposition 4.13, is reproduced here:

1. In the absence of linearity, pattern matching is an instance of the sub-graph isomorphism problem, an NP-complete problem. The assumption is therefore necessary and expected. ↩

Proposition 1.1 *Pattern matching*

Let $P_1, \dots, P_\ell$ be patterns with width w and depth d . The pre-computation runs in time and space complexity

$$O((d \cdot \ell)^w \cdot \ell + \ell \cdot w \cdot d).$$

For any subject graph G , the pre-computed prefix tree can be used to find all pattern embeddings $P_i \rightarrow G$ in time

$$O\left(|G| \cdot \frac{c^w}{w^{1/2}} \cdot d\right) \quad (2)$$

where $c = 6.75$ is a constant.

The runtime complexity is dominated by an exponential scaling in maximal pattern width w . Meanwhile, the advantage of our approach over matching one pattern at a time grows with the number of patterns ℓ . It is thus of particular interest for matching numerous small width patterns.

We illustrate this point by comparing our approach to a standard algorithm that matches one pattern at a time [182], with runtime complexity $O(\ell \cdot |P| \cdot |G|)$. Using $|P| \leq w \cdot d$ (cf. section 4.2), and comparing to eq. (2), we thus have a speedup in the regime $\Theta(c^w/w^{3/2}) < \ell$. On the other hand, ℓ is upper bounded by the maximum number $N_{w,d}$ of patterns of bounded width and depth. Using a crude lower-bound for $N_{w,d}$ derived in Appendix , we obtain a computational advantage for our approach when

$$\Theta\left(\frac{c^w}{w^{3/2}}\right) < \ell < \left(\frac{w}{2e}\right)^{\Theta(wd)} \leq N_{w,d}.$$

In the case of quantum circuits, the width of the patterns is given by the number of qubits. The low-qubit regime where our approach shines coincides exactly with the typical applications of GTs in quantum compilation: in [332] and [333], all rules used have at most 4 qubits.

We present benchmarks on a real world dataset of 10,000 quantum circuits in section 4.7, showing a 20x speedup over a leading C++ implementation of pattern matching for quantum circuits.

Confluently persistent graph rewriting

Our second major contribution, in chapter 5, uses a well-known construction in GTs, the *unfolding* [23], to derive a novel data structure $\mathcal{D}$ that compresses the representation of the space $\mathcal{G}$ of all graphs reachable from an input within a GTS. We call $\mathcal{D}$ the *factorised search space* of $\mathcal{G}$. Optimisation problems over the space of reachable graphs in a GTS can then equivalently be expressed as optimisation problems over $\mathcal{D}$.

We show in section 5.5 that under some assumptions on the GTS and input, there is an exponential complexity separation in the input size between the size of the factorised search space $\mathcal{D}$ – which admits an asymptotically *linear* upper bound – and the size of the rewrite space $\mathcal{G}$ that it encodes – which grows at least exponentially.

$\mathcal{D}$ is furthermore the first *confluently persistent* data structure [117, 139] [?] it performs non-destructive rewrites on immutable graph objects by maintaining an explicit history of all graph rewrites and their dependencies. This allows concurrent application of multiple rewrites and can merge rewritten graphs that were obtained independently. This represents an exciting development in its own right that opens the door to functional programming and massively parallelised approaches to graph rewriting (see section 6.2).

The intuition behind the exponential reduction in search space size is as follows: if rewrites $r_1, \dots, r_n$ apply to disjoint subgraphs of a common graph G , then $\mathcal{D}$ will be of size n , storing the set possible rewrites, rather than the up to 2^n distinct

graphs in $\mathcal{G}$ obtained by applying a subset of the rewrites. To generalise to arbitrary rewrites, the data structure $\mathcal{D}$ must keep track of the dependencies and overlaps between rewrites and update these as more rewrites are added to $\mathcal{D}$.

A lot of parallels can be drawn between this approach and *equality saturation*, a technique for term rewriting with applications in classical compilers. We explore these connections in section 5.2.

Unlike the results of chapter 4, the construction and bounds proven in chapter 5 can be applied to a wide range of graph rewriting domains. It has particularly significant implications for applications of GTs that are unable to derive rewriting strategies from first principles, and hence have to resort to an exhaustive (or heuristic) exploration of the rewrite space $\mathcal{G}$. They can proceed as follows:

1. *Exploration phase*. Construct the factorised search space $\mathcal{D}$ by finding and applying rewrites, in time proportional to $|\mathcal{D}|$. With our results, this results in an exponential speedup over the naive exploration of $\mathcal{G}$ (section 5.3).
2. *Extraction phase*. Unlike the case of $\mathcal{G}$ where the optimal solution is an element $G_{opt} \in \mathcal{G}$, constructing the optimal solution $\mathcal{D} \rightarrow G_{opt}$ in $\mathcal{D}$ is a non-trivial extraction problem. We show in section 5.4 that the extraction can be expressed as a boolean satisfiability (SAT) problem; depending on the cost function, the optimisation can then be encoded as a side condition on SAT or by a generalisation of the problem to Satisfiability Modulo Theories (SMT).

In the worst case, SAT and SMT problems will require exponential time to solve [89, 233, 52], thus cancelling the exponential compression of the search space $\mathcal{G} \rightarrow \mathcal{D}$. However, SAT and SMT are standardised problems for which heavily optimised solvers and optimisers have been developed [235, 286]. We expect that the instances of SAT and SMT that encode the extraction problem will scale well in practice:

- Clauses in the problem encode *local* properties that SAT solvers are well-suited to solve [340]: the boolean variables represent rewrites, which only impose restrictions on other rewrites that apply in the same neighbourhood of the graph.
- Furthermore, in quantum compilation applications, $\mathcal{D}$ can be *sparsified*: most rewrties in $\mathcal{D}$ do not change the cost function (think of IR transformations that reorder operations but do not reduce the runtime) and thus do not need to be encoded in the SAT problem.

In a first exploratory analysis, we present some empirical results that support our claims: by searching over the factorised search space instead of the naive search space, the optimiser is able to find the global optimum for circuits that are twice as large. Our results also exhibit a linear scaling in the size of the factorised search space, confirming that the approach should scale well to larger problems.

Conclusion

The thesis concludes in chapter 6 with a discussion on how our contributions serve our overall goal of a scalable and modular quantum compiler platform. We discuss in particular two extensions of our work that we see as particularly promi-

sing: the generalisation of fast multi-pattern matching to non-linear values and to the persistent data structure $\mathcal{D}$ of chapter 5 (section 6.1) and the deployment of confluent persistent graph rewriting to a massively parallel distributed compute architecture (section 6.2).

Chapter 2

Quantum Computing: a Computer Scientist's Perspective

Many (too many?) introductions to quantum computing have been written, so we will refrain from adding another entry to the collection. Instead, beyond the absolute basics, our focus is on the expressive power and *syntax* of quantum programs. This demystifies quantum compilation into program transformation problems, amounting to traditional compiler methods that will be very familiar to computer scientists.

In this chapter, we lay the groundwork for this thesis by introducing what programs meant to run on quantum computers look like today, what we expect they will look like in the (near) future, and how quantum compilers have been built to optimise them. We start in section 2.1 with a review of the basic computation primitives of quantum computers and how they are composed to form quantum circuits, the simplest form of quantum programs. This is followed by a review of the leading quantum circuit optimisation techniques in section 2.2. Finally, sections 2.3 and 2.4 introduce and discuss the impact of *hybrid* quantum computations, and how they challenge existing quantum compiler designs and optimisations.

2.1. Foundations of quantum computing

The most widespread computational model in quantum computing – and arguably its simplest – is built on the qubit abstraction. As its name suggests, it is the quantum analogue of the classical bit, i.e., a value that can take the values 0 or 1.

We will stick to our promise of not delving into the details of the physical realisations of qubits in real-world architectures. Nonetheless, it is important to note one fundamental difference with classical systems. Classical bit values (the famous 0s and 1s of our computers) are typically encoded using two voltages; another way of saying this is that bit values, and hence data, correspond to electrical currents in the wires¹ of a chip. Gates, i.e. the lowest level of operations that can be

1. In the case of integrated circuits and printed circuits boards, the *wires* we refer to here would be called “interconnects” or “traces”. ↩

applied to bits, then correspond to barriers that let the electrical current flow through to outgoing wires, or block it.

We can thus picture a classical gate as a black box with n input wires going into the box and m output wires leaving it. For any combination of on and off voltages on the input wires, the box will turn on some of the output wires. The vital point to take away from this classical state of affairs is that we can think of the carriers of input and output data (i.e. the input and output wires) as physically distinct objects that can exist and can be read simultaneously.

Quantum physics rules out such an implementation of qubits. In the case of matter-based qubits, such as ions in traps or Josephson junctions on superconductors, quantum gates are operations that *modify* – or “mutate”, to borrow a term from programming languages – the physical qubits themselves. An input qubit to a gate is thus submitted to physical interactions that change its internal state. After the gate execution is completed, the qubits that held the input states now contain the operation’s output.

Similarly, photonic systems encode qubits using modes of the electromagnetic field. A gate in this setting acts by transforming these modes – mixing them, shifting their phases, or entangling them with ancillary modes. It is never possible to modify qubit data coherently whilst keeping access to the original input data.

2. The NOT gate is the notable exception to this. It is often found in quantum programs and called X. ↩

This has several profound implications for quantum computing. First and foremost, every quantum gate must have the same number of inputs as outputs. Most iconic classical gates (AND, OR, XOR, etc.) are thus impossible to implement on a quantum computer without some adjustments². This also means that the number of qubits must remain unchanged throughout the computation. A computation that starts with n qubits must also end in n qubits – and have n qubits at every point throughout the computation.

3. This is true physically: the carriers of quantum information, typically atoms or photons, live forever in the absence of interactions with their environment. However, we would be seriously deluding ourselves if we believed that the control systems we use to manipulate and keep these particles trapped could do so for any significant amount of time. Instead, experimentalists must constantly devise creative ways to stop the qubits from escaping or interacting with their surroundings (and destroying themselves in the process). ↩

At this point, taking the preservation of qubits just described seriously, we should be asking how a quantum computation can even come to be at all, given that no qubit can be created out of thin air. In our attempt to remain blissfully ignorant of physical realities, we suggest adopting the following abstracted mental model of qubits: qubits can neither be created nor deleted³, they simply *i)* exist at all times, and *ii)* can be reset to the 0 state.

For our convenience, we can ignore qubits that are unimportant to us. If all we need are n qubits, then we will limit our considerations to these and pretend none other exists. Pushing further our myopic focus on qubits with a direct utility, we can also adjust the window of qubits of interest as we progress through the computation. If, for instance, a new qubit becomes useful halfway through our program execution, we can enlarge the set of qubits we are keeping track of and refer to this as “creating” a qubit. Conversely, qubits often become irrelevant, in which case we move them outside of our field of consideration and say that the qubits were discarded.

A final consequence of mutating qubits that we will highlight is that once a gate has been applied, the input states to the gate no longer exist! In other words, any state that we reach throughout our execution can only be used at most once. Here, your classical intuition might kick in:

Let us just maintain a copy of the original state before modifying it!

This would allow us to do more than one computation from a temporary value. However, copying is a big NO in quantum computing. It is a profound restriction (or property, depending on your point of view) with deep roots in the physics of quantum mechanics. This principle, the no-cloning theorem, is one of three fundamental properties of quantum physics that quantum computing builds upon.

The physical constraints of quantum computation

No-cloning theorem

The no-cloning principle [330] provides a formal foundation for the vague statement “*qubits live forever*” we made earlier. It is a fundamental tenet of quantum information, deserving a more rigorous treatment than we are giving it here. We recommend that the curious reader refers themselves to more respectful references such as [238].

No-cloning theorem: it is impossible to copy an arbitrary unknown state onto another (possibly known) qubit, or to copy a (possibly known) qubit to a qubit with unknown arbitrary state.

If we use $|\psi\rangle$ to denote an arbitrary state and $|0\rangle$ to denote a known state, the principle can be restated as: there are no quantum computations mapping $|\psi\rangle|0\rangle \mapsto |\psi\rangle|\psi\rangle$, nor $|\psi\rangle|0\rangle \mapsto |0\rangle|0\rangle$. The consequences of this are profound.

A consequence of the first half is what we alluded to in the previous section: any qubit states can only be used once in a computation. This statement also justifies why every quantum gate implementation, no matter the hardware specifics, will mutate its input qubits to produce the output states.

The second half of the statement is often referred to as the “no delete” theorem. Indeed, if we view $|\psi\rangle$ as a state encoding some data, we can interpret it as some amount of information. The state $|0\rangle$, on the other hand, is a fixed state and thus cannot store any information. From the perspective of information theory, the map $|\psi\rangle|0\rangle \mapsto |0\rangle|0\rangle$ thus destroys information: it turns an information storing left-hand side into a product of $|0\rangle$ states, devoid of any information.

We can also revisit the first map $|\psi\rangle|0\rangle \mapsto |\psi\rangle|\psi\rangle$ and understand it from an information theoretic perspective as an attempt to create information out of thin air! Using this interpretation, the no-cloning theorem is thus the statement that quantum information is a preserved quantity in quantum computations: its amount will never increase or decrease.

Reversibility

The fact that the amount of quantum information can never increase by transforming quantum states matches our intuition: if no new information is added from outside the system, then the total information encoded should not be increasing. Why, however, is it impossible to erase some information and thus reduce its total? The answer is reversibility of closed quantum systems: if we exclude the option of discarding parts of the physical system, every quantum of operation is undoable. In other words, a computation must have an inverse operation that recovers the input when applied to the output.

If a quantum operation were thus to erase any information, then an inverse operation would exist that creates information from nothing! The two halves of the no-cloning theorem, as we presented it, thus state the same principle once we consider that every operation must be reversible.

Universality

Finally, a third distinguishing property of quantum computation is how arbitrarily large computations can be generated from single-qubit gates and pairwise entangling interactions between qubits (two-qubit gates) [36]. It is furthermore the case that the choice of a *fixed* two-qubit gate, along with single-qubit gates, is sufficient to generate any arbitrary quantum computation. We call a set of gates that can be used to construct any arbitrary quantum computation a **universal gate set**.

This is a boon for hardware design, as manipulating single-qubit systems is often much more manageable than controlling physical interactions between multiple entities. This decomposition into single-qubit and (a fixed) two-qubit gates means that the architecture *i*) does not need to support interactions between $n > 2$ qubits, and *ii*) can be specialised and hand-tuned to execute the two-qubit interaction of choice as faithfully as possible. Having a two-qubit gate as the entangling operation is not the only choice. Some architectures, such as neutral atoms, choose instead to replace it with a global entangling operation that applies to many qubits simultaneously [128], resulting in a universal gate set that is more convenient to implement experimentally in their system.

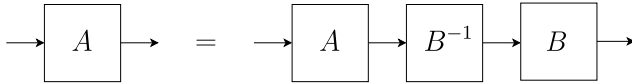
Gate set universality can be generalised further to approximate universality, which is at the centre of the development of error-correcting codes. Indeed, any quantum computations can be approximated to arbitrary precision using only discrete finite sets of one and two-qubit gates [191, 110]. This represents a significant simplification for error correction, as it removes the need for continuously parametrised gates and discretises the problem space.

Leveraging quantum properties for compilation

We have introduced the universality, reversibility and no-cloning properties of quantum computations for a reason: these laws of physics that govern quantum computations and are absent from classical computer science are an excellent foundation for developing quantum-specific computation optimisations and compilation techniques in general.

As we have just discussed, the wide variety of universal gate sets are degrees of freedom that the compiler can use. Using universality to translate computations between universal gate sets, enabling programmers to seamlessly target different hardware, is one of quantum compiler's first and most fundamental functions [294].

Reversibility is also a source of flexibility when expressing quantum programs. Suppose the user wants to execute an operation A but it is more convenient, or the hardware is only capable of executing a different gate B . Then, using the inverse B^{-1} of B , it is always possible to rewrite the program as



where these diagrams should be read as operations to be executed from left to right. This is nothing but the mathematical trick of multiplying the left-hand side with the identity operation expressed as $B \circ B^{-1}$ ⁴.

Now, of course, this rewrite is only sensible if the operation $B^{-1} \circ A$ is reasonably cheap to perform. There are plenty of instances where this is indeed the case. Morally, the quantum compiler always has the freedom to execute any quantum operation – at the risk of producing very inefficient code – given that reversibility always guarantees that the operation can be reversed and the competition undone whenever necessary.

Finally, no-cloning is a very useful guarantee that the compiler can use to simplify reasoning about computations⁵. In chapter 4 we will see that it dramatically simplifies pattern matching, which helps identify all possible optimisations quickly. More generally, no-cloning restricts the set of programs that the compiler must consider, resulting in elegant graph transformation semantics – a topic we explore in chapter 3.

4. The $\circ$ denotes the composition of functions, so unlike the left-to-right diagram, it must be read from right to left. ↩

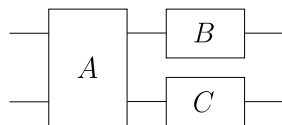
5. In particular, no-cloning resolves the problem of [aliasing](#) once and for all! ↩

The quantum circuit representation

We could not conclude our overview of the basics of quantum computing without mentioning the quantum circuit, a representation of quantum computation ubiquitous in the field. With the understanding that we have gained in this section, the two building blocks of the circuit model and the conventions around their graphical representation should be of no surprise to the reader:

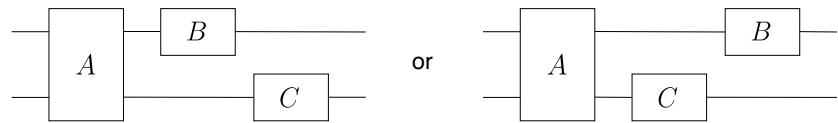
1. Qubits are represented by straight, horizontal lines. Their evolution through time can be followed along the line from left to right: At the leftmost point on the line, the qubit is in its input state; when the qubit reaches its rightmost point, operations have mutated it into the output state of the circuit.
2. Gates on qubits are boxes placed vertically across one or multiple qubit lines. The qubits it is on represents the set that the gate may act on (and mutate), whereas the left-to-right ordering of the gates reflects their ordering in time.

A simple circuit composed of two qubits and three gates A , B and C could for instance look like this



The previous diagram was in fact also a circuit, in which each arrow pointing to the right was a segment of a qubit line. In this case, A would be executed before B and C ; A would act on both qubits, whereas B and C would only modify the

ng makes no difference. It is thus common to display them as acting at the same time. We could have equivalently chosen to draw them as:

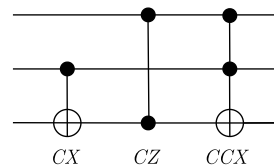


All these circuits represent the same computation.

Certain quantum gates are particularly useful and appear very regularly in practice. These have standard names that are widely used in the field. The most common single qubit gates are arguably the Hadamard, represented in circuits by a H box, and the X , Y and Z -axis rotations, drawn as $R_x(\theta)$, $R_y(\theta)$ and $R_z(\theta)$ boxes respectively. Note that rotation gates are parametrised by an angle $0 \leq \theta < 4\pi$ that must be specified to execute the rotation.

6. This becomes immediately apparent if you attempt to draw a gate that should act on the first and third qubit line of a circuit, but leave the second one untouched. ↩

There are also commonly used multi-qubit gates. For these, it becomes slightly awkward to draw them as boxes, as they may act on qubits that are not drawn next to each other in the circuit⁶ or might be applied to qubits in a specified order. As a solution, common gates were given representations that do not spell out their name but mark which qubit they are acting on and in what order. Here are the representations of three of the most famous ones, in order: the CX (also known as CNOT) gate, the CZ and the CCX (also known as the three-qubit Toffoli):



You will probably notice that there seems to be a system to this graphical notation. There is, but unfortunately, explaining it would require us to discuss Pauli matrices and commutation relations and quickly lead us astray. The references in section 2.5 are a good starting point for further reading.

2.2. Quantum circuit optimisation: a review

Much of the foundations of classical computer science rely on boolean logic and discrete mathematics [208]. In some regards, this is a poor man's maths, as much of the structure that comes with continuous infinite mathematical objects is lost along the way when discretised.

7. If intrigued, look at this nice introduction [199] and the references therein. It's not as scary as it sounds. ↩

In contrast, quantum computation, on the other hand, encompasses the whole breadth of (finite dimensional) quantum physical system evolution. Underlying this is a rich mathematical theory steeped in the theory of Hilbert spaces and Lie groups¹. A direct consequence of the mathematics of quantum computations is the flourishing of an entire field of research dedicated to quantum circuit optimi-

sations [184]. They leverage the unique structure and symmetries of quantum physics to reduce the noise and resource requirements of quantum computations significantly.

In this section, we will review the main optimisation techniques that established themselves within quantum compilers, focusing on the representation of quantum computations they use and their assumptions about the computations they are optimising.

Cost function

A key point to settle first when discussing circuit optimisations is the objective of the optimisation – the cost function to be minimised. Unlike much of classical compiler research, which can rely on an established set of hardware targets and benchmarking datasets to profile the empirical, “real world” performance of compiled programs, the quantum world must often contend with simplified noise and architecture models to design proxy metrics, given the limited scale and availability of current quantum devices.

The quantum compilers research community has mostly coalesced around cost functions based on gate count statistics [184]. Counting a type of gate is a simple and popular choice. Making some additional assumptions on the gate parallelism of future hardware, one may also consider cost functions based on gate depth, i.e. the length of the longest chain of gates that cannot be run simultaneously [288, 40]. In spite (or precisely *because*) of their simplicity, gate counts serve well as cost functions in many quantum compilation use cases. Most circuit optimisations target one of two hardware regimes.

On most current hardware architectures, the major challenge is achieving high accuracy on entangling operations, i.e. quantum gates that make two or more qubits interact [4, 253, 196, 54]. In superconducting qubit and ion trap architectures², for example, the gate set is typically composed of one and two-qubit gate types, with error rates dominated by an order of magnitude by the latter [300, 294]. Circuit optimisations for computations on such *noisy* hardware thus often define cost functions based on the number of two-qubit gates – typically the *CX* gate, though many other two-qubit gates could be used equivalently.

On the other hand, future generations of hardware for larger scale computations are expected to be more resilient to noise, with the help of error detection and correction techniques. In this regime, the computational power of the hardware is no longer limited by hardware noise but rather by the affordances of the error-correcting code. Depending on how the quantum data is redundantly encoded in the code space, the fault-tolerant execution of specific operations may be anywhere between very straightforward and nigh-impossible. The bottleneck is widely expected to be the execution of single-qubit (non-Clifford) gates, such as the *T* gate³. These cases can thus just as well be modelled by cost functions based on gate counts.

Unitary synthesis: the perfect optimisation

The *ne plus ultra* of quantum circuit optimisation is *unitary synthesis*. It leverages the representation of a quantum computation as a square, complex-valued, unitary matrix, which is then *re-synthesised* as a new, equivalent (and ideally optimi-

2. Experimental realisations of many-qubit interactions have also been demonstrated [127, 55, 15, 128] and are at the core of other proposed architectures [39, 62]. ↩

3. Much of quantum error correction theory is built on the Clifford group, a subset of quantum operations that preserve “Pauli errors” and can thus be corrected easily. The flip side of this is that correcting any non-Clifford operation is very hard, something that is resolved by constructing “error-free” *magic* states ahead of time. For more details, refer to a quantum error correction textbook such as [155]. ↩

sed!) quantum circuit. This approach thus breaks down quantum optimisation into two separate sub-problems:

1. Reduce a n -qubit quantum circuit into a $2^n \times 2^n$ matrix. This matrix is a *unique representation* of the computation, meaning that any two equivalent computations will be mapped to the same matrix.
2. Find the optimal matrix decomposition into primitive quantum gates, thus obtaining a new quantum circuit, equivalent to the original.

The uniqueness of the unitary matrix representation makes it invaluable as a resource for computation optimisation. Not only does it reduce any potentially large collection of equivalent inputs to a single form; it also – crucially – provides a sound distance metric on the space of all circuits, in the form of the Haar measure. This can be used in search-based approaches to measure the distance between synthesised circuits and thus direct a search heuristic towards the optimal solution.

Early work explored general unitary decomposition schemes obtained analytically from linear algebra. These express arbitrary unitaries as a product of unitaries that typically correspond to one and two-qubit gates in the quantum circuit model [173, 174]. Approaches have been proposed using the Cosine-Sine decomposition [234], the Quantum Shanon decomposition [201], and the QR decomposition [287]. While some schemes have been shown to be asymptotically efficient for almost all unitaries [173], such strategies typically generate fixed-sized circuits and fail to synthesise short circuits when such circuits exist. The size of synthesised circuits grows exponentially with the number of qubits, making most such schemes impractical beyond three qubits.

Unitary matrix decomposition can also be combined with tools from classical circuit design: in [212], Loke et al. proposed an approach merging reversible circuit synthesis (see below), a classical compilation problem, with unitary matrix synthesis. They show that searching for decompositions $U = PU'Q$, where P and Q are classical reversible circuits can yield shorter circuits when using the Cosine-Sine decomposition for the unitaries U and U' .

Search-based approaches have been developed to address the shortcomings of analytical decompositions. Unlike the algebraic approaches, the circuit decomposition problem is viewed as an optimisation problem in search-based circuit synthesis. The space of all possible quantum circuits is explored to find the one that implements the desired unitary whilst minimising the cost function. The major challenge of such methods is the gigantic (typically super-exponential) size of the search space of all possible programs. Without mitigation, most work in this space struggles to scale beyond a handful of qubits.

Up to 3 qubits, T-depth optimal circuits can be found using exhaustive brute force search first proposed in [7] and improved in [149]. Asymptotic bounds on the number of T gates required for general unitary synthesis were recently given in [153].

Scaling to 4 qubits and handling gate sets with continuous parameters, required for non-fault tolerant circuits, an A* search with smart pruning heuristics was proposed in [109]. This approach's outputs are no longer provably optimal, but the results match optimal decompositions in all known instances. This line of work has subsequently been further refined with heuristics based on pre-defined circuit

templates [295,215], parameter instantiation [337,338,263], machine learning [319] and tensor networks [202].

Some of these heuristics also make it possible to synthesise circuits with device constraints in mind and can trade off decomposition accuracy for shallower circuit depth and lower noise. In [331, 202], the authors have also explored partitioning the circuit into smaller parts optimised independently to scale these techniques to large circuit sizes. Despite the reduced optimisation performance that the boundaries of the partitioned circuits introduce, the combination of circuit partitioning with the techniques listed above yields some of the best-performing circuit optimisation techniques developed to date [96]. Circuit synthesis schemes have also been extended to generate circuits on a more expressive gate set, including elementary classical operations [5, 240].

However, a fundamental flaw of all unitary synthesis schemes is the 4^n -scaling in the number of qubits n of the unitary representation itself. This means that no matrix-based synthesis method, however efficient, will ever be able to handle computations with much more than a dozen qubits. Circuit partitioning schemes such as [331, 202] effectively circumvent the problem, but they are heavily dependent on the partitioning quality.

The search for scalable representations

Our study of unitary synthesis introduced us to a convenient two-step approach to quantum computation optimisation. First, the input circuit is transformed into a “global” representation that captures the computation as a whole, abstracting away the precise sequences of gates that compose the original circuit. This representation is then the input for the second half of the problem, which produces a circuit of the desired shape, equivalent to the original input but with reduced cost.

In addition to simplifying the original problem, such global intermediate representations are well-positioned to leverage the quantum-specific structure and symmetries in the computation. They can thus enable more advanced optimisations and are robust to varying circuit representation and local optimisation landscape.

The unitary matrix is the most common representation of quantum computations, but as we have seen, it suffers from severe scaling problems in the number of qubits. The problem is not so much that quantum computations require exponential space to be described *in the worst case* – after all, the space of all n -qubit unitaries $SU(2^n)$ is exponentially large. However, the set of unitaries implementable *in practice* can only be a tiny subset of $SU(2^n)$ ⁴ – the set of unitaries that admit a polynomial-sized circuit representation.

Another fruitful avenue of work for quantum optimisation has thus been the development of alternative representations for quantum computations that can encode polynomially sized quantum programs efficiently whilst enabling novel optimisations.

Phase Polynomials and Pauli Gadgets

A particularly convenient global representation of many quantum circuits is as products of Pauli exponentials, also known as *Pauli gadgets* [98]. These unitaries

4. Polynomial-sized quantum circuits constitute a polynomial-dimensional submanifold of the exponential-dimensional $SU(2^n)$ Lie group. They are, hence, a measure zero subset of $SU(2^n)$ with respect to the Haar measure. ↩

are of the form

$$U = \prod_{s \in P} \exp(i\alpha_s \cdot s)$$

where $\alpha_s \in [0, 2\pi)$ are real coefficients and $s \in P = \{I, X, Y, Z\}^n$ are strings of length n of the four Pauli matrices – so-called *Pauli strings*. In this formulation, n fixes the number of qubits of the computation.

These exponentials are always valid n -qubit unitaries and can express entangling operations across any number of qubits: the qubits on which an operation $\exp(i\alpha \cdot s)$ acts non-trivially are given by the indices of the characters in s that are not the identity I . For instance, the exponential

$$\exp(i\frac{\pi}{2}XIZ)$$

is a valid quantum computation on 3 qubits, entangling the first and third qubits. Beyond useful abstractions for optimisation, such entangling operations appear naturally when simulating quantum systems, for example in quantum chemistry [223].

The use of these primitives for quantum compilation was first explored in [98], and further generalised in [99]. Starting from an (unordered) sequence of Pauli gadgets, the gadgets are clustered into sets of mutually commuting gadgets. These can then be jointly synthesised into a circuit, markedly reducing the number of entangling operations as compared to naively implementing one exponential at a time.

Further improvements to this work have since been presented in [170] and [281], where new heuristics are introduced to choose the Pauli gadget ordering. In [170], the hardware-specific connectivity constraints between qubits are also taken into account to produce programs that can be executed on the targeted architecture without overhead.

A close relative of Pauli gadgets – a strictly smaller subset of it, to be precise – are the so-called *phase polynomials* [8], obtained when restricting the Pauli strings to combinations of Z Pauli matrices and identities: $s \in \{I, Z\}^n$. These are particularly amenable to optimisation as in this case, the ordering of the gadgets becomes irrelevant – all exponential terms commute. This gives the compiler a lot of freedom during circuit synthesis.

The action of phase polynomials on quantum states is quite easy to understand. Instead of the exponentials of I and Z -based Pauli string, the computation can equivalently be given by its action on the basis states. A quantum basis state – just like a classical state – is given by a bistring $b_1 \dots b_n$ of bits $b_i \in \{0, 1\}$. Writing $e_{b_1 \dots b_n}$ for the basis state corresponding to the bitstring $b_1 \dots b_n$, the action of a phase polynomial U on $e_{b_1 \dots b_n}$ is given by

$$e_{b_1 \dots b_n} \mapsto \underbrace{\exp(i \cdot \sum_{s \in \{I, Z\}^n} a_s \cdot (\tilde{s}_1 b_1 \oplus \dots \oplus \tilde{s}_n b_n))}_{\in \mathbb{R}} e_{b_1 \dots b_n} \quad (2)$$

where $\tilde{s}_1 \cdots \tilde{s}_n$ is now also a bitstring of booleans $\tilde{s}_i \in \{0, 1\}$, and $\oplus$ denotes the boolean XOR operation. The boolean $\tilde{s}_i$ has value 1 if and only if the i -th character in the Pauli string s is Z ,

The exponential expression in (2) is just a real number – indeed each term in the sum simply evaluates to either a_s or 0. A phase polynomial is thus a diagonal unitary matrix: it maps every basis state $|b_1 \cdots b_n\rangle$ to itself, multiplied by some phase $e^{i\theta}$.

Polynomially-sized circuits that implement diagonal matrices correspond to phase polynomials with $k = \mathcal{O}(n^p) \ll 2^n$ non-zero terms $a_s \neq 0$, i.e. they can represent quantum computations efficiently and scale well with the number of qubits – thus allowing efficient algorithms that scale polynomially in the number of qubits n .

The Graysynth algorithm, as presented in [8], has become *the* reference synthesis method for phase polynomials. The key observation made by its authors is that all terms of the sum within the exponential can be cycled through and obtained following the binary Gray codes [157]. The Hamming distance of one that separates successive bitstrings in the code translates into a single two-qubit CX gate when synthesised to a quantum circuit by Graysynth.

This approach was adapted to work with hardware connectivity constraints in [159], [152] and [312]. An up-to-date study of the performance of phase polynomial-based compiler optimisations and comparisons with standard approaches is performed in [225].

The study of phase polynomials can also be generalised to arbitrary diagonal operators. Tight asymptotic bounds on the resource requirements for arbitrary diagonal operator synthesis and their implementation were recently given in [302]. The authors propose using a smart meshing of different Gray codes in parallel and, where available, additional qubits as ancilla registers to parallelise computations further and minimise circuit depth. The resulting general-purpose decomposition of arbitrary diagonal operators yields circuits of depth $\mathcal{O}(\frac{2^n}{n})$ and size $\mathcal{O}(\frac{n^2}{\log n}) + 2^{n+3}$, as well as improved bounds in the presence of $m \geq 2n$ ancilla qubits.

Clifford synthesis

The group of all n -qubit unitaries $SU(2^n)$ contains a subgroup that has become an object of study across many domains of quantum computing science: the Clifford group. We have already mentioned that it is at the centre of quantum error correction theory [64]; it is also a cornerstone of measurement-based quantum computing [265] and graph states [164], as well as one of the most promising approaches for fast quantum simulations [154, 65, 186].

The Clifford subgroup of quantum circuits admits an efficient $\Theta(2n^2)$ -sized program representation known as *Clifford tableau* [1]. This has been used profusely for compiler optimisation. In [1] the first Clifford circuit synthesis procedure is given, using an analytical decomposition of Clifford tableaus into $\mathcal{O}(n^2/\log n)$ one and two-qubit gates. An improved, Bruhat-based decomposition that is optimal in

further refined to replace gates with classical computation on the measurement outcomes [66]. Finally, an alternative normal form that is well-suited to hardware with limited nearest neighbours connectivity was also derived using a diagrammatic approach [218].

Just as in unitary synthesis, circuit decompositions of Clifford operations more efficient than the general analytical expressions can be obtained case-by-case using search and optimisation. The pendant to the provably optimal decompositions of unitaries obtained through brute force search [7] also exists for Clifford circuits [193]. Due to the more efficient representation and smaller search space, all optimal Clifford circuits could be found up to 6 qubits. Using modern SAT solvers, optimal Clifford synthesis has recently been pushed much further, with known optimal circuits beyond 20 qubits [251, 282].

Heuristic optimisation approaches have also been shown to be effective on Clifford optimisation [66, 129] and scale to larger systems. For Clifford computations on devices with restricted connectivity, an architecture-aware synthesis method was proposed in [328].

Diagrammatic representations

Quantum computer science and quantum mechanics have a rich history in diagrammatic representations [138, 84, 20]. These have allowed one to picture complex physical processes as intuitive operations in a graphical language and have – as a nice side effect – led to a plethora of state-of-the-art quantum circuit optimisation techniques!

A diagrammatic representation of quantum computation is obtained by lifting the gates that form a quantum circuit into the nodes of a more abstract graph-based graphical calculus. The most commonly used flavour of calculus for circuit optimisation is the ZX calculus [85, 86, 320, 336].

5. and totally arbitrary! ↩

By breaking up multi-qubit gates into several non-unitary tensors, the ZX calculus and related variants [273, 20, 133] expose some of the symmetry and structure of quantum physics in the form of simple and intuitive graphical rules. This has enabled the discovery of many quantum optimisation techniques (e.g. [120, 321]), some of which we have already reviewed [170, 152, 159, 98, 99]. This selection of papers is not *quite* exhaustive⁵ – there are currently over 300 hundred papers on the topic, as indexed by zxcalculus.com.

Aside from being an invaluable tool for research and compiler pass design, a significant contribution of these diagrammatic representations is the introduction of graph transformation systems (GTS) [124, 274, 198] to quantum computing. More on this in chapter 3 (and much of the rest of this thesis)!

Reversible classical circuits

Many more representations have either been taken over from classical compiler optimisations or were developed for specific purposes. The last we will mention is reversible circuit synthesis, an entirely classical circuit design problem which can draw from the results of decades of research. From a quantum perspective, reversible classical circuits correspond to unitaries (and more generally, isometries) that send basis states to basis states – and thus do not introduce any complex phase

[291]. We highlight a selection of the more recent work in the field and refer the reader to the much more complete, albeit ageing, survey of [276].

Up to 4 (qu)bits, all reversible circuits and their optimal synthesis can be generated by brute force [209]. Viewing reversible circuits as a permutation of all 2^n bitstrings, Susam et al. pre-compute optimal circuits only for swaps of two bitstrings (transpositions). These can then be used as part of a standard selection sort to synthesise arbitrary permutations. The number of such permutations scales much more favourably compared to arbitrary permutation, allowing fast circuit synthesis of up to 20+ (qu)bits in a fraction of a second, with good performance.

Truth table or matrix representations of reversible circuits suffer from the same exponential scaling as unitaries. To address these, other representations that have been used include exclusive sums of product terms (ESOP) [132, 33], positive polarity Reed-Müller codes (PPRM) [180] and decision diagrams [301, 324, 249].

The quantum framework is strictly more general than the classical regime in which the problem was studied initially. This affords additional freedom for decomposition schemes, such as decompositions of CCX gates on 3 qubits into single and two-qubit gates [292]. Various optimised decompositions for sequences of Toffoli gates have also been similarly developed [285, 12, 107, 260, 108, 14, 2, 146]. Mohammadi and Eshghi introduced 4-valued truth tables to extend classical circuit synthesis to include $\sqrt{X}$ (also known as V) gates [231]. References [298] as well as [261] incorporated controlled- V gates into template matching strategies and showed significant improvements in synthesised gate count. Finally, [219] proposed decomposing Toffolis only up to relative phase, introducing a lot of freedom in the quantum decompositions that are required compared to the traditional classical decompositions.

In summary, a variety of scalable representations – such as phase polynomials, Pauli gadgets, Clifford tableaux, diagrammatic calculi, and reversible circuits – have been developed to abstract computations and enable highly tailored optimisation methods. These approaches leverage the unique structure and symmetries of quantum computations, achieving significant reductions in circuit size, depth, and hardware-specific overheads. Techniques such as phase polynomial synthesis and Clifford tableau representations are widely applicable and are a cornerstone of modern quantum compilers [9, 225]. Meanwhile, diagrammatic calculi, such as the ZX calculus, provide a flexible and theoretically robust framework for optimisations, often revealing simplifications invisible in the traditional gate-based model.

2.3. Rise of hybrid quantum-classical computation

Quantum measurements

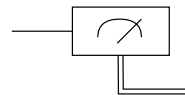
We have, until now, skipped over a crucial part of the quantum computation process: the role of quantum measurements. Quantum data, in isolation, is inherently inaccessible to us and the broader macroscopic world. A result from a quantum computation is only of value if we can probe it and get some readout value that we can display to the user or return to whoever launched the quantum computation.

1. It is ironic that Schrödinger's thought experiment [283], intended to highlight the absurdity of quantum mechanics, has become the field's most famous PR campaign. Sorry to disappoint – you won't find felines occupying multiple states of existence (though qubits do!) ↪
2. This is known as *state tomography* [6]. One must perform measurements in multiple bases, i.e., different choices of classical states to project to. ↪
3. Where did the qubit go? All the information in a qubit post-measurement is also contained in the classical bit of output data – it is, therefore, redundant and renders the qubit useless. In our model, we, therefore, bundle measurement and qubit discard into one operation. ↪

Quantum physics measurements fundamentally differ from our classical understanding of just “reading out” data that is already there. This is the famous Schrödinger's cat thought experiment of quantum mechanics: what data is within the qubits remains undefined until a measurement is performed. The act of observation will transform the quantum data: looking inside the box will, at random, either kill the cat or spare it¹.

We thus need to add the measurement operation as a special case to our computer scientist's model of quantum computing. Unlike purely quantum operations, measurements inherently involve interaction with the environment to produce a readout. Consequently, the no-delete and reversibility principles discussed earlier do not apply. Indeed, measurement is a lossy (and therefore irreversible) operation that projects the quantum state into one of a small subset of classical states. Which state the quantum state is projected into is non-deterministic. If one has access to an infinite supply of the same quantum state, then the whole state can be reconstructed by repeating measurements and analysing the distribution of outcomes². Given no-cloning, however, this is unlikely to be the case, and so the full quantum result is hardly ever known. Instead, we must rely on well-designed measurement schemes to extract useful information from our partial access to the quantum states.

We model measurement as an operation that takes one qubit and outputs one purely classical bit³. In the circuit formalism, measurements are often implicitly added at the end of every qubit. Suppose we wish to make them explicit or add them elsewhere in the computation. In that case, we must introduce a graphical representation for the classical bit of data the measurement produces. The field has adopted the double-wire $\equiv$ for this, even though a “half” wire would arguably have been more appropriate to reflect the reduced information content relative to quantum wires. I present to you the measurement box:



Measurements as first-class citizens

It is very tempting to our feeble classical brains – and admittedly, we just did it ourselves in the previous paragraphs – to view measurements as merely a readout operation, an auxiliary operation that we are forced to perform at the end of a computation for operative reasons. This could not be further from the truth! In many ways, measurements are just as powerful tools as any other quantum operation – if not more so!

One eye-opening perspective on this is the field of *measurement-based quantum computing* (MBQC). Raussendorf and Briegel showed indeed [265] that arbitrary quantum computations can be reproduced in the MBQC framework using only some resource quantum states that can be prepared ahead of time and measurements! In other words, given entangled qubits, measurements are all you need to perform quantum operations.

We will not explore MBQC further in this chapter (nor in this thesis, for that matter). Instead, we will use this as a motivation to explore what we can achieve with

measurements. We have so far spared you from any mathematical alphabet soup. As we start discussing more concrete constructions of quantum computations, some introductory linear algebra and conventions around notation will become unavoidable.

Dirac formalism. Quantum states are nearly unanimously written using *kets*: instead of referring to a quantum state as ψ , we write it wrapped in special brackets as $|\psi\rangle$. This notation is also used when referring to the 0 and 1 states of qubits, written $|0\rangle$ and $|1\rangle$.

Several states can be joined and considered together as one overall state. This is expressed using the tensor $\otimes$ symbol: $|\psi_1\rangle \otimes |\psi_2\rangle$ is the joint system of $|\psi_1\rangle$ and $|\psi_2\rangle$. When the states in question are all explicitly qubit states, we use the shorthand binary notation $|0\rangle \otimes |1\rangle \otimes |0\rangle = |010\rangle$.

We will introduce more notation along the way.

With this out of the way, let us look in more details at the first smart use of measurements: the block-encoding technique. Consider the following scenario: you would like to perform an operation A on an arbitrary quantum state $|\psi\rangle$. Now, there are, unfortunately, many cases where implementing A as a quantum circuit made of primitive gates that can be executed on hardware is very expensive⁴.

However, what we can always do is express A as a matrix of dimensions $2^n \times 2^n$, where n is the number of qubits in the state $|\psi\rangle$. Then, there is a neat trick that we can sometimes apply: instead of trying to execute A , we enlarge the matrix to a bigger $\tilde{A}$:

$$\tilde{A} = \begin{pmatrix} A & G_1 \\ G_2 & G_3 \end{pmatrix}$$

where G_1, G_2 and G_3 are “garbage” matrices that we do not care about, but should combine into a matrix $\tilde{A}$ that we know how to execute on a quantum computer. Quantum computations must be matrices with a row and column number that is a power of two; so at a minimum, $\tilde{A}$ must be of size $2^{n+1} \times 2^{n+1}$, i.e. be a computation on $n + 1$ qubits.

We restrict our considerations in the following to the case on $m = n + 1$ qubits – other cases are similar. We thus need to add a qubit to our $|\psi\rangle$ state to be able to pass it to our new operation. Such qubits that are added temporarily to facilitate a computation are a recurring feature in quantum computing and have thus earned themselves a name – ancilla qubits.

Let us take a look at the quantum states that result from executing $\tilde{A}$. If we add to $|\psi\rangle$ a $|0\rangle$ ancilla state, our quantum operation acts as⁵

$$\tilde{A}(|0\rangle \otimes |\psi\rangle) = |0\rangle \otimes A|\psi\rangle + |1\rangle \otimes G_1|\psi\rangle. \quad (3)$$

The expression $A|\psi\rangle$ means the operation A applied to $|\psi\rangle$ – exactly the output state we are seeking. If we input the ancilla qubit in state $|1\rangle$, we get garbage:

4. or outright impossible, in cases where A is not a unitary linear operation, for example. ↩

5. This is obtained by a simple matrix multiplication. The vector representation of the quantum state $|0\rangle \otimes \psi$ is obtained using the Kronecker product. You can also just trust me that this works out this way. ↩

So $|0\rangle \otimes |\psi\rangle$ is definitely the input state we are more interested in.

6. This is simplifying slightly. There is a necessary condition for this to be a valid measurement: the states in the sum must form a measurement basis, i.e. they must be orthogonal. This is satisfied here. ↪

How can we recover A from (3)? This is precisely what measurements do! When quantum states are expressed as sum of states, the terms of the sum form the possible measurement outcomes⁶. If we only measure a subset of the qubits, then the term corresponding to that measurement is isolated and all other terms disappear. Hence, if we measure the first qubit (that we introduced ourselves) in the zero state, then the remaining n qubits will be precisely in the desired state $A|\psi\rangle$. Success!

Using this “term isolating” property of measurements, known as *state collapse*, we can thus effect computations that would have been otherwise difficult or impossible to perform. There is however one important wrinkle that we cannot forget about: measurements are non-deterministic! We cannot assume that all measurements of the ancilla qubit will return the zero state. When $|1\rangle$ is measured on the ancilla, the remaining qubits are left in the $G_1|\psi\rangle$ state. The computation has thus failed, and the execution must be aborted and restarted. How often the block-encoding protocol that we have presented fails depends on the details of A and the choices of G_1, G_2 and G_3 and is the main disadvantage of an otherwise very powerful quantum technique.

We will now explore two strategies to deal with “fails” in measurements. At the core of them is the idea of hybrid quantum-classical programs.

Who said quantum computers could not fix their mistakes

Failed computations are an expensive mistake in quantum computing as the no-cloning theorem prevents us from keeping a “backup” of the initial state. The fact that failures are in fact unlucky measurement outputs makes matters worse, given that measurements are the only irreversible quantum operation. It is therefore impossible in general to recover from a “wrong” measurement.

There are, however, prominent cases in which the computation *can* be corrected based on the measurement outcome, thus yielding deterministic results. Recall equation (3) of the previous section: there is a computation A on n qubits, that can be probabilistically computed using $m = n + 1$ qubits using $\tilde{A}$:

$$\tilde{A}(|0\rangle \otimes |\psi\rangle) = |0\rangle \otimes (A|\psi\rangle) + |1\rangle \otimes (G|\psi\rangle),$$

7. Notice that, informally, we would hope to get a computation G such that $G \approx A$ in the sense that it should somehow be closely related to A . This way, the resulting correction $A \circ G^{-1} \approx Id$ would be close to the identity, and would be cheap to compute. ↪

for some “garbage” G . What if G is a reversible operation, i.e. there is an operation G^{-1} to undo G ? Well then, we can still, at least in theory, recover $A|\psi\rangle$ by applying $A \circ G^{-1}$:

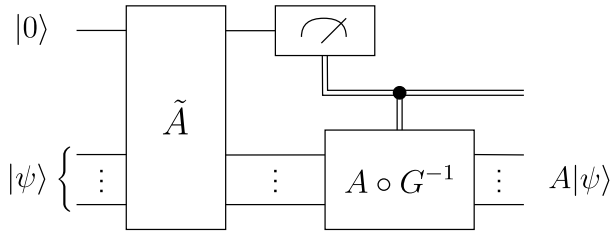
$$G|\psi\rangle \mapsto (A \circ G^{-1}) \circ G|\psi\rangle = A|\psi\rangle,$$

but *only if* **1** was measured on the ancilla qubit⁷!

This is the beginning of quantum-classical hybrid computing: we start by performing quantum operations followed by measurements, the outcomes of which dic-

certain classical bit (the *condition*) is set. This bit will typically be a value derived from a previous measurement: it could be as simple as the outcome that a previous measurement yield, or a function of multiple past outcomes that must be evaluated on classical hardware (e.g. a CPU).

Mixing classical and quantum operations is a sure way to bring the quantum circuit representation to its knees. We adopt the following representation, in which a quantum gate that has an additional classical bit wire attached to it represents a classically controlled operation that is only executed if the bit value is **1**.

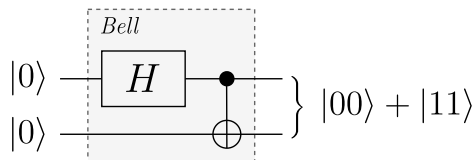


Quantum Teleportation

Quantum teleportation is a simple example of performing classically controlled quantum operations to do circuit corrections based on measurement outcomes. It is also coincidentally one of the most fundamental protocols of quantum theory. Its name is slightly misleading. Think of it as data transfer for quantum data, with a mind-bending twist: at the time of the transfer, only classical data must be communicated between the sending and receiving parties. As a result of this protocol, quantum information can be transferred using plain old-school copper wires (or any other classical communication channels)!

This is predicated on one crucial action being performed before the start of the communication. For every qubit that should be transmitted, the parties must beforehand create and share among themselves a pair of qubits that will serve as the quantum resource during the protocol execution. This resource state is widespread enough that it got its name: the Bell pair state. It is written in Dirac notation as $|00\rangle + |11\rangle$. As the notation indicates, it is a state with perfectly correlated measurements: when measured, the two qubits will always yield the same outcome, either both **0** or both **1**.

There turns out to be a straightforward circuit that maps the two-qubit $|00\rangle$, which every two-qubit computation starts in, into the Bell pair state:



It is enough for us to think of it as a black box – or a grey box in this case.

We are interested in “teleporting” an arbitrary, single-qubit quantum state. Such a state can always be expressed as $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, i.e. in the most general case, a one-qubit state will be in some superposition of the states $|0\rangle$ and $|1\rangle$. The parameters α and β are complex coefficients that encode the probabilities of measuring 0 or 1 – we can view them as the weights of a weighted sum.

We are now interested in combining a Bell resource state in a joint system with the arbitrary state $|\psi\rangle$. The resulting three-qubit state is obtained with the $\otimes$ operation, which distributes over sums just like usual multiplication:

$$\underbrace{(|00\rangle + |11\rangle)}_{\text{first two qubits}} \otimes \underbrace{(\alpha|0\rangle + \beta|1\rangle)}_{\text{third qubit}} = \alpha|000\rangle + \alpha|110\rangle + \beta|001\rangle + \beta|111\rangle. \quad (4)$$

We chose to place the Bell pair on the first two qubits and the arbitrary state on the third. The goal is to move the data that sits on that last qubit to the first qubit. Looking at the first qubit in the above expression, notice that the desired state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ appears in the first qubit if we can discard the second and third terms:

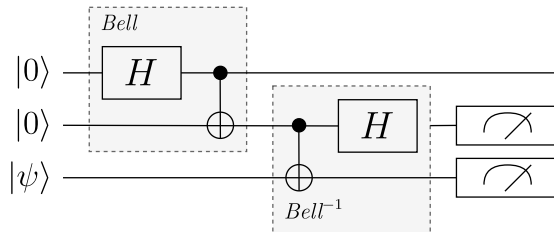
$$\alpha|000\rangle + (\alpha|110\rangle + \beta|001\rangle) + \beta|111\rangle$$

8. Apologies, it seems at this point that we are conjuring up a complex expression out of nowhere. It is in fact just a change of basis – plain old linear algebra. The formula can be obtained easily by writing out the basis change matrix. ↩

This sounds very much like the measurement operations we have used before to isolate terms – but we need to isolate two terms simultaneously. We can resolve this issue by reorganising the expression⁸

$$\begin{aligned} & \alpha|000\rangle + \alpha|110\rangle + \beta|001\rangle + \beta|111\rangle \\ &= \underbrace{(\alpha|0\rangle + \beta|1\rangle)}_{=|\psi\rangle} \otimes \underbrace{(|00\rangle + |11\rangle)}_{\text{Bell pair}} \\ &+ (\beta|0\rangle + \alpha|1\rangle) \otimes (|01\rangle + |10\rangle) \\ &+ (\alpha|0\rangle - \beta|1\rangle) \otimes (|01\rangle - |10\rangle) \\ &+ (\beta|0\rangle - \alpha|1\rangle) \otimes (|00\rangle - |11\rangle) \end{aligned}$$

Obtaining the $|\psi\rangle$ state on the first qubit is thus as simple as isolating the first of these four terms. We do not know a priori how to measure $|00\rangle + |11\rangle$ but we do know how to map that state to $|00\rangle$: that’s the inverse of the Bell pair state preparation circuit! This results in the following circuit:



This brings us to the same situation as we had for the block encoding application above: conditioned on the measurement outcome of the second and third qubits being 0, the computation performs a state “teleportation”, moving $|\psi\rangle$ from the third to the first qubit. We can compute the effect of $Bell^{-1}$ on the overall expression of (4) to find all possible output states:

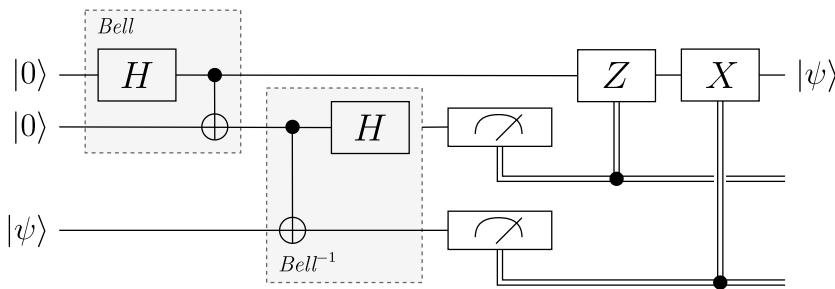
$$\begin{aligned} \alpha |000\rangle + \alpha |110\rangle + \beta |001\rangle + \beta |111\rangle &\xrightarrow{Bell^{-1}} (\alpha |0\rangle + \beta |1\rangle) \otimes |00\rangle \\ &\quad + (\beta |0\rangle + \alpha |1\rangle) \otimes |01\rangle \\ &\quad + (\alpha |0\rangle - \beta |1\rangle) \otimes |10\rangle \\ &\quad + (\beta |0\rangle - \alpha |1\rangle) \otimes |11\rangle \end{aligned}$$

As expected, we do get $|\psi\rangle$ on the first qubit for the measurement 00 (corresponding to the state $|00\rangle$), but as it stands, this only has a $\frac{1}{4}$ probability of success.

You might notice, however, that the other states in which the first qubit can end up look remarkably similar, up to some sign flips and swaps $|0\rangle \leftrightarrow |1\rangle$. In particular, all states still have the amplitudes α and β somewhere, so it does not seem unfathomable that these “wrong” states can be mapped back to ψ .

We can use the measurement outcomes of the second and third qubit to infer which of the “mistakes” occurred, and hence what state the first qubit has ended in. The 01 measurement outcome, for instance, results in the $\beta |0\rangle + \alpha |1\rangle$ state – this is just a bit flip away from $|\psi\rangle$! This gate is known as X . Its colleague the Z gate on the other hand leaves $|0\rangle$ states untouched but flips the *sign* of $|1\rangle$. This would fix the 10 outcome. Finally, 11 requires both a Z and a X correction.

Putting these observations together, we can leverage classically controlled operations to obtain a fully deterministic protocol! The correct circuit implementing quantum teleportation is given by



In the scenario where a first party (Alice) wants to send a one-qubit quantum state to Bob, they can achieve that by creating a Bell pair state, the first qubit of which is given to Bob and the second to Alice. When Alice then gets in possession of another qubit $|\psi\rangle$ whose data she wants to transmit to Bob, she can achieve that by executing $Bell^{-1}$, measuring her two qubits and communicating the (classical) measurement outcomes to Bob. Bob can perform the necessary corrections and will then have state $|\psi\rangle$.

It is beautiful and often overlooked how one of the most fundamental protocols of quantum information theory is, in fact, a hybrid classical-quantum computation. Quantum teleportation without classical communication is physically impossible: it would let Alice communicate with Bob instantly, even though he could be light years away – in other words, it would fundamentally break relativity.

Repeat until success: If you fail, retry!

9. This is fiendishly effective: the [Hoeffding bounds](#) guarantee that the probability of success will converge to 1 exponentially with the number of runs. ↪

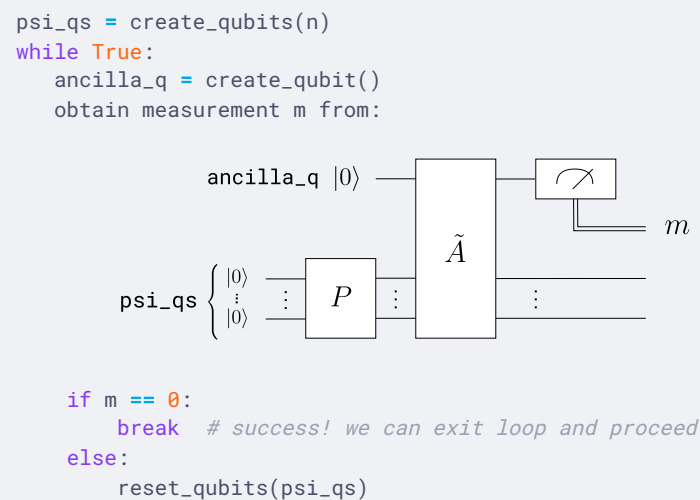
Classical computer science has a straightforward solution whenever probabilistic computations that can fail are used: probability amplification or boosting [280]. The idea is so simple that it barely deserves a name: execute several independent runs of the computation and choose the most common outcome. If the probability of failure is below a certain threshold (e.g. 50% for a binary output), then with basic statistics, one can extrapolate the number of runs required to obtain any desired accuracy⁹.

We have been ignoring this approach so far since no-cloning prohibits us from repeating a procedure more than once on an input state $|\psi\rangle$. However, in the scenario that the computation should only be executed on a specific, known input state and the computation that prepares that state is known, we can recover from computation failures by just preparing a new state identically.

Suppose we know how to execute the quantum computation P mapping

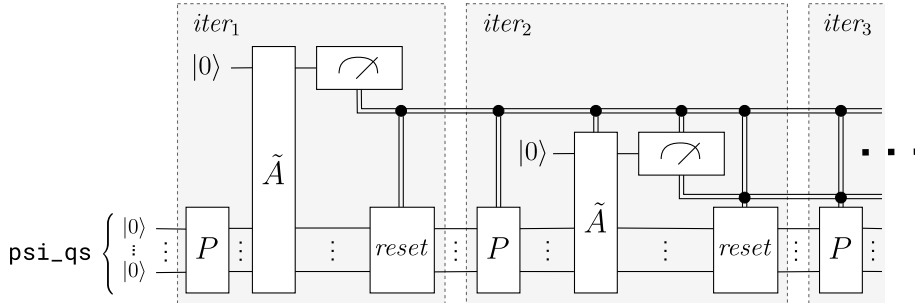
$$|0 \dots 0\rangle \mapsto P |0 \dots 0\rangle = |\psi\rangle.$$

As before, we would like to compute A given an implementation of the computation $\tilde{A}$ that acts on a n -qubit state $|\psi\rangle$ and an ancilla qubit in the $|0\rangle$ state. If the measurement of $\tilde{A}(|0\rangle \otimes |\psi\rangle)$ returns **1**, then the computation failed. We can then discard all qubits and restart from the $|0\rangle$ state, applying P followed by $\tilde{A}$ and an ancilla measurement, repeating until we measure 0. As a pseudo-quantum circuit, we could express this as:



At each iteration, we can either exit the loop if the state collapse was successful ($m == 0$), or reset the qubits to zero and try again. But pseudo circuits do not run

on hardware! The only way to express this computation as an actual circuit is to unroll the loop, i.e. repeat the block within the loop as many times as we expect might be necessary¹⁰. The first two iterations would look as follows:



It should be obvious why we haven't unrolled the loop any further – it quickly becomes unwieldy. The resulting program is not only hard to display and read, but it also suffers from fundamental issues in practice. For one, the program size becomes hugely bloated, and beyond slowing down the compiler, it will also cause a host of issues on the control hardware in real-time, such as long load times, inefficient execution, and low cache efficiency.

Even more worryingly, when picking the maximum number of iterations, we face an impossible tradeoff: if the number of iterations is small, then the probability of failure will remain non-negligible. As we scale this value up, however, we are introducing more and more gates into the program to cover the odd case of multiple successive repeated failures. We do not intend to execute these gates on most computation runs. They come at a significant cost to the runtime. For each gate listed in the circuit, the condition for the gate's execution must be checked, whether or not the gate ends up being executed. Furthermore, hardware schedulers might be forced to be pessimistic and schedule a time window for all conditional operations ahead of time. This will significantly delay any operation to be performed after the loop.

We, therefore, argue that the quantum circuit model is ill-suited as the representation for quantum programs that combine classical and quantum data. Such programs, however, are a fundamental building block towards developing meaningful large-scale quantum computations and are bound to become the norm.

Beyond the examples discussed above—including block-encodings, repeat-until-success schemes, distributed quantum computing and measurement-based quantum computing – one application of hybrid quantum-classical operations stands out as critically important for the large-scale deployment of quantum computing: quantum error correction (QEC) schemes. We discuss this use case in the next section.

10. In other words, we must pick a constant M for the maximum number of times we expect the loop to be executed. If a single loop iteration has a failure probability of p , the failure probability of the program with M unrolled iteration is then p^M . ↩

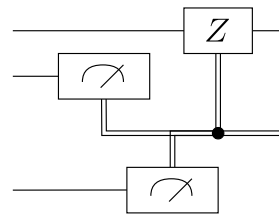
2.4. Quantum compilers cannot do it alone

We have (hopefully!) by now convinced our readership that quantum programs must interface with our established classical infrastructure and should rather be understood as an interleaved execution of both classical and quantum operations. The obvious question that thus poses itself is

How do we equip quantum compilers to deal with classical operations?

The simplest solution is to adopt the extended quantum circuit formalism with support for classically controlled operations, as we have introduced in the previous section. Using this representation, the basic types available for computation are the qubit and the classical bit. We can also, at that point, introduce purely classical operations on bits, for instance, to compute boolean logic on measurement outcomes, such as “if both the first AND the second measurement outcomes are 1, then ...”.

However, the circuit model is inherently designed with the no-cloning principle in mind: specifically, with the assumption that at any one time, there are exactly n (for some fixed value of n) resources available for computation. This for example means that in the following program



1. Or, as we would say in programming parlance, to *allocate*. ↪

in which two measurements write to the same classical bit, it would be impossible to append a gate controlled on the first measurement outcome after the Z gate, as that value was overwritten on the classical wire by the second measurement. The solution could be to introduce¹ a new, fresh classical wire for each measurement and avoid overwriting outcomes. However, there are also many other ways to break this wires-based representation: suppose you have an operation with one input and two outputs, such as a copy operation $x \mapsto (x, x)$. We would need two wires for the output, but the input would only provide us with one... We now have to start creating additional wires ahead of time for this purpose and solve memory allocation problems to decide which wire should be given to which operation.

These are run-of-the-mill classical compiler problems! One might at first hope that the set of overlapping problems between classical and quantum compilers is manageably small. After all, in all the use cases we have covered so far, the amount of classical computation was very minimal, limiting itself to conditionals and loops based on simple boolean expressions. Surely the full-blown powers of a classical compiler are not required!

Unfortunately (and as usual), scientists have shown no lack of imagination in this field – and so have found very compelling use cases for complex classical comput-

ations within quantum programs. To drive this point home, let us consider the concrete example of quantum error correction.

The quantum error correction use case

Error-correcting protocols do as their name suggests: they detect whenever data is subjected to errors and thus modified in an unexpected way. They then attempt to recover the intended valid state. In the classical world, such schemes are employed whenever the hardware is not reliable enough: this is hardly the case for computations themselves but is widespread in communications (e.g. within the TCP/IP protocol for the internet [123]) or for memory and storage in critical applications.

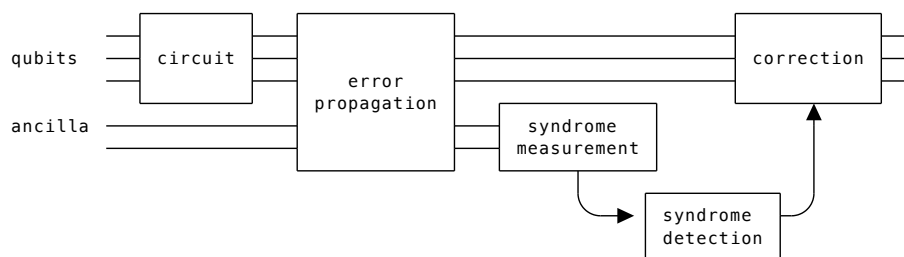
No one expects to be able to manipulate matter-based qubits without introducing errors for a very long time. Photons, on the other hand, are prone to data losses through absorption and can only be entangled using complex and noisy schemes such as the Knill–Laflamme–Milburn protocol [194]². Simply put, it is safe to assume that error correction will be found everywhere – as soon as our quantum computers manage to implement such protocols.

A sketch of quantum error correction goes roughly as follows: the data that would be stored on k qubits is instead encoded in a redundant way on a larger number $n > k$ of qubits. Thus, when errors occur on a subset of the n qubits, the data can be restored using the qubits that have not been corrupted. Before errors can be corrected, they must be detected. To this end, we first add fresh ancilla qubits to the program. Through smartly designed interactions with the data qubits, the ancilla qubits pick up the errors from the data. When we subsequently perform measurements on the ancilla qubits, these errors result in modified outcomes, called the error *syndrome*.

The challenging bit comes next: from a run of syndrome measurements, one must infer the most likely errors – a step known as *syndrome decoding*. This is a purely classical maximum likelihood problem that requires a non-trivial amount of computations to resolve. For small problem instances, all possible input syndromes can be tabled, and the outputs precomputed – in which case the problem at runtime is reduced to fast table lookups. However, the higher the fault tolerance we require, the more qubits must be used in the encodings, and so invariably, the problem quickly becomes very demanding computationally.

Meanwhile, these “cycles” of error detection and correction are under strict latency constraints: idling qubits waiting for corrections to be applied will accumulate new errors that must themselves be corrected – for error correction to be workable, we must be capable of detecting and correcting for errors faster than they are being introduced. The entire error correction cycle just described can be summarised by the following diagram:

2. We should at this point – at the risk of stoking controversy – acknowledge the commendable efforts of scientists chasing the Majorana particle [279, 160, 236]. The topological quantum computers these would enable are, to our knowledge, the only quantum architecture proposed that could do away with error correction. ↩



The decoding time is a crucial factor in determining the overall cycle time and, thus, the clock rate of fault-tolerant quantum hardware. Consider, for example, a 32-qubit Toric code [192], one of the most well-studied quantum error-correcting codes. Without going into the details of the code itself, we can use the C++ implementation made available by the MQT toolkit [69] to study the decoder performance for this code.

3. Here we are using Apple clang v15.0.0, running macOS 14.7 on an Apple M3 Max chip. ↩

Consider first a “naive” compilation of the decoder – the kind of program that we could hope to get from a quantum compiler that “understands” classical operations but only implements optimisations directly relevant to quantum computations. Such a compiler does not currently exist, but the decoder being a C++ program, we can approximate what the compiled binary would look like by turning off all optimisations from an established classical compiler³.

The runtime averaged over 1000 runs of the decoder is 0.73 ± 0.06 ms. This is within the latency requirements of certain trapped ion architectures [275], but far beyond the sub-microsecond regime that will be required to make error correction a reality on superconduction-based quantum computers [74]. This can be contrasted with the program output by the same compiler, but with all compiler optimisations enabled: the average runtime is reduced by a factor close to 10x to 0.078 ± 0.004 ms – still a factor 100x away from the required performance on superconductors, but huge gains nonetheless! The details of the experiment with all build flags, the hardware used and how to reproduce the results are available [here](#).

There is no hope of obtaining these types of speedups without an in-depth understanding of classical hardware and battle-tested implementations for every optimisation pass under the sun – in short, the full thrust of a modern state-of-the-art compiler such as clang or gcc.

To make matters worse, such classical computations are bound to move to dedicated accelerators that require specialised compilation, such as GPUs and FPGAs, for the most time-critical subroutines: quantum error decoding using GPUs is already well-developed [42, 71] and more esoteric platforms FPGAs [245, 226], superconducting circuits [310] and compute-in-memory architectures [315] are being actively studied.

These observations should leave the reader convinced that in order to compile and realise the kind of hybrid quantum-classical programs that we expect will become the norm in the field, quantum compilers will need to embrace and encompass the full breadth and depth of classical compilers. This leaves us with no choice but to fully transform and integrate the existing quantum tooling and quantum optimisation research into the established compiler ecosystem. What this means exactly is the subject of the rest of this chapter.

A new quantum programming paradigm?

We have seen it – quantum circuits are very limited in their expressiveness. They are well suited to presenting sequences of purely quantum operations and how the computation is parallelised across qubits, but they quickly become limiting once both quantum and classical data types are mixed and any type of control flow (conditionals, loops, function calls, etc.) is introduced.

How users express programs in the front end has deep implications for the kind of computations that the compiler must be capable of reasoning about and, hence, for the compiler’s architecture. The great merging of classical and quantum compilers is the perfect opportunity to reconcile program representations and integrate the learnings from decades of classical programming language research into quantum computing.

There have been several trailblazing initiatives to formalise quantum programming and create dedicated languages, such as QCL [244], Quipper [158, 270, 145], Q# [229] and Silq [51]. Their adoption in the quantum ecosystem have so far remained limited, overshadowed by the popularity of python-based APIs for quantum circuit-based representations, as offered by Qiskit [179], PennyLane [46] and Cirq [79]. There is, as a result, a justified dose of scepticism in the quantum community on how well the ideas from classical programming really translate to quantum.

It is thus all the more notable that we are seeing a new generation of quantum programming tooling being developed [197, 177, 102], driven by the need to write more expressive programs for the improving hardware (as we have been discussing), as well as for performance reasons, to scale quantum compilation to large scale [176], accelerate quantum simulations [177] and integrate with classical high-performance computing (HPC) [241].

The history of programming is, first and foremost, a masterclass in constructing abstractions. Many of the higher-level primitives, that have proven invaluable classically, solve problems that we expect to encounter very soon in our hybrid programs – when we have not already. Examples include

- **structured control flow** to simplify reasoning about branching in quantum-classical hybrid programs,
- **type systems** to encode program logic and catch errors at compile time – this is particularly important for quantum programs as there is no graceful way of handling runtime errors on quantum hardware: by the time the error has been propagated to the caller, all quantum data stored on qubits is probably corrupted and lost,
- **memory management** such as reference counting and data ownership models. Current hardware follows a static memory model, in which the number of available qubits is fixed, and every operation acts on a set of qubits assigned at compile time. This becomes impossible to keep track of in instances such as qubit allocations within loops with an unknown number of iterations at compile time. It thus becomes necessary to manage qubits dynamically, just like classical memory.

To facilitate such a large swath of abstractions, the first step quantum compilers must take is to make a distinction between the language frontend and the *intermediate representation* (IR) that the compiler uses to reason about the program

and perform optimisations. This will be the topic of chapter 3. The graph-based IR that we introduce in that chapter will then form the foundation for the new quantum compilation techniques that will be developed throughout the remainder of the thesis.

2.5. Summary and further reading

This introductory chapter covered some of the basic principles of quantum computation and, in doing so, hopefully, made a convincing argument as to why we should expect the programs running on quantum hardware to become more complex in the future, with the intertwining of classical and quantum computations – processes we refer to as hybrid quantum-classical programs. Prior to that, we also presented quantum compilation, an emerging discipline that is introducing many new problems and ideas to the established corpus of work on compiler research.

1. And while we're on the topic of my supervisor's brilliant work, there is also a very recent textbook, a sort of spiritual successor to [84], particularly focused on quantum compilation [189]. It is just as worth a read and might appeal more to the computer science-y reader. ↪

If this quantum taster has intrigued you or you would like to learn the basics from people who *actually* know what they are talking about, nothing beats the reference book for quantum information and quantum computing by Nielsen and Chuang [238]. A fascinating alternative perspective on quantum theory has also been developed within the programme of categorical quantum mechanics, for which the illustrious "Dodo book" [84] would be the go-to introductory material¹.

At the risk of turning this thesis into absolutely shameless Oxford self-promotion, guess what else was a product of this university's *world-class* research? The quantum circuit itself! These diagrams came from theoretical physicists (no surprise here) interested in capturing thought experiments in quantum information theory [113].

2. That is *classical* software written to control and optimise quantum computations. ↪

The idea caught on, and soon, software tools were created to facilitate building such diagrams. The Quantum Computation Language (QCL) was one of the first [244]. Quantum software² has since proliferated, especially as the possibility of actually *performing* these thought experiments on quantum hardware became more tangible. The result were software packages for quantum computing, designed for the automatic transformation and *optimisation* of quantum computations for execution on real hardware [179, 79, 300, 294] – we called them quantum compilers.

A recent development for quantum compilers focuses on scalability and first-class support for hybrid quantum-classical computations. Quantum circuits that include some form of classical control have been variously called "dynamic circuits" (e.g. [90]), "adaptive circuits" (e.g. [296]), "circuits with measurements and feedforward" (e.g. [156]), and "circuits assisted by local operations and classical communication" (e.g. [254]).

Besides supporting advances in quantum hardware [90, 156, 253], hybrid classical-quantum computations are central to many quantum computing applications. As put recently by Alam and Clark [5]:

"[...] dynamic quantum circuits are a crucial milestone on the roadmap to fault-tolerant quantum computers."

We have covered a small subset of applications of hybrid quantum-classical computations. Quantum teleportation is undoubtedly one of the oldest [45]. The

block-encoding technique that we discussed in section 2.3 is the foundation of several algorithms, including the Quantum Singular Value Decomposition (QSVD) [150] and the Linear Combination of Unitaries (LCU) [75,304]. Measurement-based quantum computing (MBQC) was introduced in [265] and is forming the base for some photonic quantum computing architectures [39,62]. Hybrid programs have also been shown to be useful for implementing the Quantum Fourier Transform (QFT) [41] and the Quantum Phase Estimation (QPE) algorithms [90], two of the most fundamental computation primitives for quantum algorithms.

On the other hand, repeat until success schemes [247] are widespread in state preparation routines and will play a key role in fault-tolerant (FT) quantum computing. Arguably, the most well-known scheme for FT is magic state distillation [64], a procedure expected to be a core building block of many FT architectures. State preparation is generally a ubiquitous problem for FT, as the error-correcting codes that are employed initiate computations starting from a *logical* zero state, which may be expensive to prepare on the qubits of the hardware [143].

Finally, quantum error-correcting (QEC) codes themselves must be implemented using hybrid programs. The quantum error correction (QEC) literature is vast and can get very technical very quickly, but diving into it promises bountiful rewards. The field is one of quantum information's fastest-evolving areas of research. These work-in-progress lecture notes [155] by a coryphaeus of the field make for excellent introductory material.

Quantum Compilation as a Graph Transformation Problem

The specialised optimisation techniques that we reviewed in section 2.2 are effective for the scenarios they were designed for, but they are challenging to adapt to new hardware primitives, constraints, or cost functions.

This thesis proposes interpreting quantum compilation as a graph transformation system (GTS). GTSs endow quantum compilation with well-defined semantics and strong theoretical foundations [203]. They establish a practical, purely declarative framework in which compiler transformations can be defined and studied.

This allows us to decouple the semantics of quantum programs and the architecture specifics from the compiler infrastructure itself. We can thus focus on building and designing scalable and efficient graph transformation algorithms that can then be applied on a wide range of compilation problems and hardware targets.

In this chapter, we formalise quantum computation and optimisation based on graphs and graph transformations, providing the foundation for all considerations in later chapters. Albeit slightly simplified, the intermediate representation IR we propose here is based on joint work [216], as well as [ongoing development](#).

The words *graph rewrite* and *graph transformation* are often used interchangeably in the literature. In the context of this thesis, we will take these words to distinguish two slightly different problems:

The study of equivalences and other relations between graphs under well-defined semantics is the subject of **graph transformations**. For instance:

- a *graph transformation rule* $L \rightarrow R$ (Definition .) expresses that an instance of L can always be transformed into an instance of R , reflecting the semantics of the system that the graph is modelling.
- a minIR equivalence class (Definition .) is an instance of a *graph transformation system* (GTS), which uses known semantic relations, expressed, for example, as graph transformation rules, to define how graphs can be transformed.

Graph rewriting, on the other hand, encapsulates the algorithmic procedures and data structures that mutate graphs. A *rewrite* (Definition 3.9) is the tuple of data required to turn a graph G into a new graph G' .

Given matches of patterns L on a graph G , a *graph transformation system* can consider the set of *graph transformation rules* that define the semantics of G to produce a set of *rewrites* that can be applied to G and mutate G .

Our contributions in the subsequent chapters are mostly preoccupied with problems of *graph rewriting*, i.e. the definition and application of the data required to mutate graphs, as opposed to *graph transformations*. This chapter nonetheless considers both, using the mature graph transformation framework as a foundation to define IR rewriting semantics.

Section 3.1 starts with a review of previous related work at the intersection of graph transformation software and quantum program optimisation. We then discuss in section 3.2 a fundamental difference between classical computation graphs and the requirements of quantum computation. This motivates a new graph-based IR tailored to quantum computation that we present in section 3.3, along with formal graph rewriting semantics based on sesqui-pushout (SqPO) transformations (section 3.4). Whilst the SqPO transformation definition is constructive, its existence is not guaranteed. We conclude the chapter in section 3.5 by discussing a more restricted “operational” notion of graph rewriting that will be useful for the rest of the thesis.

3.1. Related work

Graph rewriting on computation graphs. Optimisation of computation graphs is a long-standing problem in computer science that is seeing renewed interest in the compiler [206], machine learning (ML) [181, 131] and quantum computing communities [332, 333]. In all these domains, graphs encode computations that are either expensive to execute or evaluated repeatedly over many iterations, making the optimisation of the execution cost of the computation a primary concern.

Domain-specific heuristics are the most common approach in compiler optimisations [250, 294] – a more flexible alternative are optimisation engines based on declarative sets of graph transformations [56, 57]. In such systems, a graph transformation system (GTS) is used to find a sequence of allowed transformations that rewrite a computation graph given as input into a computation graph with minimal cost.

Transformation systems were first studied on strings [111], then generalised to trees and terms [49], before being applied to graph domains [124, 274, 198]. Their use in quantum computing is part of a long tradition of diagrammatic reasoning in physics [252, 138], and particularly in quantum mechanics with the advent of categorical quantum mechanics [3, 86, 84].

GTS in quantum computing. In quantum computing, the ZX calculus [85] and other diagrammatic theories that derive from it are particularly important. Properties of GTSs such as completeness, confluence and termination [314] are well-studied within this field [21, 20, 50]. These results have formed the basis for

software implementations of circuit optimisations with soundness and performance guarantees [119, 187, 294, 60].

Great strides are also being made in our theoretical understanding of transformation systems for quantum circuits. Recently, Clément et al. established completeness for the first time [80] as well as minimality [81] of a GTS for quantum circuits. A set of circuit transformation rules were presented such that no rule is redundant, and for any two equivalent quantum circuits, there exists a sequence of local transformations rewriting one into the other. Such systems are however not confluent, and this is unlikely to change: most circuit optimisation problems are known to be computationally hard [322].

There is also another inherent tension in integrating diagrammatic calculi into compilers. Diagrammatic theories arise from abstract primitives that admit a simple rewriting logic [167, 59, 134, 72]; compilers meanwhile must capture all the expressivity, constraints and messiness of real-world hardware targets, with all the edge cases and exceptions that this entails.

An example of this is the ZX circuit extraction problem [259, 22]: it is in general hard to recover an executable quantum circuit from a ZX diagram as the latter is strictly more general and primitives cannot be mapped one-to-one. Similarly, while simple quantum-classical hybrid computations can be expressed using extensions of ZX [61, 73, 200], it will never be possible to capture the full breadth and generality of classical CPU instruction sets in a practical and extensible (and algebraically satisfying) way.

Peephole optimisations. As an alternative to the very principled approach of elegant calculi, graph transformations can also be used in the absence of theoretical guarantees in a more ad hoc fashion. Indeed, many existing (classical and quantum) compiler optimisations can already be understood as graph transformations. For as long as compilation has existed, compilers have relied on local transformations of the IR, typically referred to as peephole optimisations [224, 306]. Such optimisation strategies are based on the heuristic that local optimisations to the program will produce a well-optimised result overall. Mature compiler ecosystems have developed tools for declarative definitions, as well as automatic generation and correctness proving of peephole optimisations [227, 213, 269]. We refer to the classical compiler literature, e.g. [237], for more details on the various types of common peephole optimisations.

Quantum compilers adopted peephole-style optimisations from the beginning [78, 300, 294]. They encompass some of the most common optimisations in quantum computing, including the Euler Angle reduction [77], the two-qubit KAK decomposition [308, 101] and all gate set rebases [88]. A quantum-specific flavour of peephole optimisation with close links to GTSs, template matching [220, 175], achieved state-of-the-art results for Clifford circuit optimisation [67]. Recently, quantum peephole optimisations were also proposed that leverage provable state information to perform contextual optimisations [211], similar to strength reduction and optimisation with preconditions in classical compilation [213].

Internal representations. The graph formalisation of quantum computations we will define in this chapter also draws a lot from the internal representations (IR) of programs in classical compilers. The classical compilation community has found significant advantages in sharing a common standardised IR format. Indeed, while the exact syntax constructs and abstractions vary across programming languages, and, at the other end of the compiler stack, the specific assembly instructions

emitted differ between hardware targets, much of the compiler middleware can be broadly shared across use cases. This gave rise to the LLVM [207] and, more recently, the MLIR [206] projects, which provide common compiler IRs, along with all the infrastructure compilers typically require: IR transformation tooling, translation into hardware-specific assembly, efficient serialisations, in-memory formats etc.

The idea of adopting LLVM for quantum was championed by QIR [256], a standard introducing quantum primitives into the LLVM IR. This was subsequently adopted by many quantum hardware providers for its superior expressive power compared to circuit-based formats [257]. Building on top of QIR, an IR specifically for quantum-classical programs was proposed in [216], with additional soundness guarantees based, among others, on the no-cloning principle of quantum information. In parallel, projects with similar aims have also emerged [222, 176] that use the full MLIR and LLVM toolchain.

Challenges of GTS for compilation. Peephole optimisations of compiler IRs have proven to be a fast, general and scalable approach to compilation and code optimisation in practice. However, the optimisation results depend heavily on well-designed transformation orderings and the performance may vary widely across (equivalent) input programs. This is commonly known in compiler research as the phase ordering problem [82]. When a compiler can modify code in multiple ways, it must determine which transformations to apply and in what sequence to achieve optimal results [323, 210]. This is a common design challenge in GTSs, often addressed through mechanisms such as rule controls [162].

1. at the cost of many hours of compute, of course. ↩

This issue is also a key challenge within quantum compilation, as can be verified by comparing the performance of peephole-based compilers with provably optimal circuit synthesis strategies. On problem sizes where exhaustive search is feasible, unitary synthesis tools can sometimes outperform current, mostly peephole-based compilers [294] by up to 50%¹ [331].

3.2. Computation graphs and linearity

Computation graphs represent the flow of data between operations in a program, with nodes as operations and edges as data dependencies. Widely used in machine learning frameworks and GPU optimisations [47, 339], they are conceptually equivalent to dataflow graphs used in compiler design, which were pioneered by [135] and [183] and are now central to most compiler IRs.

In classical computations, these graph representations of computations are essentially *term graphs* [37] – sets of algebraic expressions that are stored as trees, combined with an important optimisation known as *term sharing*. When identical subexpressions appear multiple times, they can be represented as one computation and referenced from multiple locations, creating a directed acyclic graph rather than a term tree [255]. This sharing enables a more efficient representation. It can also be used as a compiler optimisation to identify subexpressions that can be cached and shared across expression evaluations for a more efficient execution – a technique known as common subexpression elimination (CSE) [83].

Each edge of a computation graphs corresponds to a unique *value*: the output of a previous computation that is being passed on to new operations. These values flow along edges in the graph – hence *dataflow graph*. Values are immutable:

they are defined once and then passed as input to further operations, where they can only be consumed, never modified. In compiler speak, programs expressed using such immutable values are often called single static assignment (SSA) programs [103,271]. In SSA:

1. Every value is defined exactly once,
2. Every value may be used any number of times (including zero).

Quantum computing throws this second pillar of SSA into the bin. Values in quantum computations are the result of computations on quantum data, and as such must obey the no-cloning and no-deleting theorems (section 2.1). We call values subject to these restrictions *linear*¹. They introduce the following constraint on valid computation graphs:

*Every **linear** value must be used exactly once.*

Linear values change fundamentally how transformations of the computation graph must be specified. Where compilers on classical data can:

- freely share common subexpressions (term sharing),
- undo term sharing, i.e. duplicate shared terms into independent subterms, and
- delegate the identification and deletion of obsolete code to specialised passes (e.g., dead code elimination [103, 68]),

quantum compilers must enforce much stricter invariants on IR transformations – or risk producing invalid programs.

In classical compilers, IR modification APIs (such as MLIR’s [PatternRewriter](#)) decouple program transformation from code deletion. Program transformations are specified by copying existing values and introducing new values and operations as needed, while the actual deletion of unused code is deferred to specialized *dead code elimination* passes. This approach is no longer feasible in the presence of linear values. Computation graphs for quantum computations must adopt *proper* graph rewriting semantics, in which the explicit deletion of obsolete values and operations is just as much a part of the rewriting data as the new code generation.

1. The terminology comes from “linear” logic [151]. I apologise for slamming additional semantics on what I recognise is an already very overloaded term. ↩

3.3. A graph representation for quantum programs

For the purposes of this thesis, we introduce a simplified graph-based IR for quantum computations that we will call minIR. It captures all the expressiveness that we require for hybrid programs whilst remaining sufficiently abstract to be applicable to a variety of IRs and, by extension, quantum programming languages and compiler frameworks.

MinIR can be thought of as being built from statements of the form

```
x, y, ... := op(a, b, c, ...)
```

to be understood as an operation `op` applied on the SSA values `a, b, c, ...` and producing the output values `x, y, ...`.

Computation and dataflow graphs are commonly defined with operations as vertices and values as edges. To faithfully capture the function signature of `op`, this requires storing and preserving an ordering of the incoming and outgoing edges (also known as port graphs [136]).

Instead, we adopt the hypergraph formalisation of computation graphs, which is more common within category theory (see string diagrams [289] and their formalisation in the hypergraph category [56, 327]). This definition is particularly well-suited for our purposes because it frames the graph transformations of interest to us in the well-studied language of rewriting within adhesive categories [204].

Hypergraphs and minIR

At a minimum, a directed hypergraph – for simplicity sometimes in the following referred to simply as *graph* – is defined by a set of vertices $\mathbf{V}$ and a set of (hyper) edges $\mathbf{E}$. We will always consider hypergraphs where the edges $e \in \mathbf{E}$ are directed and the vertices attached to e are given by ordered lists. We formalise this incidence relation between vertices in $\mathbf{V}$ and edges in $\mathbf{E}$ by writing $\mathbf{E}$ as the partition over the disjoint sets $\mathbf{E}_{st}$

$$\mathbf{E} = \bigsqcup_{s,t \in \mathbb{N}} \mathbf{E}_{st}$$

and introducing s source and t target maps for each $\mathbf{E}_{st}$. Why we write sets in boldface will become clear in a moment.

Definition 3.1 Directed hypergraph

A directed hypergraph is given by sets $\mathbf{V}$ and $\mathbf{E}_{st}$ for $s, t \in \mathbb{N}$, along with maps

$$\begin{aligned} \text{src}_{st,i} : \mathbf{E}_{st} &\rightarrow \mathbf{V} & \text{for } 1 \leq i \leq s \\ \text{tgt}_{st,j} : \mathbf{E}_{st} &\rightarrow \mathbf{V} & \text{for } 1 \leq j \leq t \end{aligned} \tag{1}$$

for all $s, t \in \mathbb{N}$.

Note that in this thesis, as in most common uses of hypergraphs, the sets $\mathbf{V}$ and $\mathbf{E} = \bigsqcup \mathbf{E}_{st}$ will always be finite, and thus $\mathbf{E}_{st} \neq \emptyset$ for a finite number of $s, t \in \mathbb{N}$ only.

For simplicity, we can further omit the st subscript of the source $\text{src}_{st,i}$ and target $\text{tgt}_{st,j}$ maps whenever it can be inferred from the domain of definition of the map. For $e \in \mathbf{E}_{st}$, we call $\text{src}_1(e), \dots, \text{src}_s(e) \in \mathbf{V}$ the s source vertices of e and $\text{tgt}_1(e), \dots, \text{tgt}_t(e) \in \mathbf{V}$ the t target vertices of e .

We introduce the notation $u \rightsquigarrow v$ to signify that there is an edge from u to v , i.e. there is $e \in \mathbf{E}_{st}$ for some $s, t \in \mathbb{N}$ and $1 \leq i \leq s, 1 \leq j \leq t$ such that $u = \text{src}_i(e)$ and $v = \text{tgt}_j(e)$. We define the equivalence relation $\sim \subseteq \mathbf{V}^2$ of connec-

write $[v]$, resp. $[e]$ for the connected component that contains the vertex v , resp. the edge e .

To proceed, it is useful to frame the hypergraph definition in a categorical setting. We write $[\mathbb{C}, \text{Set}]$ for the presheaf topos of the category $\mathbb{C}$, i.e. the category with functors $\mathbb{C} \rightarrow \text{Set}$ as objects and natural transformations as morphisms. Definition 3.1 can be equivalently restated as:

hypergraphs are objects in the presheaf topos $\mathbb{H} = [\mathbb{C}, \text{Set}]$,

where the category $\mathbb{C}$ has objects V and E_{st} for $s, t \in \mathbb{N}$ and arrows given by (1), now interpreted as morphisms in $\mathbb{C}$ rather than as functions in Set . In this framing, a graph is a functor that defines a set for each object of $\mathbb{C}$ and specifies functions between these sets – one for each arrow in $\mathbb{C}$.

This is where the distinction between bold and non-bold typeface comes from: we use bold letters to refer to images in Set of a hypergraph functor, whereas the non-bold typeface is refers to objects in the indexing category $\mathbb{C}$. The distinction between $\mathbb{C}$ and Set is less important for morphisms – it will typically be clear from the context. We thus use the same symbols for both.

Linearity constraints

The introduction of $\mathbb{H}$ not only gives us a notion of hypergraph homomorphisms – maps between hypergraphs that preserve the structure of the graph. It also provides us with a way to express the linearity constraints that arise from our discussion in section 3.2, and which we must enforce on our computation graphs.

The definition that follows adds the coproduct E explicitly as an object of the category (which we did not need to do in Definition 3.1), as we need it as the codomain of the new morphisms *use* and *def*. The adhesitivity of the category does no longer comes for free – we will get back to this in section 3.4.

Definition 3.2 Hypergraph with linearity constraints

The category $\text{lin-}\mathbb{C}$ is the category given by objects

$$\{V, V_{src}, V_{tgt}\} \cup \{E\} \cup \{E_{st} \mid s, t \in \mathbb{N}\}.$$

Its arrows are the incidence morphisms given in (1), along with

$$\begin{aligned} use : V_{src} &\rightarrow E & def : V_{tgt} &\rightarrow E \\ \lambda_{src} : V_{src} &\multimap V & \lambda_{tgt} : V_{tgt} &\multimap V \end{aligned}$$

and $\iota_{st} : E_{st} \multimap E$ for all $s, t \in \mathbb{N}$. The morphisms $\lambda_{src}, \lambda_{tgt}, \iota_{st}$ are split monomorphisms and the following diagrams commute for all $s, t \in \mathbb{N}$ and $1 \leq i \leq s, 1 \leq j \leq t$:

$$\begin{array}{ccc}
 E_{st} & \xrightarrow{\iota_{st}} & E \\
 \text{\scriptsize tgt}_j \downarrow & & \uparrow \text{\scriptsize def} \\
 V & \xleftarrow{\lambda_{tgt}} & V_{tgt}
 \end{array}
 \qquad
 \begin{array}{ccc}
 E_{st} & \xrightarrow{\iota_{st}} & E \\
 \text{\scriptsize src}_i \downarrow & & \uparrow \text{\scriptsize use} \\
 V & \xleftarrow{\lambda_{src}} & V_{src}
 \end{array}
 \tag{2}$$

Directed hypergraphs with linearity conditions are objects in the full subcategory $\mathbf{lin}\text{-}\mathbb{H}$ given by objects $H_{\mathbf{lin}} \in [\mathbf{lin}\text{-}\mathbb{C}, \mathbf{Set}]$ such that

$$\mathbf{E} = \bigsqcup \mathbf{E}_{st}$$

is the coproduct in $\mathbf{Set}$ and $H_{\mathbf{lin}}(\iota_{st}) : H_{\mathbf{lin}}(E_{st}) \rightarrow H_{\mathbf{lin}}(E)$ are the injections into $H_{\mathbf{lin}}(E)$.

We probably owe an explanation for this definition – at least for the sake of the few computer scientists that are still following us.

First of all, notice that every hypergraph with linearity constraint corresponds to a hypergraph in the sense of Definition 3.1: there is an obvious functor $\mathcal{L} : \mathbb{C} \rightarrow \mathbf{lin}\text{-}\mathbb{C}$ that maps each object and morphism in $\mathbb{C}$ to the object or morphism with the same name in $\mathbf{lin}\text{-}\mathbb{C}$. By contravariance, we can thus (functorially) map every hypergraph with linearity constraints $H_{\mathbf{lin}} \in \mathbf{lin}\text{-}\mathbb{H}$ to the hypergraph $H = H_{\mathbf{lin}} \circ \mathcal{L} \in \mathbb{H}$.

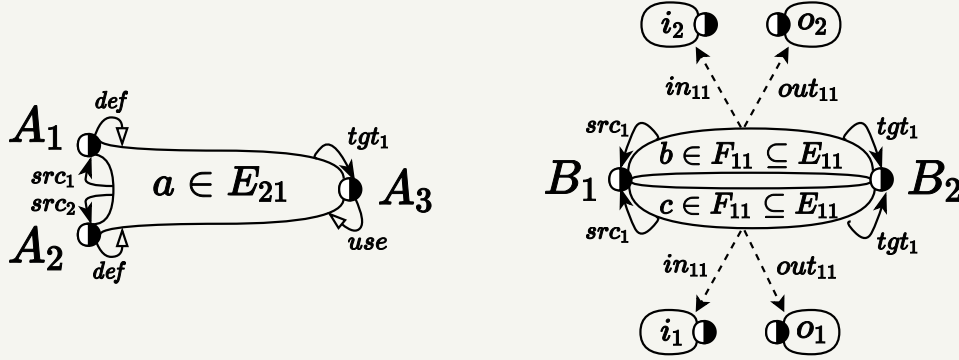
Another way of looking at this is to realise that by requiring that $\lambda_{src}, \lambda_{tgt}$ be split monomorphisms, we obtain that the resulting functions in $\mathbf{Set}$ are injective. Up to isomorphism, we can consider that $\mathbf{V}_{src}, \mathbf{V}_{tgt} \subseteq \mathbf{V}$ are subsets of vertices in $H_{\mathbf{lin}}$. A hypergraph with linearity constraints is thus a directed hypergraph with two selected subsets of vertices $\mathbf{V}_{src}$ and $\mathbf{V}_{tgt}$.

Vertices within these subsets are special. For every $v \in \mathbf{V}_{src}$, there exist unique indices $s, t \in \mathbb{N}$, $1 \leq i \leq s$ and edge $use(v) = e \in \mathbf{E}_{st}$ such that $src_i(e) = v$. In words, for every $v \in \mathbf{V}_{src}$ there is a unique edge in the hypergraph that has v as one of its sources. We then say that e is the unique *use* of vertex v . Similarly, vertices in $\mathbf{V}_{tgt}$ have a unique edge e in the hypergraph with v as one of its targets – it is the unique *definition* of v .

Typed graphs

MinIR graphs are strongly typed. We introduce *typed* graphs for this purpose, a concept for which graph transformation was first formalised in [125]. A *type system* for directed hypergraphs is just another object $\Sigma \in \mathbb{H}$. A *typed graph* is then an object of the slice category $\mathbb{H} \searrow \Sigma$, that is to say, typed graphs are morphisms $H \rightarrow \Sigma$ of $\mathbb{H}$ and morphisms between typed graphs are given by the subset of morphisms $H_1 \rightarrow H_2$ of $\mathbb{H}$ that make the triangle diagram formed by H_1 , H_2 and Σ commute.

too strict. We consider instead the category $\text{lin-}\mathbb{C}_{\text{type}}$ given by the same objects as $\text{lin-}\mathbb{C}$, as well as the same morphisms, with the omission of *use* and *def*. There is an obvious functor $\text{lin-}\mathbb{C}_{\text{type}} \rightarrow \text{lin-}\mathbb{C}$ and thus, by contravariance, every hypergraph with linearity constraints $H_{\text{lin}} \in \text{lin-}\mathbb{H}$ can be mapped to a hypergraph in $\text{lin-}\mathbb{H}_{\text{type}}$. We say that a hypergraph with linearity constraints $H_{\text{lin}} \in \text{lin-}\mathbb{H}$ is *Σ -typed* for a type system $\Sigma \in \text{lin-}\mathbb{H}_{\text{type}}$ if there is a morphism $\Sigma \rightarrow H_{\text{lin}}$, when interpreting H_{lin} is as an object of $\text{lin-}\mathbb{H}_{\text{type}}$.



Two example hypergraphs. Vertices (labelled with capital letters) are circles and hyperedges (labelled with small letters) span between them. Vertices that are attached to an edge in the black half of the circle correspond to source vertices of the edge; those in the white half correspond to target vertices. The functions src_i and tgt_j map edges to the incident vertices, defining the directed hypergraph. On the left, there are further functions *use* and *def* that map vertices to the unique edge that uses or defines them. This defines a hypergraph with linearity constraints, with $V_{\text{tgt}} = \{A_1, A_2\}$ and $V_{\text{src}} = \{A_3\}$. These cannot be defined on the graph on the right. On the other hand, the edges *b* and *c* are in F_{11} , and are in the domain of functions in_{11} and out_{11} , thus defining a child region of a hierarchical graph. Note that it would be invalid to have any edge connecting i_1 or o_1 to i_2 or o_2 . The *i* and *o* vertices also have incidence morphisms, not displayed here.

Hierarchical hypergraphs

A final bit of structure that minIR graphs require is a notion of hierarchy between regions of the graph. This will be useful to define functions, control flow blocks such as `if-else`, or any subroutine that can itself be viewed as an operation in the computation.

Hierarchical hypergraphs were first proposed in [116] and further generalised in [70, 248]. However, we opt to use a more restrictive definition, closer to the notion of flattened hypergraphs of [116]. The reason for this is twofold. Firstly, hierarchical (hyper)graphs are typically defined recursively. It is not obvious under which conditions (and if) such definitions form adhesive categories, although progress in this direction was made in [246] with the introduction of coalgebraic graphs. As a result, to the extent that graph transformation results can be applied to such structures, it must be done so carefully.

The second, more practical, reason is that the notion of typed graph introduced above cannot be directly lifted to the hierarchical graph setting: while some subset of the hierarchical relation in minIR should be enforced by the type system,

the type graph of a nested graph should be identical to the parent's (as opposed to being itself nested within the type graph of the parent).

1. And in fact, we will see in section 3.4 that it is not. ↩

It is therefore more convenient for us to encode hierarchy in directed hypergraphs as follows. Note that it is not clear that our definition is adhesive¹ – but at least it is framed as a subcategory of a base category that is.

Definition 3.3 *Hierarchical hypergraph*

The category $\mathbf{hier}\text{-}\mathbb{C}$ is the category with objects and arrows of $\mathbb{C}$ along with additional objects F_{st} for $s, t \in \mathbb{N}$ and arrows:

- $F_{st} \rightarrowtail E_{st}$ that are split monomorphisms,
- input arrows $F_{st} \xrightarrow{in_{st}} E_{0s}$, and output arrows $F_{st} \xrightarrow{out_{st}} E_{t0}$.

Hierarchical hypergraph are the objects in the full subcategory $\mathbf{hier}\text{-}\mathbb{H}$ given by objects $H_{\mathbf{hier}} \in [\mathbf{hier}\text{-}\mathbb{C}, \mathbf{Set}]$ such that

- for any edge $e \in \mathbf{E}$ of $H_{\mathbf{hier}}$, the set $P([e]) = \overline{in}^{-1}([e]) \cup \overline{out}^{-1}([e])$ has at most one element.
- the transitive and reflexive closure of $[e] \preceq [P([e])]$ for $e \in \mathbf{E}$ is a partial order on the connected components of $H_{\mathbf{hier}}$.

Here $\overline{in}$ and $\overline{out}$ are the functions with domain $\mathbf{E}$ defined piecewise as in_{st} and out_{st} for all s, t on their respective (disjoint) domains of definition.

The same definition can also be applied to $\mathbf{lin}\text{-}\mathbb{H}$ to obtain the category of hierarchical directed hypergraphs with linearity constraints $\mathbf{hier}\text{-}\mathbf{lin}\text{-}\mathbb{H}$. Similarly, we define the associated category for type systems; however, we do not impose any of the two conditions related to $P(\cdot)$ for the type category, i.e. $\mathbf{hier}\text{-}\mathbf{lin}\text{-}\mathbb{H}_{\text{type}} = [\mathbf{hier}\text{-}\mathbf{lin}\text{-}\mathbb{C}_{\text{type}}, \mathbf{Set}]$ is the full presheaf category, rather than a subcategory of it.

As with the incidence morphisms src and tgt , we will drop the st subscript for the IO arrows in and out when it can be inferred from the domain of definition.

Just as in the discussion of Definition 3.2, we interpret the split monomorphisms as equivalent to requiring $\mathbf{F}_{st} \subseteq \mathbf{E}_{st}$. Taking over terminology from [116], we call elements $f \in \mathbf{F}_{st}$ the *frames* of $H_{\mathbf{hier}} \in \mathbf{hier}\text{-}\mathbb{H}$. For each frame f , there is thus a unique *input* edge $in(f)$ and a unique *output* edge $out(f)$ in $H_{\mathbf{hier}}$ that have respectively s targets and zero sources, and t sources and zero targets.

By the first condition we imposed on $P(\cdot)$, the partial function *parent* mapping connected components to their *parent* edge:

$$parent([e]) = \begin{cases} p & \text{if there exists } p \in P(e) \\ \perp & \text{otherwise.} \end{cases}$$

is well-defined. We call the subgraphs of H_{hier} that share a same parent a *region* of H_{hier} ². The subgraph of vertices and edges without parent is the *root region* of H_{hier} .

2. Note that a region may not be a connected subgraph. Albeit, it is a simple exercise to convince yourself that any non-root region contains either one or two connected components. ↩

The minIR computation graph

In minIR, the vertices are the values of the computation, while the hyperedges define the operations. This imposes some constraints for a hypergraph to be a valid minIR graph:

- all values in minIR must have a unique operation that *defines* them;
- values that are *linear* must also have a unique operation that *uses* them;
- the graph must be acyclic, meaning that no value can be defined in terms of itself.

This can be expressed as a hypergraph with linearity constraints by choosing $\mathbf{V}_{\text{tgt}} = \mathbf{V}$ and $\mathbf{V}_{\text{src}} = \mathbf{V}_L$, where $\mathbf{V}_L \subseteq \mathbf{V}$ is the subset of linear values.

The following definition then comes as no surprise:

Definition 3.4 *MinIR graph*

Let $\Sigma \in \text{hier-lin-}\mathbb{H}_{\text{type}}$ be a type system. A minIR graph H typed in Σ is an object of $\text{hier-lin-}\mathbb{H}$ that is Σ -typed and such that the adjacency relation $\rightsquigarrow$ is acyclic. We call $\mathbf{V}_{\text{src}} = \mathbf{V}_L$, the linear values of H .

In the context of minIR, $\rightsquigarrow$ relations encode the data flow of values across the computation. The lack of explicit operation ordering differentiates minIR (and HUGR) from most classical IRs, which, unless specified otherwise, typically assume that instructions may have side effects and thus cannot be reordered. All quantum operations (and the classical operations we are interested in) are side-effect free, which significantly simplifies our IR.

Input and output values

Notice that in Definition 3.4, it is not enforced that every value has a definition, i.e. there might be $v \in \mathbf{V} \setminus \mathbf{V}_{\text{tgt}}$; nor that every value with a linear type is in $\mathbf{V}_L$, i.e. if $\tau : H \rightarrow \Sigma$ is the typing morphism and $\mathbf{T}_L$ are the linear types in the type system, there might be $v \in \tau^{-1}(\mathbf{T}_L) \setminus \mathbf{V}_L$.

This would be easy to fix: we could on the one hand enforce the equality $\mathbf{V}_{\text{tgt}} = \mathbf{V}$, thus guaranteeing that every value has a unique definition in the graph. On the other hand, we could define $\mathbf{V}$ as the coproduct $\mathbf{V} = \mathbf{V}_L \sqcup \mathbf{V}_{NL}$, where $\mathbf{V}_{NL}$ is a new object introduced to explicitly capture the set of non-linear values. Morphisms in this category would guarantee that the linearity of values is always preserved, and thus in particular the type morphism would map a value to a linear type if and only if it is linear.

Instead, we opt to allow undefined values and unused linear values to be able to express rewrite rules that match subgraphs of minIR graphs within the same category.

Definition 3.5 *Inputs and outputs of minIR graphs*

For a minIR graph H with typing morphism $\tau : H \rightarrow \Sigma$, we call the set $I = \mathbf{V} \setminus \mathbf{V}_{tgt}$ the *input values* of H and $O = \tau^{-1}(\mathbf{T}_L) \setminus \mathbf{V}_L$ its *output values*, where $\mathbf{T}_L$ are the linear values in Σ .

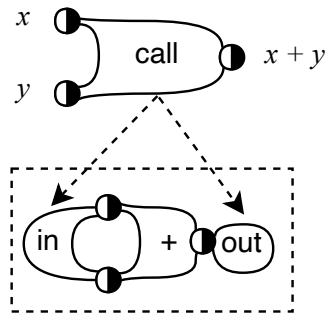
If $I = O = \emptyset$, we say that H is IO-free.

Note that by this definition, an output value in O always has a linear type! This is because non-linear values do not need to be treated specially when they are outputs: unlike linear values that must always be used in a non-output position, non-linear values may have no outgoing edge, in which case they are simply discarded in the computation.

Structured control flow

The operations and values of a minIR graph define the data flow of a program. However, a program must also be able to control and change the data flow at run time in order to express loops, conditionals, function calls etc. This is the program *control flow*, which minIR expresses using regions and so-called *structured control flow*.

Using regions, any non-trivial control flow (function calls, conditionals, loops etc.) is captured by a frame, a “black box” operation within the data flow of the program. Its implementation is then defined in the nested region of the frame. This can be used for function calls, but also for branches of control flow. A simple function call, that unconditionally redirects the control flow to the operations within a nested region, could for example be represented as follows:



In this figure and below, circles are SSA values (the vertices of the hypergraph), while the edges spanning between them are the operations. Edges attached to the white half of circles are value definitions, while hyperedges attached to the black half of circles are value uses. The `call` and `+` operation can be read left to

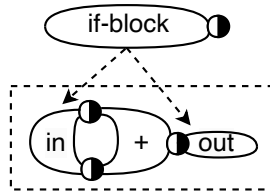
right: for instance, the two values x and y on the left of `call` are inputs (the operation *uses* those values, and is thus attached to the black half of the circles), whereas the $x + y$ value on the right is the output of the `call` operation (the operation *defines* this value, which is thus attached to the white half of the circle).

Dashed arrows indicate hierarchical dependencies that map the frame edge to the two input and output edges in the child region; dashed rectangles mark the non-root regions of the graph.

Importantly, the frame edge representing the `call` operation must intuitively “forward” all its input values to the `in` operation of the child region, and similarly passes on the value at the `out` operation to the output value of the `call` output in the parent graph. Passing function arguments and retrieving returned values in this fashion will be very familiar to any computer scientist. Unlike most programming languages, this is also how in minIR values are passed to and from *any* control flow constructs we would wish to model.

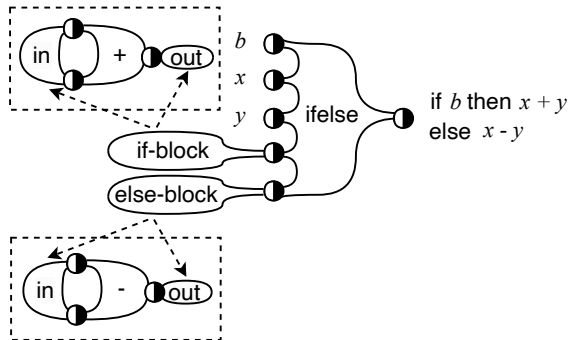
In terms of graph structure, this relation between values in parent and child regions means that the arity and types of the inputs and output of `call` fix the signatures of the child `in` and `out` operations. Definition 3.3 already ensures that the input and output arities of `in` and `out` are correct. The correct typing of these input and output values will be ensured by the type system, which we discuss in a separate section below.

To handle constructs that require more than one child region, such as an `if-else` statement, we can use frames that have zero input and one output:



The output of the `ifblock` is intuitively a higher-order type representing an operation that takes two inputs and outputs the sum.

An if-else statement might then look as follows:



The `if` and `else` blocks must expect the same input and output values. This is key to respecting any linearity constraints that values passed to `ifelse` might be subject to. By definition, all operations that use or define a value v will be in the same region – in other words, values are only available within their defining region. This in effect implements “variable scoping”. With some imagination, this construction can easily be adapted to model loops, complex control flow graphs, or any other control flow structures.

Why not plain branch statements?

3. You may know this from prehistoric times as the `goto` statement, in languages such as Fortran, C, and, yes, [even Go](#). ↩

There is a simpler – and at least as popular – way of expressing control flow in IRs without requiring regions and operation hierarchies, using *branch* statements³. For instance, LLVM IR provides a conditional branch statement

```
br i1 %cond, label %iftrue, label %iffalse
```

that will jump to the operations under the `iftrue` label if `%cond` is true and to the `iffalse` label otherwise.

This is a simple and versatile approach to control flow that can be used to express any higher-level language constructs. Unfortunately, conditional branching does not mix well with linear values.

Linearity, as defined in Definition 3.4, is a simple constraint to impose on minIR graphs. In the presence of conditional branching, however, the constraint would have to be relaxed to allow for single use *in each mutually exclusive branch* of control flow. For instance, the following two uses of `b` should be allowed (in pseudo-IR):

```
b := h(a)
<if cond> c := x(b)
<else>    d := h(b)
```

4. You might be thinking “*oh, but all that is required here are phi nodes!*”, if you are familiar with those. No – you’d also need a sort of “phi inverse”. Besides, see [this discussion](#) for more arguments on why no phi nodes. ↩

This is a much harder invariant to check on the IR: linearity would no longer be enforceable as a syntactic constraint on the minIR graph as in Definition 3.4, but would instead depend on the semantics of the operations to establish mutual exclusivity of control flow⁴. Forbidding arbitrary branching in minIR and resorting instead to structured control flow as described above to express control flow is just as expressive and gives the linearity constraint a much simpler form.

Type graph

We have seen how minIR graphs impose *some* structure on the types of computations that can be expressed: a linear value cannot be used by two (or zero) operations, frames will always have a unique `in` and `out` operation in their child region with correct arities, etc.

However, without a “good” type system and associated semantics, it is still possible to express nonsensical programs: we have mentioned for instance earlier that it is up to the type system to enforce that the types of the `in` and `out` operations match the types of the frame. Similarly, it is possible to construct programs that break linearity: take the `ifelse` operation discussed in the previous section,

but now replace its semantics to be `do-in-parallel`, i.e. it will execute both the `if-block` and `else-block` in parallel on the inputs that it is given. This would violate the linearity of its inputs, but would nonetheless be a syntactically valid miniR graph!

To resolve this we present here some typed operations, along with their semantics, that can be used to construct well-behaved type systems: programs typed in this system model the kind of quantum programs that we are interested in expressing and are guaranteed to be valid computations. Categorising all valid constructions or an exhaustive enumeration of conditions that type systems must satisfy to guarantee the validity of programs is beyond the scope of this thesis. It is in practice often straightforward to combine and extend the elements presented here to support further custom syntactic constructs and types.

Basic types and operations

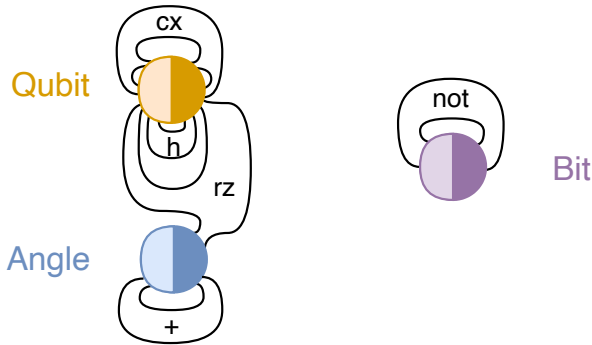
The most elementary types in our computations are `Bit`s and `Qubit`s. The former is typically known as a Boolean and represents the purely classical values `0` and `1`. The latter is the canonical quantum example of a *linear type*. Indeed, just like values in miniR graphs, the type system in $\text{hier-lin-}\mathbb{H}_{\text{type}}$ distinguishes between linear and non-linear types.

Other typical classical types such as integers, floats, strings, custom algebraic data types (ADT) etc could also be introduced as required. In the figure below, we for instance introduce the `Angle` type to represent rotation angles that parametrise quantum gates. Further examples of linear types, on the other hand, include higher-dimensional qudits, but also any ADT that contains a linear type within it.

As we saw in section 2.1, the number of input qubits in pure quantum operations will match the number of output qubits: the single-qubit `h` (hadamard gate) and two-qubit `cx` (controlled NOT) operations thus have one or two `Qubit`s as both inputs and outputs `rz` (Z rotation) is also a single-qubit operation, but it takes an additional input of type `Angle` to specify the rotation angle.

On the pure classical side, we are free to add any side-effect free operations on our types; in our example we model addition `+` on `Angle`s and negation `not` on `Bit`s. In the type system $\Sigma \in \text{hier-lin-}\mathbb{H}_{\text{type}}$, each type is represented by a single vertex.

In our example, we thus have three vertices:



We introduce a different colour for each type. Operations such as `cx` are represented by a hyperedge with two sources on `Qubit` and two targets on `Qubit`. As in the previous diagrams, we can distinguish operation inputs from outputs by whether they are attached to the dark or light half of the type vertex: the `rz` operation thus has one `Qubit` input, one `Angle` input and one `Qubit` output.

As you can tell from the diagram, whilst `Qubit` is a linear type in the type system, it is not a linear value in the sense of a minIR graph: the `Qubit` type has multiple uses and defines in the `cx` operation alone. This is the key difference between $\text{lin-}\mathbb{H}_{\text{type}}$ and $\text{lin-}\mathbb{H}$.

Qubit allocation and measurement

We also introduce non-pure quantum operations `qalloc` and `measure` which respectively “create” a qubit (so no input, one `Qubit` output) and “destroy” it (one `Qubit` input, one `Bit` output – depending on whether the qubit was projected onto the $|0\rangle$ or $|1\rangle$ state). Remember that the reason these operations seem to “break” the laws of pure quantum physics is because they result from interactions with the classical environment.



`measure` is fundamental, as it connects quantum values with classical ones!

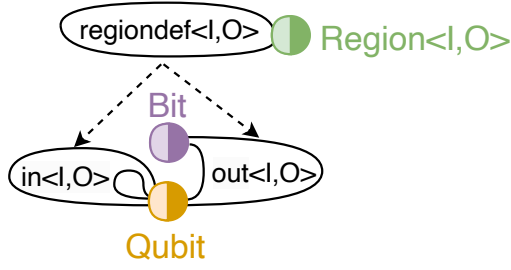
Region definition and structured control flow

Our type system is so far missing a crucial aspect of minIR: the hierarchical structures. For this we need frame types, i.e. frames in the type graph. We must introduce a distinct type for each possible type signature of a frame. To keep this as simple as possible, we will introduce exactly one type for each signature.

If we write T for the set of types in our type system (i.e. `Bit`, `Qubit` and `Angle` in our example), then a type signature of an edge is given by a pair (I, O) of ordered lists of types $I, O \in T^*$. For each such (I, O) pair, we introduce

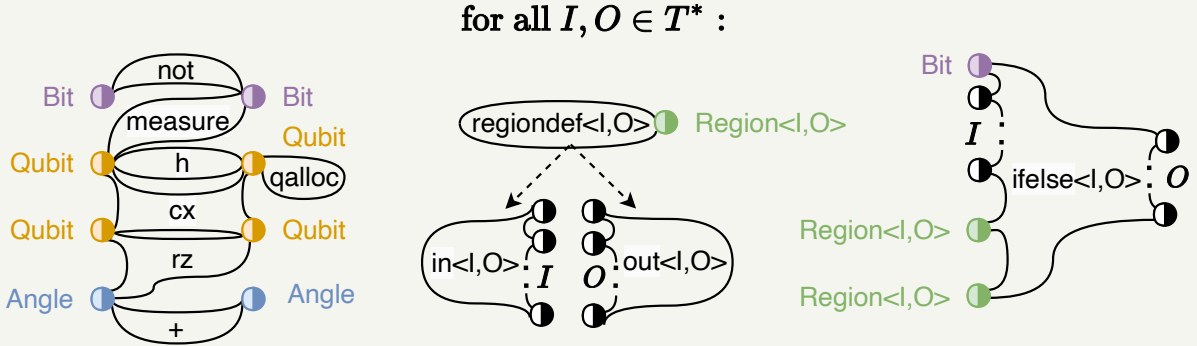
- the frame type `regiondef<I, 0>`,
- the in and out types `in<I,0>` and `out<I,0>`,
- along with a new non-linear type `Region<I, 0>`, the higher-order type representing a region with inputs I and outputs O .

The `regiondef<I, 0>` op takes zero inputs and returns one output `Region<I, 0>`, whereas `in<I,0>` takes zero inputs and returns values of type I and `out<I,0>` takes inputs of type O and returns nothing. For instance, for $I = (\text{Qubit}, \text{Qubit})$ and $O = (\text{Qubit}, \text{Bit})$, we have the following type graph.



Note that there is an important distinction in $\text{hier-lin-}\mathbb{H}_{\text{type}}$ in comparison to $\text{hier-lin-}\mathbb{H}$: there is no notion of regions in the type system: the `Qubit` and `Bit` types in the above diagram would be in the child region of `regiondef<I, 0>` if it were a graph in $\text{hier-}\mathbb{H}$, but in the type system, they might also be used by other operations in other regions (such as `cx`, `rz`, `h` etc. defined earlier).

Using the `Region<I, 0>` types, it is then easy to define typed operations for any structured control flow of interest, such as the `if-else` example above. The following figure gives an overview of the entire type system of our example. For display purposes, we have included multiple copies of each type vertex; we remind the reader that in the actual type graph, all circles of the same type (colour) are one and the same.



A complete *minIR* type graph, following the example in this section. Value vertices with the same label (and same colour) form a single vertex in the type graph. They have been split into multiple vertices in this representation for better readability. The data types and op types with the `<I, 0>` suffix are parametrised on the signature type (I, O) for $I, O \in T^*$.

An example miniR program

Taking a step back, let us make the introduced ideas more concrete through an example. We demonstrate how a simple program written in textual form can be translated and expressed as a miniR graph. All statements are of the form

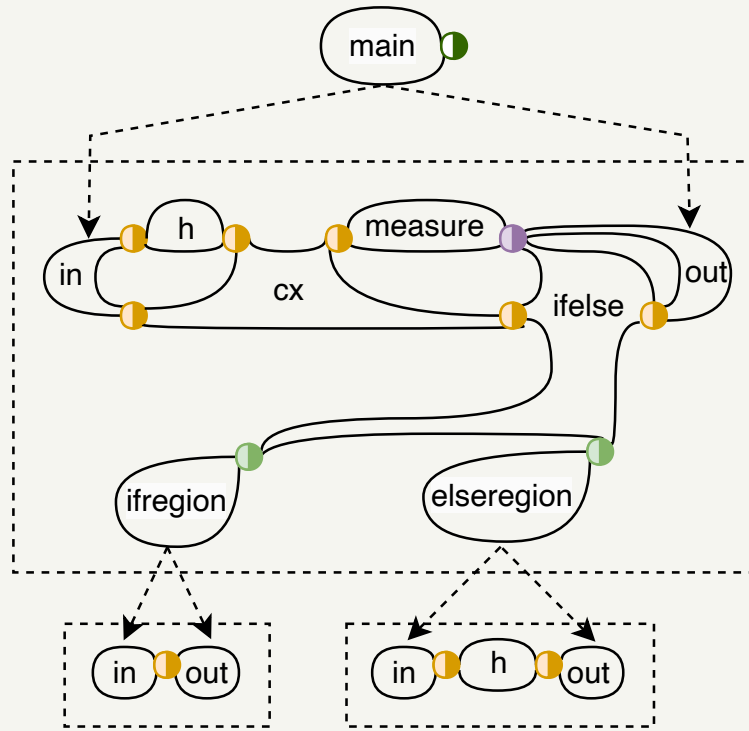
```
x, y, ... := op(a, b, c, ...)
```

where `a`, `b`, `c` etc are the SSA values passed to `op` (or *used by* `op`), and `x`, `y` etc are the SSA values returned by `op` (or *defined by* `op`). We use curly bracket to define the child region of a `regiondef` operation. A valid miniR program might then look as follows:

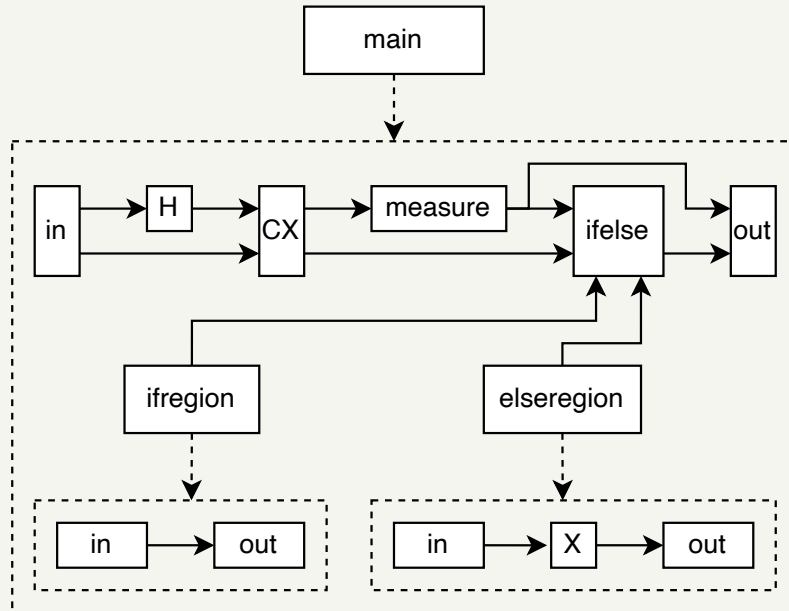
```
1 main := regiondef<(Qubit, Qubit), (Qubit, Bit)> {
2   q0, q1 := in()
3
4   q0_1 := h(q0)
5   q0_2, q1_1 := cx(q0_1, q1)
6
7   m0 := measure(q0_2)
8
9   ifregion := regiondef<(Qubit,), (Qubit,> {
10    q1 := in()
11    out(q1)
12  }
13  elserregion := regiondef<(Qubit,), (Qubit,> {
14    q1 = in()
15    q1_1 := h(q1)
16    out(q1_1)
17  }
18  q1_2 := ifelse(m0, q1_1, ifregion, elserregion)
19
20  out(q1_2, m0)
21 }
```

Note that the `in()` and `out(...)` operations are only allowed within nested regions (as required by the type system). We have omitted the type parameters on these operations, as it mirrors exactly the parameter of the `regiondef`.

It corresponds to the two miniR graphs on the following page. We use “wiggly hyperedges” that stretch between values, as in the first figure. They may look unusual if you are used to computation graphs. One can opt to draw the same graph with boxes for hyperedges and wires for values, yielding the second figure. The two representations are equivalent, but the rewriting semantics are most explicit when viewing values as vertices.



An example of an IO-free minIR graph. The vertex colours indicate their types in the type system presented in the previous figure. The `main`, `ifregion` and `elseregion` ops are all of op type `regiondef` (with type parameters omitted), labelled here with custom names for clarity. The type parameters of the `ifelse`, `in` and `out` op type have similarly been omitted. All other operation types are given as labels on the edges.



An equivalent representation of the computation above, now representing operations as boxes and values as wires. The arrow direction indicates the flow from value definition to value use(s). Dashed arrows have been changed to point to regions instead of individual operations.

Differences to the quantum circuit model

5. Note that these comments apply specifically to characteristics of quantum *circuits*. Other diagrammatic representations of quantum processes in use, such as string diagrams, quantum combs etc may not share the same properties. ↪

We conclude this presentation of minIR by highlighting the differences between this IR-based representation and the quantum circuit model that most quantum computing and quantum information scientists are familiar with⁵.

When restricted to purely quantum operations and no nested regions, the string diagram representation of a minIR graph (i.e. operations as boxes and values as wires) *looks* very similar to a quantum circuit. There is, however, a fundamental shift under the hood from [reference to value semantics](#) – to borrow terminology from C++.

In the reference semantics of quantum circuits, operations are typically thought of as “placed” on a qubit (the “lines” in the circuit representation), for instance, by referring to a qubit index. This qubit reference exists for the entire computation duration, and the quantum data it refers to will change over time as operations are applied to that qubit.

In the value semantics of computation graphs and SSA, on the other hand, qubits only exist in the form of the data they encode. When applying an operation, the (quantum) data is consumed by the operation and new data is returned. Given that the input data no longer exists, linearity conditions are required to ensure that no other operation can be fed the same value.

To make the difference clear, compare the program representations of the following computation:

6. This is python code: `pip install pytket`. ↪

Quantum circuit (pytket)⁶

```
import pytket as tk
circ = tk.Circuit(2)
circ.H(0)
circ.CX(0, 1)
circ.X(1)
```

SSA (minIR)

```
q0_0, q1_0 := in()
q0_1      := h(q0_0)
q0_2, q1_1 := cx(q0_1, q1_0)
q1_2      := x(q1_1)
out(q0_2, q1_2)
```

In value semantics, it becomes much harder to track physical qubits across their lifespan. This has very practical implications: without the convenient naming scheme, it would, for example, be non-trivial to count how many qubits are required in the SSA representation of the computation above. However, it is a drastically simpler picture from the point of view of the compiler and the optimiser – hence its popularity in classical compilers. When operations are defined based on qubit references, the compiler must carefully track the ordering of these operations: operations on the same qubit must always be applied in order. Through multi-qubit gates, this also imposes a partial ordering on operations across different qubits that must be respected.

SSA values remove this dependency tracking altogether: the notion of physical qubit disappears, and the ordering of statements becomes irrelevant. All that matters is connecting each *use* of a value (i.e. an input to an operation) with its *unique* definition, the output of a previous operation. In other words, the global ordering imposed by reference semantics is replaced by a causal order on the diagram [190].

All the concepts of minIR embed themselves very easily within the MLIR-based quantum IRs and the HUGR IR [216]. In this sense, our toy IR serves as the minimum denominator across IRs and compiler technologies so that proposals and contributions we are about to make can be applied regardless of the underlying technical details.

By waiving goodbye to the circuit model, we have been able to integrate much of the theory of traditional compiler design, bringing us in the process much closer to traditional compiler research and the large-scale software infrastructure that is already available. This gives us access to all the classical optimisation and program transformation techniques developed over decades. Using structured control flow, we were also able to model linear resources such as qubits well – by using value semantics and SSA, checking that no-cloning is not violated is as simple as checking that each linear value is used exactly once.

Finally, this new design is also extremely extensible. Not only does it support arbitrary operations, but the type system is also very flexible. There is dedicated support for linear types, but this does not have to be restricted to qubits: lists of qubits could be added or even, depending on the target hardware, higher dimensional qudits, continuous variable quantum data, etc.

3.4. Graph transformation in minIR

As discussed in section 3.2, computation graphs with linear values, such as minIR, must adopt strict graph transformation semantics to ensure that linear constraints are satisfied at all times. In this section, we use the minIR graph category presented in the previous section to define transformation semantics that lean on the double pushout (DPO) [126] and sesqui-pushout (SqPO) [92] semantics in adhesive categories [203].

Adhesivity of hypergraph categories

The natural place to start this section is by studying which of the categories defined in section 3.3 are adhesive. From adhesivity follows that transforming graphs using DPO and SqPO constructions is well-defined and unique, at least in the regimes of interest to us.

A category is said to be *adhesive* if it has all pullbacks and pushouts along monos, as well as some compatibility conditions between them, the so-called “Van Kampen squares”. We refer to the literature (e.g. [203]) for a complete definition. For our purposes, the following two results are sufficient:

- Every presheaf topos $[\mathbb{C}, \text{Set}]$ is adhesive (Corollary 3.6 in [203]);
- Every full subcategory $\mathbb{D} \subseteq \mathbb{C}$ of an adhesive category is adhesive if the pullbacks and pushouts in $\mathbb{C}$ of objects in $\mathbb{D}$ are again in $\mathbb{D}$ (a simple result; if the Van Kampen squares commute in $\mathbb{C}$, they must commute in $\mathbb{D}$).

A first result immediately follows from the first result:

Proposition 3.1 *Adhesivity of directed hypergraphs*

The category $\mathbb{H}$ of directed hypergraphs is adhesive.

Proof. It is a presheaf.

This does not immediately generalise to $\text{lin-}\mathbb{H}$, as unlike $\mathbb{H}$, Definition 3.2 imposes that E be a coproduct. However, the result still holds:

Proposition 3.2 *Adhesivity of hypergraphs with linearity constraints*

The categories $\text{lin-}\mathbb{H}$ and $\text{lin-}\mathbb{H}_{\text{type}}$ are adhesive.

Proof. $\text{lin-}\mathbb{H}$ is a full subcategory of the adhesive category $[\text{lin-}\mathbb{C}, \text{Set}]$. We must show the existence of pullbacks and pushouts along monos in $\text{lin-}\mathbb{H}$.

Pullbacks. Consider a pullback $A \xleftarrow{p_a} P \xrightarrow{p_b} C \xleftarrow{b} B$ in $[\text{lin-}\mathbb{C}, \text{Set}]$, with $A, B, C \in \text{lin-}\mathbb{H}$. We must show that P is in $\text{lin-}\mathbb{H}$. Colimits are computed pointwise in presheaves, so we know that $P(E)$ is the pullback of $A(E) \rightarrow C(E) \leftarrow B(E)$ in Set . If we can show that $P(E)$ is the coproduct of $P(E_{st})$ for $s, t \in \mathbb{N}$, then we are done.

Let $v \in P(E)$. Because $A(E)$ and $B(E)$ are coproducts in Set , i.e. a disjoint union, there must be $s, t, s', t' \in \mathbb{N}$ such that $p_a(v) \in A(E_{st})$ and $p_b(v) \in B(E_{s't'})$. By naturality of a and b , it follows that $a(p_a(v)) \in C(E_{st})$ and $b(p_b(v)) \in B(E_{s't'})$. But by commutativity of the pullback diagram, $a(p_a(v)) = b(p_b(v))$, and thus $s = s'$ and $t = t'$. We conclude by unicity of the pullback that $v \in P(E_{st})$ and thus $P(E) = \bigsqcup_{st} P(E_{st})$.

Pushouts. The same argument as for pullbacks also applies to pushouts: given a pushout P of $A \xrightarrow{a} C \xleftarrow{b} B$ in $[\text{lin-}\mathbb{C}, \text{Set}]$ with $A, B, C \in \text{lin-}\mathbb{H}$, an element $v \in P(E)$ that makes the pushout square commute must have preimages in $A(E_{st}), B(E_{st})$ and $C(E_{st})$ for some $s, t \in \mathbb{N}$. Thus the pushout distributes over the coproduct, and we can conclude that $P(E)$ is the coproduct of pushouts.

The same argument also applies to $\text{lin-}\mathbb{H}_{\text{type}}$ ¹.

Now to the spicy stuff:

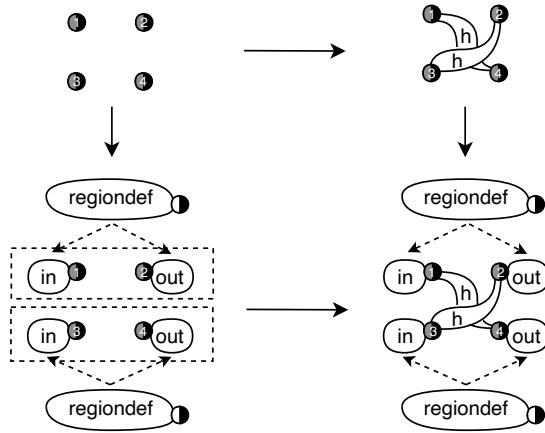
Proposition 3.3 *Non-adhesivity of hierarchical hypergraphs*

Whilst $\text{hier-lin-}\mathbb{H}_{\text{type}}$ is adhesive, the category $\text{hier-lin-}\mathbb{H}$ is NOT adhesive.

1. In fact, a much simpler argument applies: the category $\text{lin-}\mathbb{H}_{\text{type}}$ is isomorphic to the presheaf category $[\text{lin-}\tilde{\mathbb{C}}_{\text{type}}, \text{Set}]$, where $\text{lin-}\tilde{\mathbb{C}}_{\text{type}}$ is obtained from $\text{lin-}\mathbb{C}_{\text{type}}$ by removing the object E . Adhesivity follows. ↩

Proof. $\text{hier-lin-}\mathbb{H}_{\text{type}}$ is a presheaf – hence adhesive.

The following pushout square shows that $\text{hier-lin-}\mathbb{H}$ cannot be adhesive: the pushout square is valid in $[\text{hier-lin-}\mathbb{C}, \text{Set}]$, but the pushout at the bottom right is not in $\text{hier-lin-}\mathbb{H}$, because the child regions cannot each be assigned a unique parent.



Double pushout semantics

From Proposition 3.3, it follows that minIR graph transformations can be performed through the double pushout (DPO) construction [126] in the $[\text{hier-lin-}\mathbb{C}, \text{Set}]$ category.

Definition 3.6 Double pushout (DPO) transformation

A transformation rule p in an adhesive category $\mathbb{A}$ is a span $L \leftarrow I \rightarrow R$. For objects $G, H \in \mathbb{A}$, we then write $G \xrightarrow{(p,m)} H$ or $G \xrightarrow{p} H$ if there is a *matching* morphism $m : L \rightarrow G$ and a *context* object C along with morphisms $G \leftarrow C \rightarrow H$ and $I \rightarrow C$ such that the following diagram commutes and both squares are pushouts:

$$\begin{array}{ccccc}
 L & \longleftarrow & I & \longrightarrow & R \\
 m \downarrow & & \downarrow & & \downarrow \\
 G & \longleftarrow & C & \longrightarrow & H
 \end{array}$$

If the DPO transformation $G \xrightarrow{(p,m)} H$ exists for some rule p and match m , then we say $G \Rightarrow H$ is a valid DPO rewrite.

To ensure that a DPO rewrite is valid in minIR , we must impose certain conditions. Let G be an IO-free minIR graph, i.e. $G \in \text{hier-lin-}\mathbb{H}$, there is a morphism $G \rightarrow \Sigma$ in $\text{hier-lin-}\mathbb{H}_{\text{type}}$ for some type system Σ and $I = O = \emptyset$.

A DPO rewrite $G \Rightarrow H$ is a valid minIR DPO rewrite if there is a transformation $G \xRightarrow{p} H$ in $[\text{hier-lin-}\mathbb{C}, \text{Set}]$ and

2. This is often called *left-linear* in the literature. We avoid this term in this thesis to avoid confusion with the linearity property of values in minIR . ↩

1. p is left-mono, i.e. the morphism $I \rightarrow L$ is mono,²
2. the pushout complement C and pushout H also exist in the slice category $\text{hier-lin-}\mathbb{H}_{\text{type}} \searrow \Sigma$,
3. H satisfies the hierarchy condition of Definition 3.3,
4. H is IO-free.

Proposition 3.4

If G is a minIR graph and $G \Rightarrow H$ is a valid minIR DPO rewrite, then H is a valid minIR graph.

Proof. We know by construction that $H \in [\text{hier-lin-}\mathbb{C}, \text{Set}]$. We must show that H further satisfies the constraints to be an object in the full subcategory of minIR graphs.

The first condition is standard in DPO and guarantees that C and D are unique if they exist.

The third condition we impose on H corresponds directly to the constraint that defines hierarchical graphs in $\text{hier-lin-}\mathbb{H}$. The fourth condition ensures that valid minIR DPO rewrites map IO-free graphs to IO-free graphs.

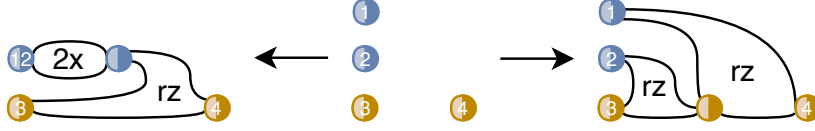
Finally, the second condition is imposed to ensure well-typedness of H . The functor $\text{hier-lin-}\mathbb{H} \rightarrow \text{hier-lin-}\mathbb{H}_{\text{type}}$ that forgets the *def* and *use* morphisms is a left adjoint (it possesses a right Kan extension defined pointwise), and thus preserves colimits. The images of C and H thus form pushout squares in $\text{hier-lin-}\mathbb{H}_{\text{type}}$, and by unicity, must match the pushout squares in $\text{hier-lin-}\mathbb{H}_{\text{type}} \searrow \Sigma$. Hence H is well-typed.

The restriction to rewrites of IO-free graphs is not a restriction of generality: if we are interested in rewriting computations with inputs and outputs, we can always express them as IO-free graphs by adding `input` and `output` ops with the values in I as outputs, respectively O as inputs. We assign them dedicated types distinct from all other operations; these operations will never be matched by transformation rules and can be removed at the end of rewriting.

Generalising to sesqui-pushouts

We restricted minIR rewrites to DPO transformations obtained from left-mono rules, to ensure that the construction is unique. This excludes rules that may identify two values in G but split them into two different values in H . Such rules

allow for cloning values, which is a useful transformation in minIR for non-linear values. An example of a transformation rule that we would like to allow in minIR:



For this example we added a `2x` operation that multiplies an angle value passed as input by two. The transformation rule replaces a rotation of angle 2α by two rotations of angle α by cloning the input angle.

Such semantics are possible using the sesqui-pushout construction (SqPO) by Corradini et al. [92]. We can reuse the same $\xRightarrow{(p,m)}$ notation: when DPO is restricted to left-mono rules as we have done, SqPO is a generalisation of DPO (i.e. the construction coincides whenever the DPO exists).

Definition 3.7 Sesqui-pushout (SqPO) transformation

A transformation rule p in an adhesive category $\mathbb{A}$ is a span $L \leftarrow I \rightarrow R$. For objects $G, H \in \mathbb{A}$, we then write $G \xRightarrow{(p,m)} H$ or $G \xRightarrow{p} H$ if there is a *matching* morphism $m : L \rightarrow G$ and a *context* object C along with morphisms $G \leftarrow C \rightarrow H$ and $I \rightarrow C$ such that C is the final pullback complement of $I \rightarrow L \xrightarrow{m} G$ and the right square is a pushout:

$$\begin{array}{ccccc} L & \xleftarrow{\quad} & I & \xrightarrow{\quad} & R \\ m \downarrow & & \downarrow & & \downarrow \\ G & \xleftarrow{\quad} & C & \xrightarrow{\quad} & H \end{array}$$

If the SqPO transformation $G \xRightarrow{(p,m)} H$ exists for some rule p and match m , then we say $G \Rightarrow H$ is a valid (SqPO) rewrite.

The left square is redundant in the diagram above, as it follows from the requirement that C be the final pullback complement (FPC). It is kept to highlight the similarities to DPO. As the commuting diagram indicates, the final pullback complement (FPC) construction forms a pullback square. Furthermore, unlike pushout complements, the FPC is defined by a universality property that ensures uniqueness if it exists. We refer to [92] for the exact FPC construction.

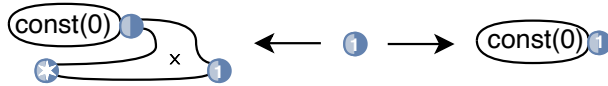
With SqPO, we can define the set of valid minIR rewrites as given by the SqPO transformations $G \xRightarrow{p} H$ in $[\text{hier-lin-}\mathbb{C}, \text{Set}]$ satisfying the relaxed set of conditions

1. the pushout complement C and pushout H also exist in the slice category $\mathbf{hier-lin-HI}_{\text{type}} \searrow \Sigma$,
2. H satisfies the hierarchy condition of Definition 3.3,
3. H is IO-free.

We conclude this section with a discussion of some of the properties of minIR transformations using SqPO (referring again to Corradini [92] or [198] for a more detailed explanation of the concepts discussed):

Deletion in unknown context. A key difference between DPO and SqPO transformations is that SqPO transformations on graphs will delete edges attached to a vertex v_d that is deleted by the transformation rule (i.e. $v_d \in L$ but $v_d \notin R$ of the rule). The DPO transformation on the other hand is only well-defined when all edges incident to v_d are in the image of m and thus explicitly deleted (this is known as the *dangling condition*).

As minIR rewrites follow SqPO semantics, transformation rules such as the following are allowed:



Here $\times$ denotes the multiplication of angles and $\text{const}(0)$ the zero angle. Any operation that would be connected to the starred value on the left would be deleted by this rule. However such an implicit operation deletion only yields valid minIR graphs if all incident values are non-linear and none of the target values of the deleted operation are used.

Non-left-mono rules. As discussed in the introduction to SqPO, the cloning of values is allowed in minIR rewrites. However, linear values may never be cloned (the FPC or pushout will not exist in these cases). Thus any minIR transformation rule will be left-mono on linear values. It must further be left-linear on all (linear and non-linear) values in I that are mapped to outputs in R : if a value w is produced by op applied to v , then cloning v and 'op' will result in two definitions of w .

Non-right-mono rules. Non-right-mono rules are allowed in both DPO and SqPO. They result in vertex merges. In minIR, the situation for right-mono is symmetric to left-mono: the map must be mono on linear values (otherwise the same value will have multiple uses or definitions) and it must be mono on all values in I that are mapped to inputs in L (otherwise a value in the rewritten minIR graph will have more than one value definition).

3.5. MinIR rewriting, operationally

The previous section proposed to view minIR rewrites as the result of a (DPO or SqPO) graph transformation. This yields valid rewriting semantics elegantly (and with little effort!). However, the conditions that must be imposed on the transf-

ormation to be valid, along with the fact that pushouts may not exist mean that the existence of a rewrite given a transformation rule and a match is not guaranteed.

In this section, we address this by considering a more restricted notion of minIR rewriting, for which the existence of the right-hand side of the rewrite is guaranteed. In addition, in place of the categorical presentation of the last section, we express the rewriting operation *operationally*, i.e. as data and a procedure on sets that translates directly into an algorithmic implementation.

We find that this rewrite definition is sufficient in practice. We conclude the section with an example of how more complex rewrites can be achieved by composition of simpler rewrites that can be expressed in this framework.

Graph glueings and rewrites

Throughout, we consider graph glueings on disjoint vertex and (hyper)edge sets. To underline this, we will use the $\sqcup$ symbol to denote disjoint set unions.

As we will be working exclusively with vertex and edge sets in this section (as opposed to the objects in the indexing category), we will drop the bold typeface for sets, writing e.g. V instead of $\mathbf{V}$ for the set of vertices of a hypergraph.

Finally, all minIR graphs in this section are IO free.

We define local graph rewrites in terms of graph glueings. Consider first the case of two arbitrary graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, along with a relation $\mu \subseteq V_1 \times V_2$. Let $\sim_\mu \subseteq (V_1 \sqcup V_2)^2$ be the equivalence relation induced by μ , i.e. the smallest relation on $V_1 \sqcup V_2$ that is reflexive, symmetric and transitive, and satisfies for all $v_1 \in V_1$ and $v_2 \in V_2$,

$$(v_1, v_2) \in \mu \Rightarrow v_1 \sim_\mu v_2.$$

Then, we can define

- $V = (V_1 \sqcup V_2) / \sim_\mu$ is the set of all equivalence classes of $\sim_\mu$, and
- for $v \in V_1 \sqcup V_2$, $\alpha_\mu(v) \in V$ is the equivalence class of $\sim_\mu$ that v belongs to.

Definition 3.8 Graph glueing

The glueing of G_1 and G_2 according to the glueing relation μ is given by the vertices $V = (V_1 \sqcup V_2) / \sim_\mu$ and the edges

$$E = \{(\alpha_\mu(u), \alpha_\mu(v)) \mid (u, v) \in E_1 \sqcup E_2\} \subseteq V^2.$$

We write the glueing graph as $(G_1 \sqcup G_2) / \sim_\mu$.

In other words, the glueing is the disjoint union of the two graphs, with identification (and merging) of vertices that are related in μ .

This allows us to define a rewrite on a graph G :

Definition 3.9 *Graph rewrite*

A rewrite r on a graph $G = (V, E)$ is given by a tuple $r = (G_R, V^-, E^-, \mu)$, with

- $G_R = (V_R, E_R)$ is a graph called the *replacement graph*,
- $V^- \subseteq V$ is the *vertex deletion set*,
- $E^- \subseteq E \cap \text{dom}(\mu)^2$ is the *edge deletion set*, and
- $\mu : V^- \rightarrow V_R$ is the *glueing relation*, a partial function that maps a subset of the deleted vertices of G to vertices in the replacement graph.

The domain of definition $\text{dom}(\mu)$ is known as the boundary values of r .

A graph rewrite per this definition can always be generated by a single pushout (SPO) transformation [214].

- define L as the graph (V^-, E^-) . Then the injection $L \subseteq G$ is the match morphism $L \rightarrow G$;
- the partial map μ maps a subset of V^- to vertices in the replacement $R = G_R$. By injectivity of the match morphism, it also defines a partial map $L \rightarrow R$.

We opted for SPO-like semantics in this definition, as they are the simplest to write in set-theoretic terms and coincide with DPO and SqPO in our restricted domain of interest.

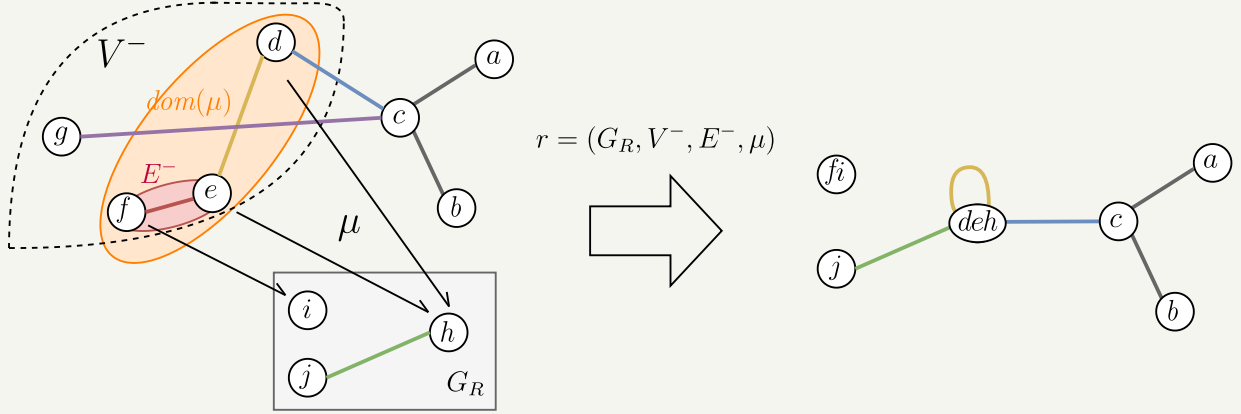
The result of the rewrite is computed by gluing the right-hand side G_R to the *context* subgraph $G_C = (V_C, E_C)$ of G given by

$$\begin{aligned} V_C &= (V \setminus V^-) \cup \text{dom}(\mu) \\ E_C &= (E \setminus E^-) \cap V_C^2. \end{aligned} \tag{9}$$

The partial function μ is a special case of a glueing relation $\mu \subseteq V_C \times V_R$, and thus defines a glueing of G_C with G_R . The rewritten graph resulting from applying r to G is

$$r(G) = (G_C \sqcup G_R) / \sim_\mu.$$

An example of a graph rewrite is given in the next figure. This is equivalent to an SPO transformation with the graph induced by V^- on the left-hand side, the graph G_R on the right-hand side and the partial map $L \hookrightarrow R$ given by μ .



Application of a graph rewrite. On the left, the original graph G along with the replacement graph G_R (grey box). On the right, the rewritten graph $r(G)$. Only the vertex g has been deleted, as other vertices in V^- are in the boundary $\text{dom}(\mu)$ (in orange). The (singleton) edge deletion set is red. The blue edge connects a vertex of $V \setminus V^-$ to a boundary vertex, and is thus also present on the right-hand side. The purple edge, on the other hand, connects a vertex of $V \setminus V^-$ to a non-boundary vertex of V^- , and is thus deleted.

When there are no edges between $V \setminus V^-$ and $V^- \setminus \text{dom}(\mu)$ (purple in the example above), this definition corresponds to graph rewrites that can be produced using DPO transformations (see discussion in section 3.4). Otherwise, such edges are deleted.

The notions of graph glueing and graph rewrite can straightforwardly be lifted to hypergraphs and, by extension, to minIR graphs. Notice that in this case, values are glued together, not operations (the former were defined as the graph's vertices, the latter as its hyperedges).

However, the glueing of two valid minIR graphs – and the result of applying a valid rewrite – may not be a valid minIR graph. Glueing two values of a linear type, for instance, is a sure way to introduce multiple uses (or definitions) of it. Thus, we must be careful to only consider glueings and rewrites of minIR graphs that preserve all the constraints we have imposed in Definition 3.4.

Ensuring rewrite validity: interfaces

As a sufficient condition for valid minIR rewrites, we introduce minIR *interfaces*, a concept closely related to the “hypergraph with interfaces” construction of [58] or the supermaps of quantum causality [163]. We eschew the presentation of holes as a slice category in favour of a definition that fits naturally within minIR and is sufficient for our purposes.

Let G be a Σ -typed minIR graph with data types T and linear types $T_L \subseteq T$. Consider type strings $S, S' \in T^*$. We define the index sets

$$\begin{aligned} \text{Idx}(S) &= \{i \in \mathbb{N} \mid 1 \leq i \leq |S|\} \\ \text{Idx}_L(S) &= \{i \in \text{Idx}(S) \mid S_i \in T_L\} \subseteq \text{Idx}(S) \end{aligned}$$

corresponding respectively to the set of all indices into S and the subset of indices of linear types. For any $i \in \text{Idx}(S)$, we denote by S_i the type at position i in S .

1. To be precise, $\preceq$ is a partial order on the type strings up to isomorphism. $\hookrightarrow$

We define a partial order $\preceq^1$ on T^* where $S \preceq^1 S'$ and say that S' can be coerced into S if there exists an *index map* $\rho : \text{Idx}(S) \rightarrow \text{Idx}(S')$ such that

- types are preserved: $S_i = S'_{\rho(i)}$, and
- ρ is well-defined and bijective on the restriction to indices of linear types

$$\rho|_{\text{Idx}_L(S)} : \text{Idx}_L(S) \rightarrow \text{Idx}_L(S').$$

Definition 3.10 *Interface*

Let T be a set of data types. An interface $I = (U, D)$ is a pair of type strings $U, D \in T^*$.

We say that an interface $I' = (U', D')$ can be *coerced* into an interface $I = (U, D)$, written $I \triangleleft I'$, if $U \succcurlyeq U'$ and $D \preceq D'$.

We can define the interface associated with an operation o in a minIR graph G by considering the values used and defined by o . Calling τ the type morphism on G and assuming $o \in E_{st}$ to be an operation in G with s inputs and t outputs, we define the interface of o in G as the pair of strings in T^*

$$I(o) = (\tau(\text{src}_1(o)) \cdots \tau(\text{src}_s(o)), \tau(\text{tgt}_1(o)) \cdots \tau(\text{tgt}_t(o))).$$

Similarly, we can assign interfaces to *subgraphs* of minIR graphs:

Definition 3.11 *MinIR subgraph*

Consider a subset of values and operations $V_H \subseteq V$ and $E_H \subseteq E$. Define the *use* and *define* boundary sets

$$\begin{aligned} B_U &= \{v \in V_H \mid \text{def}(v) \in E \setminus E_H\}, \\ B_D &= \{v \in V_H \mid \text{use}(v) \in E \setminus E_H\}. \end{aligned} \tag{10}$$

The tuple $H = (V_H, E_H)$ of G is called a *minIR subgraph* of G if there exists a region R of G such that all boundary values of H are in R :

$$B = B_U \cup B_D \subseteq R.$$

We write $H \subseteq G$ to indicate that H is a minIR subgraph of G .

Note that B_U is exactly the set of inputs I in the non-IO free minIR graph given by the subgraph (V_H, E_H) of the minIR graph. B_D is a superset of the outputs O

is exactly the set of inputs I in the non-IO free minIR graph given by the subgraph (V_H, E_H) of the minIR graph. B_D is a superset of the outputs O of H : it includes all linear values in H that do not have a use in H , but also any non-linear value that has a use outside of H .

Unlike interfaces, subgraph boundary values are not ordered. An *ordering* of $B \subseteq V$ is a string $S \in V^*$ along with a bijective map

$$\text{ord} : B \rightarrow \text{Idx}(S) \quad \text{such that} \quad v = S_{\text{ord}(v)}.$$

If there are strings $S_U, S_D \in V^*$ and orderings of B_U and B_D

$$\text{ord}_U : B_U \rightarrow \text{Idx}(S_U) \quad \text{ord}_D : B_D \rightarrow \text{Idx}(S_D), \quad (11)$$

then we can set $\text{src}_i(H) = (S_U)_i$ and $\text{tgt}_i(H) = (S_D)_i$ in complete analogy to operations. We will write $\text{src}(H)$ and $\text{tgt}(H)$ for the strings $\text{src}_1(H) \cdots \text{src}_{|S_U|}(H)$ and $\text{tgt}_1(H) \cdots \text{tgt}_{|S_D|}(H)$ respectively. We say that the subgraph H *implements* the interface

$$I_H = (\tau(\text{src}(H)), \tau(\text{tgt}(H))),$$

where the type morphism τ was extended element-wise to strings V^* .

Remark, though, that unlike operations, the same subgraph may implement more than one interface as a result of various choices of orderings ord_U and ord_D .

As mentioned, the subgraph H forms a non-IO free minIR graph. We can always construct an IO-free minIR graph from H by adding two operations o_{in} and o_{out} in the root region respectively in $E_{0,|S_U|}$ and $E_{|S_D|,0}$ inputs-outputs, defined by

$$\text{tgt}_i(o_{in}) = \text{src}_i(H), \quad \text{src}_i(o_{out}) = \text{tgt}_i(H).$$

We call the resulting graph $\bar{H}$ an *interface graph*. It *implements* the interface I_H if H implements I_H . Calling to mind the illustrations of section 3.3, $\bar{H}$ looks like one of the nested regions within `regiondef` operations that we were considering.

MinIR operation rewrite

Consider

- an operation o in a minIR graph G with values V ,
- an interface graph $\bar{H}$ with values V_H and its associated subgraph $H \subseteq \bar{H}$, such that H implements an interface $I(o) \triangleleft I_H$,
- the index maps $\rho : \text{Idx}(\text{src}(H)) \rightarrow \text{Idx}(\text{src}(o))$ and $\sigma : \text{Idx}(\text{tgt}(o)) \rightarrow \text{Idx}(\text{tgt}(H))$ that define the generalisation $I(o) \triangleleft I_H$ (per Definition 3.10).

We can define a glueing relation $\mu_o \subseteq V \times V_H$

$$\mu_o = \{ (src_{\rho(i)}(o), src_i(H)) \mid i \in \text{Idx}(src(H)) \} \cup \{ (tgt_i(o), tgt_{\sigma(i)}(H)) \mid i \in \text{Idx}(tgt(o)) \}. \quad (12)$$

This is almost enough to define a rewrite that replaces the operation o in G with the values and operations of H – the interface compatibility constraint $I(o) \triangleleft I_H$ that we have imposed ensures that the resulting minIR graph is valid.

Unfortunately, μ_o is not a partial function as required by Definition 3.4.

This is resolved in the following proposition:

Proposition 3.5 *MinIR operation rewrite*

Let G , o and H such that $I(o) \triangleleft I_H$, as defined above. Then

$$((G \sqcup H) / \sim_{\mu_o}) \setminus \{o\}, \quad (13)$$

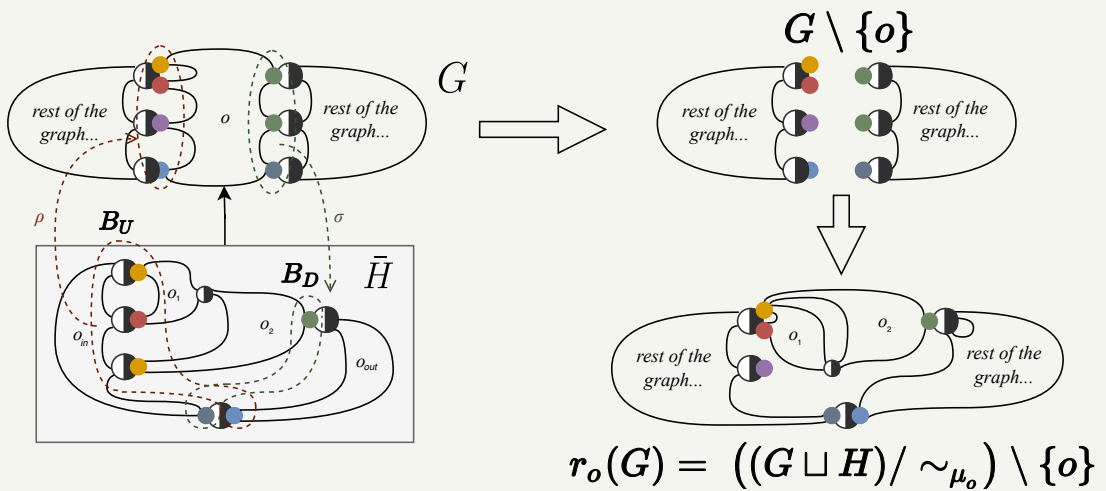
i.e. the graph obtained by removing the operation o from the glueing of G and H along μ_o , is a valid minIR graph.

There is a graph G_R with values V_R and a partial function $\mu'_o : V \rightarrow V_R$ such that the graph (13) is the graph $r_o(G)$, obtained from the rewrite

$$r_o = (G_R, \text{dom}(\mu_o), \{o\}, \mu'_o). \quad (14)$$

We call r_o the rewrite of o into H .

The definition of the rewrite of o into a graph H behaves as one would expect – the only subtleties relate to handling non-linear (i.e. copyable) values at the boundary of the rewrite. The following example illustrates some of these edge cases.



When the index maps ρ and σ are not injective (yellow and green dots), values are merged, resulting in multiple uses of the value (i.e. copies). This is why the index maps must be injective on linear values (dots in shades of blue). Value merging also happens when a value is used multiple times in o (yellow and red dots). This will never happen with linear values (as they can never have more than one use in o), nor with any value definitions (the same value can never be defined more than once). Finally, values not in the image of ρ or σ (purple dot) are discarded. This case is also excluded for linear values by requiring surjectivity.

Proof. We start this proof with the explicit construction of G_R and μ'_o . Define $\sim_R \subseteq (V_H)^2$ as the smallest equivalence relation such that

$$src_{\rho(i)}(o) = src_{\rho(j)}(o) \Rightarrow tgt_i(o_{in}) \sim_R tgt_j(o_{in}).$$

Then we define $\bar{G}_R = \bar{H} / \sim_R$, the graph obtained by glueing together values within the same equivalence class of $\sim_R$.

Claim 1: $\bar{G}_R$ is a valid minIR graph.

Claim 1 follows from the observation that only values of non-linear types are glued together. If $v \sim_R v'$, then either $v = v'$ or there exist $i \neq j$ such that

$$tgt_i(o_{in}) \sim_R tgt_j(o_{in}).$$

If $\rho(i) = \rho(j)$, then ρ is not injective on i and j , and by the definition of ρ , $\tau(v) \notin T_L$ and $\tau(v') \notin T_L$. Otherwise, there are $i' = \rho(i) \neq \rho(j) = j'$ such that $src_{i'}(o) = src_{j'}(o)$. The same value is used twice, which is only a valid minIR graph if v and v' are not linear, thus proving Claim 1.

Define G_R as the subgraph obtained from $\bar{G}_R$ by removing the operations $\{o_{in}, o_{out}\}$. Let $V_R = V_H / \sim_R$ be the set of values of G_R (and of $\bar{G}_R$). Writing $\alpha_R(v) \in V_R$ for the equivalence class of $\sim_R$ that $v \in V_H$ belongs to, we can define $\mu'_o \in V \times V_R$ as:

$$(v, w) \in \mu_o \Leftrightarrow (v, \alpha_R(w)) \in \mu'_o.$$

Claim 2: μ'_o is a partial function $V \rightarrow V_R$.

In other words, for all $(v, \alpha_1), (v, \alpha_2) \in \mu'_o$, then $\alpha_1 = \alpha_2$. Let $w_1 \in \alpha_1$ and $w_2 \in \alpha_2$ be values in V_H . First of all, $src_i(o) \neq tgt_j(o)$ for all i, j , otherwise G is not acyclic. So either $use(v) = o$, or $def(v) = o$, but not both.

The simpler case: if $def(v) = o$, then there exists i such that $tgt(o)_i = v$. Furthermore i is unique because by minIR definition, v has a unique definition in G . It follows from (12) that $w_1 = src_{\rho(i)}(o_{out}) = w_2$ and hence $\alpha_1 = \alpha_2$.

Otherwise, there exists i and j such that $v = src_{\rho(i)}(o) = src_{\rho(j)}(o)$ and $tgt_i(o_{in}) = w_1$ as well as $tgt_j(o_{in}) = w_2$. By definition of $\sim_R$, we have $w \sim_R w'$, and thus

$$\alpha_1 = \alpha_R(w_1) = \alpha_R(w_2) = \alpha_2,$$

proving Claim 2.

Claim 3: $r_o(G)$ is given by $((G \sqcup H) / \sim_{\mu_o}) \setminus \{o\}$.

It follows directly from our construction of $\sim_R$ and μ'_o that the equivalence classes of (the smallest equivalence relation closure of) $\mu'_o \circ \alpha_R$ is equal to the equivalence classes of (the smallest equivalence relation closure of) μ_o . The claim follows by Definition 3.8 and the definition of r_o .

And finally, *Claim 4:* $r_o(G)$ is a valid minIR graph.

Per Definition 3.4, We must check four properties: (i) every value is defined exactly once, (ii) every linear value is used exactly once, (iii) the graph is acyclic, and (iv) every region has (at most) one parent.

(iii) follows from the fact that G and H are acyclic and a single operation o in G is replaced: any cycle across G and H would also be a cycle in G by replacing the subpath in H with o . (iv) follows from the fact that o_{in} and o_{out} are in the root region of $\bar{H}$, by definition of interface implementation. (i): removing o from G removes the unique definitions of all values that are targets of o . Each such value v is glued to a unique value $src_i(o_{out})$ in H – the new and unique definition of v in $r_o(G)$. (ii) follows from the same argument as in (i), but relying on injectivity of ρ on linear values to establish uniqueness.

Arbitrary minIR rewrites

We have so far defined rewrites of single operations into graphs H . We can generalise these rewrites to rewrite subgraphs $P \subseteq G$, provided the minIR subgraphs satisfy some constraints. We require for this a notion of convexity, as discussed in [57].

As usual, let us consider a minIR graph G with values V , linear values $V_L \subseteq V$, edges E , the incidence maps src_i and tgt_j as well as their inverses use and def . Consider further a subgraph of G that we will now call $P = (V_P, E_P) \subseteq G$, to distinguish from H .

Let us further define the partial *parent* morphism that maps a value $v \in V$ to the parent of the region of v .

Definition 3.12 Convex minIR subgraph

A minIR subgraph $P \subseteq G$ is *convex* if the following conditions hold:

- for all $v_1, v_2 \in V_P$, any path along $\rightsquigarrow$ from v_1 to v_2 contains only vertices in V_P ,
- parent-child relations are contained within the subgraph, i.e.

$$v \in V_P \cap \text{dom}(\text{parent}) \Leftrightarrow \text{parent}(v) \in V_P.$$

Define the sets of boundary values B_U, B_D and $B = B_U \cup B_D$, as in (10); then fix the boundary orderings $src(P)$ and $tgt(P)$ as in (11). The subgraph P implements the interface

Consider an interface graph $\bar{H}$ that implements I_H such that $I_P \triangleleft I_H$. Instead of defining a gluing relation from values of an operation o to values of H , we replace the interface $I(o)$ with I_P . This generalises the definition of μ_o from (12) to a glueing $\mu \subseteq B \times V_H$ defined as

$$\mu = \{((src_{\rho(i)}(P)), src_i(H)) \mid i \in \text{Idx}(src(H))\} \cup \{((tgt_i(P)), tgt_{\sigma(i)}(H)) \mid i \in \text{Idx}(tgt(P))\}, \quad (15)$$

With the set of *boundary operations* defined as²

$$E_B = \{o \in E_P \mid (tgt(o) \cup src(o)) \subseteq B\},$$

we are able to define minIR rewrites in their most general form.

2. The set operations $\subseteq$ and $\cup$ are again understood to apply to the unordered set of elements contained in the lists $tgt(o)$ and $src(o)$. ↩

Proposition 3.6 *MinIR subgraph rewrite*

Let $P \subseteq G$ and H such that $I_P \triangleleft I_H$ and P is convex, as defined above. Then,

$$((G \sqcup H) / \sim_\mu) \setminus (V_P \setminus B, E_B), \quad (16)$$

i.e. the graph obtained by removing the values $V_P \setminus B$ and operations E_B from the glueing of G and H along μ , is a valid minIR graph.

There is a graph G_R with values V_R and a partial function $\mu' : V_P \rightarrow V_R$ such that the graph (16) is the graph $r_P(G)$, obtained from the rewrite

$$r_P = (G_R, V_P, E_B, \mu').$$

We call r_P the rewrite of P into H .

Proof. Consider an operation o that implements $I_P = (U_P, D_P)$. We can define the interface graph $\bar{H}_o$ given by three operations o_{in} , o_{out} and o . Its associated subgraph $H_o \subseteq \bar{H}_o$ only includes o . Let $\tilde{\mu}$ be the glueing relation

$$\tilde{\mu} = \{(src_i(P), src_i(o)) \mid i \in \text{Idx}(U_P)\} \cup \{(tgt_i(P), tgt_i(o)) \mid i \in \text{Idx}(D_P)\}.$$

Consider the rewrite $r = (H_o, V_P \setminus B, E_B, \tilde{\mu})$. If we write $G' = (V', E')$ for the subgraph of G given by

$$\begin{aligned} V' &= (V \setminus (V_P \setminus B)) \\ E' &= (E \setminus E_B) \cap (V')^*, \end{aligned}$$

then according to (9), the graph resulting from applying r to G can be expressed as the glueing

$$G_o = r(G) = (G' \sqcup H_o) / \sim_{\tilde{\mu}}.$$

Our claim is that G_o is a valid minIR graph.

The graph (16) is then obtained by applying the rewrite r_o as given by (14) to G_o . Defining the rewrite r_P as the composition of r followed by r_o , the result follows from our claim and Proposition 3.5.

We now prove the claim, by showing the four properties of minIR graphs as per Definition 3.4. Property *i*) requires showing that every value is defined exactly once. As G' is obtained by removing values and operations from a valid minIR graph G , no value in V' can be defined more than once. A value $v \in V'$ that is not defined in G' must be in the boundary $v \in B$ of P . By the boundary definitions of (10), v cannot be in B_U and thus must be in B_D . It follows by the definition of the glueing $\tilde{\mu}$ that in G_o , v will be in the definitions of o : $\text{def}(v) = o$. The glueing $\tilde{\mu}$ is bijective between the values of P and o and thus we can conclude that v has a unique definition in G_o .

The same argument applies to property *ii*). Property *iii*) follows from the convexity requirement of P . Finally, property *iv*) (every region has at most one parent) follows from two observations. First, by convexity of P , no deleted value or operation could be the parent of any value not in P , and thus the *parent* relation is well-defined on G' : $\text{im}(\text{parent}) \subseteq E'$. Secondly, all new values and operations added to the boundary region of G' are from the root region of H , and thus do not have a parent, ensuring that parent uniqueness is preserved.

This simple and limited graph transformation framework captures a remarkably large set of minIR program transformations. It may seem at first that the restriction to boundary values within a single region of Definition 3.11, as well as the convexity requirements of Definition 3.12 represent significant limitations on the expressivity of the rewrites. In practice, however, the semantics of minIR operations can be used to decompose more complex rewrites into a sequence of simple rewrites to which Proposition 3.6 applies.

Consider minIR graphs with a type system that includes `regiondef` and `call` operations as discussed in examples of the previous section – respectively defining a code block by a nested region and redirecting control flow to a code block defined using a `regiondef`. Then all constraints that we impose on rewriting can be effectively side-stepped using the *region outlining* and *value hoisting* transformations.

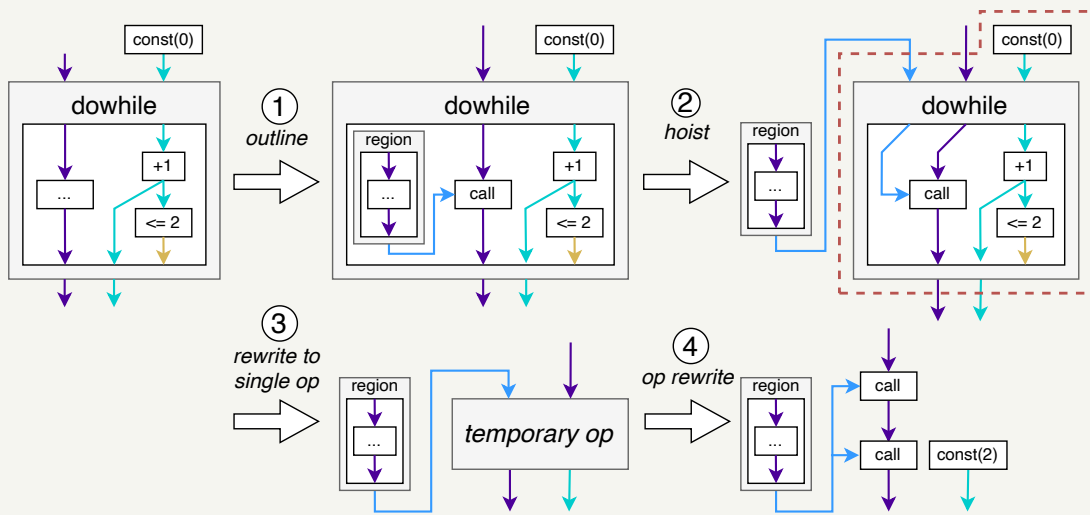
Region outlining moves a valid minIR subgraph into its own separate region, and replaces the hole left by the subgraph in the computation by a `call` operation to the newly outlined region.

Value hoisting moves a value definition within a region to its parent region and passes the value down to the nested region through an additional input. In case of linear values, we can similarly hoist the unique *use* of the value to the parent region.

Using these transformations, non-convex subgraphs can always be made convex by taking the convex hull and outlining any parts within it that are not part of the subgraph. Outlined regions can then be passed as additional inputs to the subgraph. Step 1 of the figure below illustrates this transformation. Similarly, a subgraph that includes operations without their parent can be extended to cover the entire region and its parent, outlining any parts of the region that are not part of the subgraph.

Finally, whenever a boundary value v belongs to a region that is not the top level region of the subgraph³, we can repeatedly hoist v to its parent region until it is in the top level region. The value is then recursively passed as argument to descendant regions until the region that it is required in. Subgraphs can thus always be transformed to only have input and output boundary values at the top level region. Step 2 of the figure below illustrates this transformation.

3. We can always extend a subgraph to contain more ancestor regions, until there is indeed a unique top-level region in the subgraph. ↩



A non-convex minIR graph rewrite, obtained by decomposition into valid convex rewrites, using outlining and hoisting. For simplicity, `regiondef` operations were made implicit and represented by nested boxes: a region within an operation corresponds to a region definition that is passed as an argument to the operation. Edge colours correspond to value types. Step 1 outlines the `...` operations into a dedicated region, which step 2 hoists outside of the region being rewritten. Step 3 and 4 together correspond to a minIR subgraph rewrite. They have been split into two steps following the proof strategy. Step 4 is an instance of a minIR operation rewrite.

Pattern Matching in large Graph Transformation Systems

To our knowledge, the first practical proposal for a GTS-based quantum compiler was presented in [332] and then refined in [333]. In these, the set of possible graph transformations is obtained by exhaustive enumeration. Using SAT solvers and fingerprinting techniques, the set of all small programs up to a certain size can be generated ahead of time and clustered into disjoint partitions of equivalent programs. This concisely expresses every possible peephole optimisation up to the specified size: for every small enough subset of operations of an input program, its equivalence class can be determined. Any replacement of that set of operations with another program in the same equivalence class is a valid transformation and, thus, a potential peephole optimisation. Transformation systems on minIR graphs based on equivalence classes were formalised in section 3.4.

First results of this approach are promising. [332] demonstrated that optimisation performance improves markedly with larger sets of transformation rules. Such workloads however rely heavily on *pattern matching*, the computational task that identifies subgraphs on which transformation rules apply. In [332] and [333], pattern matching is carried out separately for each pattern. This becomes a significant bottleneck for large rule sets. In [332], performance peaks at around 50,000 transformation rules, after which the additional overhead from pattern matching becomes dominant, deteriorating the compilation results.

In this chapter, we solve these scaling difficulties by presenting an algorithm for pattern matching on minIR graphs that uses a pre-computed data structure to return all pattern matches in a single query. The set of transformation rules is directly encoded in this data structure. After a one-time cost for construction, pattern-matching queries can be answered in running time independent of the number of rules in the transformation system.

The asymptotic complexity results presented in this chapter depend on some simplifying assumptions on the properties that the pattern graphs and embeddings must satisfy. This represents a restriction on the generality of minIR graphs, but we do not find that they restrict the usefulness of the result in practice. As discussed in section 4.7, none of these assumptions are required in practice for the

implementation. We have not observed any impact on performance when the imposed constraints are lifted, so we conjecture that at least some of these assumptions can be relaxed and our results generalised.

After a discussion of related work in section 4.1, Section 4.2 presents the assumptions that we are making in detail, along with some relevant definitions for the rest of the chapter. Sections 4.3, 4.4, and 4.5 present the core ideas of our approach, respectively introducing: a reduction of minIR graphs to equivalent trees, a canonical construction for the tree reduction and an efficient way to enumerate all possible subtrees of a graph. We also prove bounds on the size and number of the resulting trees.

In section 4.6, we introduce a pre-computation step and show that the pattern-matching problem reduced to tree structures can be solved using a prefix tree-like automaton that is fixed and pre-computed for a given set of patterns. Combining the automaton construction with bounds from section 4.5 leads to the final result. We conclude in section 4.7 with benchmarks on a real-world dataset of 10000 quantum circuits, obtaining a 20x speedup over a leading C++ implementation of pattern matching for quantum circuits.

4.1. Related work

1. RETE networks have been shown to have exponential worst-case space (and thus time) complexity [262], although performance in practical use cases can vary widely [309]. ↩

Our proposed solution can be seen as a specialisation of RETE networks [142, 313] and derivatives [171, 13, 230] to the case of graph pattern matching. The additional structure obtained from restricting our considerations to graphs results in a simplified network design that allows us to derive worst-case asymptotic runtime and space bounds that are polynomial in the parameters relevant to our use case¹ – overcoming a key limitation of RETE.

Another well-studied application of large-scale pattern matching is in the context of stochastic biomolecular simulations [297, 19], particularly the Kappa project [105]. Stochastic simulations depend on performing many rounds of fast pattern-matching for continuous Monte Carlo simulations [334]. However, unlike our use case, the procedure typically does not need to scale well to a large number of patterns. In [104], Danos et al. introduced a pre-computation step to accelerate matching by establishing relations between patterns that activate or inhibit further patterns. This idea was later expanded upon and formalised in categorical language in [63]. The ideas presented in [63] are similar to ours; their formalism has the advantage of being more general but does not present any asymptotic complexity bounds and suffers from similar worst-case complexities as RETE.

A similar problem has also been studied in the context of multiple-query optimisation for database queries [290, 266], but it has limited itself to developing caching strategies and search heuristics for specific use cases. Finally, using a pre-compiled data structure for pattern matching was already proposed in [228].

However, with a $n^{\Theta(m)}$ space complexity – n is the input size and m the pattern size – it does not scale to large input graphs, even for small patterns.

4.2. Preliminaries and simplifying assumptions

For simplicity, we will throughout consider minIR graphs that admit a type system Σ , though most of the results can also be adapted to other graph domains. We will write T for the types of Σ (i.e. its values) and Γ for the operation types (i.e. its edges).

Linear paths and operation splitting

An operation type $\nu \in \Gamma$ in the type system Σ is a hyperedge. Its endpoints

$$\text{src}(\nu) = \text{src}_1(\nu) \cdots \text{src}_s(\nu) \text{ and } \text{tgt}(\nu) = \text{tgt}_1(\nu) \cdots \text{tgt}_t(\nu)$$

are strings of data types that define the input and output signature of the operation ν . We can refer to the set of all hyperedge endpoints of ν using the string indices $\text{Idx}(\cdot)$ ($\sqcup$ denotes the disjoint set union):

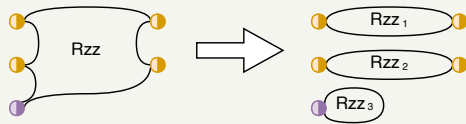
$$P_\nu = \text{Idx}(\text{src}(\nu)) \sqcup \text{Idx}(\text{tgt}(\nu)) = \{1, \dots, s\} \sqcup \{1, \dots, t\}.$$

Fix a partition of P_ν into disjoint pairs

$$P_\nu = \{p_1, p'_1\} \sqcup \{p_2, p'_2\} \sqcup \cdots,$$

where the last set of the partition may be a singleton if $|P_\nu|$ is odd. For every $\nu \in \Gamma$, we then define $f = \lceil |P_\nu|/2 \rceil$ new *split operation types* $\nu_1, \dots, \nu_f$, each with two endpoints: the i -th operation type ν_i has endpoints p_i and p'_i in P_ν . For every operation $o \in E$ of type ν , we can then *split* o into f operations $o_1, \dots, o_f$, each of arity 1 or 2 and of types $\nu_1, \dots, \nu_f$ respectively. We will refer to the graph transformation that replaces an operation o in a minIR graph with the operations o_i for $1 \leq i \leq f$ as *operation splitting*.

It is important to note that the splitting of an operation o is *unique* and given by the type of o , and thus invariant under (typed) morphisms: there is a morphism $P \rightarrow G$ of a pattern into a graph if and only if there is a morphism $P' \rightarrow G'$ from the split pattern P' into the split graph G' .



A transformation rule splitting an operation with 3 sources and 2 targets. The choice of endpoint partition made here, obtained by pairing the i -th use with the i -th define, is arbitrary but convenient for quantum gates as they correspond to the input and output values of a same qubit.

The endpoint partitions P_v also define *linear paths*. Two values v, v' in a minIR graph are on the same linear path if there are values $u_1, \dots, u_k$ with $v = u_1$ and $v' = u_k$ such that u_i is connected to u_{i+1} through an operation o and they correspond to the same pair of endpoints in the endpoints partition (i.e. the indices of $P_{type(o)}$ correspond to values u_i and u_{i+1} in $src(o) \sqcup tgt(o)$).

Linearity assumption and rigidity

1. This restriction is necessary for our results: copyable values may admit an arbitrary number of adjacent hyperedges. As a result, minIR graph pattern matching with copyable values is a generalisation of the subgraph isomorphism problem, a well-known NP-complete problem [89]. The approach generalises to non-linear types, but the complexity analysis no longer holds (we pay a computational price for every non-linear value matched). ↩

Recall that in Definition 3.2, V_{src} and V_{tgt} refer to the subset of values that are within the domain of definition of src and tgt respectively. For this chapter, we will assume $V_{src} = V_{tgt} = V$; in other words, minIR graphs are IO-free (this is w.l.o.g., see discussion after Proposition 3.4) and all values are linear¹ (this is definitely *not* w.l.o.g.!).

As a result of this assumption, the subcategory of minIR graphs that we consider forms a *rigid* category, as introduced by Danos et al. [106]. The definition, which we reproduce here, is given in terms of morphisms that *intersect all components* of the codomain. We refer to [106] for the precise definition of that notion: in the context of linear-valued minIR graphs, this is equivalent to requiring that the image of the graph homomorphism intersects every connected component of the codomain.

Definition 4.1 Rigid Category

A category $\mathbb{C}$ is rigid if for all morphisms $A \xrightarrow{h} B$ in $\mathbb{C}$ that intersects all components of B and for all $A \xrightarrow{g} C$ that factorises as $g = f \circ h$, then $B \xrightarrow{f} C$ is unique.

In other words, there is a unique way to extend a morphism $A \xrightarrow{g} C$ to a morphism $B \xrightarrow{f} C$, if it exists. If we interpret A and B as graph patterns that we are interested in matching with $A \subseteq B$, then rigidity guarantees that there is (at most) a unique way to extend a match morphism $A \rightarrow G$ into a match morphism on the larger pattern $B \rightarrow G$.

The linearity assumption also has other useful consequences. Every linear value has exactly one use and one definition. As a result, all linear paths are disjoint and form a partition of the values of the graph. They correspond to the paths that form the connected components of the *fully split graph*, i.e. the graph obtained by splitting every operation. We call the number of linear paths (and hence the number of connected components in the fully split graph) the *circuit width*, written $width(G)$. We also use the linear path decomposition to define *circuit depth*, written $depth(G)$, as the longest linear path in G .

As discussed in section 3.4, minIR rewrites are instantiated from transformation rules by minIR match morphisms $m : P \rightarrow G$. Restricting our considerations to linear-valued minIR graphs has the further implication that all such match morphisms m will be injective. We call m an *embedding* and write it using greek letters and a hooked arrow

$$m = \varphi : P \hookrightarrow G.$$

Finding such embeddings $P \hookrightarrow G$ is the *pattern matching problem* that we are solving. This problem is equivalent to finding minIR subgraphs $H \subseteq G$ of G such that H is isomorphic to the pattern $P \simeq H$.

Convexity

According to Proposition 3.6, a necessary condition for a subgraph H to define a valid minIR rewrite is *convexity*. In this chapter we weaken this requirement and propose a condition based on circuit width:

Proposition 4.1 *Necessary condition for convexity*

Let $\varphi : P \hookrightarrow G$ be an embedding of a pattern P into a linear-valued minIR graph G such that $\varphi(P)$ is a convex subgraph of G . Then for every subgraph $H \subseteq G$ such that $\varphi(P) \subseteq H$, it holds that $\text{width}(P) \leq \text{width}(H)$.

Proof. Up to isomorphism, we can assume $P \subseteq G$. Suppose there is $H \subseteq G$ such that $P \subseteq H$ and $\text{width}(P) > \text{width}(H)$. Let $\mathcal{L}_P, \mathcal{L}_H \subseteq \mathcal{P}(V)$ be partitions of V_P and V_H into sets of values that are on the same linear path of P and H respectively. It must hold that for all $\ell \in \mathcal{L}_P$ there is $\ell' \in \mathcal{L}_H$ such that $\ell \subseteq \ell'$, because $P \subseteq H$ and operation splitting is preserved under embeddings. As the map from $\mathcal{L}_P$ to $\mathcal{L}_H$ cannot be injective, there must be $\ell_1, \ell_2 \in \mathcal{L}_P$ and $\ell' \in \mathcal{L}_H$, such that $\ell_1 \neq \ell_2$ and $\ell_1, \ell_2 \subseteq \ell'$. We conclude that there must be a path in the fully split graph of H between a value of ℓ_1 and a value of ℓ_2 that is not in the fully split graph of P . Given that P is convex, this path must be in P , which contradicts the preservation of operation splitting under embeddings.

In this chapter, whenever we define a subgraph $H \subseteq G$ of a graph G , we will assume that H satisfies the above weakened convexity condition.

The converse of Proposition 4.1, however, is not true. The pattern-matching technique presented below will find a strict superset of convex embeddings. To restrict considerations to convex embeddings, it suffices to filter out the non-convex ones in a post-processing step.

Ignoring minIR Hierarchy

So far, we have omitted discussing one part of the minIR structure: the nested hierarchy of operations. Syntactically, the hierarchy formed by *parent* relations between minIR operations can be viewed as just another value type that operations are incident to: parent operations define an additional output that children operations consume as additional input. Because of the bijectivity requirement of minIR morphisms on parent-child relations of Definition ., these parent-child relat-

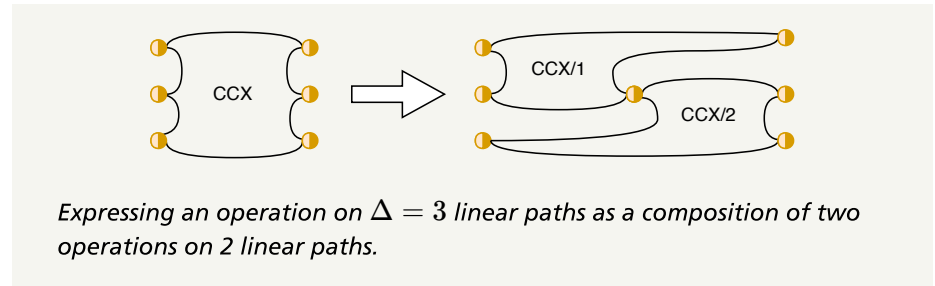
ions behave, in fact, like linear values – and hence do not violate the linearity assumption we have imposed.

However, by treating them as such, we have further weakened the constraints on pattern embeddings. We do not enforce that boundary values must be in the same regions or that parent-child relations cannot be boundary values. Similarly to convexity, we defer checking these properties to a post-processing step.

Further assumptions (harmless)

We will further simplify the problem by making presentation choices that do not imply any loss of generality. First of all, we assume that all patterns have the same width w and depth d , are connected graphs and have at least 2 operations. These conditions can always be fulfilled by adding “dummy” operations if necessary. Embeddings of disconnected patterns can be computed one connected component at a time.

We will further assume that all operations are on at most two linear paths (and thus in particular, have at most 4 endpoints). Operations on $\Delta > 2$ linear paths can always be broken up into a composition of $\Delta - 1$ operations, each on two linear paths as follows:



This transformation leaves circuit width unchanged but may multiply the graph depth by up to a factor Δ .

We furthermore define the set of all *port labels*

$$P_{all} = \bigcup_{\nu \in \Gamma} P_{\nu}$$

2. Remark that the ordering of the operations thus defined is a particular case of a depth-first search (DFS) ordering of the graph: given an operation o that has been visited, all its descendants will be visited before proceeding to any other operation. ↩

so that we can associate every operation endpoint in a minIR graph G with a *port label* from the set P_{all} . We further endow the labels P_{all} with a total order (for instance, based on the string index values). The total order on P_{all} then induces a total order on the paths $v_1 \cdots v_k \in V^*$ in G that start in the same value v_1 : the paths are equivalently described by the sequence of port labels of the operations traversed. These form strings in P_{all}^* , which we order lexicographically. Given a root value r , for every value v in G there is thus a unique smallest path from r to v in G^2 . This path is invariant under isomorphism of the underlying graph (i.e. relabelling of the values and operations but preserving the port labels). With this we conclude the discussions of the specificities of minIR graphs related to typing, linearity and hierarchy, and the related assumptions that we are making.

To summarise, minIR graphs as they are considered in this chapter are hypergraphs (Definition 3.1) that satisfy the following properties

- every vertex (value) is incident to exactly two hyperedges (operations). It is the target of one hyperedge (its *definition*) and the source of another one (its *use*),
- every hyperedge is incident to at most four vertices,
- every hyperedge can be *split* in a unique way (and invariant under isomorphism) into at most two split operations, with each at most two endpoints.

When modelling subgraphs of IO-free minIR graphs (typically patterns for pattern matching), some hyperedge connections at the boundary of the subgraph will be missing. We say a value is *open* if a use or define operation is missing (i.e. it is a boundary value in a minIR subgraph).

We will simply refer to hypergraphs that satisfy the above assumptions as *graphs*. In the unique instance of this chapter where a graph that does not satisfy this construction is referred to, we will specifically call it a *simple graph*.

We conclude with the following notable bound on circuit width.

Proposition 4.2 *Bound on circuit width*

Let G be a graph with n_{odd} operations of odd arity (i.e. $|def(o) + use(o)|$ is odd) and n_{ω} open values. Then, the circuit width of G is

$$width(G) = \lfloor (n_{\text{odd}} + n_{\omega})/2 \rfloor.$$

Proof. For any linear path $P \subseteq V^*$ in G consider its two ends v_1 and v_2 , i.e. the two values in P with only one neighbouring value in P (by definition linear paths cannot be empty). In the fully split graph of G , these values are either open or must be connected to two operations. In the latter case, at least one of the operations must have a single endpoint (otherwise by acyclicity, the operation would have two neighbours).

In a fully split graph, operations with a single endpoint result from a split operation with an odd number of endpoints. We conclude that for every linear path, there are either two operations with an odd number of endpoints in G , or one such operation and one open value, or two open values. The result follows.

4.3. Tree reduction

We reduce the problem of graph pattern matching to matching on rooted trees – as we will see in section 4.6, a much simpler problem to solve. The map between *graphs* and (rooted) *trees* is given by *rooted dual trees*. Call G *tree-like* if G is connected and the underlying undirected graph G_U of G is acyclic.

Definition 4.2 Rooted dual tree

Let G be a tree-like graph with operations O . Then given a root operation $r \in O$, the rooted dual tree of G rooted at r , written $\tau_r(G)$ is the tree given by

- the nodes of the tree are the operations O of G ,
- the parent and children of $o \in O$ are the operations that share a value with o in G ; the parent is the unique operation on the path from o to r ,
- the children of an operation are ordered according to the port labels.

Unlike graphs, tree nodes are identified uniquely by their path from the root. Trees isomorphic as graphs with identical root are thus considered equal.

A tree reduction using path splitting

To reduce a graph G to a tree using the rooted dual tree construction, it suffices to reduce G to a tree-like graph. The following result shows that this can always be achieved by repeatedly applying operation splitting transformations.

Proposition 4.6 Path splitting

A tree-like graph can be obtained from any connected graph G by applying operation splittings. The resulting graph is a path-split graph (PSG) of G .

Proof. Consider the undirected simple graph $\mathcal{I}$, where vertices are linear paths, and there is an edge between two linear paths for every operation that belongs to both paths. We call $\mathcal{I}$ the *interface graph* of G .

Splitting an operation o in a graph G corresponds to removing the corresponding edge in $\mathcal{I}$. On the other hand, the underlying undirected graph G_U of G has a cycle if and only if there is a cycle in $\mathcal{I}$. Indeed, a cycle in G_U cannot belong to a single linear path in G , by acyclicity of minIR graphs. There is, therefore, a cycle of operations that span multiple linear paths, thus forming a cycle in $\mathcal{I}$.

Hence, the operations to be split to turn G into a tree-like graph are given by the set of edges E^- in $\mathcal{I}$ that must be removed to obtain a spanning tree of $\mathcal{I}$.¹

1. It is a simple result from graph theory that such a set of edges always exists – it suffices to remove one edge from every cycle in the graph. ↪

As we consider typed graph, the splitting of an operation is unique; however the choice of spanning tree of $\mathcal{I}$ is not unique, and thus multiple PSGs exist for a given graph G .

If G' is a PSG of some graph G , then call an operation o of G an *anchor* operation if it is on two linear paths and it is not split in G' . The set of all anchors operations $\pi \subseteq O$ fully determines the path-split graph. We write $G^\pi = G'$ for the PSG of G obtained by anchors π .

Proposition 4.3 *Rooted dual trees are ternary*

Consider a PSG G^π of a graph G . There is a *root* operation $r \in O$ such that the rooted dual tree $\tau_r(G^\pi)$ is a ternary tree, i.e. every node of $\tau_r(G^\pi)$ has at most three children.

Proof. We have assumed in section 4.2 that every operation in G is on at most two linear paths and thus can be connected to at most four values. Each value is linear and hence connected to at most one other operation. It results that every operation in $\tau_r(G^\pi)$ has at most four neighbouring operations – one parent and three children. A tree leaf can be chosen as the root operation to ensure the root does not have four children.

We can make the path splitting transformation $G \rightarrow G^\pi$ *reversible* by separately storing the set of split operations in G^π that correspond to a single operation in G . As every operation of G can get split in at most two split operations, we can store the pairs of split operations in G^π that correspond to an operation in G in a partial map that defines weights for (a subset of) the operations O^π of G^π :

$$split : O^\pi \rightharpoonup P^*.$$

This maps a split operation o to the unique undirected path in G^π from o to the other half of the split operation.

This defines a map $\sigma_1 : (G^\pi, split) \mapsto G$, the inverse of the path splitting transformation $G \rightarrow G^\pi$.

Contracted path-split graphs

We can further simplify the structure of the data of a PSG by contracting all operations of G^π that are on a single linear path. The result is the *contracted path-split graph* (cPSG) of G , written $c(G^\pi)$.

We employ a similar trick as above to make this transformation reversible, this time by introducing weights on the *values* of $c(G^\pi)$ that store the string of operations that were contracted²

2. Because all contracted operations apply on a single, shared, linear path, they indeed form a string of operations. ↩

where V_G are the values of $c(G^\pi)$ and Γ^π are the otypes of operations in G^π , i.e. the otypes of the minIR graph G along with the otypes of the split operations. This defines a second map $\sigma_2 : (c(G^\pi), \text{contract}) \mapsto G^\pi$ that is the inverse of the path-split graph contraction transformation $c(\cdot)$. In summary, we have the composition

$$(c(G^\pi), \text{contract}, \text{split}) \xrightarrow{\sigma_2 \times \text{Id}} (G^\pi, \text{split}) \xrightarrow{\sigma_1} G.$$

Contracted PSGs are particularly useful for the study of the asymptotic complexity of the pattern matching algorithm we propose, as they have a very regular structure. This is expressed by the following proposition that further extends the statement of Proposition 4.3:

Proposition 4.4 *Contracted PSG*

Consider a PSG G^π of a graph G . There is a root operation $r \in O$ such that the rooted dual tree of the contracted PSG $\tau_r(c(G^\pi))$ is a ternary tree with $\text{width}(G) - 1$ nodes.

Proof. That the tree is ternary follows from Proposition 4.3. Every node of the tree corresponds to an operation in $c(G^\pi)$, which is on exactly two linear paths. As a result of acyclicity of the tree, a tree of k nodes spans $k + 1$ linear paths – and hence, we conclude $k = \text{width}(G) - 1$.

We conclude the construction presented in this section with the following result, expressing graph pattern matching in terms of tree equality:

Proposition 4.5 *Reduction to Tree Pattern matching*

Let P be a pattern graph and G a graph. Let P^π be a PSG of G . There is an embedding $P \hookrightarrow G$ if and only if there is $H \subseteq G$ and a PSG $H^{\pi'}$ of H such that

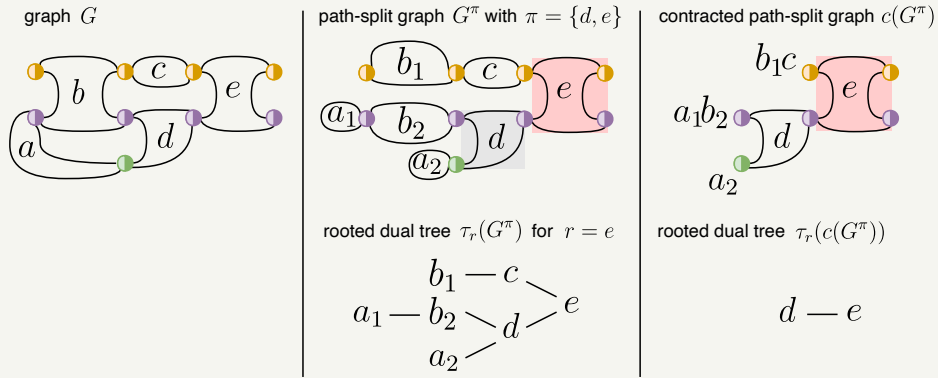
$$\tau(c(P^\pi)) = \tau(c(H^{\pi'}))$$

and the trees have equal weight maps *split* and *contract*.

The proof of this follows directly from our construction, the unicity of trees under isomorphism and the bijection between the graphs P , H and their cPSGs.

We have thus successfully reduced the problem of pattern matching to the problem of matching on trees. Given that the ordering of children of a node in a tree is fixed, checking trees for equality is a simple matter of checking node and weight equality, one node (and edge) at a time.

We conclude this section with a figure summarising the constructions we have presented.



A graph G , along with the path-split graph G^π , the contracted path-split graph $c(G^\pi)$ and their rooted dual trees. The anchor operations are d (grey) and e (red). The root of the rooted dual trees is e .

4.4. Canonicalising the tree reduction

The reduction of graph matching to ternary trees from the previous section is a big step towards an algorithm for graph matching. However, Proposition 4.5 is expressed in terms of *existence* of PSGs – it is as yet unclear how the trees can be constructed. This is the purpose of this section.

We introduce for this purpose a *canonical*, that is, invariant under isomorphism, choice of PSG G^π of G . The result is a unique canonical transformation $G \mapsto G^\pi \mapsto c(G^\pi)$ from G to a cPSG that we can use for pattern matching.

We proceed by using the total order that we have defined on port labels and can be extended lexicographically to paths outgoing from a shared root operation (see section 4.2 for more details). Whenever more than one path from r to o exist in G , it suffices to consider the smallest one. For a choice of root operation $r \in O$, we thus obtain a total order of all operations O in G .

We then restrict our attention to operations on two linear paths and consider them in order. We keep track of linear paths that have been visited and proceed as follows to determine whether $o \in O$ must be split:

- if o is on a linear path that was not seen before, it is left unchanged and the set of visited linear paths is updated;
- otherwise, i.e. o is on two linear paths that have already been visited, the operation is split, resulting in two operations on a single linear path.

The pseudocode `CanonicalPathSplit` implements this algorithm. We use `Operations(G)` to retrieve all the operations on the graph G and `LinearPaths(G, op)` to retrieve the linear paths of the operation op . The linear paths are identified using integer indices that can be pre-computed and stored in linear time in the graph size. `SplitOperation(G, op)` returns the graph resulting from splitting op into two operations on a single linear path. Finally, `PathAsPortLabels(G, root, v)` returns the string of the port labels that encode the path from $root$ to v in the graph G . The strings are ordered lexicographically.

1. The `sort_key` parameter of the `sort` function defines the total order according to which the elements are sorted, from smallest to largest. ↩

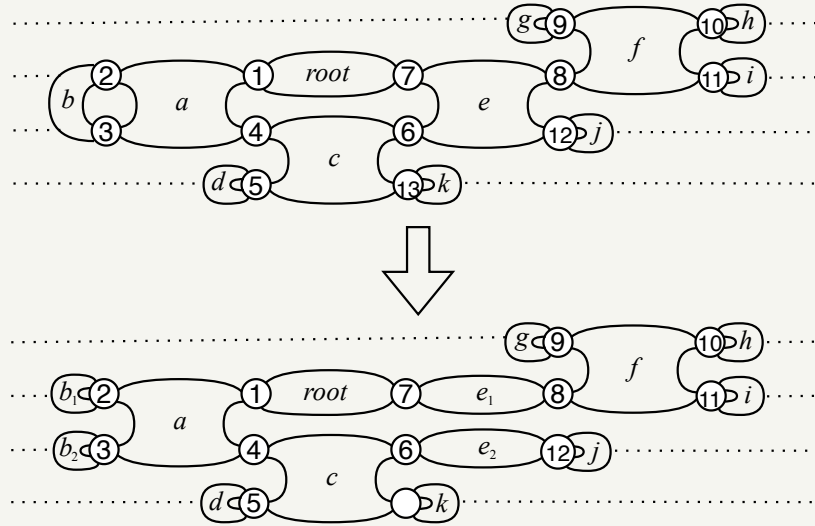
lly. The non-capitalized functions `set`, `union`, `sort`¹, `len`, and `issubset` have their standard meanings.

```

1 def CanonicalPathSplit(G: Graph, root: Operation) -> Graph:
2   new_G := G
3   all_operations := Operations(G)
4   sorted_operations := sort(
5     all_operations,
6     sort_key= lambda v: PathAsPortLabels(G, root, v)
7   )
8
9   # keep track of the visited linear paths
10  seen_paths := set()
11  for op in sorted_operations:
12    # Get the (pre-computed) indices of the linear paths
13    op_linear_paths := LinearPaths(G, op)
14    if len(op_linear_paths) == 2:
15      if issubset(op_linear_paths, seen_paths):
16        # The two linear paths of `op` are already visited
17        new_G = SplitOperation(new_G, op)
18      else:
19        # Mark the new linear paths as visited
20        seen_paths = union(seen_paths, op_linear_paths)
21  return new_G

```

The following figure shows an example of splitting a graph into its canonical PSG using `CanonicalPathSplit`.



Splitting a graph into its canonical PSG. Ports are ordered counter-clockwise on each edge, and numbered according to the lexicographic order of the paths from root to the port, as returned by `PathAsPortLabels`. This induces an order on the hyperedges, reflected in the alphabetic order of the edge labels. Linear paths are formed by ports in a horizontal line (as marked by the dotted lines). Vertex root is chosen as the root of the canonical splitting. Vertices d and g are not split because they are the smallest edges that contain the fourth, respectively first linear path.

Proposition 4.7 *Correctness of `CanonicalPathSplit`*

For a graph G , the graph returned by `CanonicalPathSplit( $G$ )` is a valid PSG of G . It is deterministic and invariant under isomorphism of G . The runtime of `CanonicalPathSplit` is $O(|G|)$, where $|G|$ is the number of operations in the graph G .

Proof. Let G^π be the graph returned by `CanonicalPathSplit( $G$ )`. From the discussion in the proof of Proposition 4.6, we know it is sufficient to show that the interaction graph $\mathcal{I}$ of G^π is acyclic and connected.

$\mathcal{I}$ is acyclic. If there was a cycle in $\mathcal{I}$, then there would be operations $o_0, \dots, o_{k-1}$ in G that pairwise $(o_i, o_{i+1 \bmod k})$ share a linear path. One of these operations must be considered last in the `for` loop of lines 11–20, suppose it is o_{k-1} . But every linear path of o_{k-1} is either also a linear path of o_{k-2} or a linear path of o_1 : o_{k-1} thus does not satisfy the condition on line 15, and thus cannot be in $\mathcal{I}$, a contradiction. Hence $\mathcal{I}$ is acyclic.

$\mathcal{I}$ is connected. We proceed inductively to show the following invariant for the main `for` loop (lines 11–20): for all linear paths in `seen_paths`, there is a path in $\mathcal{I}$ to a linear path of the root operation. `seen_paths` is only modified on line 20. If `op` is the root operation, then trivially there is a path from the linear paths `op_linear_paths` to a linear path of the root operation. Otherwise, we claim that there must be one of the paths in `op_linear_paths` that is already in `seen_paths`. From there it follows that there is a path in $\mathcal{I}$ from the root path to the unseen linear path, given by the path to the linear path in `seen_path` followed by the edge in $\mathcal{I}$ that corresponds to `op`.

By connectedness of G , there is a path from the root operation to `op`. The path is not empty because `op` is not the root operation, so we can consider the prefix of the path of all operations excluding `op`. Call `op'` the last operation preceding `op` and `op_linear_paths'` its linear paths. Two successive operations on a path must share a linear path: `op_linear_paths` $\cap$ `op_linear_paths'` cannot be empty. According to line 4, `op'` must have been visited before `op`, thus `op_linear_paths'  $\subseteq$  seen_paths`. It follows that at least one element of `op_linear_paths` must be in `seen_paths`.

Deterministic and isomorphism invariant. The pseudocode above is deterministic and only depends on paths in G encoded as strings of port labels, which are invariant under isomorphism.

Runtime complexity. Lines 2 and 3 run in $O(|G|)$ time. With the exception of the `sort` function on lines 4–7, every other line can be run in $O(1)$ time:

- lines 13 and 15 run in constant time because the size of `op_linear_paths` is always at most 2;

- line 20 (and the `in` check on line 15) can be run in constant time by representing the `seen_paths` set as a fixed-size boolean array of size w , with the i -th bit indicating whether the i -th linear path has been seen;
- line 17 is a constant time transformation if we allow in-place modification of `new_G`.

The `for` loop will run $|G|$ iterations, for a total of $O(|G|)$ runtime. Finally, the sorting operation would naively take time $O(|G| \log |G|)$. However, given that the ordering is obtained lexicographically from the paths starting at the root, we can obtain the sorted list of operations by depth-first traversal of the graph starting at the root. The result follows.

Using `CanonicalPathSplit`, we can now sketch what the pattern matching algorithm should look like. For each pattern, we first compute their canonical PSG for an arbitrary choice of pattern root operation; then, given a graph G , we can find all embeddings of patterns into G by iterating over all possible PSGs within G . Naively, this involves enumerating all possible subgraphs of G , and then for each of them, iterating over all possible root choices.

2. Think for example of the same root operation r that is considered repeatedly for every overlapping subgraph of G that contains r . ↪

This can be significantly sped up by realising that many of the PSGs that are computed when iterating over all possible subgraphs and root choices are redundant². We will see in the next section that we can *i*) iterate once over all possible root choices r in G and *ii*) introduce a new procedure `AllPathSplits` that will efficiently enumerate all possible rooted ual trees of PSGs that are rooted in r for subgraphs within G . In the process, we will also see that we can replace the tree equality check of line 12 with a subtree inclusion check, further reducing the number of PSGs that must be considered.

Naive pattern matching.

```

1 # Precompute all PSGs
2 allT = [CanonicalPathSplit(
3     P, root_P
4 ) for (P, root_P) in patterns]
5
6 for S in Subgraphs(G):
7     for root_S in Operations(S):
8         TG = CanonicalPathSplit(
9             S, root_S
10        )
11     for T in allT:
12         if T == TG:
13             yield T

```

Improved using `AllPathSplits` (section 4.5).

```

1 # Precompute all PSGs
2 allT = [CanonicalPathSplit(
3     P, root_P
4 ) for (P, root_P) in patterns]
5
6 for root_G in Operations(G):
7     for TG in AllPathSplits(
8         G, root_G
9     ):
10         for T in allT
11             # Replace == with subtree
12             if IsSubTree(T, TG)
13                 yield T

```

4.5. Enumerating all path-split graphs

The `CanonicalPathSplit` procedure in the previous section defines for all graphs G and choice of root operation r a canonical PSG G^π , and thus a canonical set of anchors π that we write as $\pi_r(G) = \pi$.

Instead of `CanonicalPathSplit`, we can equivalently consider a `CanonicalAnchors` procedure, which computes $\pi_r(G)$ directly instead of the graph $G^{\pi_r(G)}$.

We formulate this computation below, using recursion instead of a `for` loop. This form generalises better to the `AllAnchors` procedure that we will introduce next.

The equivalence of the `CanonicalAnchors` procedure with `CanonicalPathSplit` follows from the observation made in section 4.2 that ordering operations in lexicographic order of the port labels is equivalent to a depth-first traversal of the graph.

`CanonicalAnchors` implements a recursive depth-first traversal (DFS), with the twist that the recursion is explicit only on the anchor nodes and otherwise relying on the lexicographic ordering just like in `CanonicalPathSplit`: lines 5–15 of `CanonicalAnchors` correspond to the iterations of the `for` loop (line 11–20) of `CanonicalPathSplit` until an anchor operation is found (i.e. the `else` branch on lines 18–20 is executed). From there, the graph traversal proceeds recursively.

We introduce the `ConnectedComponent`, `Neighbours` and `RemoveOperation` procedures; the first returns the connected component of the current operation, whereas the other two procedures are used to traverse, respectively modify, the graph G . Importantly, `Neighbours(G, op)` returns the neighbours of `op` ordered by port label order.

To ensure that the recursive DFS does not visit the same operation twice, we modify the graph with `RemoveOperation` on lines 11 and 15, ensuring that no visited operation remains in G . As a consequence, `CanonicalAnchors` may be called on disconnected graphs, which explains why an additional call to `ConnectedComponent` (line 4) is required.

Proposition 4.8 *Equivalence of `CanonicalPathSplit` and `CanonicalAnchors`*

Let G be a connected graph and let r be a root operation in G . Then `CanonicalAnchors` maps the graph to the canonical anchor set:

$$(G, r, \{\}) \mapsto (\pi(G)_r, L, \emptyset),$$

where L is the set of all paths in G and $\emptyset$ designates the empty graph.

The proof follows directly from the previous paragraphs.

```

1 def CanonicalAnchors(
2     G: Graph, root: Operation, seen_paths: Set[int]
3 ) -> (Set[Operation], Set[int], Graph):
4     operations = Operations(ConnectedComponent(G, root))
5     # sort by PathAsPortLabels, as previously
6     sorted_operations := sort(operations)
7     operations_queue := queue(sorted_operations)
8
9     # Skip all operations that are not anchors
10    op := operations_queue.pop() # never empty, contains root
11    G = RemoveOperation(G, op)
12    while len(LinearPaths(G, op)) == 1 or
13        issubset(LinearPaths(G, op), seen_paths):
14        op = operations_queue.pop() or return ({}, {}, G)
15        G = RemoveOperation(G, op)
16
17    # op is anchor, update seen_paths and recurse
18    seen_paths = union(seen_paths, LinearPaths(G, op))
19    anchors := [op]
20    # sort by port labels
21    for child in Neighbours(G, op):
22        (new_anchors, seen_paths, G) = CanonicalAnchors(
23            G, child, seen_paths
24        )
25        anchors += new_anchors
26
27    return (anchors, seen_paths, G)

```

Maximal PSGs

In addition to “simplifying” the data required to define path splitting, the definition of PSGs using anchor operations has another advantage that is fundamental to the pattern matching algorithm.

Consider the rooted dual tree $\tau_r(G^\pi)$ of a PSG with root operation r in G^π . Recall that tree nodes are uniquely identified by their path from the root and thus are considered equal if they are isomorphic as graphs. We can in the same way define a tree inclusion relation $\subseteq$ on rooted dual trees that corresponds to checking that the trees have the same root and that the left-hand side is isomorphic to a subtree of the right-hand side. We also require that the operation weights given by the *split* map *split* map coincide on the common subtree.

Proposition 4.9 Maximal PSG

Let G be a connected graph, π a set of operations in G and $r \in \pi$ a root operation. Consider the set $\mathcal{G}_\pi = \{H \subseteq G \mid \pi_r(H) = \pi\}$.

There is a subgraph $M \subseteq G$ such that for all subgraphs $H \in \mathcal{G}_\pi$: $H \subseteq M$. Furthermore, for all graph P , there is r' and $\pi' = \pi(P)_{r'}$ such that

$$P \simeq H \in \mathcal{G}_\pi \quad \Leftrightarrow \quad \tau_{r'}(P^{\pi'}) \subseteq \tau_r(M^\pi). \quad (2)$$

We call M^π the *maximal PSG* with anchors π in G .

The proof gives an explicit construction for M .

Proof. Assume $\mathcal{G}_X \neq \emptyset$, otherwise the statement is trivial.

Construction of M . Let L be the set of linear paths in G that go through at least one operation in π . Consider the set of operations O_L in G given by the operations whose linear paths are contained in L . This defines a subgraph $G|_{O_L}$ of G . Since $\mathcal{G}_X \neq \emptyset$, there exists $H \in \mathcal{G}_X$. By assumption, H is connected, and thus the anchors π of H are connected in H . There is therefore a connected component $M \subseteq G|_{O_L}$ that contains the set π .

Well-definedness of M^π . Consider the PSG M^π of M . We must show that M^π is a tree-like graph for the proposition statement to be well-defined. In other words, we must show that the interaction graph $\mathcal{I}$ of M^π is acyclic and connected. M is connected by construction, which implies connectedness of M^π and thus of $\mathcal{I}$. It is acyclic because $\text{width}(M) = |\pi| + 1$ and M has exactly $|\pi|$ operations on more than one linear path. $\mathcal{I}$ is thus a tree.

$H \subseteq M$. For any subgraph $H \in \mathcal{G}_X$, its operations must be contained in O_L . Since any $H \in \mathcal{G}$ is connected and contains π , it must further hold that $H \subseteq M$.

We can now prove the $\Leftrightarrow$ equivalence of (2).

$\Leftarrow$: If $\tau_{r'}(P^{\pi'}) \subseteq \tau_r(M^\pi)$, then there exists $H' \subseteq M^\pi$ with rooted dual tree

$$\tau_r(H^\pi) = \tau_r(P^{\pi'}).$$

Furthermore, by definition of $\subseteq$ on rooted trees, a *split* map is defined on H' , given by the *split* map of M^π on the domain H' . Recall from section 4.3 that there is a map σ that maps

$$(H', \text{split}) \xrightarrow{\sigma} H \quad \text{and} \quad (M^\pi, \text{split}) \xrightarrow{\sigma} M.$$

It merges split operations pairwise, and thus it is immediate that $H' \subseteq M^\pi$ implies $H \subseteq M$. Thus $H \in \mathcal{G}_X$ and $H' = H^\pi$. By construction, one can also derive that $P \simeq H$. The statement follows.

$\Rightarrow$: Since $H \in \mathcal{G}_X$, we know from point 1 that $H \subseteq M$. Thus we can define an injective embedding $\varphi : P \rightarrow M$.

Operation splitting leaves the set of values from H to H^π , as well as from M to M^π unchanged. Similarly, there is a bijection between values in H^π and M^π and thus between edges in $\tau_r(H^\pi)$ and $\tau_r(M^\pi)$. The pattern embedding φ hence defines an injective map ϕ_E from tree edges in $\tau_r(H^\pi)$ to tree edges in $\tau_r(M^\pi)$. We extend this map to a map on the trees $\phi : \tau_r(H^\pi) \rightarrow \tau_r(M^\pi)$ by induction over the nodes set of $\tau_r(H^\pi)$. We start by the root map $\phi(r) = r$. Using ϕ_E , we can then uniquely define the image of any child node of r in $\tau_r(H^\pi)$, and so forth inductively.

from the root r to v and v' respectively such that their image under ϕ_E are two paths from r to $\phi(v) = \phi(v')$. But $\tau_r(M^\pi)$ is a tree, so both paths must be equal. By bijectivity of ϕ_E , it follows $v = v'$, and thus ϕ is injective. Finally, the value and operation weights are invariant under pattern embedding and thus are preserved by definition.

This result means that instead of listing all PSGs for every possible subgraph of G , it is sufficient to proceed as follows:

1. for every pattern P , fix a root operation r_P and construct the rooted tree dual of the canonical PSG

$$\tau_P := \tau_{r_P}(P^{\pi_{r_P}(P)}).$$

2. enumerate every possible root operation r in G ,
3. enumerate every possible sets of anchors π in G with root r ,
4. for each set π , find the maximal PSG M with anchors π in G , and take its rooted tree dual $\tau_M := \tau_r(M^\pi)$,
5. find all patterns P such that $\tau_P \subseteq \tau_M$.

In other words, if `AllAnchors` is a procedure that enumerates all possible sets of anchors π in G and `MaximalPathSplit` computes the maximal PSG M as presented in the proof of Proposition 4.9, then `AllPathSplits(G)` can simply be obtained by calling `AllAnchors` and then returning their respective maximal PSGs in G :

```
def AllPathSplits(G: Graph, root: Operation) -> Set[Graph]:
  all_anchors = AllAnchors(G, root)
  return {MaximalPathSplit(G, pi) for pi in all_anchors}
```

The missing piece: `AllAnchors`

We can now complete the definition of `AllPathSplits` by defining the `AllAnchors` procedure, which enumerates all possible sets of anchors in G given a root operation r .

1. Every anchor operation is on at least one previously unseen linear path, thus there can be at most w operations in the set of anchors. ↪

The procedure is similar to `CanonicalAnchors`, described in detail in the previous paragraphs. In addition to the arguments of `CanonicalAnchors`, `AllAnchors` requires a width $w \geq 1$ argument. It then returns all sets of at most w operations¹ that form the canonical anchors of some width- w subgraph of G with root r . The main difference between `CanonicalAnchors` and `AllAnchors` is that the successive recursive calls (line 22 in `CanonicalAnchors`) are replaced by a series of nested loops (lines 42–48 in `AllAnchors`) that exhaustively iterate over the possible outcomes for different subgraphs of G . The results of every possible combination of recursive calls are then collected into a list of anchor sets, which is returned.

The part of the pseudocode that is without comments is unchanged from `CanonicalAnchors`. Using Proposition 4.3, we know that we can assume that every operation has at most 3 children, and thus 3 neighbours in G , given that the operations equivalent to parent nodes were removed.

```

1 def AllAnchors(
2     G: Graph, root: Operation, w: int,
3     seen_paths: Set[int] = {}
4 ) -> List[(Set[Operation], Set[int], Graph)]:
5     # Base case: return one empty anchor list
6     if w == 0:
7         return [({}, {}, G)]
8
9     operations = Operations(ConnectedComponent(G, root))
10    sorted_operations := sort(operations)
11    operations_queue := queue(sorted_operations)
12
13    op := operations_queue.pop()
14    G = RemoveOperation(G, op)
15    while len(LinearPaths(G, op)) == 1 or
16        issubset(LinearPaths(G, op), seen_paths):
17        op = operations_queue.pop() or return [({}, {}, G)]
18        G = RemoveOperation(G, op)
19
20    seen0 = union(seen_paths, LinearPaths(G, op))
21    # There are always at most three neighbours: we
22    # unroll the for loop of CanonicalAnchors.
23    [child1, child2, child3] = Neighbours(G, op)
24    # Iterate over all ways to split w-1 anchors over
25    # the three children and solve recursively
26    all_anchors = []
27    for 0 <= w1, w2, w3 < w with w1 + w2 + w3 == w - 1:
28        for (anchors1, seen1, G1) in
29            AllAnchors(G, child1, w1, seen0):
30            for (anchors2, seen2, G2) in
31                AllAnchors(G1, child2, w2, seen1):
32                for (anchors3, seen3, G3) in
33                    AllAnchors(G2, child3, w3, seen2):
34                    # Concatenate new anchor with anchors from all paths
35                    anchors = union([op], anchors1, anchors2, anchors3)
36                    all_anchors.push((anchors, seen3, G3))
37    return all_anchors

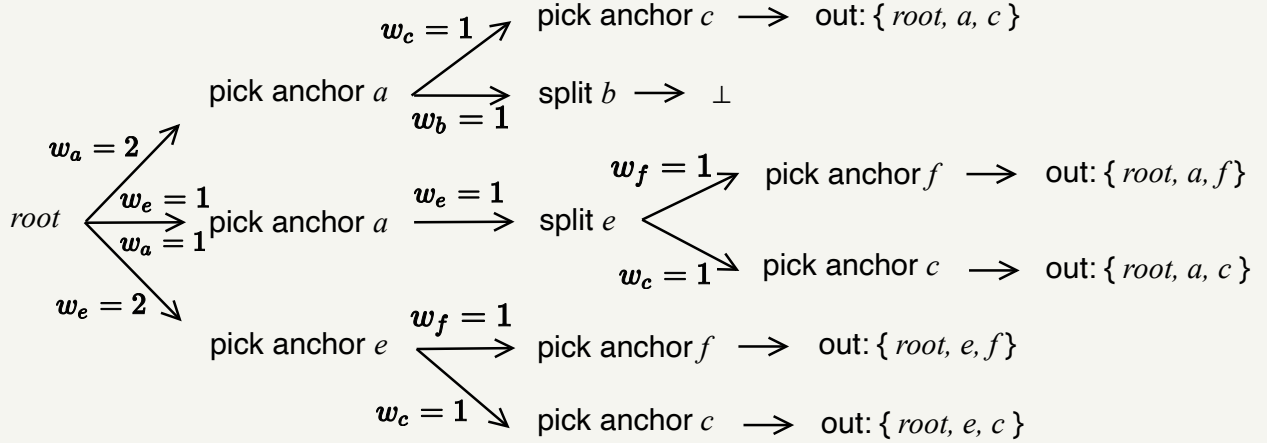
```

We can represent the sequence of recursive calls to `AllAnchors` as a tree. The call tree for the graph used as example to illustrate `CanonicalAnchors` earlier is given on the next page.

We now show correctness of the procedure. Let us write $\Pi_r^w(G)$ for the set of sets of anchors returned by `AllAnchors(G, r, w, {})`.

Proposition 4.11 *Correctness of AllAnchors*

Let G be a graph and $H \subseteq G$ be a subgraph of G of width w . Let r be a choice of root operation in H . We have $\pi_r(H) \subseteq \Pi_r^w(G)$.



A call tree for an execution of `AllAnchors` on the example graph of the previous figure with $w = 3$. Starting from the root, each node in the tree corresponds to either picking an operation as anchor or not (thus splitting it). Edges are labelled by the values assigned to w for the respective children of the source node. One path from root to leaf leads to no solution (it is impossible to find an unseen linear path from operation b). The other paths each lead to a valid set of three anchors.

The proof is by induction over the width w of the subgraph H . The idea is to map every recursive call in `CanonicalAnchors` to one of the calls to `AllAnchors` on lines 29, 31 or 33. All recursive results are concatenated on line 36, and thus, the value returned by `CanonicalAnchors` will be one of the anchor sets in the list returned by `AllAnchors`.

Proof. Let $H \subseteq G$ be a connected subgraph of G of width w . We prove inductively over w that if $(X, S', H') = \text{CanonicalAnchors}(H, r, S)$ then there is a graph G' such that $H' \subseteq G' \subseteq G$ such that

$$(X, S', G') \in \text{AllAnchors}(G, r, w, S)$$

for all valid root operations r of H and all subsets of the linear paths of H in `seen_paths`. The statement in the proposition directly follows this claim.

For the base case $w = 1$, `CanonicalAnchors` will return the anchors `anchors = [op]` as defined on line 19: there is only one linear path, and it is already in `seen_paths`, thus for every recursive call to `CanonicalAnchors`, the `while` condition on line 12 will always be satisfied until all operations have been exhausted and empty sets are returned. In `AllAnchors`, on the other hand, The only values of `w1`, `w2` and `w3` that satisfy the loop condition on line 27 for $w = 1$ are `w1 = w2 = w3 = 0`. As a result, given the `w = 0` base case on lines 6–7, the lines 35 and 36 of `AllAnchors` are only executed once, and the definition of `anchors` on line 36 is equivalent to its definition in `CanonicalAnchors`.

We now prove the claim for $w > 1$ by induction. As documented in `AllAnchors`, we can assume that every operation has at most 3 children. This simplifies the

loop on lines 21–25 of `CanonicalAnchors` to, at most, three calls to `CanonicalAnchors`.

Consider a call to `CanonicalAnchors` for a graph $H \subseteq G$, a root operation r in H and a set S of linear paths. Let w_a , w_b and w_c be the length of the values returned by the three recursive calls to `CanonicalAnchors` of line 22 for the execution of `CanonicalAnchors` with arguments H , r and S . Let c_a , c_b and c_c be the three neighbours of r in H . If the child c_x does not exist, then one can set $w_x = 0$ and it can be ignored – the argument below still holds in that case. The definition of `seen0` on line 20 in `AllAnchors` coincides with the update to the variable `seen_paths` on line 18 of `CanonicalAnchors`; similarly, the updates to `G` on lines 14 and 18 of `AllAnchors` are identical to the lines 11 and 15 of

`CanonicalAnchors` that update H . Let the updated `seen_paths` be the set S_a , the updated `G` be G_a and the updated H be H_a , with $H_a \subseteq G_a$.

As every anchor operation reduces the number of unseen linear paths by exactly one (using the simplifying assumptions of section 4.2), it must hold that $w_a + w_b + w_c + 1 = w$. Thus, for a call to `AllAnchors` with the arguments G , r , w and S , there is an iteration of the `for` loop on line 27 of `AllAnchors` such that $w1 = w_a$, $w2 = w_b$ and $w3 = w_c$. It follows that on line 29 of `AllAnchors`, the procedure is called recursively with arguments (G_a, c_a, w_a, S_a) . From the induction hypothesis, we obtain that there is an iteration of the `for` loop on line 29 in which the values of `anchors1` and `seen1` coincide with the values of the `new_anchors` and `seen_paths` variables after the first iteration of the `for` loop on line 21 of `CanonicalAnchors`. Call the value of `seen1` (and `seen_paths`) S_b . Similarly, call the updated value of `G` in `AllAnchors` G_b and the updated value of `G` in `CanonicalAnchors` H_b . We have, by the induction hypothesis, that $H_b \subseteq G_b$.

Repeating the argument, we obtain that there are iterations of the `for` loops on lines 30 and 32 of `AllAnchors` that correspond to the second and third recursive calls to `CanonicalAnchors` on line 22 of the procedure. Finally, the concatenation of anchor lists on line 36 of `AllAnchors` is equivalent to the repeated concatenations on line 25 of `CanonicalAnchors`, and so we conclude that the induction hypothesis holds for w .

We will see that the overall runtime complexity of `AllAnchors` can be easily derived from a bound on the size of the returned list. For this, we use the following result:

Proposition 4.10 *Number of anchor sets in `AllAnchors`*

For a graph G , a root operation r in G and $1 \leq w \leq \text{width}(G)$, the length of the list `AllAnchors` (G, r, w) is in $O(c^w \cdot w^{-3/2})$, where $c = 6.75$ is a constant.

Proof. Let C_w be an upper bound for the length of the list returned by a call to `AllAnchors` for width w . For the base case $w = 0$, $C_0 = 1$. The returned `all_anchors` list is obtained by pushing anchor lists one by one on line 36. We

can count the number of times this line is executed by multiplying the length of the lists returned by the recursive calls on lines 28–32, giving us the recursion relation

$$C_w \leq \sum_{\substack{0 \leq w_1, w_2, w_3 \leq w \\ w_1 + w_2 + w_3 = w-1}} C_{w_1} \cdot C_{w_2} \cdot C_{w_3}.$$

Since C_w is meant to be an upper bound, we replace $\leq$ with equality above to obtain a recurrence relation for C_w . This recurrence relation is a generalisation of the well-known Catalan numbers [299], equivalent to counting the number of ternary trees with w internal nodes: a ternary tree with $w \geq 1$ internal nodes is made of a root along with three subtrees with w_1, w_2 and w_3 internal nodes respectively, with $w_1 + w_2 + w_3 = w - 1$. A closed form solution to this problem can be found in [17]:

$$C_w = \frac{\binom{3w}{w}}{2w+1} = \Theta\left(\frac{c^w}{w^{3/2}}\right)$$

satisfying the above recurrence relation with equality, where $c = 27/4 = 6.75$ is a constant obtained from the Stirling approximation:

$$\begin{aligned} \binom{3w}{w} &= \frac{(3w)!}{(2w)!w!} = \Theta\left(\frac{1}{\sqrt{w}}\right) \left(\frac{(3w)^3}{e^3}\right)^w \left(\frac{e^2}{(2w)^2}\right)^w \left(\frac{e}{w}\right)^w \\ &= \Theta\left(\frac{(27/4)^w}{w^{1/2}}\right). \end{aligned}$$

To obtain a runtime bound for `AllAnchors`, it is useful to identify how much of G needs to be traversed. If we suppose all patterns have at most depth d , then it immediately follows that any operation in G that is in the image of a pattern embedding must be at most a distance d away from an anchor operation. We can thus equivalently call `AllAnchors` on a subgraph of G such that no linear path is longer than $2d$. We thus obtain the following runtime.

Proposition 4.12 *Runtime of `AllAnchors`*

For patterns with at most width w and depth d , the total runtime of `AllAnchors` is in

$$O\left(\frac{c^w \cdot d}{w^{1/2}}\right).$$

Proof. We restrict `Operations` on line 9 to only return the first d operations on the linear path in each direction, starting at the anchor operation: operations more than distance d away from the anchor cannot be part of a pattern of depth d .

We use the bound on the length of the list returned by calls to `AllAnchors` of Proposition 4.10 to bound the runtime. We can ignore the non-constant runtime of the concatenation of the outputs of recursive calls on line 35, as the total size of the outputs is asymptotically at worst of the same complexity as the runtime of the recursive calls themselves. Excluding the recursive calls, the only remaining lines of `AllAnchors` that are not executed in constant time are the `while` loop on lines 15–18 and the `Operations` and `sort` calls on lines 9–11. Using the same argument as in `CanonicalAnchors`, we can ignore the latter two calls by replacing the queue of operations by a lazy iterator of operations. The next operation given `op` and the graph `G` can always be computed in $O(1)$ time using a depth-first traversal of `G`.

Consider the recursion tree of `AllAnchors`, i.e. the tree in which the nodes are the recursive calls to `AllAnchors` and the children are the executions spawned by the nested `for` loops on line 28–32. This tree has at most

$$C_w = \Theta\left(\frac{c^w}{w^{3/2}}\right)$$

leaves. A path from the root to a leaf corresponds to a stack of recursive calls to `AllAnchors`. Along this recursion path, `seen_paths` set is always strictly growing (line 35) and the operations removed from `G` on lines 14 and 18 are all distinct. For each linear path, at most $2d$ operations are traversed. Thus the total runtime of the `while` loop (lines 15–18) along a path from root to leaf in the recursion tree is in $O(w \cdot d)$. We can thus bound the overall complexity of executing the entire recursion tree by $O(C_w \cdot w \cdot d) = O(\frac{c^w \cdot d}{w^{1/2}})$.

4.6. An automaton for multi-pattern matching

We have shown in the previous sections that graph pattern-matching can be reduced to a problem of tree inclusions, with trees of fixed width w . To complete the pattern-matching algorithm, we must provide a fast way to evaluate the subtree relation for many trees representing the set of all patterns we wish to match.

More precisely, for patterns $P_1, \dots, P_\ell$ with width w , fix a root operation r_i in P_i for each $1 \leq i \leq \ell$ and consider the rooted tree duals of the canonical PSGs $\tau_{r_i}(P_i^{\pi_i})$, with $\pi_i = \pi_{r_i}(P_i)$ the canonical anchors. Then given a subject graph G , we wish to compute the set

$$\{1 \leq i \leq \ell \mid \tau_{r_i}(P_i^{\pi_i}) \subseteq \tau_r(G^\pi)\}, \quad (2)$$

for all anchor sets $\pi \in \Pi_r^w(G)$ and root operation r in G . This corresponds to the `IsSubTree` predicate introduced in the sketch of the algorithm in section 4.4.

Instead of considering the trees of PSGs, it will prove easier to consider the *contracted* PSGs (cPSGs)

$$\tau_{r_i}(c(P_i^{\pi_i})) \quad \text{and} \quad \tau_r(c(G^\pi)).$$

section 4.3).

It will be useful to remind ourselves the following properties of contracted PSGs. Every operation of a cPSG (and thus every node in its rooted dual tree) is an anchor operation of the PSG. Per Proposition 4.4, the rooted dual tree of a cPSG is a ternary tree and has exactly $\text{width}(G) - 1$ nodes. Finally, recall the concept of an open value of a graph, i.e. a value that is missing either a use or define operation (see section 4.2).

Reduction of tree inclusion to string prefix matching

Now consider two contracted spanning tree reductions $c(G_1^{\pi_1})$ and $c(G_2^{\pi_2})$ with values V_1 and V_2 . To simplify notation, define

$$\tau_1 = \tau_{r_1}(c(G_1^{\pi_1})) \quad \text{and} \quad \tau_2 = \tau_{r_2}(c(G_2^{\pi_2}))$$

for some choice of root operations r_1 and r_2 in G_1 and G_2 , respectively. We lift the $\subseteq$ relation on rooted dual trees of PSGs introduced in section 4.5 to rooted dual trees of cPSGs in such a way that there is an inclusion relation between two rooted dual trees of PSGs if and only if the same relation holds on the rooted duals of cPSGs.

We say that $\tau_1 \subseteq \tau_2$ if and only if

- the trees share the same root operation,
- τ_1 is a subtree of τ_2 ,
- the *spilt* map coincides on the common subtree, and
- the *contract* map satisfies for all $v \in V_1$:

$$\begin{cases} \text{contract}(v) \subseteq \text{contract}(f(v)) & \text{if } v \text{ is an open value,} \\ \text{contract}(v) = \text{contract}(f(v)) & \text{otherwise,} \end{cases}$$

where $f : V_1 \hookrightarrow V_2$ designates the embedding of V_1 into V_2 given by the tree embedding.

The first three conditions are taken as-is from the $\subseteq$ relation on non-contracted trees, whilst the fourth condition on the *contract* map is specific to contracted trees.

1. The values can be ordered as usual by using the total lexicographic order on port labels of the tree. ↩

Using Proposition 4.2, there are at most 2 open values for each linear path in the graph, and thus at most $2 \cdot w$ open values in a rooted dual tree of a cPSG of width w . For each such contracted rooted dual, we can thus define a *contracted string tuple* $S = (s_1, \dots, s_{2w}) \in (O^*)^{2w}$ given by the values of the *contract* map evaluated in the (up to) $2w$ open values¹.

If $\text{contract}|_C$ is the restriction of *contract* to the domain of definition of non-open values of a cPSG, the fourth condition for the inclusion relation $\subseteq$ on rooted dual cPSGs, given above becomes an equality condition when restricted to non-open values. A special case of this property of particular interest to us is stated as

the following result. The $\subseteq$ relation on strings refers to prefix inclusion, i.e. $s \subseteq t$ if and only if s is a prefix of t .

Proposition 4.14 *Inclusion of equal-width trees*

Let $S = (s_1, \dots, s_{2w})$ and $T = (t_1, \dots, t_{2w}) \in (O^*)^{2w}$ be the contracted string tuples of τ_1 and τ_2 respectively. Then $\tau_1 \subseteq \tau_2$ if and only if the trees share the same root, are isomorphic, have the same *split* and *contract* $|_C$ maps and for all $i \in \{1, \dots, 2w\}$: $s_i \subseteq t_i$.

The proof of this follows directly from observing that rooted duals of cPSGs have the same set of nodes and that the restriction to non-open values *contract* $|_C$ must satisfy equality.

Why restricting ourselves to trees of the same width w ? It is sufficient for our purposes! All patterns are of width w by assumption and so are the rooted dual trees of the form $\tau_r(G^\pi)$, given that $\pi \in \Pi_r^w(G)$.

The string prefix matching problem is a simple computational task that can be generalised to check for multiple string patterns at the same time using a prefix tree. An overview of this problem can be found in appendix A. We can thus obtain a solution for the pattern matching problem for ℓ patterns:

Proposition 4.15 *Fixed anchor pattern matching*

As above, let

- G be a graph, with $\pi \in \Pi_r^w(G)$ a set of $w - 1$ operations and $r \in \pi$ a choice of root operation,
- $P_1, \dots, P_\ell$ be patterns of width w and depth d , with choices of root operations $r_1, \dots, r_\ell$ and canonical anchors $\pi_i = \pi_{r_i}(P_i)$.

The set of all pattern embeddings mapping the canonical anchor set π_i to π and root r_i to r for $1 \leq i \leq \ell$ can be computed in time $O(w \cdot d)$ using at most ℓ pre-computed prefix tree of size at most $(\ell \cdot d)^w + 1$, each constructed in time complexity $O((\ell \cdot d)^w)$.

Proof. For each pattern, we consider its canonical spanning tree reduction and construct a *multi-dimensional prefix tree* (see Appendix) for each group of patterns that share the same spanning tree reduction.

Given a graph G , we can compute the cPSG of G for anchors π and map its rooted dual tree to the corresponding prefix tree. This can be done in $O(|T_G|)$ time by using a search tree. We can restrict G^π to a graph of size $O(w \cdot d)$ by

truncating the linear paths to at most $2d$ length, as in the proof of Proposition 4.12. Thus we can assume $|G^\pi| \in O(w \cdot d)$.

The rest of the proof and the runtime follow from the multi-dimensional prefix tree construction detailed in Appendix).

Combining everything

Finally, putting Proposition 4.15 and Proposition 4.12 together, we obtain our main result.

Proposition 4.13 *Pattern matching*

Let $P_1, \dots, P_\ell$ be patterns with width w and depth d . The pre-computation runs in time and space complexity

$$O((d \cdot \ell)^w \cdot \ell + \ell \cdot w \cdot d).$$

For any subject graph G , the pre-computed prefix tree can be used to find all pattern embeddings $P_i \rightarrow G$ in time

$$O\left(|G| \cdot \frac{c^w}{w^{1/2}} \cdot d\right)$$

where $c = 6.75$ is a constant.

Proof. The pre-computation consists of running the `CanonicalAnchors` procedure on each of the ℓ patterns and then transforming them into a map of prefix trees using Proposition 4.15. By Proposition 4.7, `CanonicalAnchors` runs in $O(w \cdot d)$ for each pattern, where we used that $|P_i| \leq w \cdot d$ for all patterns. The total runtime of prefix construction is thus

$$O((d \cdot \ell)^w \cdot \ell + \ell \cdot w \cdot d).$$

The complexity of pattern matching itself on the other hand is composed of two parts: the computation of all possible anchor sets $\Pi_r^w(G)$, and the execution of the prefix string matcher for each of the trees resulting from these sets $\pi \in \Pi_r^w(G)$. As `AllAnchors` must be run for every choice of root vertex r in G , the runtime is thus obtained by multiplying *i*) $|G|$ with *ii*) the runtime of the prefix tree matching (Proposition 4.15), and with *iii*) $|\Pi_r^w(G)|$, i.e. the number of anchor lists returned by `AllAnchors` (Proposition 4.10):

$$O(|G| \cdot w \cdot d \cdot C_w),$$

where C_w is the bound for the number of anchor lists returned by `AllAnchors`. The result follows.

4.7. Benchmarks

Proposition 4.13 shows that pattern-independent matching can scale to large datasets of patterns but imposes some restrictions on the patterns and embeddings that can be matched. In this section, we discuss these limitations and give empirical evidence that the pattern-matching approach we have presented can be used on a large scale and outperform existing solutions.

Pattern limitations

In section 4.2, we imposed conditions on the pattern embeddings to obtain a complexity bound for pattern-independent matching. We argued how these restrictions are natural for applications in quantum computing, and most of the arguments will also hold for a much broader class of computation graphs.

In future work, it would nonetheless be of theoretical interest to explore the importance of these assumptions and their impact on the complexity of the problem. As a first step towards a generalisation, our implementation and all our benchmarks in this section do not make any of these simplifying assumptions. Our results below give empirical evidence that a significant performance advantage can be obtained regardless.

Implementation

We provide an open-source implementation in Rust of pattern independent matching using the results of this chapter, available on [GitHub](#). The code and datasets used for the benchmarks themselves are available in a [dedicated repository](#).

The implementation works for weighted or unweighted port graphs – of which typed minIR graphs are a special case – and makes none of the simplifying assumptions employed in the theoretical analysis. Pattern matching proceeds in two phases: precomputation and runtime.

Precomputation. In a first step, all graph patterns are processed and compiled into a single state automaton that will be used at *runtime* for fast pattern independent matching. The automaton in the implementation combines in one data structure two distinct computations of this chapter:

- the recursive branching logic used in the `AllAnchors` procedure to enumerate all possible choices of anchors.
- the automaton described in section 4.6 that matches patterns for a fixed set of anchors, and

The former is implemented with non-deterministic state transitions – each transition corresponding to choosing an additional anchor –, whereas the latter is implemented deterministically.

Concretely, the automaton is constructed by following the construction of section 4.4 to decompose each pattern into its canonical path-split graph. We then order the nodes of the PSG and express each node as a condition that ensures the connectivity and node weight in the graph matches the pattern. We thus obtain a chain of conditions, with a transition between any two consecutive conditions;

transitions are deterministic by default and marked as non-deterministic whenever they lead to a condition on an anchor node. The state automaton for all patterns is then obtained by joining all chains of conditions into a tree.

Runtime. Pattern matching is then as simple as simulating the state automaton, evaluating all conditions on the graph G passed as input. The states in the automaton corresponding to the last condition of a pattern must be marked as *end states*, along with a label identifying the pattern that was matched. This can then be used at runtime to report all patterns found.

Our implementation has been tested for correctness, i.e. on the one hand that all matches that are reported are correct, and on the one hand that all pattern matches are found. This was done by comparing the matches of our implementation with the results obtained from matching every pattern separately on millions of randomly generated graphs and edge cases. We also ensured during benchmarking that the number of matches reported by our implementation and by Quartz were always the same.

Benchmarks

Baseline. To assess practical use, we have benchmarked our implementation against a leading C++ implementation of pattern matching for quantum circuits from the Quartz superoptimiser project [332]. This implementation is the principal component of an end-to-end quantum circuit optimisation pipeline. The results and speedups we obtain here thus apply and transfer directly to this application.

Dataset. We further ensure that our results apply in practice by using a real-world dataset of patterns. The Quartz optimiser finds opportunities for circuit optimisation by relying on precomputed equivalence classes of circuits (ECC). These are obtained exhaustively by enumerating all possible small quantum circuits, computing their unitaries and clustering them into classes of circuits with identical unitaries.

1. Such small circuit sizes are imposed in part by the fact that ECCs of larger circuits quickly become unfeasible to generate as their number grows exponentially. In practice, large circuit transformations can often be expressed as the composition of smaller atomic transformations, hence the good performance of this approach in practice. ↩

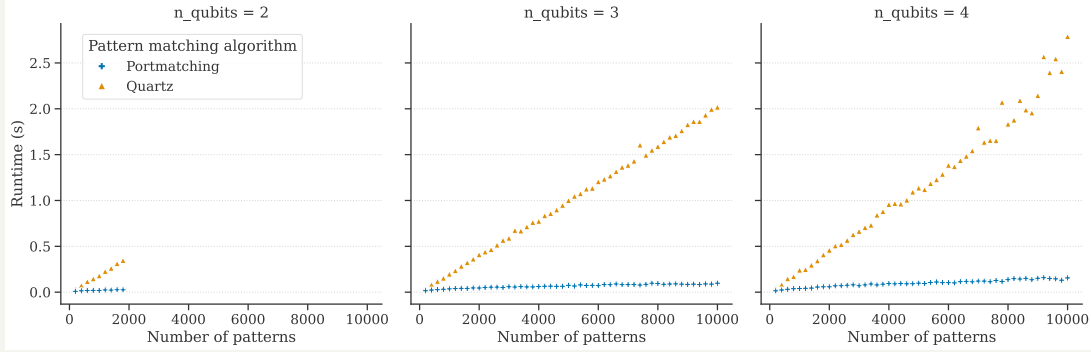
The generation of ECC sets is parametrised on the number of qubits, the maximum number of gates and the gate set in use. For these benchmarks we chose the minimal set of gates T, H, CX and considered circuits with up to 6 gates and 2, 3 or 4 qubits. The size of these pattern circuits is typical for the application¹.

Thus, for our patterns, we have the bound $d \leq 6$ for the maximum depth and width $w = 2, 3, 4$. In all experiments, the graph G subject to pattern matching was `barenco_tof_10` input, i.e. a 19-qubit circuit input with 674 gates obtained by decomposing a 10-qubit Toffoli gate using the Barenco decomposition [36].

Results. We study the runtime of our implementation as a function of the number ℓ of patterns being matched, up to $\ell = 10^4$ patterns. We expect the runtime of pattern matching algorithms that match one pattern at a time to scale linearly with ℓ . On the other hand, Proposition 4.13 results in a complexity that is independent of ℓ .

For each value of ℓ , we select a subset of all patterns in the ECC sets at random. For $w = 2$, there are only a total of $\ell = 1954$ patterns, explaining why we do not report result beyond that number. For $\ell = 200$ patterns, our proposed algorithm is $3\times$ faster than Quartz. As expected, the advantage of our approach increases

as we match more patterns, scaling up to a $20\times$ speedup for $\ell = 10^4$. The results are summarised in the following figure:

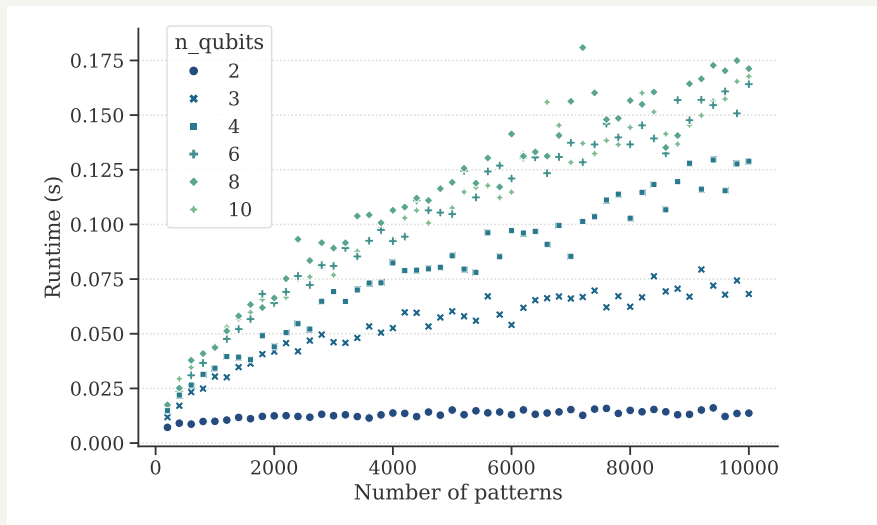


Runtime of pattern matching for $\ell = 0 \dots 10^4$ patterns on 2, 3 and 4 qubit quantum circuits from the Quartz ECC dataset, for our implementation (Portmatching) and the Quartz project. All $\ell = 1954$ two-qubit circuits were used, whereas for 3 and 4 qubit circuits, $\ell = 10^4$ random samples were drawn.

Dependency on w and ℓ . We further study the runtime of our algorithm as a function of its two main parameters, the number of patterns ℓ and the pattern width w , on an expanded dataset. To this end, we generate random sets of 10,000 pattern circuits with 15 gates and between $w = 2$ and $w = 10$ qubits, using the same gate set as previously. The resulting pattern matching runtimes are shown in the figure below.

From Proposition 4.13, we expect that the pattern matching runtime is upper bounded by a ℓ -independent constant. Our results support this result for $w = 2$ and $w = 3$ qubit patterns, where runtime seems indeed to saturate, reaching an observable runtime plateau at large ℓ .

We suspect on the other hand that the exponential c^w dependency in the complexity bound of Proposition 4.13 makes it difficult to observe a similar plateau for $w \geq 4$, as we expect this upper bound on the runtime to increase rapidly with qubit counts. A runtime ceiling is not directly observable at this experiment size, but the gradual decrease in the slope of the curve is consistent with the existence of the ℓ -independent upper bound predicted in Proposition 4.13.



Runtime of our pattern matching for random quantum circuits with up to 10 qubits.

Fully and Confluently Persistent Graph Rewriting

This chapter leverages another construction from graph rewriting theory that finds a direct application in quantum compilation: the unfolding of graph transformation systems [23, 329]. Whereas most applications to date have focused on using unfolding for model verification (see section 5.1 for a review), we will use the same techniques to instead speed up optimisation problems over the space of reachable graphs in a GTS.

In the unfolding, graph rewrites are expressed as *persistent* modifications of the graph. Mutable data structures are typically *ephemeral*: modifying the data structure overwrites information and invalidates any references to the old data. In contrast, a persistent data structure applies changes to the data so that both the old and new versions remain accessible – a famous example of this are [version control systems such as git](#).

A data structure is *fully* persistent if modifications can be applied not only to the latest version but also to previous versions of the data structure. In that case, a version of the data may be used to create several new versions. Instead of a linear *edit history* of all mutations, the result is an edit history tree, with possibly many “most recent” versions – leaves in the edit history.

Finally, a fully persistent data structure is also *confluently* persistent if different versions of the data in the edit history can be joined together. As a result, the edit history forms a directed acyclic graph (DAG) of versions of the data, linked by data mutation and joining operations. Adopting terminology from git, we call a join of two or more versions a *merge* of multiple versions.

In this chapter, we will consider all graphs to be hypergraphs (V, E) with vertex set V and hyperedge set $E \subseteq V^*$. All results can easily be adapted to accommodate graph attributes, weights, and types as required by applications. This means the data structure and algorithms we present apply directly to minIR graphs and, more broadly, to most instances of graph rewriting.

1. This makes use of the fact that unlike in DPO, our rewrite definition allows the (implicit) deletion of edges with one endvertex in V^- . ↩

The central object of study in this chapter is the *graph rewrite*. We restate a simplified version of Definition 3.9 here. We opt for convenience for a rewrite definition that omits the *edge deletion set* E^- of Definition 3.9. This is not a restriction of the general case, as can be seen by adding a “dummy” vertex v_e for each edge e in a graph: a rewrite that removes an edge $e \in E^-$ can equivalently be expressed by the rewrite that removes the dummy vertex $v_e \in V^-$.

As in previous chapters, $\sqcup$ denotes disjoint union, $f : A \rightarrow B$ denotes a partial function and dom denotes the domain of definition of a (partial) function.

Definition 5.4 *Graph rewrite*

A rewrite r on a graph $G = (V, E)$ is given by a tuple $r = (G_R, V^-, \mu)$, with

- $G_R = (V_R, E_R)$ is a graph called the *replacement graph*,
- $V^- \subseteq V$ is the *vertex deletion set*, and
- $\mu : V^- \rightarrow V_R$ is the *glueing relation*, a partial function that maps a subset of the deleted vertices of G to vertices in the replacement graph.

Define the *context* subgraph G_C as the subgraph induced by the vertices

$$V_C = (V \setminus V^-) \cup \text{dom}(\mu).$$

The rewritten graph resulting from applying r to G is the glueing

$$r(G) = (G_C \sqcup G_R) / \sim_\mu.$$

obtained from the union of G_C and G_R by merging all vertices within the same class in the equivalence relation $\sim_\mu$ that is the closure of μ . We refer to section 3.5 for more details and an illustration of glueings and rewrites.

In this chapter, we will consider sequences of multiple rewrites. We will use the notation $V(G)$ and $E(G)$ to designate the vertices, respectively the edges, of a graph G . It is further assumed that the vertices $V(G)$ and $V(G')$ for $G \neq G'$ are always disjoint, a fact that we underline by always writing unions of graphs and vertices with $\sqcup$.

We make use of the fact that for every rewrite $r = (G_R, V^-, \mu)$, the equivalence classes α of $\sim_\mu$ are of the form

$$\alpha = \{m\} \sqcup \{v \in V^- \mid \mu(v) = m\},$$

for some $m \in V(G_R)$. For every set of merged vertices $\alpha \subseteq V(G)$ in $r(G)$, there is thus a unique vertex not in V^- :

$$\alpha \setminus V^- = \{m\}.$$

We choose to always identify the merged vertex in $r(G)$ with m . Using this convention, the set of vertices of $r(G)$ is simply

$$V(r(G)) = (V(G) \setminus V^-) \sqcup V(G_R).$$

5.1. Related work

The unfolding of a graph transformation system (GTS) was first proposed in [23] as a generalisation of a well-known construction on Petri nets [329]. Originally defined for DPO rewriting, the unfolding was later generalised to SPO [24, 25] and SqPO [44] in arbitrary adhesive categories. The unfolding is a powerful GTS technique that has found applications in model verification [26, 27, 95] and other formal analysis tools such as model-based diagnosis [28] and model transformation analysis [53].

Unfoldings of finite GSTs are often infinite. A lot of work has therefore concerned itself with finding sufficient conditions for finiteness or the existence of finite complete prefixes of unfoldings [26, 29, 30, 31, 284]. On the other hand, unfoldings of GSTs of quantum computation are expected to be intractably large [335], with no complete prefixes in general. Rather, our interests lie in finding heuristics that determine the subspace of the unfolding of interest, combined with fast algorithms to expand finite unfolding prefixes into larger ones. This chapter is to our knowledge the first work in this direction.

Persistent data structures on the other hand have a rich history in computer science [118, 205], and particularly within functional programming [242, 243, 168]. Confluently persistent data structures were first explored in [117]. A general treatment of the approach was subsequently presented in [139] and improved in [87]. [76] proposed a data structure for confluently persistent tries.

Within the field of graph transformations, there is a well-developed theory for persistent (and confluently persistent) transformations in the form of the concurrent graph transformation formalism of Corradini et al. [93] and Baldan et al. [32]. This categorical formalism has also been extended to include support for overlapping transformations in [121].

The first practical application of persistent graph rewriting was developed by the graph rewriting engine GRAPE [317] and was originally based on ephemeral data structures with transactional ACID semantics. Later, its successor, GrapeVine [318], enhanced this with the first fully persistent data structure for graph rewriting. In this work, the vertices and edges that result from graph rewrites are stored individually in a specialised database. Depending on the requested data version, the graph can be retrieved from the database's individual vertex and edge entities using the database's graph query language. To our knowledge, no confluently persistent data structure has been proposed for graph rewriting.

As we will see in section 5.5, confluent persistence is a particularly valuable property in the absence of a rewriting strategy, i.e. a procedure to select and prioritise among possible graph transformations [122]. This distinguishes the approach presented in this thesis from most previous work. Rewriting strategies feature

transformation order [267]; and finally, tools such as GrGen provide advanced control flow primitives to specify rewrite rule execution [148].

Specifying rewriting strategies yields efficient graph transformation procedures and is particularly effective for systems with provable properties such as confluence and termination [314]. As a result, rewriting strategies have also been used successfully within classical compiler optimisations [16] and quantum circuit optimisation [129, 119].

However, such properties of the transition system – or successful heuristic approximations for it – cannot always be derived. In these cases, the space of graphs reachable from an input graph within the transition system must be explored non-deterministically. In the absence of a control program, GROOVE will fall back to an exhaustive exploration of the search space – for an exploration up to depth Δ , the result is a search tree of size $\mathcal{O}(\gamma^\Delta)$, where γ is the number of possible rewrites at every graph in the search space (assuming γ is constant for every reachable graph).

Exhaustive exploration is used extensively in model checking, typically to verify properties that must hold for all reachable graphs [268]. It has also proven to be very useful for compiler optimisation, where the constantly evolving rewrite rules, instruction sets and complex, architecture-dependent cost functions render it challenging to fix a deterministic program transformation schedule.

Jia et al. showed in [181] that computation graph optimisation using graph transformations was achievable without predefined rewriting strategies. They discovered new state-of-the-art implementations for computation graphs of interest to the deep learning community using a simple exhaustive search of the space of possible rewrites with backtracking. This approach was then adapted to quantum circuit optimisation in [332] and [333].

These recent results fit within a long line of compiler research called *superoptimisation* [144, 221, 277, 34, 278]. On top of excellent optimisation performance, this approach to compiler optimisation using graph transformation systems (GTS) is exceptionally flexible, as rewrite rules can be generated and tailored on demand to the constraints and instruction set of the target hardware. For any supplied cost function, the compiler can explore all valid program transformations to find the rewrites sequence that minimises cost. This keeps the cost function-specific logic separate from the transformation semantics of the program, making it straightforward to replace or update the optimisation objective.

The adaptation of superoptimisation to quantum optimisation of [332] and [333] is, however, showing scaling difficulties: unlike classical superoptimisation, which is usually designed to optimise small subroutines within programs, e.g. focusing on arithmetic instructions, single instruction multiple data (SIMD) etc., the technique should in principle be able to optimise quantum programs in their entirety and requires tens of thousands of rewrite rules. This leads to immense search spaces that superoptimisation does not scale well to.

For the special case of *term rewriting*, i.e. rewriting of tree expressions, a technique known as *equality saturation* was introduced in [307] to compress and reduce the size of the search space significantly. Equality saturation can be viewed as a twist on persistent data structures designed to optimise terms through term rewriting. It is persistent insofar as it preserves all data inserted into it, though, unlike persistent data structures, it does not retain the history of transformations.

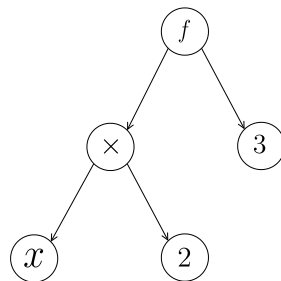
This introduces a new step in the optimisation process known as the *extraction phase*, where the best term stored within the data structure must be identified and recovered.

An efficient implementation was presented in [325], and it has recently been adopted in modern compiler optimisation pipelines [130]. Though the approach was extended to optimise computation graphs for deep learning in [335], equality saturation does not generalise to graph rewriting. Equality saturation and the difficulties of adapting it to graph rewriting are interesting (and subtle!) enough to warrant their own section.

5.2. A closer look at equality saturation

Below, we provide a succinct introduction to equality saturation and discuss its shortcomings in the context of quantum computation and graph rewriting in general. For further details on equality saturation, we recommend the presentation of [325], its implementation [326], and this blog discussion [48].

Unlike a general-purpose compiler utility, equality saturation is specifically a technique for term rewriting. Terms¹ are algebraic expressions represented as trees, in which tree nodes correspond to operations, the children of an operation are the subterms passed as arguments to the operation, and leaf nodes are either constants or unbound variables. For instance, the term $f(x \times 2, 3)$ would be represented as the tree:

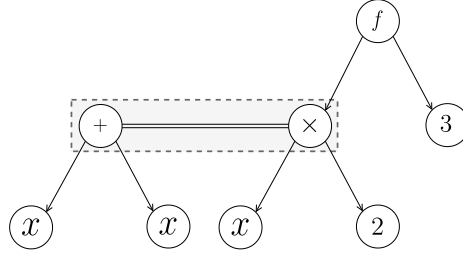


This representation is particularly suited for any pure functional (i.e. side-effect-free) classical computation. Every node of a term is identified with its own term: the subterm given by the subtree the node is the root of. Given term transformation rules, term rewriting consists of finding subterms that match known transformation patterns. The matching subtrees can then be replaced with the new equivalent trees.

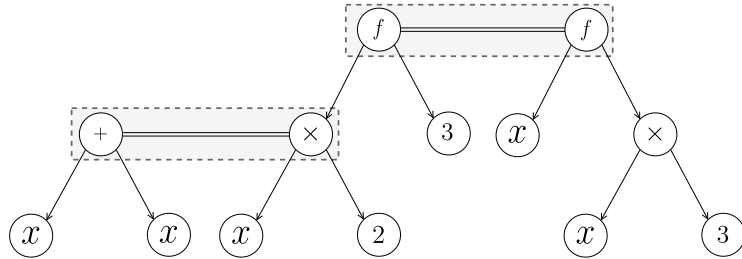
In equality saturation, all terms obtained through term rewriting are stored within a single persistent data structure. Term optimisation proceeds in two stages. First, an exploration phase adds progressively more terms to the data structure to discover and capture all possible terms that the input can be rewritten to until *saturation* (see below), or a timeout, is reached. In the second phase, the saturated data structure is passed to an *extraction* algorithm tasked with finding the term that minimises the cost function of interest among all terms discovered during exploration.

1. Depending on the context, computer scientists also call them abstract syntax trees (AST) – for our purposes, it's the same thing. ↩

The data structure that enables this is a generalisation of term trees. Just as in terms, nodes correspond to operations and have children subterms corresponding to the operation's arguments. To record that a new term obtained through a rewrite is equivalent to an existing subterm, we extend the data structure we employ to also store equivalence classes of nodes, typically implemented as Union-Find data structures [147, 91]. If we, for instance, applied the rewrite $x * 2 \mapsto x + x$ to the term above, we would obtain

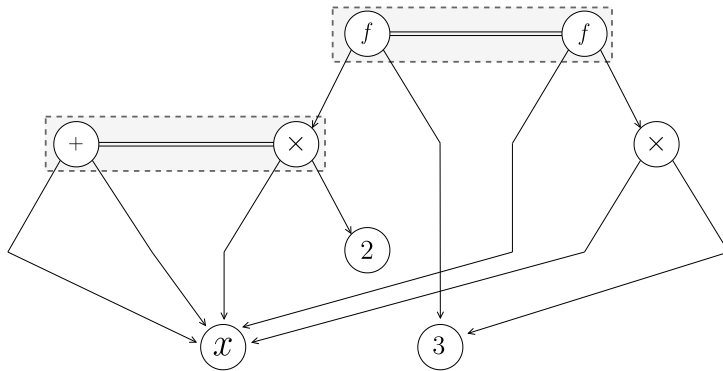


Nodes within a grey box indicate equivalent subterms. This diagram encodes that any occurrence of the $x * 2$ term can equivalently be expressed by the $x + x$ term. Henceforth, when matching terms for rewriting, both the $*$ term and the $+$ term are valid choices for the first argument of the f operation. Suppose, for example, the existence of a rewrite $f(x + y, z) \mapsto f(x, y * z)$, then this would match the above data structure, resulting in



A consequence of using equivalence relations in the data structure is that the ordering in which the rewrites are considered and applied becomes irrelevant!

As presented, the exploration process would never terminate, and the data structure size would grow indefinitely: as more rewrites are applied, more and more terms are created, resulting in an ever-increasing set of possible rewrites to be considered and processed. Equality saturation resolves this by enforcing a term uniqueness invariant: every term or subterm explored is expressed by exactly one node in the data structure. We can see in the above example that this is currently not the case: the term x for instance, is present multiple times – so is 3 . As a result, the nodes no longer form a forest of trees but instead directed acyclic graphs:



This is commonly known as *term sharing*, and the resulting data structure is known as a *term graph* [325]. Maintaining this invariance is not hard in practice: whenever a new term is about to be added by applying a rewrite, it must first be checked whether the term exists already – something that can be done cheaply by keeping track of all hashes of existing terms. In the affirmative case, rather than adding a new term to the matched term’s equivalence class, both terms’ classes must be merged.

It might be that equivalence classes must be merged recursively: given the terms $f(x, 3)$ and $f(y, 3)$, if the classes of x and y are merged (and thus x and y have been proven equivalent), then the classes of their respective parent $f(x, 3)$ and $f(y, 3)$ must also be merged. Doing so efficiently is non-trivial, so we will not go into details here and refer again to [325].

In the absence of terms of unbounded size, the uniqueness invariant guarantees that the exploration will eventually *saturate*: as rewrites are applied, there will come a point where all equivalent terms have been discovered, i.e. every applicable rewrite will correspond to an equivalence within an already known class, thereby not providing any new information. This marks the end of the exploration phase².

Term optimisation then proceeds to the extraction phase. Reading out an optimised term out of the saturated term data structure is not trivial. For every equivalence class in the data structure, a representative node must be chosen so that the final term, extracted recursively from the root by always selecting the representative node of every term class, minimises the desired cost function³.

The strategy for choosing representative terms depends heavily on the cost function. In simple cases, such as minimising the total size of the extracted term, this can be done greedily in reverse topological order, i.e. proceeding from the leaves towards the root [325]. There are also more complex cases, however: if the cost function allows for the sharing of subexpressions that may be used more than once in the computation, for instance, then finding the optimal solution will require more expensive computations such as solving boolean satisfiability (SAT) or Satisfiability Modulo Theories (SMT) problem instances [52].

Equality saturation on graphs?

Equality saturation is a fast-developing subfield of compilation with a growing list of applications. Unfortunately for us⁴, adapting these ideas to quantum computat-

2. Of course, it is also practical to include a timeout parameter in implementations to guarantee timely termination even on large or ill-behaved systems. ↩

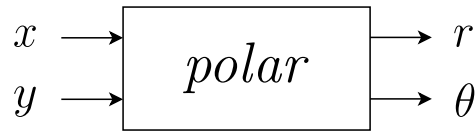
3. Note that we are omitting a subtle point here that arises due to term sharing: depending on the cost function, choosing different representative nodes for the same class could be favourable for the other occurrences of the term in the computation. ↩

4. but fortunately for this thesis ↩

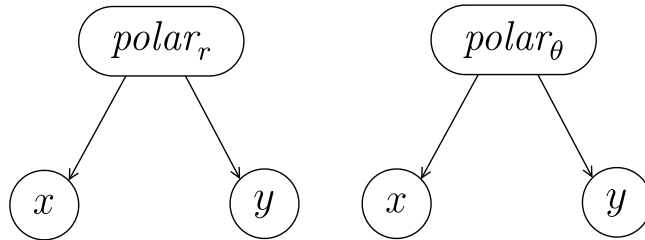
ion (and graph rewriting more generally) presents several unsolved challenges.

The root of the problem lies in the program representation. The minIR representation we presented in section 3.3 – but also the quantum circuit representation – captures quantum computations, not as a term, but in a directed acyclic graph (DAG) structure.

A generalisation of equality saturation to computation DAGs was studied in [335] in the context of optimisation of computation graphs for deep learning. Their approach is based on the observation that the computation of a (classical) computation DAG can always be expressed by a term for each output of the computation. Consider, for example, the simple computation that takes two inputs (x, y) representing 2D cartesian coordinates and returns its equivalent in polar coordinates (r, θ) .



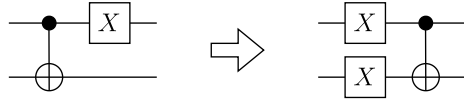
By introducing two operations $polar_r$ and $polar_\theta$ that compute $polar$ and subsequently, discard one of the two outputs, the DAG can equivalently be formulated as two terms



corresponding to the two outputs r and θ of the computation. This involves temporarily duplicating some of the data and computations in the DAG – though all duplicates will be merged again in the term graph due to the term sharing invariant.

This duplicating and merging of data is fundamentally at odds with the constraints we must enforce on linear data, such as quantum resources. Each operation (or data) of a DAG that is split into multiple terms introduces a new constraint that must be imposed on the extraction algorithm: a computation DAG will only satisfy the *no-discarding* theorem (section 2.1) for linear values if, for each split operation it contains, it either contains all or none of its split components.

To illustrate this point, consider the following simple rewrite on quantum circuits that pushes X gates ($\oplus$) from the right of a CX gate to the left:



Both the left and right hand sides would be decomposed into two terms, one for each output qubit. The left terms could be written as

$$X(CX0(0, 1)) \quad \text{and} \quad CX1(0, 1)$$

whereas the right terms would be

$$CX0(X(0), X(1)) \quad \text{and} \quad CX1(X(0), X(1)).$$

We introduced the term $X(\cdot)$ for the single-qubit X gate and two terms $CX1(\cdot, \cdot)$ and $CX2(\cdot, \cdot)$ for the terms that produce the first, respectively second, output of the two-qubit CX gate. 1 and 0 denote the input qubits of the computation. This would be interpreted as two different rewrites

$$\begin{aligned} X(CX0(0, 1)) &\mapsto CX0(X(0), X(1)) \\ \text{and } CX1(0, 1) &\mapsto CX1(X(0), X(1)). \end{aligned}$$

Unlike classical computations, however, either of these rewrites on their own would be unphysical: there is no implementation of either split operations $CX0$ or $CX1$ on their own. We would thus have to enforce at extraction time that for every application of this pair of rewrite rules, either both or none of the rewrites are applied.

Conversely, satisfying the *no-cloning* theorem requires verification that during extraction, terms that share a subterm but correspond to distinct graph rewrites are never selected simultaneously – otherwise, the linear value corresponding to the shared subterm would require cloning to be used twice.

The no-discarding and no-cloning restrictions result in a complex web of **AND** respectively **XOR** relationships between individual terms in the term graph. These constraints *could* be ignored during the exploration phase and then be modelled in the extraction phase by an integer linear programming (ILP) problem. However, [335] observed that this approach causes the term graph to encode a solution space that grows super-exponentially with rewrite depth (see Fig. 7 in [335]), rendering the ILP extraction problem computationally intractable beyond 3 subsequent rewrites. Recent work has attempted to tackle this issue using reinforcement learning [35].

Linearity-preserving rewrites are an exponentially small subset

A simple calculation shows that in the case that all values in the computation graph are linear and only graphs up to a maximal size are considered, the number of possible rewrites only grows exponentially in the rewrite depth. In other words, for optimisation of quantum computations, the solution space of valid

5. Exponential is super-exponentially smaller than super-exponential! Or put mathematically $e^{o(n)}/e^{\Theta(n)} = e^{o(n)-\Theta(n)} = e^{o(n)}$. ↩

computations is *much* smaller⁵ than the space explored by the equality saturation approach of [335].

Indeed suppose there is a maximal graph size $|V(G)| \leq \Theta$ and suppose that all rewrite patterns, i.e. the subgraph induced by the vertex deletion set V^- of a rewrite, are connected. This is an assumption that was also made in chapter 4, see section 4.2 for a discussion.

In a computation graph of linear values G , every vertex (value in the computation) $v \in V(G)$ has a *unique* incoming and outgoing edge. This means that any pattern embedding $\varphi : P \hookrightarrow G$ is uniquely defined by the image $\varphi(v_P)$ of a single vertex $v_P \in V(P)$. Thus for a GTS with m transformation rules, there can be at most a constant number

$$m \cdot |V(G)| \leq m \cdot \Theta =: \alpha$$

of possible rewrites that can be applied to any graph G . Let $\mathcal{G}_d$ be the set of all graphs that can be reached in the GTS in at most d rewrites from some input graph G_0 . $\mathcal{G}_{d+1}$ is the set of all graphs obtained by applying a rewrite to a graph $G \in \mathcal{G}_d$. Thus we have the relation:

$$|\mathcal{G}_{d+1}| \leq \alpha \cdot |\mathcal{G}_d|,$$

The total number of rewrites R_d that can be applied on any graph in $\mathcal{G}_d$ is thus

$$R_d \leq \alpha \cdot |\mathcal{G}_d| = O(e^{\alpha \cdot d}).$$

In summary, equality saturation is a specialisation of persistent data structures uniquely suited to the problem of term rewriting. It succinctly encodes the space of all equivalent terms, and using term sharing does away with the need to apply equivalent rewrites on multiple copies of the same term, which inevitably occurs on more naive rewriting approaches.

However, equality saturation cannot model rewrites that require deleting parts of the data. This is not a problem for terms representing classical operations, as data can always be implicitly copied during exploration and discarded during extraction as required. This is not the case for quantum computations – and for graph rewriting in general, where explicit vertex (and edge) deletions are an integral part of graph transformation semantics.

As a result, numerous constraints would have to be imposed to restrict the solution space encoded by term graphs to valid outcomes of graph rewriting procedures. This would make extraction algorithms complex and cumbersome. More importantly, we showed that in the case of computation graphs on linear values, such as quantum computations, the solution space explored by equality saturation is *super-exponentially larger* than the space of valid computations, rendering the extraction algorithm and meaningful exploration of the relevant rewriting space computationally intractable.

5.3. The data structure

We now present a data structure that is closely related to equality saturation but supports arbitrary graph rewriting. It is modelled on the graph unfolding construction as presented in [26].

Rather than maintaining equivalence relations between terms, as done in term graphs, we maintain equivalence relations between graph vertices. Our data structure stores the set of all applied rewrites – the main subject of this section is to show how all operations of interest on this data structure can be implemented efficiently.

The persistent graph rewriting data structure is given by a set $\mathcal{D}$ of events $\delta = (G_R, V^-, \mu) \in \mathcal{D}$, with

- vertex deletion set $V^- \subseteq V(\mathcal{D} \setminus \delta)$ and
- glueing relation $\mu : V^- \rightarrow V(G_R)$.

We have extended the $V(\cdot)$ notation to $\mathcal{D}$ by defining it as the union of all vertex sets of replacement graphs in $\mathcal{D}$. We will similarly use $V(\delta)$ to denote the set of vertices in the replacement graph of a rewrite δ .

Events resemble rewrites as defined in Definition 5.4 but differ in that they do not apply to a single graph G , i.e. there is no graph such that $V^- \subseteq V(G)$. Instead,

$$V^- \subseteq \bigsqcup_{\delta \in \mathcal{D}} V(\delta) = V(\mathcal{D}). \quad (2)$$

We will see below how a graph G can be constructed such that an event $\delta \in \mathcal{D}$ is indeed a valid rewrite on G .

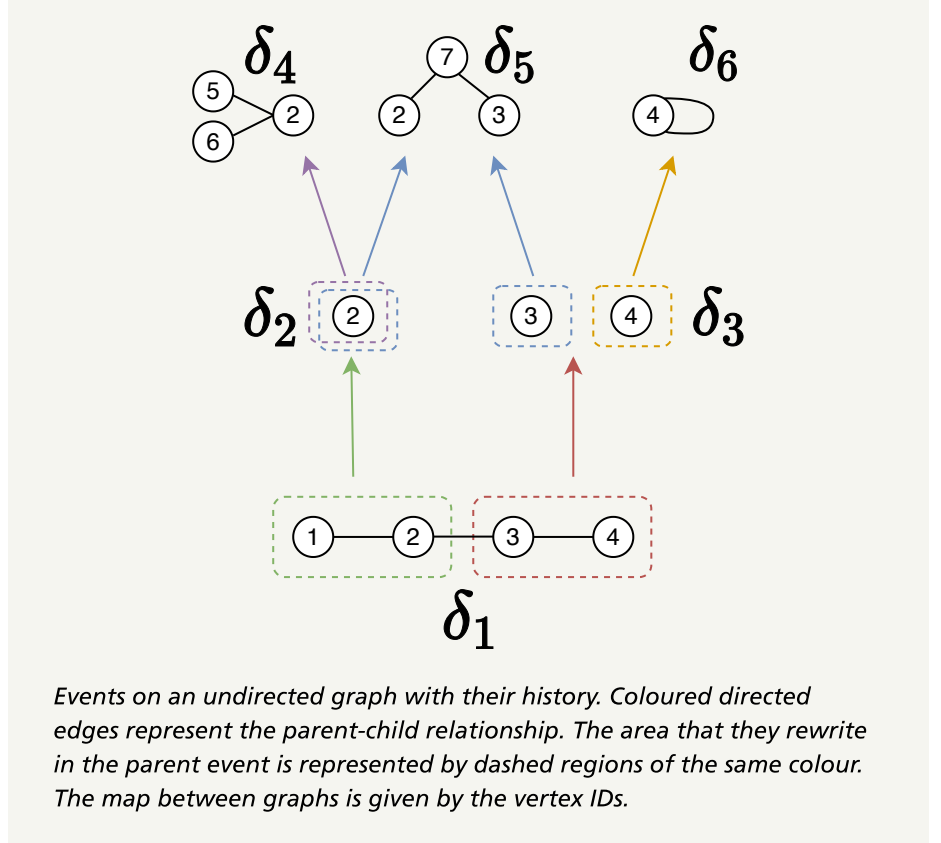
Using the disjointness of the union in (2), for all $v \in V(\mathcal{D})$, there is a unique $\delta \in \mathcal{D}$ such that $v \in V(\delta)$ that we call the *owner* of v . The *parents* (or directed *causes*) $P(\delta)$ of an event δ are then the owners of the vertices in the deletion set V^- of δ :

$$P(\delta) = \{\delta_p \in \mathcal{D} \mid V^- \cap V(\delta_p) \neq \emptyset\}.$$

Inversely, we define the *children* of δ as the set of event whose parents include δ

$$P^{-1}(\delta) = \{\delta_c \in \mathcal{D} \mid \delta \in P(\delta_c)\}.$$

The following figure shows an example of a data structure $\mathcal{D}$ on undirected graphs.



Merges, confluent persistence and event creation

A rewrite $r = (G_R, V^-, \mu)$ that applies to a replacement graph of an event δ_p , i.e. $V^- \subseteq V(\delta_p)$ immediately defines a valid event $\delta_r := r$. In that case, δ_r has a unique parent

$$P(\delta_r) = \{\delta_p\}.$$

Creating an event δ_r from a rewrite r is the simplest type of data mutation that can be recorded in $\mathcal{D}$. For $\mathcal{D}$ to be a confluent persistent data structure, it must also be allowed to *merge* multiple data mutations together. Rather than handling merges of versions of the data structure explicitly, an event $\delta \in \mathcal{D}$ can define graph mutation operations that apply on collections of events $P \subseteq \mathcal{D}$ – the resulting mutation is equivalent to explicitly creating a merged version of the versions in P , followed by the desired rewrite. In this case, the parents of δ are precisely the set $P = P(\delta)$.

In other words, the parent-child relationships of $\mathcal{D}$ is precisely the *event history* of $\mathcal{D}$: a directed graph with vertex set $\mathcal{D}$ and edges $\delta_1 \rightarrow \delta_2$ if $\delta_1 \in P(\delta_2)$. For $\mathcal{D}$ to define a valid confluent persistent data structure, we need to

1. Ensure that the event history is acyclic, and
2. Define conditions that guarantee that events correspond to valid *data mutations*.

We hit both birds with one stone by restricting how $\mathcal{D}$ can be constructed and modified in such a way that acyclicity is guaranteed. Specifically, we introduce two procedures:

- `CreateEmpty` constructs an empty $\mathcal{D} = \emptyset$, and
- `AddEvent`, adds an event δ to $\mathcal{D}$.

The first is straightforward – and importantly, the only way to construct an instance $\mathcal{D}$. `AddEvent`, on the other hand, enforces two conditions that δ must satisfy to be added to a set $\mathcal{D}$:

- $P(\delta) \subseteq \mathcal{D}$, and
- all parents of δ must be *compatible*.

We defer the discussion on the second condition, enforced by the `AreCompatible` procedure, to its dedicated section below. The restriction $P(\delta) \subseteq \mathcal{D}$ defines a partial order on events by guaranteeing that an event δ can only be defined and added to $\mathcal{D}$ after all its parents $P(\delta)$ have been added.

We say that $\mathcal{D}$ is *valid* if it can be constructed from a single call to `CreateEmpty`, followed by a sequence of calls to `AddEvent`. This is equivalent to requiring that

1. the parent-child relationship is acyclic and
2. the parents of every event satisfy `AreCompatible`.

For the remainder of this chapter, we will always assume that $\mathcal{D}$ is valid, and thus the event history of $\mathcal{D}$ is always well-defined and acyclic.

```
def CreateEmpty() -> Set[Event]:
    return set()

def AddEvent(
    events: Set[Event],
    replacement_graph: Graph
    deletion_set: Set[V],
    glueing_relation: EquivalenceRelation[V]
) -> Set[Event]:
    new_event = (
        replacement_graph,
        deletion_set,
        glueing_relation
    )
    parents = parents(new_event)
    assert(issubset(parents, events))
    assert(AreCompatible(parents))

    events = union(events, {new_event})
```

Compatible events

Assuming the parent-child relationship is acyclic, we can define the ancestors (or causes) $\lfloor \delta \rfloor$ of an event δ recursively

$$\lfloor \delta \rfloor = \{ \delta \} \cup \bigcup_{\delta_p \in P(\delta)} \lfloor \delta_p \rfloor.$$

Events $D \subseteq \mathcal{D}$ are *compatible* (or a *configuration*) if all vertex deletion sets V^- for all ancestors of $\delta \in D$ are disjoint. That is, writing

$$\lfloor D \rfloor = \bigcup_{\delta \in D} \lfloor \delta \rfloor,$$

we require that all sets $\{V^- \mid (G_R, V^-, \mu) \in \lfloor D \rfloor\}$ are disjoint. In the example above, events δ_5 and δ_6 are compatible, whereas δ_5 and δ_4 are not. As pseudo-code, this is implemented by the following procedure.

```
def AreCompatible(events: Set[Event]) -> bool:
    all_ancestors = union([ancestors(d) for d in events])
    deleted_vertices = set()
    for d in all_ancestors:
        for v in deletion_set(d):
            if v in deleted_vertices:
                return False
            deleted_vertices.add(v)
    return True
```

Note that this definition of *event compatibility* is a strictly stronger version of parallel independence as is typically defined in DPO rewriting [94]. It does not allow for events δ_1 and δ_2 such that a vertex v is both

- in the *read only context* of δ_1 , i.e. $v \in V_1^- \cap \text{dom}(\mu_1)$, and thus present both before and after the application of δ_1 ,
- in the deletion set of δ_2 , i.e. $v \in V_2^-$.

This excludes *asymmetric conflicts* as discussed in e.g. [26], which arise in the more general definition. This restriction simplifies our considerations as makes the event history of any event unique.

The runtime is $O(D_A \log D_A)$, where D_A is the sum of the sizes of all vertex deletion sets of events in $\lfloor D \rfloor$

$$\sum_{\delta \in \lfloor D \rfloor} |V_\delta^-|.$$

The log factor can typically be removed if the vertices v span a contiguous integer range or by using a hash function. Alternatively, the log factor can also be reduced by using separate sets to track deleted vertices of each event.

When talking about compatible sets of events $D \subseteq \mathcal{D}$, it simplifies considerations to always choose D such that the ancestors of $\delta \in D$ are also in D , i.e. $D = \lfloor D \rfloor$. We introduce the notation

$$\Gamma(\mathcal{D}) = \{\lfloor D \rfloor \mid D \subseteq \mathcal{D} \text{ and } D \text{ is compatible}\} \subseteq \mathcal{P}(\mathcal{D})$$

for the set of all compatible sets of rewrites of the form $\lfloor D \rfloor$.

Events are rewrites on the flattened history

We have so far explored how events can be added to $\mathcal{D}$, as well as when they are compatible. However, until we have established that adding events to $\mathcal{D}$ is *in some sense* equivalent to applying rewrites on a graph, it is hard to see how the data structure $\mathcal{D}$ would be useable for graph rewriting. This is precisely our next point.

In a valid non-empty $\mathcal{D}$, events $\delta \in \mathcal{D}$ form a directed acyclic graph and therefore there must always be (at least) one “root” event $\delta_1 \in \mathcal{D}$ with no parents $P(\delta_1) = \emptyset$. δ_1 is thus a valid rewrite that can be applied to any graph.

For the applications of $\mathcal{D}$ that we consider, it will always be sufficient to have a unique root event δ_1 . Viewing δ_1 as a rewrite that applies to the empty graph $G_0 = \emptyset \rightarrow G_1$, we can understand it as *injecting* the input graph G_1 into $\mathcal{D}$.

Non-root events in $\mathcal{D}$ on the other hand typically correspond to valid (semantics preserving) rewrites in the GTS under consideration.

Consider a set of compatible events $D \in \Gamma(\mathcal{D})$. Define a topological ordering $\delta_1, \dots, \delta_k$ of the events in D , i.e. if $\delta_j \in P(\delta_i)$ then $i < j$.

Proposition 5.1 *Events as valid rewrites*

There are graphs $G_0, \dots, G_k$ such that for all $1 \leq i \leq k$, the event δ_i defines a valid rewrite r_i on G_{i-1} and $G_i = r_i(G_{i-1})$.

Proof. Define the empty graph $G_0 = \emptyset$. The event δ_1 has no parent and thus must have an empty vertex deletion set and glueing relation. It is thus a valid rewrite r_1 on G_0 . Define $G_1 = r_1(G_0)$.

We can similarly define $G_i = r_i(G_{i-1})$ inductively for graphs $G_2, \dots, G_k$ if we show for $2 \leq i \leq k$ that the i -th event δ_i defines a valid rewrite r_i on G_{i-1} . The set of vertices in G_{i-1} is the union of all vertices in the replacement graph of $\delta_1, \dots, \delta_{i-1}$ minus their vertex deletion sets

$$V(G_{i-1}) = \left(\bigsqcup_{1 \leq j < i} V(\delta_j) \right) \setminus \left(\bigsqcup_{1 \leq j < i} V_j^- \right),$$

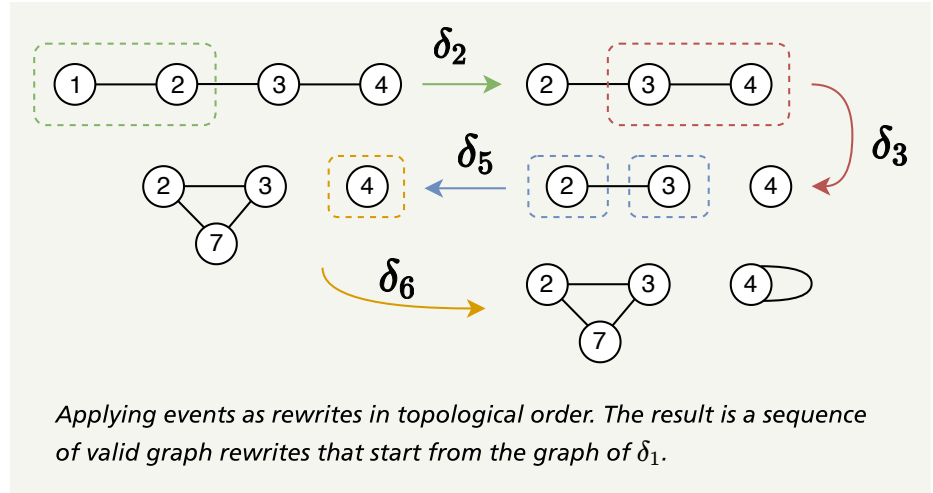
where V_j^- is the vertex deletion set of δ_j .

Now, by definition of the event δ_i ,

$$V_i^- \subseteq \bigsqcup_{1 \leq j < i} V(\delta_j).$$

On the other hand, because of the compatibility of all events in D , we know that $V_i^- \cap V_j^- = \emptyset$ for all $1 \leq j < i$. It thus follows $V_i^- \subseteq V(G_{i-1})$. Hence δ_i is indeed a valid rewrite of G_{i-1} , and thus r_i and G_i are well-defined.

This construction is illustrated in the following figure for the compatible set $[\delta_5] \cup [\delta_6]$ of the previous example.



We now show that the graph G_k is determined uniquely by $D \in \Gamma(\mathcal{D})$ and provide an explicit procedure to construct it.

Proposition 5.2 *Flat graph extraction*

The graph G_k obtained by applying the set of compatible rewrites $D \in \Gamma(\mathcal{D})$ in topological order on the empty graph is independent of the topological ordering chosen.

Given the set of rewrites $D \subseteq \mathcal{D}$, the procedure `FlattenHistory` returns G_k in time

$$O(m + n)$$

where n and m are the total number of vertices and edges across all replacement graphs in D .

Let us start with the definition of `FlattenHistory`:

```

1 def FlattenHistory(events: Set[Event]) -> Graph:
2   all_ancestors = union([ancestors(d) for d in events])
3   graph = Graph()
4   for a in toposort(all_ancestors):
5     add_graph(graph, replacement_graph(a))
6     for (del_v, repl_v) in glueing_relation(a):
7       move_edges(graph, repl_v, del_v)
8     for v in deletion_set(a):
9       remove_vertex(graph, v)
10  return graph

```

`toposort` is a function that returns a topological ordering of the rewrites in D according to the parent-child rewrite relation, `add_graph` inserts the graph passed as second argument into the graph passed as first argument, `remove_vertex` removes the vertex along with all incident edges from the graph and `move_edges` moves all edges of the second vertex to the first vertex.

Proof. Correctness of FlattenHistory. It is easy to see that if the graph G_k that is obtained from applying the rewrites in order is independent of the choice of the topological ordering, then `FlattenHistory` is a correct implementation of the procedure, as it applies one rewrite at a time, in topological order.

Rewrite order invariance. Consider two rewrites $\delta_1, \delta_2 \in D$ such that neither is an ancestor of the other. Let

$$D_{pre} = [\delta_1] \cup [\delta_2] \subseteq D$$

and proceed by induction over D_{pre} : assume the graph G_{pre} obtained by applying the rewrites in D_{pre} is invariant on the choice of the topological ordering of D_{pre} . Clearly this is true for $D_{pre} = \emptyset$. All that remains to be shown is that G_{post} obtained by applying first δ_1 then δ_2 on G_{pre} is equal to G'_{post} , obtained by applying the same rewrites in the reverse order on G_{pre} .

The vertex sets V_1^- and V_2^- of δ_1 and δ_2 must be disjoint because $\delta_1, \delta_2 \in D$ and hence are compatible. Furthermore, the replacement graphs (by definition of the rewrites) and the glueing relations of δ_1 and δ_2 (by rewrite compatibility) cannot contain vertices in $V_1^- \sqcup V_2^-$. It follows that the order in which vertices of $V_1^- \sqcup V_2^-$ are removed from G_{pre} does not affect the graph G_{post} . Furthermore, vertex merging is a commutative operation, and so is disjoint graph addition. It follows $G_{post} = G'_{post}$ and hence the result.

Runtime. In total n vertices and m edges will be added to `graph` by `add_graph` on line 5. As a result, at most n vertices can ever be deleted by line 9. Finally, while a naive implementation of `move_edges` of line 7 might result in the same edge being moved many times, all edge moves can be cached and only executed once at the end: notice that every time edges are moved away from a vertex, that vertex is subsequently removed from the graph. Instead of removing the vertex, keep it “hidden”, with a link to the vertex that the edges should be moved to. Once all graph operations are completed, traverse all hidden vertices and follow the links to the vertices that the edges should be moved to. This can

be done in $O(n)$ time. Then move all edges to the correct vertex, in time $O(m)$, and delete the hidden vertices.

Now instead of exploring the space of all graphs $\mathcal{G}$ reachable by repeatedly applying rewrites, we can explore the rewrite space by adding events to $\mathcal{D}$. Write $flat(D)$ for the graph returned by `FlattenHistory` on set D . If $\mathcal{G}'$ is the set of all graphs returned by `FlattenHistory` on compatible events

$$\mathcal{G}' = \{ flat(D) \mid D \in \Gamma(\mathcal{D}) \},$$

then Proposition 5.1 and Proposition 5.2 combined guarantee that $\mathcal{G}' \subseteq \mathcal{G}$. To conclude, we show that indeed any graph in $\mathcal{G}$ is in $\mathcal{G}'$, and hence $\mathcal{G} = \mathcal{G}'$.

Proposition 5.3 *Rewrites as valid events*

Let $D \in \Gamma(\mathcal{D})$ be a set of compatible events and $G = flat(D)$. Any rewrite r that can be applied on G defines an event $\delta = r$ that can be added to $\mathcal{D}$.

Proof. We recall that a rewrite $r = (G_R, V^-, \mu)$ defines an event $\delta = (G_R, V^-, \mu)$ that can be added to $\mathcal{D}$ if

- $P(\delta) \subseteq \mathcal{D}$, and
- all rewrites in $P(\delta)$ are compatible.

By the rewrite definition, $V^- \subseteq V(G)$. It follows in particular that

$$V^- \subseteq \bigcup_{\delta' \in D} V(\delta'),$$

and thus $V^- \subseteq V(\mathcal{D})$, as well as $P(\delta) \subseteq D \subseteq \mathcal{D}$. This proves both conditions.

Starting from the empty graph $\mathcal{D} = \emptyset$, we can create a root event $\delta_0 = (G, \emptyset, \emptyset)$ with an empty vertex deletion set and glueing relation and add it to $\mathcal{D}$.

Clearly, $flat(\{\delta_0\}) = G$. We then apply Proposition 5.3 repeatedly. If we have a sequence $r_1, \dots, r_k$ of valid rewrites that can be applied on G , then the sequence of events $\delta_1 = r_1, \dots, \delta_k = r_k$ that it defines can also be added to $\mathcal{D}$ in this order. As we have further seen in Proposition 5.1 and Proposition 5.2, the graph G_k that is obtained as a result of the rewrites is the same graph returned by `FlattenHistory` called on $D = \mathcal{D}$.

In other words, we conclude that exploring the rewrite space on G is fully equivalent to exploring the space of valid events starting from $\mathcal{D} = \{\delta_0\}$.

5.4. Exploration and extraction

In the previous section, we proposed a data structure $\mathcal{D}$ that is confluent persistent and can be used to explore the space of all possible transformations of a graph transformation system (GTS). We are now interested in using $\mathcal{D}$ to solve optimisation problems over the space of reachable graphs in the GTS. Following the blueprint of equality saturation (see section 5.2), we proceed in two phases:

1. **Exploration.** Given an input graph G , populate $\mathcal{D}$ with events that correspond to rewrites applicable to graphs reachable from G ,
2. **Extraction.** Given a cost function f , extract the *optimal* graph in $\mathcal{D}$, i.e. the graph that is a flattening of a set of compatible edits $D \subseteq \mathcal{D}$ and minimises f .

Each phase comes with its respective challenges, which we discuss in this section. We will first look at the exploration phase, which requires a way to find and construct new events δ that can be added to $\mathcal{D}$. We will consider the extraction phase in the second part of this section and see that the problem of optimisation over the power set $\mathcal{P}(\mathcal{D})$ can be reduced to boolean satisfiability formula that admit simple cost functions in the use cases of interest.

There is an additional open question that we do not cover in this section and would merit a study of its own: the choice of heuristics that guide the exploration phase to ensure the “most interesting parts” of the GTS rewrite space are explored. We propose a very simple heuristic to this end in the benchmarks of section 5.5, but further investigations are called for.

Exploring the data structure with pattern matching

We established in the previous section that rewrites that apply on G can equivalently be added as events to $\mathcal{D}$. In other words, a graph G' is reachable from G using the rewrites of a GTS if and only if there is a set of compatible events $D \subseteq \mathcal{D}$ such that G' is the graph obtained from `FlattenHistory` on input D .

To expand $\mathcal{D}$ to a larger set $\mathcal{D}' \supseteq \mathcal{D}$, we must find all applicable rewrites on all graphs within $\mathcal{D}$. A naive solution would iterate over *all* subsets of $D \subseteq \mathcal{D}$, check whether they form a compatible set of events, compute `FlattenHistory` if they do, and finally run pattern matching on the obtained graph to find the applicable rewrites. We can do better.

The idea is to traverse the set of events in $\mathcal{D}$ using the glueing relations μ that connect vertices between events. Define the function $\bar{\mu} : V(\mathcal{D}) \rightarrow \mathcal{P}(V(\mathcal{D}))$ that is the union of all glueing relations μ in events in $\mathcal{D}$:

$$\bar{\mu}(v) = \{\mu_c(v) \mid \delta_c = (V_c, V_c^-, \mu_c) \in P^{-1}(\delta_v)\}.$$

where we write δ_v for the owner of v , i.e. the (unique) event $\delta_v \in \mathcal{D}$ such that $v \in V(\delta_v)$. We define the set $\mathcal{E}_D(v)$ of equivalent vertices of v that are compatible with D by applying $\bar{\mu}(v)$ recursively and filtering out vertices whose owner is not compatible with D . It is easiest to formalise this definition using pseudocode

for the `EquivalentVertices` procedure. The set of vertices in $\mathcal{E}_D(v)$ are vertices of descendant events of δ_v .

```
def EquivalentVertices(
    v: Vertex, events: Set[Event]
) -> Set[Vertex]:
    all_vertices = set({v})
    for w in mu_bar(v):
        new_events = union(events, {owner(w)})
        if AreCompatible(new_events):
            all_vertices = union(all_vertices,
                                EquivalentVertices(w, new_events))
    return all_vertices
```

Whilst it looks as though `EquivalentVertices` does not depend on $\mathcal{D}$, it does so through the use of the function calls to `mu_bar`.

We use `EquivalentVertices` to repeatedly extend a set of *pinned* vertices $\pi \subseteq V(\mathcal{D})$. A set of pinned vertices must satisfy two properties:

- the set $D_\pi = \{\delta_v \mid v \in \pi\}$ is a set of compatible events,
- there is no vertex $v \in \pi$ and event $\delta \in D$ such that $v \in V^-(\delta)$.

As a result, for the flattened graph $G = flat(D_\pi)$, it always holds that $\pi \subseteq V(G)$. Furthermore, if $G(\pi) \subseteq G$ is the subgraph of G induced by π , then for any superset of pinned vertices $\pi' \supseteq \pi$, we have $G(\pi) \subseteq G'(\pi')$ where $G' = flat(D_{\pi'})$. In other words: extending a set of pinned vertices results in an extension of the flattened graph – a very useful property when pattern matching. This property follows from the second property above and the definition of `FlattenHistory`.

This gives us the following simple procedure for pattern matching:

1. Start with a single pinned vertex $\pi = \{v\}$.
2. Construct partial embeddings $P \rightarrow G(\pi)$ for patterns P .
3. Pick a new vertex v in $G = flat(D_\pi)$ but not in $G(\pi)$ (that we would like to extend the domain of definition of our pattern embeddings to).
4. For all vertices $v' \in \mathcal{E}_D(v)$, build new pinned vertex sets $\pi' = \pi \cup \{v'\}$, filter out the sets π' that are not valid pinned vertex sets.
5. Repeat steps 2–4 until all pattern embeddings have been found.

Step 1 is straightforward – notice that pattern matching must be started at a vertex in $V(\mathcal{D})$, so finding all patterns will require iterating over all choices of v . The pattern embeddings are constructed over iterations of step 2: each iteration can be seen as one step of the pattern matcher – for instance, as presented in chapter 4 – extending the pattern embeddings that can be extended and discarding those that cannot. If all possible pattern embeddings have been discarded, then matching can be aborted for that π set.

How step 3 should be implemented depends on the types of graphs and patterns that are matched on. It is straightforward in the case of computation graphs with

endpoints and vertices that are incident to exactly one incoming and one outgoing edge. In that case, v can always be chosen in such a way as to ensure progress on the next iteration of step 2, i.e. the domain of definition of at least one partial pattern embedding $P \hookrightarrow G(\pi)$ will be extended by one vertex. The text in the blue box below explains this case in more detail.

Step 4 produces all possible extensions of π to pinned vertex sets π' that include a descendant v' of v (or v itself). All vertices in $\mathcal{E}_D(v)$ are in events compatible with D by definition, so to check that π' is a valid pinned vertex set, we only need to check the second property of pinned vertices. Let P be a pattern, let S be the set of all π sets under consideration. Step 4 increments the sizes of all pinned vertex sets $\pi \in S$ whilst maintaining the following invariant.

Invariant for step 4. If there is a superset $D' \supseteq D_\pi$ of compatible events such that P embeds in $G' = \text{flat}(D')$, then there is a superset $\pi' \supseteq \pi$ of vertices such that P embeds in $\text{flat}(D_{\pi'})$.

Finally, step 5 ensures the process is repeated until, for all partial pattern embeddings, either the domain of definition is complete, or the embedding of P is not possible. Given that step 4 increments the size of π sets at each iteration, this will terminate as long as the vertex picking strategy of step 3 selects vertices that allow to extend (or refute) the partial pattern embeddings constructed and extended in step 2. This is satisfied, for example, in the case of linear minIR graphs, as explained in the box.

Choosing the next vertex to pin in linear minIR (step 3). Assuming patterns are connected, for any partial pattern embedding $P \hookrightarrow G(\pi)$ there is an edge $e_P \in E(P)$ with no image in $G(\pi)$ but such that at least one of the endvertex v_P of e_P has an image v_G in π – say, e_P is the outgoing edge of v_P . Let v'_P be an endvertex of e_P in P that has no image in $G(\pi)$ – and say, it is the i -th outgoing endvertex of e_P in P .

Then v_P uniquely identifies an edge e_G in $G = \text{flat}(D_\pi)$ – the unique outgoing edge of v_G – which, in turn, uniquely identifies a vertex $v'_G \in V(G)$ – the i -th outgoing endvertex of e_G . By choosing v'_G in step 3, step 4 will create pinned vertex sets that include all possible vertices equivalent to v'_G , which are all vertices that v_G might be connected to through its outgoing edge¹. The next iteration of step 2 will then either extend the partial pattern embedding to v_P or conclude that an embedding of P is not possible.

1. To realise this, notice that all vertices equivalent to v'_G are vertices that will be merged with v'_G . Hence, they will all be attached to the outgoing edge of v_G at its i -th outgoing endvertex. ↪

Using the approach just sketched, pattern matching can be performed on the persistent data structure $\mathcal{D}$. The runtime of steps 2 and 3 depend on the type of graphs and patterns that are matched on – these are, however, typical problems that appear in most instances of pattern matching, independently of the data structure $\mathcal{D}$ used here. A concrete approach to pattern matching and results for the graph types of interest to quantum compilation was presented in chapter 4.

The runtime of step 4 and the number of overall iterations of steps 2–4 required for pattern matching will depend on the number of events in $\mathcal{D}$ (`AreCompatible` runs in runtime linear in the number of ancestors), the number of equivalent ver-

Extraction using SAT

Moving on to the extraction phase, we are now interested in extracting the *optimal* graph from $\mathcal{D}$, according to some cost function of interest. Unlike exploring the “naive” search space of all graphs reachable in the GTS, the optimal solution within the persistent data structure $\mathcal{D}$ cannot simply be *read out*.

We showed in section 5.3 that finding an optimal graph G' that is the result of a sequence of rewrites on an input graph G is equivalent to finding an optimal set of compatible events $D \in \Gamma(\mathcal{D}) \subseteq \mathcal{P}(\mathcal{D})$ – the optimal graph G' is then recovered by taking $G' = flat(D)$.

There are $2^{|\mathcal{D}|}$ elements in $\mathcal{P}(\mathcal{D})$, which we encode as a boolean assignment problem by introducing a boolean variable x_δ for all events $\delta \in \mathcal{D}$. The set of events D is then given by

$$D = \{\delta \in \mathcal{D} \mid x_\delta\} \in \mathcal{P}(\mathcal{D}).$$

We can constrain the boolean assignments to compatible sets D by introducing a boolean formula

$$\neg(x_\delta \wedge x_{\delta'}) \tag{3}$$

for all $\delta, \delta' \in \mathcal{D}$ such that their vertex deletion sets intersect $V^-(\delta) \cap V^-(\delta') \neq \emptyset$. Any assignment of $\{x_\delta \mid \delta \in \mathcal{D}\}$ that satisfies all constraints of this format defines a compatible set of events.

How many such pairs of events (δ, δ') are there? By definition of parents, two events δ and δ' can only have overlapping vertex deletion sets if they share a parent. Assuming all events have at most s children, ensuring D is a set of compatible events requires at most $O(s^2 \cdot |\mathcal{D}|)$ constraints.

To further restrict to $D \in \Gamma(\mathcal{D})$, i.e. to sets of compatible events $D = \lfloor D \rfloor$ that contain all ancestors, we can add the further constraints: $\delta \in D$ implies $P(\delta) \subseteq D$. This introduces up to $s \cdot |\mathcal{D}|$ implication constraints

$$x_\delta \vee (\neg x_{\delta'}), \tag{4}$$

for all $\delta, \delta' \in \mathcal{D}$ such that $\delta' \in children(\delta)$.

For any set of events $\mathcal{D}$, the conjunction of all constraints presented above, i.e. the event compatibility constraints (3) and the parent-child relation constraints (4), defines a boolean satisfiability problem (SAT) with variables x_δ . We have shown:

Proposition 5.4 *Extraction as SAT problem*

Consider a GTS with a constant upper bound s on the number of rewrites that may overlap any previous rewrite.

[233] with $|\mathcal{D}|$ variables of size $O(|\mathcal{D}|)$.

Finding the optimal assignment

We now have to find the *optimal* assignment among all satisfiable assignments for the SAT problem given above. In the most general case where the cost function f to be minimised is given as a black box oracle on the graph G' , i.e. on the flattened history of the solution set $D \subseteq \mathcal{D}$, this optimisation problem is hard².

However, if f can be expressed as a function of x_δ instead of the flattened history $G' = flat(D)$, then the ‘hardness’ can be encapsulated within an instance of a SMT problem (satisfiability modulo theories [239, 38]), a well-studied generalisation of SAT problems for which highly optimised solvers exist [235, 286]. A class of cost functions for which the SMT encoding of the optimisation problem becomes particularly simple are *local* cost functions:

Definition 5.1 Local cost function

A cost function f on graphs is *local* if for all rewrites r there is a cost Δf_r such that for all graphs G that r applies to

$$f(r(G)) = f(G) + \Delta f_r.$$

The cost Δf_r of a rewrite r also immediately defines a cost to the event that r defines $\delta = r$. We can thus associate a cost Δf_δ with each event $\delta \in \mathcal{D}$, given by the cost of any of the rewrites that δ defines.

An instance of such a local cost function often used in the context of the optimisation of computation graphs are functions of the type

$$f(G) = \sum_{v \in V(G)} w(v)$$

for some vertex weight function w – i.e. cost functions that can be expressed as sums over the costs $w(\cdot)$ associated to individual vertices in G ³. Indeed, it is easy to see that in this case we can write

$$\begin{aligned} f(r(G)) &= \sum_{v \in r(V(G))} w(v) \\ &= \sum_{v \in V(G)} w(v) - \underbrace{\sum_{v \in V^-} w(v) + \sum_{v \in V_R} w(v)}_{:= \Delta f_r} \\ &= f(G) + \Delta f_r, \end{aligned}$$

2. Hardness can be seen by considering the special case of the extraction problem in which all events are compatible and no two events have a parent-child relation: then there are no constraints on the solution space and the optimisation problem requires finding the minimum of an arbitrary oracle over $2^{|\mathcal{D}|}$ inputs. ↩

3. A similar argument also applies to cost functions that sum over graph edges, as would be the case in minIR, where operations are modelled as hyperedges. ↩

where V^- and V_R are the vertex deletion set and replacement graph of r respectively.

As discussed in section 2.2, many of the most widely used cost functions in quantum compilation are local, as the cost of a quantum computation is often estimated by the required number of instances of the most expensive gate type (such as `CX` gates on noisy devices, or `T` gates for hardware with built-in fault tolerance protocols).

In these cases, the cost function is integer valued and the extraction problem is indeed often *sparse*:

Definition 5.2 *Sparse cost function*

The local cost function f is said to be ε -sparse on $\mathcal{D}$ if

$$|\{\delta \in \mathcal{D} \mid \Delta f_\delta = 0\}| \geq (1 - \varepsilon)|\mathcal{D}|.$$

In case of ε -sparse local cost functions, the SAT problem on $\mathcal{D}$ can be simplified to only include

$$\mathcal{D}_{\neq 0} = \{\delta \in \mathcal{D} \mid \Delta f_\delta \neq 0\}$$

by repeatedly applying the following constraint simplification rules on any $\delta_0 \in \mathcal{D}$ such that $\Delta f_{\delta_0} = 0$:

- for every parent $\delta_p \in \text{parents}(\delta_0)$ and child $\delta_c \in \text{children}(\delta_0)$, remove the parent-child constraints between δ_p and δ_0 and between δ_0 and δ_c . Insert in their place a parent-child constraint between δ_p and δ_c .
- for every non-compatible sibling event $\delta_s \in \mathcal{D}$, $\delta_s \neq \delta_0$, remove the compatibility constraint between δ_0 and δ_s . Insert in its place a compatibility constraint between δ_s and δ_c for all $\delta_c \in \text{children}(\delta_s)$.

This reduces the SAT or SMT problem to a problem with $|\mathcal{D}_{\neq 0}| = \varepsilon|\mathcal{D}|$ variables and at most $O(\min(|\mathcal{D}|, \varepsilon^2|\mathcal{D}|^2))$ constraints.

With the completion of this section, we have described an equivalent computation on $\mathcal{D}$ for every step of a GTS-based optimisation problem:

1. a rewrite that can be applied on a graph G can be added as an event to $\mathcal{D}$,
2. a graph G' that results from a sequence of rewrites can be recovered from $\mathcal{D}$ using `FlattenHistory`,
3. the set of all graphs reachable from events in $\mathcal{D}$ can be expressed as a SAT problem; depending on the cost function, the optimisation over that space can then take the form of an SMT problem.

In essence, using the confluent persistent data structure $\mathcal{D}$ we replace a naive, exhaustive search over the space $\mathcal{G}$ of all graphs reachable in the GTS with a SAT (or SMT) problem – solvable using highly optimised dedicated solvers that could in principle handle search spaces with up to millions of possible rewrites [340].

5.5. Bounding the search space size

We show in this section that under some assumptions on the GTSs that hold in the use cases of interest to quantum compilation, there is a provable gap between the size of the search space of all reachable graphs in the GTS and the size of the corresponding confluent persistent data structure $\mathcal{D}$.

Let us introduce first the notion of *overwriting rewrites*.

Definition 5.3 *Overwriting rewrite*

For two rewrites r_1 and r_2 , we say that r_2 *overwrites* r_1 , written $r_1 \rightarrow r_2$, if the deletion set V_2^- of δ_2 includes a vertex of the vertex set V_1 of the replacement graph of r_1

$$V_2^- \cap V_1 \neq \emptyset.$$

The definition can identically be applied to events. In this case, the overwriting events are precisely given by the parent-child relation: the set of all overwriting events of $\delta \in \mathcal{D}$ is by definition the set of parents $P(\delta)$ of δ .

Our argument relies on the comparison of asymptotic bounds for the sizes of two sets $\mathcal{G}_\Delta$ and $\mathcal{D}_\Delta$, which we now define. Consider a GTS and an input graph G . A graph G' is reachable from G within depth $\Delta \geq 0$ if there is a sequence of rewrites $r_1, \dots, r_\ell$ in the GTS from G to G' such that all subsequences $r_{\beta_1}, \dots, r_{\beta_k}$ formed of overwriting rewrites $r_{\beta_i} \rightarrow r_{\beta_{i+1}}$ have length at most $k \leq \Delta$.

The set $\mathcal{G}_\Delta$ is the set of all graphs reachable within depth Δ . We derive:

1. a lower bound on the size of $|\mathcal{G}_\Delta|$, the space of all graphs reachable in at most Δ rewrites from some input G , and
2. an upper bound on the size of the equivalent confluent persistent data structure $\mathcal{D}_\Delta$, i.e. such that

$$\mathcal{G}_\Delta = \{flat(D) \mid D \in \Gamma(\mathcal{D}_\Delta)\}.$$

In order to obtain bounds, we will introduce hypotheses that the GTSs must satisfy. Throughout this section, we will illustrate and motivate the restrictions that they impose in the following two use cases.

Use case 1: ζ -complete GTS

The first GTS we consider can be defined on any graph domain that has a notion of graph size (e.g. based on number of nodes, number of edges, etc.¹) and for any graph size ζ . The GTS is such that for any subgraph $H \subseteq G$ of size $|H| = \zeta$, there is at least one transformation rule in the GTS that matches H . We will call this case *CompleteGTS*.

1. The only constraint on the notion of graph size $|\cdot|$ is that it must be compatible with the subgraph relation: if $G \subseteq G'$, then $|G| \leq |G'|$. ↩

This is the use case of quantum superoptimisation discussed in section 3.1 and used for benchmarking in section 4.7. In those cases, the transformation rules are

obtained by enumerating all small circuits up to a certain size ζ , thus guaranteeing that any subcircuit of size ζ will be matched by the GTS.

Note that there is also an (obvious) upper bound on the number of transformation rules that can match on any given subgraph: the total number of transformation rules in the GTS.

Use case 2: single-rule GTS in a uniform domain

At the other extreme of the GTS spectrum, we can consider a GTS made of a single (arbitrary) transformation rule. In this case, we require that graphs are drawn from a domain uniformly at random, so that for any subgraph $H \subseteq G$, all patterns of size $|H|$ are equally likely. We will call this case *SingleRuleGTS*.

In this case, we will not show that our hypotheses hold for *all* inputs, but rather that they hold with a high probability. We will phrase our statements as a function of $\epsilon > 0$ and will require that they hold with probability $1 - \epsilon$ for randomly drawn G .

This regime is interesting as it is the simplest instance of problem domains for which few assumptions can be made about the GTS themselves, but all inputs are expected to be equally likely.

Lower bound on the naive search tree

The event history of the set of graphs $\mathcal{G}_\Delta$ defines a tree T_Δ , where $\mathcal{G}_\Delta$ are the nodes and $G_p \in \mathcal{G}_\Delta$ is the parent of $G_c \in \mathcal{G}_\Delta$ if there is a rewrite r rewriting G_p to G_c in the GTS. Paths in T_Δ are sequences of rewrites. We call T_Δ the *naive* search tree of the GTS. We wish to derive a lower bound for $|T_\Delta|$.

Graph partitioning

For fixed search depth Δ , let $n > 0$ be the largest integer such that for all graphs $G' \in T_\Delta$, there exists disjoint subgraphs $\Pi_1, \dots, \Pi_n$ of G' that satisfies the following property, for all $1 \leq i \leq n$:

there exists a rewrite in the GTS that can be applied to Π_i .

Hypothesis 1 *Linear scaling of n*

For a fixed GTS, a fixed depth Δ and a family of input graphs G , we have the scaling $n = \Theta(|G|)$.

We conjecture that this scaling holds for many GTSs of interest:

Proposition 5.5

Hypothesis 1 holds for `CompleteGTS` and for `SingleRuleGTS` probabilistically.

Proof. In *CompleteGTS*, it suffices to partition any input G into $n = \lfloor \frac{|G|}{\zeta} \rfloor = \Theta(|G|)$ disjoint subgraphs of size at least ζ . Each subgraph will match a rule of the GTS by definition.

For *SingleRuleGTS*, let $\epsilon > 0$. Let $|L|$ be the size of the left hand side L of the GTS rule. By assumption, for any subgraph H of size $|L|$ of an input G , there is a constant probability p that the rule matches H . For a subgraph H of size $k|L|$, the probability of the rule not matching in H is $(1 - p)^k$. Picking

$$k > \frac{1}{1 + \delta} \frac{\ln \epsilon + \ln \frac{|L|}{|G|}}{\ln(1 - p)}$$

for some $\delta > 0$ ensures that whenever $\delta > -\frac{\ln k}{k \ln(1 - p)}$ (i.e. for k large enough),

$$\begin{aligned} k - \frac{\ln k}{\ln(1 - p)} &> (1 + \delta)k > \frac{\ln \epsilon + \ln \frac{|L|}{|G|}}{\ln(1 - p)} \\ \Leftrightarrow k &> \frac{\ln \frac{\epsilon}{n}}{\ln(1 - p)}, \end{aligned}$$

where $n = \frac{|G|}{k|L|}$ was chosen. It follows that a partition of G into $\lfloor n \rfloor$ disjoint subgraphs of size at least $k|L|$ satisfies the hypothesis with probability $1 - \epsilon$.

Lower bound on $|T_\Delta|$

Fix the tree depth Δ and the GTS. Any rewrite from the GTS removes at most a constant K number of vertices from the graph it applies on. Thus, Any graph in T_Δ is at least of size $(1 - K\Delta)|G| = \Theta(|G|)$. Let n' be the smallest value of n for a graph $G_{min} \in \mathcal{G}_\Delta$. Whenever hypothesis 1 applies, we have $n' = \Theta(|G_{min}|) = \Theta(|G|)$.

For each $G' \in T_\Delta$ and Π_i , pick a rewrite $r_i(G')$ that applies to Π_i and let R be the set of all such rewrites

$$R_\Delta = \{r_i(G') \mid 1 \leq i \leq n', G' \in T_\Delta\}.$$

We can consider the subtree $T'_\Delta \subseteq T_\Delta$ that only contains graphs obtained by applying rewrites in R_Δ .

For $\Delta = 1$, the search tree T'_Δ will contain $2^{n'}$ graphs: for each subgraph Π_i of the input graph G , we can choose to either apply $r_i(G)$ or not².

By repeating Δ times the search tree of size $2^{n'}$ for $\Delta = 1$, we obtain a lower bound

$$|\mathcal{G}_\Delta| = |T_\Delta| \geq |T'_\Delta| = (2^{n'})^\Delta.$$

We frame this result as the following proposition.

2. Note that this counting is already an act of clemency: we are not counting all permutations of the rewrites, which would be considered separately by a naive exploration that applies one graph rewrite at a time. In this case, the search tree for $\Delta = 1$ would contain $n! \cdot 2^n$ graphs. ↩

Proposition 5.6 *Lower bound for $|\mathcal{G}_\Delta|$*

The naive search tree size $|\mathcal{G}_\Delta|$ is in $\Omega(2^{\Delta \cdot n'})$.

As a result, for any GTS satisfying Hypothesis 1 the size of $\mathcal{G}_\Delta$ grows at least exponentially with input graph size G and search depth Δ .

Upper bound on the factorised search space

Consider two rewrites r_1 and r_2 . If neither overwrites the other, i.e. $r_1 \not\rightsquigarrow r_2$ and $r_2 \not\rightsquigarrow r_1$, then the order in which they are applied is irrelevant (see also Proposition 5.2). The persistent data structure $\mathcal{D}$ uses this symmetry explicitly when exploring the set of reachable graphs.

This drastically reduces the size of the event history of $\mathcal{D}_\Delta$. The event history defines a directed acyclic graph F_Δ that is the equivalent for $\mathcal{D}_\Delta$ to the naive search tree T_Δ of $\mathcal{G}_\Delta$. The vertices of F_Δ are the flattened histories of events in $\mathcal{D}_\Delta$:

$$V(F_\Delta) = \{flat(\{\delta\}) \mid \delta \in \mathcal{D}_\Delta\},$$

with an edge from $flat(\{\delta_p\})$ to $flat(\{\delta_c\})$ if there is a parent-child relation $\delta_p \in P(\delta_c)$. We call F_Δ the *factorised search space* of $\mathcal{D}_\Delta$.

By construction, any graph $G' \in V(T)$ in the naive search tree maps injectively to a subgraph $S \subseteq F_\Delta$ of the factorised search space, given by the subgraph of F induced by the rewrites on the path from G to G' in T .

Whereas our earlier discussion was focused on proving a lower bound for the size of the search tree, we now show an upper bound on the number of graphs $|F_\Delta|$ in the factorised search space.

Graph covering

Instead of considering partitions of graphs in T_Δ as we did above, we now consider *coverings* of graphs G' in F_Δ , i.e. a set of subgraphs $\Gamma_1, \dots, \Gamma_n$ such that their union is G' but that might not be disjoint.

Let m , s and γ be parameters and fix a covering $\Gamma_1, \dots, \Gamma_m$ for each graph $G' \in V(F_\Delta)$ such that:

- for all G' and all $1 \leq i \leq m$, there are at most s applicable rewrites to the i -th covering set Γ_i of G' . Furthermore, all rewrites within Γ_i are mutually exclusive, i.e. they modify a shared subgraph so that it is never possible to apply more than one rewrite among the (up to s) applicable ones³;
- for all G' and for all rewrites r that apply to G' , there is $1 \leq i \leq m$ such that r applies to Γ_i (i.e. the matching subgraph of r is fully contained within one

3. This can always be made to hold by replacing any set of mutually disjoint rewrites with their cartesian product, in effect viewing the application of multiple disjoint rewrites as one large rewrite. This comes at the cost of a larger value for s . ↩

of the coverings). Furthermore, the matching subgraph of r overlaps with at most γ distinct Γ_i ;

- for all rewrites r that apply to the covering set Γ_i of a graph G' , the image $r(\Gamma_i) \subseteq \Gamma'_i$ must be a subgraph of the i -th covering subgraph Γ'_i of $r(G')$.

The first condition is satisfied whenever the size of the coverings can be bounded: in that case s can be chosen based on the number of distinct subgraphs that can be contained in a covering set and the number of rules that can apply to each. The second condition is related to the connectivity between the covering sets: γ can thus often be derived by considering how many neighbours a covering set has, and how many of those neighbours can a match of a GTS rule span.

The third condition above can be understood as “rewrites must preserve the coverings”. In other words, the coverings are chosen such that a graph mutation produced by the application of a rewrite $r \in E(F_\Delta)$ on G is always contained within a single covering subgraph of $r(G')$.

Hypothesis 2 *Linear scaling of m and constant s, γ*

For a fixed GTS, a fixed depth Δ and a family of input graphs G , we have the scaling $m = \Theta(|G|)$ and $s = \Theta(1)$ and $\gamma = \Theta(1)$.

These conditions along with Hypothesis 2 are somewhat restrictive and future work should explore how to relax them. For our use cases `CompleteGTS` and `SingleRuleGTS`, we restrict our considerations to the special case where the graph domain is quantum circuits. We make the further simplifying assumptions (similar results can be obtained with variations on these assumptions)

- transformation rules have two-qubit circuits as left and right hand sides, and
- the number of qubits on the inputs is fixed (i.e. the number of gates on each qubit scales with circuit size).

Define ℓ to be the largest number of gates on any one qubit in the left hand sides of the GTS transformation rules. Consider a partition of the gates on each qubit into sequences of ℓ gates. We can obtain a covering of a quantum circuit G by considering covering sets $\Gamma_1, \dots, \Gamma_m$ defined for all $1 \leq i \leq m$ such that for all matches H of a left hand side of a transformation rule of the GTS, $H \subseteq \Gamma_i$ if and only if there is a gate v in H such that v is in the i -th sequence of ℓ gate on some qubit in G . Imposing the condition that rewrites must preserve coverings, the covering of the input fixes the covering of all reachable graphs G' in the GTS.

Proposition 5.7

Restrict the graph domain to quantum circuits, the GTS to two-qubit rules and the inputs to a fixed number of qubits. Then hypothesis 2 holds for `CompleteGTS` and for `SingleRuleGTS`.

Proof. Let G be the input circuit with q qubits. Consider the covering $\Gamma_1, \dots, \Gamma_m$ of G as constructed above. By construction $m = \lceil N/\ell \rceil = \Theta(|G|)$, where N is the maximum number of gates on a qubit of G .

The covering set Γ_i contains the set V_i of gates composed of the i -th sequence of ℓ gates for each qubit in G . Furthermore, for all $v \in V_i$, if H is a match of a two-qubit rule that contains v , then H may contain at most $2\ell - 1$ other gates. Hence by construction, $|\Gamma_i| \leq 2\ell|V| = 2q\ell^2 = \Theta(1)$. This is a constant and thus there is a constant $s = \Theta(1)$ such that for all $1 \leq i \leq m$ there are most s matches of a two-qubit rule that intersect Γ_i .

Finally, any match H spans two qubits; the gates on each qubit (at most ℓ) may belong to at most two distinct sequences of ℓ gates of that qubit. Thus, any match H spans at most $\gamma = 4$ distinct covering sets. These arguments made no assumption on properties of the rule set and thus apply equally to `CompleteGTS` and `SingleRuleGTS`.

Upper bound on $|F_\Delta|$

The preservation of coverings under rewrites allows us to consider a covering of $V(F_\Delta)$: for each $1 \leq i \leq m$, let $F_\Delta^{(i)} \subseteq V(F_\Delta)$ be the set of graphs in $V(F_\Delta)$ that are the result of a rewrite of its i -th covering subgraph. Every graph in $V(F_\Delta)$ is the result of a rewrite on some covering subgraph i , or is the input graph G . So, from a bound $U_\Delta \geq |F_\Delta^{(i)}|$ for all $1 \leq i \leq m$, we can obtain a bound

$$|V(F_\Delta)| \leq 1 + m \cdot U_\Delta.$$

The bound U_Δ can be obtained recursively: $U_1 = s$ upper bounds by definition the number of rewrites in any covering subgraph of the root graph G , and thus the number of graphs in $F_1^{(i)}$. We then proceed by induction for $1 < \delta \leq \Delta$.

A rewrite $r \in E(F_\Delta)$ overlaps with at most γ other covering subgraphs. It can overwrite at most one previous rewrite for each subgraph, and thus will have at most γ parent graphs in sets $F_{\Delta-1}^{(i_1)}, \dots, F_{\Delta-1}^{(i_\gamma)}$. Each of the γ sets is of size at most $U_{\delta-1}$. Furthermore, there are at most s rewrites in any covering subgraph. We thus obtain the recursion:

$$U_\delta \leq s \cdot U_{\delta-1}^\gamma,$$

Unrolling the recursion, we can write this as

$$U_\delta \leq s^{1+\gamma+\gamma^2+\dots+\gamma^{\delta-1}} = s^{\frac{\gamma^\delta-1}{\gamma-1}} = s^{\Theta(\gamma^{\delta-1})}.$$

Recalling that by construction $|F_\Delta| = |\mathcal{D}_\Delta|$, we obtain:

Proposition 5.8 *Upper bound for $|\mathcal{D}_\Delta|$*

The factorised search space size $|\mathcal{D}_\Delta|$ is in $m \cdot s^{\Theta(\gamma^{\Delta-1})}$.

Discussion and empirical exploratory analysis

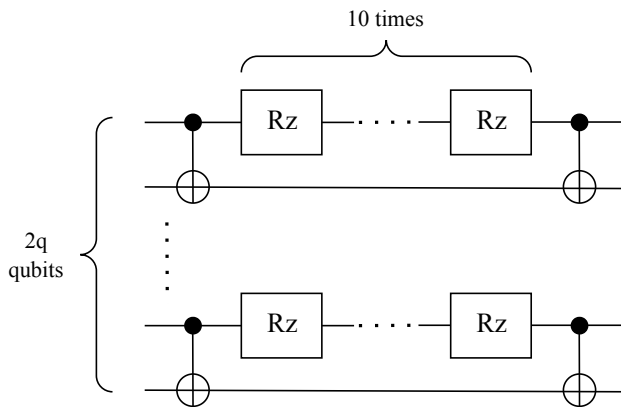
We have derived bounds on the size of the search spaces and shown that under some assumptions on the properties of the GTS, the factorised search space grows *linearly* in the size of the input graph G . This stands in stark contrast to the lower bound of the naive search tree, which scales exponentially with the size of the input graph.

However, when considering the overall optimisation problem of finding the optimal solution over the set of reachable graphs in a GTS, the exponential overhead does not disappear: it is rather shifted to the extraction phase that relies on a SAT solver. It is therefore an open question whether the factorised search space can be used to improve optimisation problems on GTSs.

To this end, we devise a simple numerical experiment that assesses the potential of using the unfolding construction as presented in this chapter in the context of quantum computation optimisation.

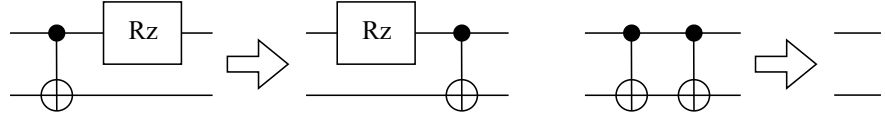
The toy problem. We consider a very simple circuit optimisation problem that is designed to require a deep search space (i.e. a large number of rewrites) to be solved. This will exacerbate the scaling difference between an optimiser that must traverse the naive search space and another that relies on the factorised representation instead.

The inputs are quantum circuits composed of two-qubit CX and single-qubit Rz rotation gates. The angles of the rotations are not relevant and set randomly. They are of the following form:



i.e. each pair of subsequent qubits have 2 CX gates at either end and 10 Rz rotation in-between, on the control qubits of the CX gates. These circuits admit a

very simple optimisation that can be expressed by the following two transformation rules:



Given the objective of minimising the number of CX gates, the optimiser must commute the leftmost CX gates through all of the rotation gates, until the two CX on each qubit are adjacent and cancel out. We study the performance of the optimisers as we increase the number $2q$ of qubits in the circuit.

Optimisers. We define two optimisers. *Badger* is a backtracking search through the naive search space of reachable graphs in the GTS: starting from the input, the search space is expanded by computing all possible rewrites at a given state. States with the lowest cost function are processed first. This is similar to an A* search [161].

Seadog on the other hand performs the backtracking search on the factorised search space instead: when expanding a state of the search space, only rewrites that overlap with the last rewrite are considered and added to the search space, as discussed in section 5.4. In a second phase, the search space is encoded as a SAT problem that is solved using Z3 [235].

4. As a Python package on [PyPI](#) and a rust crate on [crates.io](#). ↩

The *Badger* optimiser is released and publicly available as part of the open-source TKET package⁴. The *Seadog* optimiser on the other hand is still in early development; more benchmarks and a release will follow.

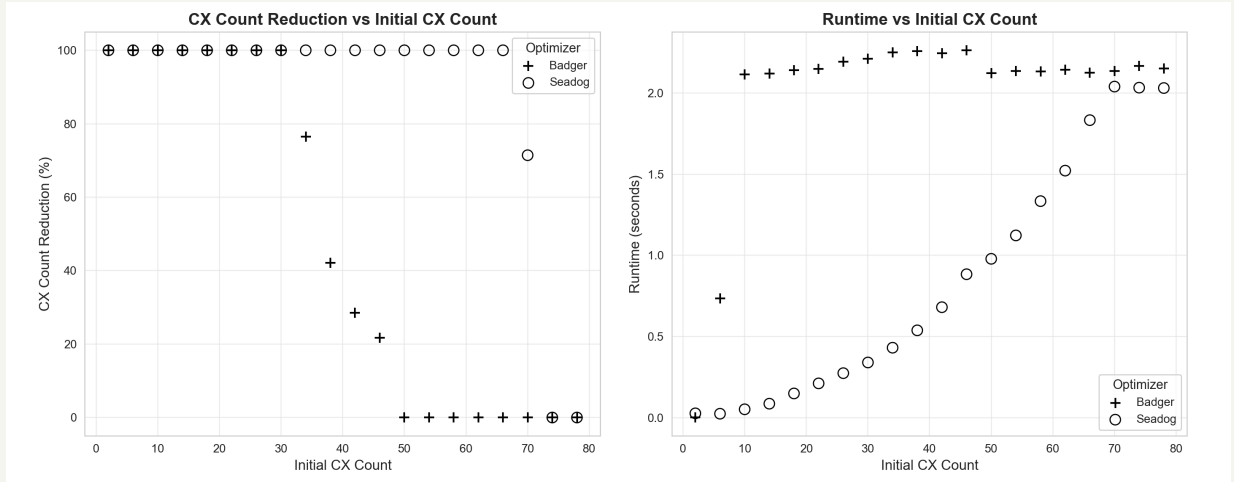
Results. We ran the experiment on an Apple M3 Max CPU (4.05GHz) for inputs between 2 (12 gates) and 78 qubits (468 gates). Both optimisers ran on a single core. For each instance, we set a timeout of 2 seconds and report the relative CX gate reduction, i.e.

$$\frac{CX_{\text{init}} - CX_{\text{final}}}{CX_{\text{init}}}.$$

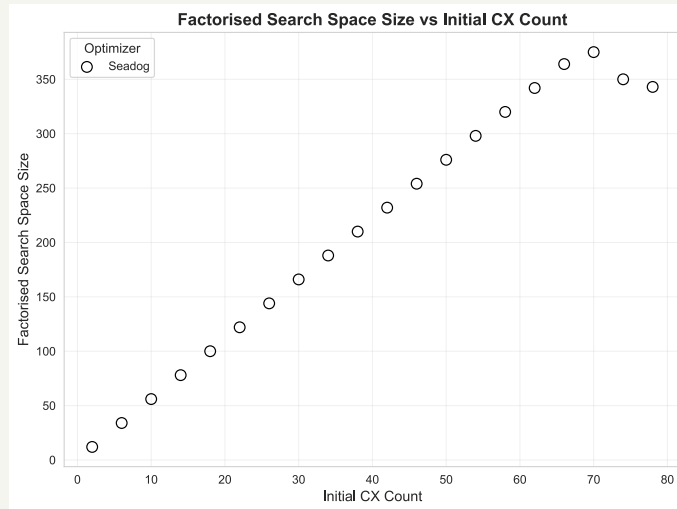
The results are shown in the figures on the right.

Discussion. On the left, we observe that both optimisers are able to find the optimum for circuits with up to 30 CX gates. Beyond that point, the time limit starts impacting *Badger* performance, which drops continuously and reaches 0% for inputs of 50 CX gates and above. *Seadog* on the other hand does not time out and is able to explore the entire (factorised) search space exhaustively up until 70 CX gates.

Observe that the *Badger* optimiser reaches the time limit for as few as 10 CX . Indeed, the complete naive search space size can be calculated to have 12^q states (each pair of qubits can be in one of 12 states). For $2q = 6$ we get 1728 states, but this already reaches over 20'000 states for $2q = 8$.



CX gate count reduction (left) and runtime (right) for the Badger and Seadog optimisers. 100% gate count reduction is optimal. A timeout was set to 2 seconds.



Size of factorised search space for Seadog.

On the other hand, the factorised search space will only contain 12 states for each qubit pair. This results in a linear scaling of the search space size, as can clearly be seen in the second figure.

Where the runtime exceeds the 2 second timeout, this is due to pre- and post-optimisation steps such as memory allocation/deallocation, I/O, file parsing etc that are included in the measurements. The quadratic runtime scaling that we observe in Seadog is due to a hash function that is run on every state of the search space to detect and discard duplicates: as the number of states in the search space grows linearly with input size and each state requires a hash in linear time, the overall runtime grows quadratically. Future work may be able to address this issue by designing updateable hash functions that do not require the full graph to be rehashed when applying a local rewrite.

Future work should also investigate how to scale Seadog to larger input sizes on a broader class of problems. We have observed that the SAT-based extraction phase of Seadog corresponds to less than 1% of the runtime budget (under 15ms for all input sizes). Whilst being asymptotically exponential in the worst case, it is thus not currently a bottleneck. On the other hand, the number of states visited per second in the exploration phase is currently up to $10\times$ slower for Seadog compared to Badger. Further investigations into the causes of this are still required, but we expect that large performance improvements can be realised on the current implementation and as a result could scale to larger inputs.

Future Work and Conclusions

The time has now come to conclude this thesis. In summary, our claim is that given

1. the modularity and expressiveness that quantum compilers will require to simultaneously express higher level abstractions, hardware primitives and interleaved quantum classical computation (cf. sections 3.3, 2.3, and 2.4),
2. the challenge of scaling up quantum programs sizes to make the most of the computational capabilities of upcoming hardware (cf. sections 1.1 and 2.2),
3. the linearity restrictions that quantum data imposes on the compiler's intermediate representation (IR) of the computation (cf. sections 2.1, 3.3, and 3.4),

graph transformation systems (GTS) are uniquely positioned to serve as the backbone of a quantum compilation framework.

To this aim, chapter 3 presented minIR, a graph-based compiler IR with explicit support for linear types. To go along with it, we proposed the first formalisation of graph transformation semantics that preserve linearity.

Chapters 4 and 5 built on this foundation and solved two critical scaling problems for the adoption of GTS techniques in quantum compilers.

Pattern matching. Successful implementations of GTSs for quantum circuit optimisation rely on thousands to hundreds of thousands of transformation rules [332, 333], for which techniques matching one pattern at a time become a significant bottleneck. Chapter 4 presented an approach based on state automata with an asymptotic runtime that is independent on the number of patterns. This resulted in a 20x speedup for a real-world pattern matching task of direct utility to quantum compilers.

Efficient rewrite space exploration. Applications of GTS to quantum compilation distinguish themselves – unfortunately – by a lack of successful rewriting strategies or other rule control mechanisms. Consequently, the optimisation of quantum computations is framed as a search problem over the space of all reachable graphs in the GTS. Chapter 5 introduced a novel confluent persistent data structure that uses the structure of the rewrite search space to speed up its exploration. In typical applications, the *factorised* search space thus obtained is conjectured to grow linearly with the size of the input – an exponential improvement

over the naive search strategy, without which GTS-based compiler optimisations on real-world computations with thousands of gates will be infeasible.

In both cases, the guarantees that linear values provide and that minIR enforces translate into asymptotic runtime guarantees that cannot be derived otherwise. In the absence of linearity, the pattern matching of chapter 4 becomes an NP-hard problem; meanwhile, the graph rewriting space considered in chapter 5 would grow super-exponentially and require pruning heuristics for the extraction problem, as studied in [335] and [35].

Combined, these contributions lay the groundwork for a quantum compiler platform that is **modular** in the hardware primitives, high-level programming abstractions and transformation rules that it can model, and **scalable** in the size of the computation and number of rules that it can match and optimise over. Work on such a platform is well underway within the TKET2 open-source compiler, available on [GitHub](#).

Further work could take many directions. The graph transformation semantics of chapter 3 that are presented operationally could for example be categorified and generalised. This would open many promising bridges and parallels to work in related domains, such as string diagrams, DPO-based GTs and even the family of ZX calculi.

There are also immediate opportunities in extending the work of chapters 4 and 5, in particular around weakening the assumptions that had to be made on the structure of the graph, respectively on the properties of the GTS and graph domain. In both cases, a more in-depth study of how the runtime of actual implementations depend on properties of the inputs would be very informative. We suspect from anecdotal observations that many assumptions we have imposed can be relaxed with little impact on performance – conversely, there may be large variations in runtimes for different regimes within the asymptotic guarantees of our results.

Another crucially important question that this thesis has not addressed is the choice of transformation rules. Beyond the results of [332] and [333] that we have referred to repeatedly throughout this corpus, very recent work by Amy and Lunderville [10] has present what amounts to the first inroads into hybrid classical-quantum optimisations. Developing comprehensive transformation rules for hybrid computations would present significant a significant advance for the field.

Among the myriad of options, we opt to conclude this thesis with the discussion of two particularly promising avenues for future work. The first (section 6.1) relates to increasing the expressivity of the pattern matching language; such an extended framework would also enable fast pattern matching *directly* on the persistent data structure $\mathcal{D}$ of chapter 5, rather than having to match patterns in each graph of $D \subseteq \mathcal{D}$ separately.

The second (section 6.2) is a proposal to use the persistent data structure of chapter 5 for large scale distributed graph rewriting. With this, the optimisation of quantum computations could be distributed across multiple machines, potentially scaling up to high-performance computing (HPC) clusters and opening the door to optimisation capabilities that could significantly advance the state of the art of quantum circuit optimisation.

6.1. More expressive pattern matching

Pattern matching as defined in chapter 4 is the problem of finding pattern embeddings $P \hookrightarrow G$ for patterns from a fixed set of patterns $P \in \mathcal{P}$. We are interested in lifting two limitations of this definition.

Firstly, it would be desirable to be able to define patterns that are not a concrete graph instance, but instead a (potentially infinite) family of graphs. Examples of such pattern families that could be useful in quantum computing are

- “a sequence of gates that commute with each other”, or
- “a subgraph that only contains Clifford gates”, or
- “all operations within the body of a loop”.

To express these patterns as concrete graph instances would require an infinite number of graphs. The study of pattern languages that allow the expression of such higher-level graphs is a mature field of graph transformations, with tools such as GrGen.NET [148] offering many advanced capabilities. It would be of great interest to establish what classes of pattern languages could be supported by generalisations of the state automaton approach presented in chapter 4.

Secondly, our approach currently only supports linear values, and thus in its current form is unsuitable for hybrid quantum-classical computations. Coincidentally, supporting non-linear values is very similar to finding embeddings $P \hookrightarrow \mathcal{D}$ of patterns into the confluent persistent data structure $\mathcal{D}$ of chapter 5. The case of a non-linear value that is used multiple times in a computation is syntactically very similar to having to consider a value in $\mathcal{D}$ that may be connected to operations in different ways, depending on the variant of the “multiverse” of equivalent graphs that are stored simultaneously in $\mathcal{D}$.

Pattern matching generalisation

The following generalisation of pattern matching might be able to achieve these two goals whilst still being compatible with the state automaton approach that we presented. We suggest defining patterns and how they match using three concepts:

Constraints. A pattern is given by a set of constraints $C_1, \dots, C_n$. They encode the conditions under which a pattern matches. They would for instance assert that two vertices are connected by an edge, or that a vertex is of a certain type. A pattern that is a concrete graph P would then at a minimum have a constraint for each edge in P .

Constraints correspond to edges (transitions) in the state automaton. Pattern matching proceeds by evaluating all outgoing constraints from the current state, and proceeds to the states for which the respective constraint is satisfied.

Indexing schemes. An indexing scheme assigns each object (e.g. vertex) in the patterns a unique key in $\mathcal{K}$, and each object (e.g. vertex) in the input domain G a unique value $\mathcal{V}$. Embeddings of patterns into G are then given by key-value maps $\mathcal{K} \rightarrow \mathcal{V}$, mapping keyed objects from patterns to objects in the input domain.

Each constraint C has a set of keys associated with it; C can then be evaluated by passing it all the values in $\mathcal{V}$ bound to its keys.

Indexing schemes are designed to give overlapping patterns the same key on their overlap, so that the overlap must only be matched once. This models how in chapter 4, patterns are clustered into patterns that share the same contracted tree and are differentiated by their contracted string tuples only.

Key-value map expansion. Indexing schemes abstract away the pattern and input data in such a way that the pattern matcher only needs to keep track of key-value maps $\mathcal{K} \rightarrow \mathcal{V}$. These maps can be created recursively using an expansion function

$$\text{expand}(\varphi, D) = \{\varphi'_1, \dots, \varphi'_n\}.$$

This provides all the ways in which the domain of definition $\text{dom}(\varphi)$ of an index map φ can be extended. The returned set of new index maps should coincide with φ on $\text{dom}(\varphi)$ but expand their domain of definition to include new keys $\mathcal{K}$. By making it possible to extend φ in more than one way, we can model the existence of non-linear values (i.e. the index map could be extended to any of the operations that uses a certain value v), as well as the fact that a persistent data structure such as $\mathcal{D}$ may be keeping track of multiple versions of the graph, and thus expand a key in multiple ways.

Execution of the pattern matcher

Starting from an empty key-value map $\varphi_\emptyset$ at the root state of the state automaton, the pattern matcher keeps track of a set of key-value maps, along with for each map the state it is in. It then proceeds by repeatedly performing the following two actions:

1. Expand the domain of definition of a key-value map φ by calling `expand`;
2. Evaluate the constraints for a key-value map φ ; if the constraint is satisfied, move φ to the next state, otherwise try another constraint. If no constraint is satisfied, delete φ .

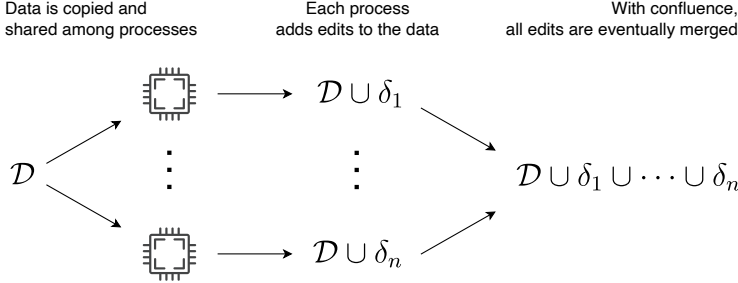
The performance of the pattern matcher will be highly dependent on choosing a smart ordering of these two actions, as well as prioritising the right key-value maps to be expanded and evaluated.

With this proposal, it would appear possible to combine the fast state automaton-based approach of chapter 4 and its scaling to a very large number of patterns, with a more expressive pattern language and support for non-linear types as well as persistent graph rewriting. An implementation of this is currently being worked on in the open-source portmatching project, available on [GitHub](#).

6.2. Massively parallel graph rewriting

Persistent data structures – and particularly fully and confluent persistent ones – are well-suited for distributed applications. In persistent data structures, data can

always be added but never deleted, and is thus immutable. This removes the need for locks and synchronisation primitives across processes. Furthermore, using confluence, edits can be made concurrently in different processes and then eventually merged asynchronously, as follows:



The contributions presented in chapter 5 thus translate directly into a proposal for a massively parallel graph rewriting system. In summary, we have shown that graph rewrites can be tracked in a persistent data structure $\mathcal{D}$ in the form of edits δ . New edits added to $\mathcal{D}$ can refer to previous edits, and thus create an acyclic edit history. Sets of edits $\mathcal{D}$ and $\mathcal{D}'$ can also be merged (confluence) and as a result, new edits that build on top of edits from both $\mathcal{D}$ and $\mathcal{D}'$ can be defined.

We describe in slightly more detail what a massively parallel graph rewriting architecture might look like.

Inter-process communication

During the rewriting process, the set of processes that are involved must regularly broadcast the edits they have added to (their copy of) the data $\mathcal{D}$. Such broadcasted edits must then be merged by the other processes into their respective local copies. This is required so that progress that is made by one process can be shared and expanded on top of by other processes.

Technologies such as message-passing interface (MPI) [115] would be well-suited to such inter-process communications. To reduce the number of messages that senders and receivers must process, edits should not be broadcasted one-by-one, but rather grouped together. For this, we propose the notion of a *salient edit*, reflecting that an edit is deemed of importance.

Non-salient edits are not broadcasted as they are added to $\mathcal{D}$. When, on the other hand, an edit δ is deemed salient, it is broadcasted along with all its ancestors $A(\delta)$ (i.e. all edits that δ depends on). As the edit history deepens, it might become inefficient to broadcast all the ancestry of an edit, in which case more advanced communication protocols would have to be devised.

Finally, a procedure must be put in place to identify identical edits that may be added and/or broadcasted by different processes to avoid deduplication. Hashing techniques and hash tables are well-suited for this kind of problem.

Process types

At a minimum, the distributed graph rewriting system should distinguish between two types of processes.

The vast majority of processes would be **rewrite factories**. Their purpose is to create new edits, add them to $\mathcal{D}$ and broadcast them whenever they are deemed salient. These processes will be responsible for driving forward the search space exploration and, in the end, the optimisation. A good candidate for a rewrite factory is the pattern matching automaton of chapter 4 and its generalisation just described in section 6.1. Different processes may specialise in different transformation rule sets; others still could implement dedicated optimisations such as ZX-based optimisations or optimal Clifford synthesis (see discussion in section 2.2).

The other type of process would be a **result extractor**; a read-only process that runs the SAT-based optimisation and graph extraction algorithm of section 5.4. Such a process would run the computation at regular intervals to track the optimisation progress.

As the distributed architecture grows in complexity, more tasks and more process types may be required. It might for instance be desirable to have a process that identifies under-explored parts of the search space to direct rewrite factories in that direction.

Using such an architecture, it might be possible for the first time to scale quantum compilation workloads to large clusters of machines. This could significantly advance compilation performance of quantum programs, a particularly valuable contribution at a time where quantum computers are on the edge of utility. Nevertheless, such distributed systems often prove difficult to design and run successfully. Open questions include how to coordinate the search across processes in such a way that the most promising parts of the search space are explored whilst avoid work duplication; will communication become the bottleneck in the computation; what are the most effective transformation rules and cost functions to use; and what are the limits of modern SAT solvers on our problem of interest.

Appendix

A. Prefix trees

Our main result is achieved by reducing a tree inclusion problem to the following problem.

String prefix matching. Consider the following computational problem over strings. Let Σ be a finite alphabet and consider $\mathcal{W} = (\Sigma^*)^w$ the set of w -tuples of strings over Σ . For a string tuple $(s_1, \dots, s_w) \in \mathcal{W}$ and a set of string tuples $\mathcal{D} \subseteq \mathcal{W}$, the w -dimensional string prefix matching consists in finding the set

$$\{(p_1, \dots, p_w) \in \mathcal{D} \mid \text{for all } 1 \leq i \leq w : p_i \text{ is a prefix of } s_i\}.$$

This string problem can be solved using a w -dimensional prefix tree. We give a short introduction to prefix trees for the string case but refer to standard literature for more details [195].

One-dimensional prefix tree. Let $P_1, \dots, P_\ell \in \mathcal{A}^*$ be strings on some alphabet $\mathcal{A}$. Given an input string $s \in \mathcal{A}^*$, we wish to find the set of patterns $\{P_{1 \leq i \leq \ell} \mid P_i \subseteq s\}$, i.e. P_i is a prefix of s .

The prefix tree of $P_1, \dots, P_\ell$ is a tree with a tree node for each prefix of a pattern. The children of an internal node are the strings that extend the prefix by one character. The root of the tree is the empty string. Each tree node also stores a list of matching patterns, with each pattern stored in the unique corresponding node. Every prefix tree has an empty string node, which is the root of the tree. For every inserted pattern of length at most L nodes are inserted, one for every non-empty prefix of the pattern. Thus a one-dimensional prefix tree has at most $\ell \cdot L + 1$ nodes and can be constructed in time $O(\ell \cdot L)$.

Given an input $s \in \mathcal{A}^*$, we can find the set of matching patterns by traversing the prefix tree of $P_1, \dots, P_\ell$ starting from the root. We report the list of matching patterns at the current node and move to the child node that is still a prefix of s , if it exists. This procedure continues until no more such child exists. In total the traversal takes time $O(|s|)$, as every character of s is visited at most once.

Note that in theory the number of reported pattern matches can dominate the runtime of the algorithm. We can avoid this by returning the list of matches as an iterator, stored as a list of pointers to the tree nodes matching lists.

Multi-dimensional prefix tree. A w -dimensional prefix tree for $w > 1$ is defined recursively as a one-dimensional prefix tree that at each node stores a $w - 1$ -di-

prefix tree on the input s_1 until no child is a prefix of the input, and then recursively traversing the $w - 1$ -dimensional prefix tree on $(s_2, \dots, s_w)$. Similarly to the one-dimensional case, the list of matching patterns is stored at prefix tree nodes and reported during traversal. The traversal thus takes time $O(|s_1| + \dots + |s_w|)$, as every character of s is visited at most once.

For ℓ tuples of size w of words of maximum length L , we can bound the number of nodes of the w -dimensional prefix tree by $1 + (\ell \cdot L)^w$. The runtime and space complexity of the construction of the w -dimensional prefix tree is thus in $O((\ell \cdot L)^w)$, summarised in the result:

Proposition 99.1 *Multi-dimensional string prefix matching*

Let $\mathcal{D} \subseteq \mathcal{W}$ be a set of string tuples and L the maximum length of a string in a tuple of $\mathcal{D}$. There is a prefix tree with at most $(\ell \cdot L)^w + 1$ nodes that encodes $\mathcal{D}$ that can be used to solve the w -dimensional string prefix matching problem in time $O(|s_1| + \dots + |s_w|)$.

B. Lower bound on the number of patterns

Proposition 99.2

Let $N_{w,d}$ be the number of port graphs of width w , depth d and maximum degree $\Delta \geq 4$. We can lower bound

$$N_{w,d} > \left(\frac{w}{2e}\right)^{\Theta(wd)},$$

assuming $w \leq o(2^d)$.

In the regime of interest, w is small, so the assumption $w \leq o(2^d)$ is not a restriction.

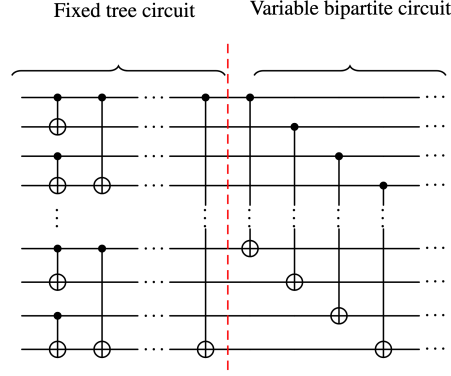
Proof. Let $w, d > 0$ and $\Delta \geq 4$ be integers. We wish to lower bound the number of port graphs of depth d , width w and maximum degree Δ . It is sufficient to consider a restricted subset of such port graphs, whose size can be easily lower bounded. We will count a subset of CX quantum circuits, i.e. circuits with only CX gates, a two-qubit non-symmetric gate. Because we are using a single gate type, this is equivalent to counting a subset of port graphs with vertices of degree 4. Assume w.l.o.g that w is a power of two. We consider CX circuits constructed from two circuits with w qubits composed in sequence:

- **Fixed tree circuit:** A $\log_2(w)$ -depth circuit that connects qubits pairwise in such a way that the resulting port graph is connected. We fix such a tree-like

circuit and use the same circuit for all CX circuits. We can use this common structure to fix an ordering of the w qubits, that refer to as qubits $1, \dots, w$.

- **Bipartite circuit:** A CX circuit of depth $D = d - \log_2(w)$ with exactly $wD/2$ CX gates, each gate acting on a qubit and a qubit.

The following circuit illustrates the construction:



All that remains is to count the number of such bipartite circuits. Every slice of depth 1 must have $w/2$ CX gates acting on distinct qubits. Every qubit 1 to $w/2$ must interact with one of the qubits $w/2 + 1$ to w , so there are $(w/2)!$ such depth 1 slices. Repeating this depth 1 construction D times and using Sterling's approximation, we obtain a lower bound for the number of port graphs of depth d , width w and maximum degree at least 4:

$$\left(\left(\frac{w}{2} \right)! \right)^D > \sqrt{w\pi} \left(\frac{w}{2e} \right)^{wD/2} = \left(\frac{w}{2e} \right)^{\Theta(w \cdot d)}$$

where we used $w = o(2^d)$ to obtain $\Theta(D) = \Theta(d)$ in the last step.

Bibliography

1. Scott Aaronson and Daniel Gottesman. 2004. Improved simulation of stabilizer circuits. *Physical Review A* 70, 5 (November 2004, 052328). doi: [10.1103/PhysRevA.70.052328](https://doi.org/10.1103/PhysRevA.70.052328)
2. Nabila Abdessaied, Matthew Amy, Mathias Soeken and Rolf Drechsler. 2016. Technology Mapping of Reversible Circuits to Clifford+T Quantum Circuits. In *2016 IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL)*, May 2016. IEEE, 150--155. doi: [10.1109/ismvl.2016.33](https://doi.org/10.1109/ismvl.2016.33)
3. Samson Abramsky and Bob Coecke. 2008. Categorical quantum mechanics. arXiv: [0808.1023](https://arxiv.org/abs/0808.1023) [quant-ph]
4. Rajeev Acharya, Dmitry A. Abanin, Laleh Aghababaie-Beni, Igor Aleiner, Trond I. Andersen, Markus Ansmann, Frank Arute, Kunal Arya, Abraham Asfaw, Nikita Astrakhantsev, Juan Atalaya, Ryan Babbush, Dave Bacon, Brian Ballard, Joseph C. Bardin, Johannes Bausch, Andreas Bengtsson, Alexander Bilmes, Sam Blackwell, Sergio Boixo, Gina Bortoli, Alexandre Bourassa, Jenna Bovaird, Leon Brill and et al. 2024. Quantum error correction below the surface code threshold. *Nature* (December 2024). doi: [10.1038/s41586-024-08449-y](https://doi.org/10.1038/s41586-024-08449-y)
5. Faisal Alam and Bryan K. Clark. 2024. Learning dynamic quantum circuits for efficient state preparation. arXiv: [2410.09030](https://arxiv.org/abs/2410.09030) [quant-ph]
6. A. E. Allahverdyan, R. Balian and Th. M. Nieuwenhuizen. 2004. Determining a Quantum State by Means of a Single Apparatus. *Physical Review Letters* 92, 12 (March 2004, 120402). doi: [10.1103/PhysRevLett.92.120402](https://doi.org/10.1103/PhysRevLett.92.120402)
7. M. Amy, D. Maslov, M. Mosca and M. Roetteler. 2013. A Meet-in-the-Middle Algorithm for Fast Synthesis of Depth-Optimal Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 6 (June 2013, 818--830). doi: [10.1109/TCAD.2013.2244643](https://doi.org/10.1109/TCAD.2013.2244643)
8. Matthew Amy, Parsiad Azimzadeh and Michele Mosca. 2018. On the controlled-NOT complexity of controlled-NOT-phase circuits. *Quantum Science and Technology* 4, 1 (September 2018, 015002). doi: [10.1088/2058-9565/aad8ca](https://doi.org/10.1088/2058-9565/aad8ca)
9. Matthew Amy. 2019. Formal methods in Quantum Circuit Design. PhD Thesis. University of Waterloo
10. Matthew Amy and Joseph Lunderville. 2025. Linear and Non-linear Relational Analyses for Quantum Program Optimization. *Proceedings of the ACM on Programming Languages* 9, POPL (January 2025, 1072--1103). doi: [10.1145/3704873](https://doi.org/10.1145/3704873)
11. Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet and Bruno Pinaud. 2011. PORGY: Strategy-Driven Interactive Transformation of Graphs. *Electronic Proceedings in Theoretical Computer Science* 48 (February 2011, 54--68). doi: [10.4204/eptcs.48.7](https://doi.org/10.4204/eptcs.48.7)
12. Mona Arabzadeh, Mehdi Saeedi and Morteza Saheb Zamani. 2010. Rule-based optimization of reversible circuits. In *2010 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, January 2010. IEEE, 849--854. doi: [10.1109/aspdac.2010.5419684](https://doi.org/10.1109/aspdac.2010.5419684)
13. Dylan Armstrong. 2014. Memory Efficient Stream Reasoning on Resource-Limited Devices. PhD Thesis. University of Dublin, Trinity College
14. Pyreddy Mary Arpita, Kamalika Datta, Rohith Vemula and Indranil Sengupta. 2015. Optimization of reversible circuits using triple-gate templates at quantum gate level. In *2015 International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV)*, January 2015. IEEE, 120--124. doi: [10.1109/edcav.2015.7060551](https://doi.org/10.1109/edcav.2015.7060551)

15. J. M. Arrazola, V. Bergholm, K. Brádler, T. R. Bromley, M. J. Collins, I. Dhand, A. Fumagalli, T. Gerrits, A. Goussev, L. G. Helt, J. Hundal, T. Isacsson, R. B. Israel, J. Izaac, S. Jahangiri, R. Janik, N. Killoran, S. P. Kumar, J. Lavoie, A. E. Lita, D. H. Mahler, M. Menotti, B. Morrison, S. W. Nam, L. Neuhaus, H. Y. Qi, N. Quesada, A. Repeatingon and et al. 2021. Quantum circuits with many photons on a programmable nanophotonic chip. *Nature* 591, 7848 (March 2021, 54–60). doi: [10.1038/s41586-021-03202-1](https://doi.org/10.1038/s41586-021-03202-1)
16. Uwe Assmann. 2000. Graph rewrite systems for program optimization. *ACM Transactions on Programming Languages and Systems* 22, 4 (July 2000, 583–637). doi: [10.1145/363911.363914](https://doi.org/10.1145/363911.363914)
17. Jean-Christophe Aval. 2008. Multivariate Fuss–Catalan numbers. *Discrete Mathematics* 308, 20 (October 2008, 4660–4669). doi: [10.1016/j.disc.2007.08.100](https://doi.org/10.1016/j.disc.2007.08.100)
18. Fariha Azad. 2024. Tensor networks for classical and quantum simulation of open and closed quantum systems. PhD Thesis. University College London.
19. John A Bachman and Peter Sorger. 2011. New approaches to modeling complex biochemistry. *Nature Methods* 8, 2 (January 2011, 130–131). doi: [10.1038/nmeth0211-130](https://doi.org/10.1038/nmeth0211-130)
20. Miriam Backens and Aleks Kissinger. 2019. ZH: A Complete Graphical Calculus for Quantum Computations Involving Classical Non-linearity. *Electronic Proceedings in Theoretical Computer Science* 287 (January 2019, 23–42). doi: [10.4204/EPTCS.287.2](https://doi.org/10.4204/EPTCS.287.2)
21. Miriam Backens. 2014. The ZX-calculus is complete for stabilizer quantum mechanics. *New Journal of Physics* 16, 9 (September 2014, 093021). doi: [10.1088/1367-2630/16/9/093021](https://doi.org/10.1088/1367-2630/16/9/093021)
22. Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski and John van de Wetering. 2021. There and back again: A circuit extraction tale. *Quantum* 5 (March 2021, 421). doi: [10.22331/q-2021-03-25-421](https://doi.org/10.22331/q-2021-03-25-421)
23. Paolo Baldan, Andrea Corradini and Ugo Montanari. 1999. Unfolding and Event Structure Semantics for Graph Grammars. In *Foundations of Software Science and Computation Structures*, Berlin, Heidelberg. Springer Berlin Heidelberg, 73–89. doi: [10.1007/3-540-49019-1_6](https://doi.org/10.1007/3-540-49019-1_6)
24. Paolo Baldan, Andrea Corradini, Ugo Montanari and Leila Ribeiro. 2007. Unfolding semantics of graph transformation. *Information and Computation* 205, 5 (May 2007, 733–782). doi: [10.1016/j.ic.2006.11.004](https://doi.org/10.1016/j.ic.2006.11.004)
25. Paolo Baldan, Andrea Corradini, Tobias Heindel, Barbara König and Paweł Sobociński. 2014. Processes and unfoldings: concurrent computations in adhesive categories. *Mathematical Structures in Computer Science* 24, 4 (June 2014). doi: [10.1017/s096012951200031x](https://doi.org/10.1017/s096012951200031x)
26. Paolo Baldan, Andrea Corradini and Barbara König. 2008. Unfolding Graph Transformation Systems: Theory and Applications to Verification
27. Paolo Baldan, Andrea Corradini and Barbara König. 2008. A framework for the verification of infinite-state graph transformation systems. *Information and Computation* 206, 7 (July 2008, 869–907). doi: [10.1016/j.ic.2008.04.002](https://doi.org/10.1016/j.ic.2008.04.002)
28. Paolo Baldan, Thomas Chatain, Stefan Haar and Barbara König. 2008. Unfolding-Based Diagnosis of Systems with an Evolving Topology
29. Paolo Baldan, Andrea Corradini and Barbara König. 2004. Verifying Finite-State Graph Grammars: An Unfolding-Based Approach. In *CONCUR 2004 - Concurrency Theory*, Berlin, Heidelberg. Springer Berlin Heidelberg, 83–98. doi: [10.1007/978-3-540-28644-8_6](https://doi.org/10.1007/978-3-540-28644-8_6)
30. Paolo Baldan, Andrea Corradini, Barbara König and Stefan Schwoon. 2008. McMillan's Complete Prefix for Contextual Nets

31. Paolo Baldan, Alessandro Bruni, Andrea Corradini, Barbara König and Stefan Schwoon. 2010. On the Computation of McMillan's Prefix for Contextual Nets and Graph Grammars. In *Graph Transformations*, Berlin, Heidelberg. Springer Berlin Heidelberg, 91--106. doi: [10.1007/978-3-642-15928-2_7](https://doi.org/10.1007/978-3-642-15928-2_7)
32. Paolo Baldan, Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig and Michael Löwe. 1999. Concurrent Semantic of Algebraic Graph Transformations. In *Handbook of Graph Grammars and Computing by Graph Transformation*, August 1999. World Scientific, 107--187. doi: [10.1142/9789812814951_0003](https://doi.org/10.1142/9789812814951_0003)
33. Chandan Bandyopadhyay, Hafizur Rahaman and Rolf Drechsler. 2014. A Cube Pairing Approach for Synthesis of ESOP-Based Reversible Circuit. In *2014 IEEE 44th International Symposium on Multiple-Valued Logic*, May 2014. IEEE, 109--114. doi: [10.1109/ismvl.2014.27](https://doi.org/10.1109/ismvl.2014.27)
34. Sorav Bansal and Alex Aiken. 2006. Automatic generation of peephole super-optimizers. *ACM SIGARCH Computer Architecture News* 34, 5 (October 2006, 394--403). doi: [10.1145/1168919.1168906](https://doi.org/10.1145/1168919.1168906)
35. George-Octavian Bărbulescu, Taiyi Wang, Zak Singh and Eiko Yoneki. 2024. Learned Graph Rewriting with Equality Saturation: A New Paradigm in Relational Query Rewrite and Beyond. arXiv: [2407.12794](https://arxiv.org/abs/2407.12794) [cs.DB]
36. Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Physical Review A* 52, 5 (November 1995, 3457--3467). doi: [10.1103/PhysRevA.52.3457](https://doi.org/10.1103/PhysRevA.52.3457)
37. H. P. Barendregt, M. C. J. D. Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer and M. R. Sleep. 1987. Term Graph Rewriting
38. Clark Barrett and Cesare Tinelli. 2018. Satisfiability Modulo Theories
39. Sara Bartolucci, Patrick Birchall, Hector Bombín, Hugo Cable, Chris Dawson, Mercedes Gimeno-Segovia, Eric Johnston, Konrad Kielsing, Naomi Nickerson, Mihir Pant, Fernando Pastawski, Terry Rudolph and Chris Sparrow. 2023. Fusion-based quantum computation. *Nature Communications* 14, 1 (February 2023). doi: [10.1038/s41467-023-36493-1](https://doi.org/10.1038/s41467-023-36493-1)
40. Daniel Basilewitsch, Clemens Dlaska and Wolfgang Lechner. 2024. Comparing planar quantum computing platforms at the quantum speed limit. *Physical Review Research* 6, 2 (April 2024, 023026). doi: [10.1103/physrevresearch.6.023026](https://doi.org/10.1103/physrevresearch.6.023026)
41. Elisa Bäumer, Vinay Tripathi, Alireza Seif, Daniel Lidar and Derek S. Wang. 2024. Quantum Fourier Transform Using Dynamic Circuits. *Physical Review Letters* 133, 15 (October 2024, 150602). doi: [10.1103/physrevlett.133.150602](https://doi.org/10.1103/physrevlett.133.150602)
42. Johannes Bausch, Andrew W. Senior, Francisco J. H. Heras, Thomas Edlich, Alex Davies, Michael Newman, Cody Jones, Kevin Satzinger, Murphy Yuezhen Niu, Sam Blackwell, George Holland, Dvir Kafri, Juan Atalaya, Craig Gidney, Demis Hassabis, Sergio Boixo, Hartmut Neven and Pushmeet Kohli. 2024. Learning high-accuracy error decoding for quantum processors. *Nature* 635, 8040 (November 2024, 834--840). doi: [10.1038/s41586-024-08148-8](https://doi.org/10.1038/s41586-024-08148-8)
43. Niel de Beaudrap and Dominic Horsman. 2020. The ZX calculus is a language for surface code lattice surgery. *Quantum* 4 (January 2020, 218). doi: [10.22331/q-2020-01-09-218](https://doi.org/10.22331/q-2020-01-09-218)
44. Nicolas Behr. 2019. Sesqui-Pushout Rewriting: Concurrency, Associativity and Rule Algebra Framework. *Electronic Proceedings in Theoretical Computer Science* 309 (December 2019, 23--52). doi: [10.4204/eptcs.309.2](https://doi.org/10.4204/eptcs.309.2)
45. Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres and William K. Wootters. 1993. Teleporting an unknown quantum state

- via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters* 70, 13 (March 1993, 1895--1899). doi: [10.1103/physrevlett.70.1895](https://doi.org/10.1103/physrevlett.70.1895)
46. 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. 2022. PennyLane: Automatic differentiation of hybrid quantum-classical computations. arXiv: [1811.04968](https://arxiv.org/abs/1811.04968) [quant-ph]
47. James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron and others. 2011. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*
48. Max Bernstein. 2024. What's in an e-graph?. (Septempter 2024). Retrieved on 14/01/2025 (blog post) from <https://bernsteinbear.com/blog/whats-in-an-egraph/>
49. Marc Bezem, Jan Willem Klop and Roel Vrijer. 2003. *Term Rewriting Systems* (1. publ. ed.). Cambridge University Press, Cambridge
50. J Biamonte and A Nasrallah. 2023. The ZX-Calculus is Canonical in the Heisenberg Picture for Stabilizer Quantum Mechanics. arXiv: [2301.05717](https://arxiv.org/abs/2301.05717) [quant-ph]
51. Benjamin Bichsel, Maximilian Baader, Timon Gehr and Martin Vechev. 2020. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2020. ACM, 286-300. doi: [10.1145/3385412.3386007](https://doi.org/10.1145/3385412.3386007)
52. Armin Biere, Marijn J. H. Heule, Hans Maaren and Toby Walsh. 2021. *Handbook of satisfiability* (Second edition ed.). IOS Press, Amsterdam
53. Dénes András Bicztray. 2009. Compositional verification of model-level refactorings based on graph transformations. PhD Thesis. University of Leicester
54. Alexandre Blais, Jay Gambetta, A. Wallraff, D. I. Schuster, S. M. Girvin, M. H. Devoret and R. J. Schoelkopf. 2007. Quantum-information processing with circuit quantum electrodynamics. *Physical Review A* 75, 3 (March 2007, 032329). doi: [10.1103/physreva.75.032329](https://doi.org/10.1103/physreva.75.032329)
55. Dolev Bluvstein, Harry Levine, Giulia Semeghini, Tout T. Wang, Sepehr Ebadi, Marcin Kalinowski, Alexander Keesling, Nishad Maskara, Hannes Pichler, Markus Greiner, Vladan Vuletić and Mikhail D. Lukin. 2022. A quantum processor based on coherent transport of entangled atom arrays. *Nature* 604, 7906 (April 2022, 451--456). doi: [10.1038/s41586-022-04592-6](https://doi.org/10.1038/s41586-022-04592-6)
56. Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski and Fabio Zanasi. 2022. String Diagram Rewrite Theory I: Rewriting with Frobenius Structure. *Journal of the ACM* 69, 2 (March 2022, 1 - 58). doi: [10.1145/3502719](https://doi.org/10.1145/3502719)

57. Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński and Fabio Zanasi. 2022. String diagram rewrite theory II: Rewriting with symmetric monoidal structure. *Mathematical Structures in Computer Science* 32, 4 (April 2022), 511--541). doi: [10.1017/s0960129522000317](https://doi.org/10.1017/s0960129522000317)
58. Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Paweł Sobociński and Fabio Zanasi. 2017. Confluence of Graph Rewriting with Interfaces
59. Robert I. Booth, Titouan Carette and Cole Comfort. 2024. Graphical Symplectic Algebra. arXiv: [2401.07914](https://arxiv.org/abs/2401.07914) [cs.LO]
60. Agustín Borgna. 2023. Towards a compiler toolchain for quantum programs. PhD Thesis. Loria, Université de Lorraine
61. Agustín Borgna, Simon Perdrix and Benoît Valiron. 2021. Hybrid Quantum-Classical Circuit Simplification with the ZX-Calculus. In *Programming Languages and Systems*, Cham. Springer International Publishing, 121--139. doi: [10.1007/978-3-030-89051-3_8](https://doi.org/10.1007/978-3-030-89051-3_8)
62. J. Eli Bourassa, Rafael N. Alexander, Michael Vasmer, Ashlesha Patil, Ilan Tzitrin, Takaya Matsuura, Daiqin Su, Ben Q. Baragiola, Saikat Guha, Guillaume Dauphinais, Krishna K. Sabapathy, Nicolas C. Menicucci and Ish Dhand. 2021. Blueprint for a Scalable Photonic Fault-Tolerant Quantum Computer. *Quantum* 5 (February 2021), 392). doi: [10.22331/q-2021-02-04-392](https://doi.org/10.22331/q-2021-02-04-392)
63. Pierre Boutillier, Thomas Ehrhard and Jean Krivine. 2017. Incremental Update for Graph Rewriting
64. Sergey Bravyi and Alexei Kitaev. 2005. Universal quantum computation with ideal Clifford gates and noisy ancillas. *Physical Review A* 71, 2 (February 2005), 022316). doi: [10.1103/PhysRevA.71.022316](https://doi.org/10.1103/PhysRevA.71.022316)
65. Sergey Bravyi, Dan Browne, Padraic Calpin, Earl Campbell, David Gosset and Mark Howard. 2019. Simulation of quantum circuits by low-rank stabilizer decompositions. *Quantum* 3 (September 2019), 181). doi: [10.22331/q-2019-09-02-181](https://doi.org/10.22331/q-2019-09-02-181)
66. Sergey Bravyi and Dmitri Maslov. 2021. Hadamard-Free Circuits Expose the Structure of the Clifford Group. *IEEE Transactions on Information Theory* 67, 7 (July 2021), 4546--4563). doi: [10.1109/TIT.2021.3081415](https://doi.org/10.1109/TIT.2021.3081415)
67. Sergey Bravyi, Ruslan Shaydulin, Shaohan Hu and Dmitri Maslov. 2021. Clifford Circuit Optimization with Templates and Symbolic Pauli Gates. *Quantum* 5 (November 2021), 580). doi: [10.22331/q-2021-11-16-580](https://doi.org/10.22331/q-2021-11-16-580)
68. Preston Briggs and Keith D. Cooper. 1994. Effective partial redundancy elimination. *ACM SIGPLAN Notices* 29, 6 (June 1994), 159--170). doi: [10.1145/773473.178257](https://doi.org/10.1145/773473.178257)
69. Lukas Burgholzer and Robert Wille. 2021. Advanced Equivalence Checking for Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 9 (September 2021), 1810--1824). doi: [10.1109/tcad.2020.3032630](https://doi.org/10.1109/tcad.2020.3032630)
70. Giorgio Busatto, Hans-Jörg Kreowski and Sabine Kuske. 2005. Abstract hierarchical graph transformation. *Mathematical Structures in Computer Science* 15, 4 (July 2005), 773--819). doi: [10.1017/s0960129505004846](https://doi.org/10.1017/s0960129505004846)
71. Hanyan Cao, Feng Pan, Yijia Wang and Pan Zhang. 2023. qecGPT: decoding Quantum Error-correcting Codes with Generative Pre-trained Transformers. arXiv: [2307.09025](https://arxiv.org/abs/2307.09025) [quant-ph]
72. Titouan Carette, Timothée Hoffreumon, Émile Larroque and Renaud Vilmart. 2023. Complete Graphical Language for Hermiticity-Preserving Superoperators. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, June 2023. IEEE, 1--22. doi: [10.1109/LIC-556636.2023.10175712](https://doi.org/10.1109/LIC-556636.2023.10175712)

73. Titouan Carette, Emmanuel Jeandel, Simon Perdrix and Renaud Vilmart. 2021. Completeness of Graphical Languages for Mixed State Quantum Mechanics. *ACM Transactions on Quantum Computing* 2, 4 (December 2021, 1--28). doi: [10.1145/3464693](https://doi.org/10.1145/3464693)
74. Almudena Carrera Vazquez, Caroline Tornow, Diego Ristè, Stefan Woerner, Maika Takita and Daniel J. Egger. 2024. Combining quantum processors with real-time classical communication. *Nature* 636, 8041 (November 2024, 75--79). doi: [10.1038/s41586-024-08178-2](https://doi.org/10.1038/s41586-024-08178-2)
75. Shantanav Chakraborty. 2024. Implementing any Linear Combination of Unitaries on Intermediate-term Quantum Computers. *Quantum* 8 (October 2024, 1496). doi: [10.22331/q-2024-10-10-1496](https://doi.org/10.22331/q-2024-10-10-1496)
76. Parinya Chalermsook, Mayank Goswami, László Kozma, Kurt Mehlhorn and Thatchaphol Saranurak. 2018. Multi-Finger Binary Search Trees. In *International Symposium on Algorithms and Computation 2018*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: [10.4230/LIPICS.ISAAC.2018.55](https://doi.org/10.4230/LIPICS.ISAAC.2018.55)
77. K. Ch. Chatzisavvas, G. Chadzitaskos, C. Daskaloyannis and S. G. Schirmer. 2009. Improving quantum gate fidelities using optimized Euler angles. *Physical Review A* 80, 5 (November 2009, 052329). doi: [10.1103/physreva.80.052329](https://doi.org/10.1103/physreva.80.052329)
78. Donny Cheung, Dmitri Maslov and Simone Severini. 2007. Translation techniques between quantum circuit architectures. In *Workshop on quantum information processing*, 1--3
79. Cirq Developers. 2024. Cirq
80. Alexandre Clément, Nicolas Heurtel, Shane Mansfield, Simon Perdrix and Benoît Valiron. 2023. A Complete Equational Theory for Quantum Circuits. In *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, June 2023. IEEE, 1--13. doi: [10.1109/lics56636.2023.10175801](https://doi.org/10.1109/lics56636.2023.10175801)
81. Alexandre Clément, Noé Delorme and Simon Perdrix. 2024. Minimal Equational Theories for Quantum Circuits. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, July 2024. ACM, 1--14. doi: [10.1145/3661814.3662088](https://doi.org/10.1145/3661814.3662088)
82. Cliff Click and Keith D. Cooper. 1995. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems* 17, 2 (March 1995, 181--196). doi: [10.1145/201059.201061](https://doi.org/10.1145/201059.201061)
83. John Cocke. 1970. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization -*. ACM Press, 20--24. doi: [10.1145/800028.808480](https://doi.org/10.1145/800028.808480)
84. Bob Coecke and Aleks Kissinger. 2017. *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press. doi: [10.1017/9781316219317](https://doi.org/10.1017/9781316219317)
85. Bob Coecke and Ross Duncan. 2008. Interacting Quantum Observables
86. Bob Coecke, Ross Duncan, Aleks Kissinger and Quanlong Wang. 2012. Strong Complementarity and Non-locality in Categorical Quantum Mechanics. In *2012 27th Annual IEEE Symposium on Logic in Computer Science*, June 2012. IEEE, 245--254. doi: [10.1109/lics.2012.35](https://doi.org/10.1109/lics.2012.35)
87. Sébastien Collette, John Iacono and Stefan Langerman. 2012. Confluent Persistence Revisited. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2012. Society for Industrial and Applied Mathematics, 593--601. doi: [10.1137/1.9781611973099.50](https://doi.org/10.1137/1.9781611973099.50)
88. TKET contributors. 2025. Documentation: pytket.passes.AutoRebase. Retrieved on 13/01/2025 (TKET docs) from <https://docs.quantinuum.com/tket/api-docs/passess.html#pytket.passes.AutoRebase>

89. Stephen A. Cook. 1971. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. ACM Press, 151--158. doi: [10.1145/800157.805047](https://doi.org/10.1145/800157.805047)
90. A. D. Córcoles, Maika Takita, Ken Inoue, Scott Lekuch, Zlatko K. Mineev, Jerry M. Chow and Jay M. Gambetta. 2021. Exploiting Dynamic Quantum Circuits in a Quantum Algorithm with Superconducting Qubits. *Physical Review Letters* 127, 10 (August 2021, 100501). doi: [10.1103/physrevlett.127.100501](https://doi.org/10.1103/physrevlett.127.100501)
91. Thomas H Cormen, Charles E Leiserson, Ronald L Rivest and Clifford Stein. 2009. *Introduction to algorithms* (Third edition ed.). MIT Press, Cambridge, Massachusetts
92. Andrea Corradini, Tobias Heindel, Frank Hermann and Barbara König. 2006. Sesqui-Pushout Rewriting. In *Graph Transformations*, Berlin, Heidelberg. Springer Berlin Heidelberg, 30--45. doi: [10.1007/11841883_4](https://doi.org/10.1007/11841883_4)
93. A. Corradini, U. Montanari and F. Rossi. 1996. Graph Processes. *Fundamenta Informaticae* 26, 3,4 (241--265). doi: [10.3233/fi-1996-263402](https://doi.org/10.3233/fi-1996-263402)
94. Andrea Corradini, Dominique Duval, Michael Löwe, Leila Ribeiro, Rodrigo Machado, Andrei Costa, Guilherme Grochau Azzi, Jonas Santos Bezerra and Leonardo Marques Rodrigues. 2018. On the Essence of Parallel Independence for the Double-Pushout and Sesqui-Pushout Approaches. In *Graph Transformation, Specifications, and Nets*, Cham. Springer International Publishing, 1--18. doi: [10.1007/978-3-319-75396-6_1](https://doi.org/10.1007/978-3-319-75396-6_1)
95. Simone André da Costa and Leila Ribeiro. 2012. Verification of graph grammars using a logical approach. *Science of Computer Programming* 77, 4 (April 2012, 480--504). doi: [10.1016/j.scico.2010.02.006](https://doi.org/10.1016/j.scico.2010.02.006)
96. Bert Costin Iancu. 2025. Berkeley Quantum Synthesis Toolkit. Retrieved on 09/01/2025 from <https://bqskit.lbl.gov/>
97. Alexander Cowtan and Simon Burton. 2024. CSS code surgery as a universal construction. *Quantum* 8 (May 2024, 1344). doi: [10.22331/q-2024-05-14-1344](https://doi.org/10.22331/q-2024-05-14-1344)
98. Alexander Cowtan, Silas Dilkes, Ross Duncan, Will Simmons and Seyon Sivarajah. 2019. Phase Gadget Synthesis for Shallow Circuits. In *Proceedings 16th International Conference on Quantum Physics and Logic, QPL 2019, Chapman University, Orange, CA, USA, June 10-14, 2019*, 213--228. doi: [10.4204/EPTCS.318.13](https://doi.org/10.4204/EPTCS.318.13)
99. Alexander Cowtan, Will Simmons and Ross Duncan. 2020. A Generic Compilation Strategy for the Unitary Coupled Cluster Ansatz. arXiv: [2007.10515](https://arxiv.org/abs/2007.10515) [quant-ph]
100. Andrew W. Cross, Lev S. Bishop, John A. Smolin and Jay M. Gambetta. 2017. Open Quantum Assembly Language. arXiv: [1707.03429](https://arxiv.org/abs/1707.03429) [quant-ph]
101. Andrew W. Cross, Lev S. Bishop, Sarah Sheldon, Paul D. Nation and Jay M. Gambetta. 2019. Validating quantum computers using randomized model circuits. *Physical Review A* 100, 3 (September 2019, 032328). doi: [10.1103/physreva.100.032328](https://doi.org/10.1103/physreva.100.032328)
102. CUDA-Q Developers. 2024. CUDA-Q Documentation. Retrieved on 31/12/24 from <https://nvidia.github.io/cuda-quantum/latest/index.html>
103. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (October 1991, 451--490). doi: [10.1145/115372.115320](https://doi.org/10.1145/115372.115320)
104. Vincent Danos, Jérôme Feret, Walter Fontana and Jean Krivine. 2007. Scalable Simulation of Cellular Signaling Networks
105. Vincent Danos and Cosimo Laneve. 2004. Formal molecular biology. *Theoretical Computer Science* 325, 1 (September 2004, 69--110). doi: [10.1016/j.tcs.2004.06.001](https://doi.org/10.1016/j.tcs.2004.06.001)

[10.1016/j.tcs.2004.03.065](https://doi.org/10.1016/j.tcs.2004.03.065)

106. Vincent Danos, Reiko Heckel and Pawel Sobocinski. 2014. Transformation and Refinement of Rigid Structures
107. Kamalika Datta, Gaurav Rathi, Robert Wille, Indranil Sengupta, Hafizur Rahaman and Rolf Drechsler. 2013. Exploiting Negative Control Lines in the Optimization of Reversible Circuits
108. Kamalika Datta, Indranil Sengupta and Hafizur Rahaman. 2015. A Post-Synthesis Optimization Technique for Reversible Circuits Exploiting Negative Control Lines. *IEEE Transactions on Computers* 64, 4 (April 2015, 1208--1214). doi: [10.1109/tc.2014.2315641](https://doi.org/10.1109/tc.2014.2315641)
109. Marc G. Davis, Ethan Smith, Ana Tudor, Koushik Sen, Irfan Siddiqi and Costin Iancu. 2020. Towards Optimal Topology Aware Quantum Circuit Synthesis. In *2020 IEEE International Conference on Quantum Computing and Engineering (QCE)*, October 2020. IEEE, 223--234. doi: [10.1109/QCE49297.2020.00036](https://doi.org/10.1109/QCE49297.2020.00036)
110. Christopher M. Dawson and Michael A. Nielsen. 2006. The Solovay-Kitaev algorithm. *Quantum Information and Computation* 6, 1 (January 2006, 81--95). doi: [10.26421/QIC6.1-6](https://doi.org/10.26421/QIC6.1-6)
111. Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite Systems
112. Ivan H. Deutsch, Gavin K. Brennen and Poul S. Jessen. 2000. Quantum Computing with Neutral Atoms in an Optical Lattice. *Fortschritte der Physik* 48, 9--11 (September 2000, 925--943). doi: [10.1002/1521-3978\(200009\)48:9/11<925::aid-prop925>3.0.co;2-a](https://doi.org/10.1002/1521-3978(200009)48:9/11<925::aid-prop925>3.0.co;2-a)
113. David Deutsch. 1989. Quantum Computational Networks. In *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*. Royal Society, 73--90
114. Strohmaier, E., Simon, H., Meuer, H. Dongarra. 2024. TOP500 List. (November 2024). Retrieved on 30/12/2024 from <https://top500.org/lists/top500/list/2024/11/>
115. Hempel, R., Hey, A., Dongarra. 1993. MPI: a message passing interface. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. ACM Press, 878--883. doi: [10.1145/169627.169855](https://doi.org/10.1145/169627.169855)
116. Frank Drewes, Berthold Hoffmann and Detlef Plump. 2002. Hierarchical Graph Transformation. *Journal of Computer and System Sciences* 64, 2 (March 2002, 249--283). doi: [10.1006/jcss.2001.1790](https://doi.org/10.1006/jcss.2001.1790)
117. James R. Driscoll, Daniel D. K. Sleator and Robert E. Tarjan. 1994. Fully persistent lists with catenation. *Journal of the ACM* 41, 5 (September 1994, 943--959). doi: [10.1145/185675.185791](https://doi.org/10.1145/185675.185791)
118. James R. Driscoll, Neil Sarnak, Daniel D. Sleator and Robert E. Tarjan. 1989. Making data structures persistent. *Journal of Computer and System Sciences* 38, 1 (February 1989, 86--124). doi: [10.1016/0022-0000\(89\)90034-2](https://doi.org/10.1016/0022-0000(89)90034-2)
119. Ross Duncan, Aleks Kissinger, Simon Perdrix and John van de Wetering. 2020. Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus. *Quantum* 4 (June 2020, 279). doi: [10.22331/q-2020-06-04-279](https://doi.org/10.22331/q-2020-06-04-279)
120. Ross Duncan, Aleks Kissinger, Simon Perdrix and John van de Wetering. 2019. Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus. arXiv: [1902.03178 \[quant-ph\]](https://arxiv.org/abs/1902.03178)
121. Rachid Echahed and Aude Maignan. 2017. Parallel Graph Rewriting with Overlapping Rules. In *Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR*, 300--318. doi: [10.29007/576h](https://doi.org/10.29007/576h)
122. Rachid Echahed. 2008. Inductively Sequential Term-Graph Rewrite Systems. In *Graph Transformations, ICGT*. Springer Berlin Heidelberg, 84--98. doi: [10.1007/978-3-540-87405-8_7](https://doi.org/10.1007/978-3-540-87405-8_7)

123. Wesley Eddy. 2022. Transmission Control Protocol (TCP). (August 2022). Retrieved as RFC 9293 from <https://www.rfc-editor.org/info/rfc9293>
124. Hartmut Ehrig, Michael Pfender and Hans Jürgen Schneider. 1973. Graph-Grammars: An Algebraic Approach. In *14th Annual Symposium on Switching and Automata Theory, Iowa City, Iowa, USA, October 15-17, 1973*. IEEE Computer Society, 167--180. doi: [10.1109/SWAT.1973.11](https://doi.org/10.1109/SWAT.1973.11)
125. Hartmut Ehrig, Ulrike Prange and Gabriele Taentzer. 2004. Fundamental Theory for Typed Attributed Graph Transformation. In *Graph Transformations*, Berlin, Heidelberg. Springer Berlin Heidelberg, 161--177. doi: [10.1007/978-3-540-30203-2_13](https://doi.org/10.1007/978-3-540-30203-2_13)
126. Hartmut Ehrig and Hans-Jörg Kreowski. 1976. Parallelism of manipulations in multidimensional information structures
127. Alexander Erhard, Joel J. Wallman, Lukas Postler, Michael Meth, Roman Stricker, Esteban A. Martinez, Philipp Schindler, Thomas Monz, Joseph Emerson and Rainer Blatt. 2019. Characterizing large-scale quantum computers via cycle benchmarking. *Nature Communications* 10, 1 (November 2019). doi: [10.1038/s41467-019-13068-7](https://doi.org/10.1038/s41467-019-13068-7)
128. Simon J. Evered, Dolev Bluvstein, Marcin Kalinowski, Sepehr Ebadi, Tom Manovitz, Hengyun Zhou, Sophie H. Li, Alexandra A. Geim, Tout T. Wang, Nishad Maskara, Harry Levine, Giulia Semeghini, Markus Greiner, Vladan Vuletić and Mikhail D. Lukin. 2023. High-fidelity parallel entangling gates on a neutral-atom quantum computer. *Nature* 622, 7982 (October 2023, 268--272). doi: [10.1038/s41586-023-06481-y](https://doi.org/10.1038/s41586-023-06481-y)
129. Andrew Fagan and Ross Duncan. 2018. Optimising Clifford Circuits with Qantomatic. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018*, 85--105. doi: [10.4204/EPTCS.287.5](https://doi.org/10.4204/EPTCS.287.5)
130. Chris Fallin. 2022. Cranelift: Using E-Graphs for Verified, Cooperating Middle-End Optimizations. (August 2022). Retrieved on 14/01/2025 (RFC) from <https://github.com/bytecodealliance/rfcs/blob/main/accepted/cranelift-egraph.md>
131. Jingzhi Fang, Yanyan Shen, Yue Wang and Lei Chen. 2020. Optimizing DNN computation graph using graph substitutions. *Proceedings of the VLDB Endowment* 13, 12 (August 2020, 2734--2746). doi: [10.14778/3407790.3407857](https://doi.org/10.14778/3407790.3407857)
132. K. Fazel, M. A. Thornton and J. E. Rice. 2007. ESOP-based Toffoli Gate Cascade Generation. In *2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, August 2007. IEEE. doi: [10.1109/pacrim.2007.4313212](https://doi.org/10.1109/pacrim.2007.4313212)
133. Giovanni de Felice and Bob Coecke. 2023. Quantum Linear Optics via String Diagrams. In *Proceedings 19th International Conference on Quantum Physics and Logic, Wolfson College, Oxford, UK, 27 June - 1 July 2022*. Open Publishing Association, 83-100. doi: [10.4204/EPTCS.394.6](https://doi.org/10.4204/EPTCS.394.6)
134. Giovanni de Felice, Razin A. Shaikh, Boldizsár Poór, Lia Yeh, Quanlong Wang and Bob Coecke. 2023. Light-Matter Interaction in the ZXW Calculus. *Electronic Proceedings in Theoretical Computer Science* 384 (August 2023, 20--46). doi: [10.4204/EPTCS.384.2](https://doi.org/10.4204/EPTCS.384.2)
135. John T. Feo, David C. Cann and Rodney R. Oldehoeft. 1990. A report on the sisal language project. *Journal of Parallel and Distributed Computing* 10, 4 (December 1990, 349--366). doi: [10.1016/0743-7315\(90\)90035-n](https://doi.org/10.1016/0743-7315(90)90035-n)
136. Maribel Fernández, Hélène Kirchner and Bruno Pinaud. 2018. Labelled Port Graph – A Formal Structure for Models and Computations. *Electronic Notes in Theoretical Computer Science* 338 (October 2018, 3--21). doi: [10.1016/j.entcs.2018.10.002](https://doi.org/10.1016/j.entcs.2018.10.002)

137. Maribel Fernández and Olivier Namet. 2010. Strategic programming on graph rewriting systems. *Electronic Proceedings in Theoretical Computer Science* 44 (December 2010, 1--20). doi: [10.4204/eptcs.44.1](https://doi.org/10.4204/eptcs.44.1)
138. R. P. Feynman. 1949. Space-Time Approach to Quantum Electrodynamics. *Physical Review* 76, 6 (September 1949, 769--789). doi: [10.1103/physrev.76.769](https://doi.org/10.1103/physrev.76.769)
139. Amos Fiat and Haim Kaplan. 2003. Making data structures confluent persist-ent. *Journal of Algorithms* 48, 1 (August 2003, 16--58). doi: [10.1016/s0196-6774\(03\)00044-0](https://doi.org/10.1016/s0196-6774(03)00044-0)
140. Stuart Flannigan. 2020. The application of quantum simulation to topological and open many-body systems. PhD Thesis. University of Strathclyde
141. Blake W. Ford, Apan Qasem, Jelena Tesic and Ziliang Zong. 2021. Migrating Software from x86 to ARM Architecture: An Instruction Prediction Approach. In *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*, October 2021. IEEE, 1--6. doi: [10.1109/NAS51552.2021.9605443](https://doi.org/10.1109/NAS51552.2021.9605443)
142. Charles L. Forgy. 1982. Rete: A fast algorithm for the many pattern/many ob-ject pattern match problem. *Artificial Intelligence* 19, 1 (September 1982, 17--37). doi: [10.1016/0004-3702\(82\)90020-0](https://doi.org/10.1016/0004-3702(82)90020-0)
143. Austin G. Fowler, Matteo Mariantoni, John M. Martinis and Andrew N. Cleland. 2012. Surface codes: Towards practical large-scale quantum computa-tion. *Physical Review A* 86, 3 (September 2012, 032324). doi: [10.1103/phys-reva.86.032324](https://doi.org/10.1103/phys-reva.86.032324)
144. Christopher W. Fraser. 1979. A compact, machine-independent peephole op-timizer. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '79*. ACM Press, 1--6. doi: [10.1145/567752.567753](https://doi.org/10.1145/567752.567753)
145. Peng Fu, Kohei Kishida, Neil J. Ross and Peter Selinger. 2023. Proto-Quipper with Dynamic Lifting. *Proceedings of the ACM on Programming Languages* 7, POPL (January 2023, 309--334). doi: [10.1145/3571204](https://doi.org/10.1145/3571204)
146. Mariam Gado and Ahmed Younes. 2021. Optimization of Reversible Circuits Using Toffoli Decompositions with Negative Controls. *Symmetry* 13, 6 (June 2021, 1025). doi: [10.3390/sym13061025](https://doi.org/10.3390/sym13061025)
147. Bernard A. Galler and Michael J. Fisher. 1964. An improved equivalence al-gorithm. *Communications of the ACM* 7, 5 (May 1964, 301--303). doi: [10.1145/364099.364331](https://doi.org/10.1145/364099.364331)
148. Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack and Adam Szalkowski. 2006. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Graph Transformations. ICGT 2006*. Springer Berlin Heidelberg, 383--397. doi: [10.1007/11841883_27](https://doi.org/10.1007/11841883_27)
149. Vlad Gheorghiu, Michele Mosca and Priyanka Mukhopadhyay. 2022. A (quasi-)polynomial time heuristic algorithm for synthesizing T-depth optimal circuits. *npj Quantum Information* 8, 1 (September 2022). doi: [10.1038/s41534-022-00624-1](https://doi.org/10.1038/s41534-022-00624-1)
150. András Gilyén, Yuan Su, Guang Hao Low and Nathan Wiebe. 2019. Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, June 2019. ACM, 193--204. doi: [10.1145/3313276.3316366](https://doi.org/10.1145/3313276.3316366)
151. Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1--101). doi: [10.1016/0304-3975\(87\)90045-4](https://doi.org/10.1016/0304-3975(87)90045-4)
152. Stefano Gogioso and Richie Yeung. 2022. Annealing Optimisation of Mixed ZX Phase Circuits. In *Proceedings 19th International Conference on Quantum*

- Physics and Logic, QPL 2022, Wolfson College, Oxford, UK, 27 June - 1 July 2022*, 415--431. doi: [10.4204/EPTCS.394.20](https://doi.org/10.4204/EPTCS.394.20)
153. David Gosset, Robin Kothari and Kewen Wu. 2024. Quantum state preparation with optimal T-count. arXiv: [2411.04790](https://arxiv.org/abs/2411.04790) [quant-ph]
 154. Daniel Gottesman. 1999. The Heisenberg representation of quantum computers. In *Group 22: Proceedings of the 12th International Colloquium on Group Theoretical Methods in Physics*,. International Press, 32--43
 155. Daniel Gottesman. 2024. Surviving as a Quantum Computer in a Classical World. (February 2024). Retrieved on 08/01/2025 (lecture notes) from <https://www.cs.umd.edu/class/spring2024/cmsc858G/QECCbook-2024-ch1-15.pdf>
 156. T. M. Graham, L. Phuttitarn, R. Chinnarasu, Y. Song, C. Poole, K. Jooya, J. Scott, A. Scott, P. Eichler and M. Saffman. 2023. Midcircuit Measurements on a Single-Species Neutral Alkali Atom Quantum Processor. *Physical Review X* 13, 4 (December 2023, 041051). doi: [10.1103/physrevx.13.041051](https://doi.org/10.1103/physrevx.13.041051)
 157. F. Gray. 1953. Pulse code communication. Retrieved from <http://www.google.com/patents/US2632058>
 158. Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2013, New York, NY, USA. Association for Computing Machinery, 333--342. doi: [10.1145/2491956.2462177](https://doi.org/10.1145/2491956.2462177)
 159. Arianne Meijer-van de Griend and Ross Duncan. 2022. Architecture-Aware Synthesis of Phase Polynomials for NISQ Devices. In *Proceedings 19th International Conference on Quantum Physics and Logic, QPL 2022, Wolfson College, Oxford, UK, 27 June - 1 July 2022*, 116--140. doi: [10.4204/EPTCS.394.8](https://doi.org/10.4204/EPTCS.394.8)
 160. Sebastiaan L. D. ten Haaf, Qingzhen Wang, A. Mert Bozkurt, Chun-Xiao Liu, Ivan Kulesh, Philip Kim, Di Xiao, Candice Thomas, Michael J. Manfra, Tom Dvir, Michael Wimmer and Srijit Goswami. 2024. A two-site Kitaev chain in a two-dimensional electron gas. *Nature* 630, 8016 (June 2024, 329--334). doi: [10.1038/s41586-024-07434-9](https://doi.org/10.1038/s41586-024-07434-9)
 161. Peter Hart, Nils Nilsson and Bertram Raphael. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics* 4, 2 (100--107). doi: [10.1109/tssc.1968.300136](https://doi.org/10.1109/tssc.1968.300136)
 162. Reiko Heckel and Gabriele Taentzer. 2020. *Graph Transformation for Software Engineers: With Applications to Model-Based Development and Domain-Specific Language Engineering*. Springer International Publishing. doi: [10.1007/978-3-030-43916-3](https://doi.org/10.1007/978-3-030-43916-3)
 163. James Hefford and Matt Wilson. 2024. A Profunctorial Semantics for Quantum Supermaps. In *Proceedings of the 39th Annual ACM/IEEE Symposium on Logic in Computer Science*, July 2024. ACM, 1--15. doi: [10.1145/3661814.3662123](https://doi.org/10.1145/3661814.3662123)
 164. M. Hein, J. Eisert and H. J. Briegel. 2004. Multiparty entanglement in graph states. *Physical Review A* 69, 6 (June 2004, 062311). doi: [10.1103/physreva.69.062311](https://doi.org/10.1103/physreva.69.062311)
 165. John Hennessy, Norman Jouppi, Steven Przybylski, Christopher Rowen, Thomas Gross, Forest Baskett and John Gill. 1982. MIPS: A microprocessor architecture. *ACM SIGMICRO Newsletter* 13, 4 (December 1982, 17--22). doi: [10.1145/1014194.800930](https://doi.org/10.1145/1014194.800930)
 166. Nicolas Heurtel, Andreas Fyrrillas, Grégoire Gliniasty, Raphaël Le Bihan, Sébastien Malherbe, Marceau Pailhas, Eric Bertasi, Boris Bourdoncle, Pierre-Emmanuel Emeriau, Rawad Mezher, Luka Music, Nadia Belabas, Benoît

- Valiron, Pascale Senellart, Shane Mansfield and Jean Senellart. 2023. Perceval: A Software Platform for Discrete Variable Photonic Quantum Computing. *Quantum* 7 (February 2023, 931). doi: [10.22331/q-2023-02-21-931](https://doi.org/10.22331/q-2023-02-21-931)
167. Nicolas Heurtel. 2024. A Complete Graphical Language for Linear Optical Circuits with Finite-Photon-Number Sources and Detectors. arXiv: [2402.17693](https://arxiv.org/abs/2402.17693) [quant-ph]
168. Ralf Hinze and Ross Paterson. 2005. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming* 16, 02 (November 2005, 197). doi: [10.1017/s0956796805005769](https://doi.org/10.1017/s0956796805005769)
169. Lev Bishop, Yudong Cao, Andrew Cross, Niel Hossein Ajallooiean. 2023. OpenQASM 3.0 Specification. Retrieved on 15/03/2025 from <https://openqasm.com/versions/3.0/intro.html>
170. Qunsheng Huang, David Winderl, Arianne Meijer-van de Griend and Richie Yeung. 2024. Redefining Lexicographical Ordering: Optimizing Pauli String Decompositions for Quantum Compiling. *CoRR* abs/2408.00354. doi: [10.48550/ARXIV.2408.00354](https://arxiv.org/abs/2408.00354)
171. Wright Ian and James A. R. Marshall. 2003. The execution kernel of RC++: RETE*, a faster RETE with TREAT as a special case. *International Journal of Intelligent Games and Simulation* 2, 1 (Feb 2003, 36-48)
172. IBM. 2024. Expanding the IBM Quantum roadmap to anticipate the future of quantum-centric supercomputing. Retrieved on 30/12/2024 from <https://www.ibm.com/quantum/blog/ibm-quantum-roadmap-2025>
173. Raban Iten, Roger Colbeck, Ivan Kukuljan, Jonathan Home and Matthias Christandl. 2016. Quantum circuits for isometries. *Physical Review A* 93, 3 (March 2016, 032318). doi: [10.1103/PhysRevA.93.032318](https://doi.org/10.1103/PhysRevA.93.032318)
174. Raban Iten, Oliver Reardon-Smith, Emanuel Malvetti, Luca Mondada, Gabrielle Pauvert, Ethan Redmond, Ravjot Singh Kohli and Roger Colbeck. 2019. Introduction to UniversalQCompiler. arXiv: [1904.01072](https://arxiv.org/abs/1904.01072) [quant-ph]
175. Raban Iten, Romain Moyard, Tony Metger, David Sutter and Stefan Woerner. 2022. Exact and Practical Pattern Matching for Quantum Circuit Optimization. *ACM Transactions on Quantum Computing* 3, 1 (January 2022, 1--41). doi: [10.1145/3498325](https://doi.org/10.1145/3498325)
176. David Ittah, Thomas Häner, Vadym Kliuchnikov and Torsten Hoefler. 2022. QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization. *ACM Transactions on Quantum Computing* 3, 3 (June 2022, 1--32). doi: [10.1145/3491247](https://doi.org/10.1145/3491247)
177. David Ittah, Ali Asadi, Erick Ochoa Lopez, Sergei Mironov, Samuel Banning, Romain Moyard, Mai Jacob Peng and Josh Izaac. 2024. Catalyst: a Python JIT compiler for auto-differentiable hybrid quantum programs. *Journal of Open Source Software* 9, 99 (July 2024, 6720). doi: [10.21105/joss.06720](https://doi.org/10.21105/joss.06720)
178. D. Jaksch, J. I. Cirac, P. Zoller, S. L. Rolston, R. Côté and M. D. Lukin. 2000. Fast Quantum Gates for Neutral Atoms. *Physical Review Letters* 85, 10 (Septempter 2000, 2208--2211). doi: [10.1103/physrevlett.85.2208](https://doi.org/10.1103/physrevlett.85.2208)
179. 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. 2024. Quantum computing with Qiskit. arXiv: [2405.08810](https://arxiv.org/abs/2405.08810) [quant-ph]
180. Jerzy Jegier and Paweł Kerntopf. 2017. PPRM-based approach to synthesis of reversible functions. In *Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2017*, August 2017. SPIE, 1044523. doi: [10.1117/12.2280943](https://doi.org/10.1117/12.2280943)
181. Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia and Alex Aiken. 2019. TASO: optimizing deep learning computation with autom-

- atic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, October 2019. ACM, 47--62. doi: [10.1145/3341301.3359630](https://doi.org/10.1145/3341301.3359630)
182. Xiaoyi Jiang and Horst Bunke. 1998. Marked subgraph isomorphism of ordered graphs. In *Advances in Pattern Recognition*, Berlin, Heidelberg. Springer Berlin Heidelberg, 122--131. doi: [10.1007/bfb0033230](https://doi.org/10.1007/bfb0033230)
 183. Gilles Kahn and David MacQueen. 1976. Coroutines and networks of parallel processes. PhD Thesis. IRIA
 184. Krishnageetha Karuppasamy, Varun Puram, Stevens Johnson and Johnson P. Thomas. 2025. A Comprehensive Review of Quantum Circuit Optimization: Current Trends and Future Directions. *Quantum Reports* 7, 1 (January 2025, 2). doi: [10.3390/quantum7010002](https://doi.org/10.3390/quantum7010002)
 185. D. Kielpinski, C. Monroe and D. J. Wineland. 2002. Architecture for a large-scale ion-trap quantum computer. *Nature* 417, 6890 (June 2002, 709--711). doi: [10.1038/nature00784](https://doi.org/10.1038/nature00784)
 186. Aleks Kissinger and John van de Wetering. 2022. Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions. *Quantum Science and Technology* 7, 4 (July 2022, 044001). doi: [10.1088/2058-9565/ac5d20](https://doi.org/10.1088/2058-9565/ac5d20)
 187. Aleks Kissinger and John van de Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. In *Proceedings 16th International Conference on Quantum Physics and Logic, Chapman University, Orange, CA, USA., 10-14 June 2019*. Open Publishing Association, 229-241. doi: [10.4204/EPTCS.318.14](https://doi.org/10.4204/EPTCS.318.14)
 188. Aleks Kissinger, John van de Wetering and Renaud Vilmart. 2022. Classical Simulation of Quantum Circuits with Partial and Graphical Stabiliser Decompositions. In . Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: [10.4230/LIPICS.TQC.2022.5](https://doi.org/10.4230/LIPICS.TQC.2022.5)
 189. Aleks Kissinger and John van de Wetering. 2024. *Picturing Quantum Software: An Introduction to the ZX-Calculus and Quantum Compilation*. Preprint
 190. Aleks Kissinger and Sander Uijlen. 2019. A categorical semantics for causal structure. *Logical Methods in Computer Science* Volume 15, Issue 3 (August 2019). doi: [10.23638/lmcs-15\(3:15\)2019](https://doi.org/10.23638/lmcs-15(3:15)2019)
 191. Alexei Y. Kitaev, A. H. Shen and Mikhail N. Vyalyi. 2002. *Classical and Quantum Computation*. American Mathematical Society
 192. A.Yu. Kitaev. 2003. Fault-tolerant quantum computation by anyons. *Annals of Physics* 303, 1 (January 2003, 2--30). doi: [10.1016/S0003-4916\(02\)00018-0](https://doi.org/10.1016/S0003-4916(02)00018-0)
 193. Vadym Kliuchnikov and Dmitri Maslov. 2013. Optimization of Clifford circuits. *Physical Review A* 88, 5 (November 2013, 052307). doi: [10.1103/physreva.88.052307](https://doi.org/10.1103/physreva.88.052307)
 194. E. Knill, R. Laflamme and G. J. Milburn. 2001. A scheme for efficient quantum computation with linear optics. *Nature* 409, 6816 (January 2001, 46--52). doi: [10.1038/35051009](https://doi.org/10.1038/35051009)
 195. Donald Knuth. 1999. *The Art of Computer Programming: Sorting and Searching, Volume 3*. Addison-Wesley, Reading MA
 196. Jens Koch, Terri M. Yu, Jay Gambetta, A. A. Houck, D. I. Schuster, J. Majer, Alexandre Blais, M. H. Devoret, S. M. Girvin and R. J. Schoelkopf. 2007. Charge-insensitive qubit design derived from the Cooper pair box. *Physical Review A* 76, 4 (October 2007, 042319). doi: [10.1103/PhysRevA.76.042319](https://doi.org/10.1103/PhysRevA.76.042319)
 197. Mark Koch, Alan Lawrence, Kartik Singhal, Seyon Sivarajah and Ross Duncan. 2024. GUPPY: Pythonic Quantum-Classical Programming. (January 2024). Retrieved (talk recording) from <https://www.youtube.com/live/D8esZrt7ogk?feature=shared&t=31448>

198. Barbara König, Dennis Nolte, Julia Padberg and Arend Rensink. 2018. A Tutorial on Graph Transformation. In *Graph Transformation, Specifications, and Nets - In Memory of Hartmut Ehrig*. Springer, 83--104. doi: [10.1007/978-3-319-75396-6_5](https://doi.org/10.1007/978-3-319-75396-6_5)
199. Korbinian Kottmann. 2024. Introducing (dynamical) Lie algebras for quantum practitioners. (February 2024). Retrieved on 08/01/2025 from https://pennylane.ai/qml/demos/tutorial_liealgebra
200. Alexander Koziell-Pipe and Aleks Kissinger. 2024. Hybrid Quantum-Classical Machine Learning with String Diagrams. arXiv: [2407.03673](https://arxiv.org/abs/2407.03673) [quant-ph]
201. Anna M. Krol, Aritra Sarkar, Imran Ashraf, Zaid Al-Ars and Koen Bertels. 2022. Efficient Decomposition of Unitary Matrices in Quantum Circuit Compilers. *Applied Sciences* 12, 2 (January 2022, 759). doi: [10.3390/app12020759](https://doi.org/10.3390/app12020759)
202. Alon Kukliansky, Ed Younis, Lukasz Cincio and Costin Iancu. 2023. QFactor: A Domain-Specific Optimizer for Quantum Circuit Instantiation. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*, September 2023. IEEE, 814--824. doi: [10.1109/QCE57702.2023.00096](https://doi.org/10.1109/QCE57702.2023.00096)
203. Stephen Lack and Paweł Sobociński. 2005. Adhesive and quasiadhesive categories. *RAIRO - Theoretical Informatics and Applications* 39, 3 (July 2005, 511--545). doi: [10.1051/ITA:2005028](https://doi.org/10.1051/ITA:2005028)
204. Stephen Lack and Paweł Sobociński. 2004. Adhesive Categories. In *Foundations of Software Science and Computation Structures*, Berlin, Heidelberg. Springer Berlin Heidelberg, 273--288. doi: [10.1007/978-3-540-24727-2_20](https://doi.org/10.1007/978-3-540-24727-2_20)
205. George Lagogiannis, Yannis Panagis, Spyros Sioutas and Athanasios Tsakalidis. 2005. A survey of persistent data structures. In *Proceedings of the 9th WSEAS International Conference on Computers*, Stevens Point, Wisconsin, USA. World Scientific and Engineering Academy and Society (WSEAS)
206. Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, February 2021. IEEE, 2--14. doi: [10.1109/CGO51591.2021.9370308](https://doi.org/10.1109/CGO51591.2021.9370308)
207. Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*. IEEE, 75--86. doi: [10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665)
208. Eric Lehman, F. Thomson Leighton and Albert R. Meyer. 2017. *Mathematics for Computer Science*. Samurai Media Limited
209. Zhiqiang Li, Hanwu Chen, Xiaoyu Song and Marek Perkowski. 2014. A Synthesis Algorithm for 4-Bit Reversible Logic Circuits with Minimum Quantum Cost. *ACM Journal on Emerging Technologies in Computing Systems* 11, 3 (December 2014, 1--19). doi: [10.1145/2629542](https://doi.org/10.1145/2629542)
210. Youwei Liang, Kevin Stone, Ali Shameli, Chris Cummins, Mostafa Elhoushi, Jiadong Guo, Benoit Steiner, Xiaomeng Yang, Pengtao Xie, Hugh Leather and Yuandong Tian. 2023. Learning Compiler Pass Orders using Coreset and Normalized Value Prediction. In *Proceedings of the 40th International Conference on Machine Learning*. JMLR.org. doi: [10.48550/ARXIV.2301.05104](https://doi.org/10.48550/ARXIV.2301.05104)
211. Ji Liu, Luciano Bello and Huiyang Zhou. 2021. Relaxed Peephole Optimization: A Novel Compiler Optimization for Quantum Circuits. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, February 2021. IEEE, 301--314. doi: [10.1109/cgo51591.2021.9370310](https://doi.org/10.1109/cgo51591.2021.9370310)

212. T. Loke, J. B. Wang and Y. H. Chen. 2014. OptQC: An optimized parallel quantum compiler. *Computer Physics Communications* 185, 12 (December 2014, 3307--3316). doi: [10.1016/j.cpc.2014.07.022](https://doi.org/10.1016/j.cpc.2014.07.022)
213. Nuno P. Lopes, David Menendez, Santosh Nagarakatte and John Regehr. 2015. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2015. ACM, 22--32. doi: [10.1145/2737924.2737965](https://doi.org/10.1145/2737924.2737965)
214. Michael Löwe. 1991. Extended algebraic graph transformation. PhD Thesis. Technical University of Berlin
215. Liam Madden and Andrea Simonetto. 2022. Best Approximate Quantum Compiling Problems. *ACM Transactions on Quantum Computing* 3, 2 (March 2022, 1--29). doi: [10.1145/3505181](https://doi.org/10.1145/3505181)
216. Seyon Sivarajah, Alan Lawrence, Alec Edgington, Douglas Wilson, Craig Roy, Luca Mondada, Lukas Heidemann, Ross Duncan Mark Koch. 2025. HUGR: A Quantum-Classical Intermediate Representation. Retrieved (talk recording) from <https://www.youtube.com/live/D8esZrt7ogk?feature=shared&t=5217>
217. Dmitri Maslov and Martin Roetteler. 2018. Shorter Stabilizer Circuits via Bruhat Decomposition and Quantum Circuit Transformations. *IEEE Transactions on Information Theory* 64, 7 (July 2018, 4729--4738). doi: [10.1109/tit.2018.2825602](https://doi.org/10.1109/tit.2018.2825602)
218. Dmitri Maslov and Willers Yang. 2023. CNOT circuits need little help to implement arbitrary Hadamard-free Clifford transformations they generate. *npj Quantum Information* 9, 1 (September 2023). doi: [10.1038/s41534-023-00760-2](https://doi.org/10.1038/s41534-023-00760-2)
219. Dmitri Maslov. 2016. Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization. *Physical Review A* 93, 2 (February 2016, 022311). doi: [10.1103/physreva.93.022311](https://doi.org/10.1103/physreva.93.022311)
220. D. Maslov, G.W. Dueck, D.M. Miller and C. Negrevergne. 2008. Quantum Circuit Simplification and Level Compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 3 (March 2008, 436--444). doi: [10.1109/tcad.2007.911334](https://doi.org/10.1109/tcad.2007.911334)
221. Henry Massalin. 1987. Superoptimizer: a look at the smallest program. In *Proceedings of the second international conference on Architectural support for programming languages and operating systems*, October 1987. ACM, 122--126. doi: [10.1145/36206.36194](https://doi.org/10.1145/36206.36194)
222. Alexander McCaskey and Thien Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, October 2021. IEEE. doi: [10.1109/QCE52317.2021.00043](https://doi.org/10.1109/QCE52317.2021.00043)
223. Jarrod R McClean, Jonathan Romero, Ryan Babbush and Alán Aspuru-Guzik. 2016. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics* 18, 2 (February 2016, 023023). doi: [10.1088/1367-2630/18/2/023023](https://doi.org/10.1088/1367-2630/18/2/023023)
224. W. M. McKeeman. 1965. Peephole optimization. *Communications of the ACM* 8, 7 (July 1965, 443--444). doi: [10.1145/364995.365000](https://doi.org/10.1145/364995.365000)
225. Arianne Meijer - van de Griend. 2025. A comparison of quantum compilers using a DAG-based or phase polynomial-based intermediate representation. *Journal of Systems and Software* 221 (March 2025, 112224). doi: [10.1016/j.jss.2024.112224](https://doi.org/10.1016/j.jss.2024.112224)
226. Kai Meinerz, Chae-Yeun Park and Simon Trebst. 2022. Scalable Neural Decoder for Topological Surface Codes. *Physical Review Letters* 128, 8 (February 2022, 080505). doi: [10.1103/physrevlett.128.080505](https://doi.org/10.1103/physrevlett.128.080505)

227. David Menendez and Santosh Nagarakatte. 2017. Alive-Infer: data-driven pre-condition inference for peephole optimizations in LLVM. *ACM SIGPLAN Notices* 52, 6 (June 2017, 49--63). doi: [10.1145/3140587.3062372](https://doi.org/10.1145/3140587.3062372)
228. Bruno T. Messmer and Horst Bunke. 1999. A decision tree approach to graph and subgraph isomorphism detection. *Pattern Recognition* 32, 12 (December 1999, 1979--1998). doi: [10.1016/S0031-3203\(98\)90142-X](https://doi.org/10.1016/S0031-3203(98)90142-X)
229. Microsoft. 2024. Introduction to the quantum programming language Q#. Retrieved on 31/12/2024 from <https://learn.microsoft.com/en-us/azure/quantum/qsharp-overview>
230. D. P. Miranker. 1987. TREAT: a new and efficient match algorithm for AI production systems. PhD Thesis. Columbia University, 2960 Broadway New York, NY United States
231. Majid Mohammadi and Mohammad Eshghi. 2008. Behavioral description of quantum V and V+ gates to design quantum logic circuits. In *2008 5th International Multi-Conference on Systems, Signals and Devices*, July 2008. IEEE, 1--5. doi: [10.1109/ssd.2008.4632850](https://doi.org/10.1109/ssd.2008.4632850)
232. Luca Mondada and Pablo Andrés-Martínez. 2025. Scalable Pattern Matching in Computation Graphs. *Electronic Proceedings in Theoretical Computer Science* 417 (March 2025, 71--95). doi: [10.4204/eptcs.417.5](https://doi.org/10.4204/eptcs.417.5)
233. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang and Sharad Malik. 2001. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th conference on Design automation - DAC '01*. ACM Press, 530--535. doi: [10.1145/378239.379017](https://doi.org/10.1145/378239.379017)
234. Mikko Möttönen, Juha J. Vartiainen, Ville Bergholm and Martti M. Salomaa. 2004. Quantum Circuits for General Multiqubit Gates. *Physical Review Letters* 93, 13 (September 2004, 130502). doi: [10.1103/PhysRevLett.93.130502](https://doi.org/10.1103/PhysRevLett.93.130502)
235. Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 337--340. doi: [10.1007/978-3-540-78800-3_24](https://doi.org/10.1007/978-3-540-78800-3_24)
236. V. Mourik, K. Zuo, S. M. Frolov, S. R. Plissard, E. P. A. M. Bakkers and L. P. Kouwenhoven. 2012. Signatures of Majorana Fermions in Hybrid Superconductor-Semiconductor Nanowire Devices. *Science* 336, 6084 (May 2012, 1003--1007). doi: [10.1126/science.1222360](https://doi.org/10.1126/science.1222360)
237. Steven S. Muchnick. 2007. *Advanced compiler design and implementation* ([Nachdr.] ed.). Morgan Kaufmann, San Francisco, Calif. [u.a.]
238. Michael A. Nielsen and Isaac L. Chuang. 2016. *Quantum Computation and Quantum Information (10th Anniversary edition)*. Cambridge University Press
239. Robert Nieuwenhuis and Albert Oliveras. 2006. On SAT Modulo Theories and Optimization Problems. In *Theory and Applications of Satisfiability Testing - SAT 2006*. Springer Berlin Heidelberg, 156--169. doi: [10.1007/11814948_18](https://doi.org/10.1007/11814948_18)
240. Siyuan Niu, Efehan Kokcu, Anupam Mitra, Aaron Szasz, Akel Hashim, Justin Kalloor, Wibe Albert de Jong, Costin Iancu and Ed Younis. 2024. AC/DC: Automated Compilation for Dynamic Circuits. arXiv: [2412.07969](https://arxiv.org/abs/2412.07969) [quant-ph]
241. NVIDIA. 2024. NVIDIA Accelerates Quantum Computing Centers Worldwide With CUDA-Q Platform. Retrieved on 31/12/2024 from <https://investor.nvidia.com/news/press-release-details/2024/NVIDIA-Accelerates-Quantum-Computing-Centers-Worldwide-With-CUDA-Q-Platform/default.aspx>
242. Chris Okasaki and Peter Lee. 1996. *Purely functional data structures*. Carnegie Mellon University, USA
243. Chris Okasaki and Andy Gill. 1998. Fast Mergeable Integer Maps. In *Workshop on ML*, September 1998, 77--86
244. Bernhard Ömer. 2000. Quantum Programming in QCL. (January 2000). Retrieved from <http://tph.tuwien.ac.at/oemer/doc/quprog.pdf>

245. Ramon W. J. Overwater, Masoud Babaie and Fabio Sebastiano. 2022. Neural-Network Decoders for Quantum Error Correction Using Surface Codes: A Space Exploration of the Hardware Cost-Performance Tradeoffs. *IEEE Transactions on Quantum Engineering* 3 (1--19). doi: [10.1109/tqe.2022.3174017](https://doi.org/10.1109/tqe.2022.3174017)
246. Julia Padberg. 2017. Hierarchical Graph Transformation Revisited: Transformations of Coalgebraic Graphs. In *Graph Transformation*, Cham. Springer International Publishing, 20--35. doi: [10.1007/978-3-319-61470-0_2](https://doi.org/10.1007/978-3-319-61470-0_2)
247. Adam Paetznick and Krysta M. Svore. 2014. Repeat-until-success: non-deterministic decomposition of single-qubit unitaries. *Quantum Information & Computation* 14, 15--16 (November 2014, 1277--1301)
248. Wojciech Palacz. 2004. Algebraic hierarchical graph transformation. *Journal of Computer and System Sciences* 68, 3 (May 2004, 497--520). doi: [10.1016/s0022-0000\(03\)00064-3](https://doi.org/10.1016/s0022-0000(03)00064-3)
249. Yu Pang, Shaoquan Wang, Zhilong He, Jinzhao Lin, Sayeeda Sultana and Katarzyna Radecka. 2011. Positive Davio-based synthesis algorithm for reversible logic. In *2011 IEEE 29th International Conference on Computer Design (ICCD)*, October 2011. IEEE, 212--218. doi: [10.1109/iccd.2011.6081399](https://doi.org/10.1109/iccd.2011.6081399)
250. Adam Paszke, Sam Gross, Francisco Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai and S. Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Neural Information Processing Systems*. doi: [10.5555/3454287.3455008](https://doi.org/10.5555/3454287.3455008)
251. Tom Peham, Nina Brandl, Richard Kueng, Robert Wille and Lukas Burgholzer. 2023. Depth-Optimal Synthesis of Clifford Circuits with SAT Solvers. In *IEEE International Conference on Quantum Computing and Engineering, QCE 2023, Bellevue, WA, USA, September 17-22, 2023*. IEEE, 802--813. doi: [10.1109/QCE57702.2023.00095](https://doi.org/10.1109/QCE57702.2023.00095)
252. Roger Penrose. 1964. Conformal treatment of infinity. *General Relativity and Gravitation* 43, 3 (901--922 (reprint)). doi: [10.1007/s10714-010-1110-5](https://doi.org/10.1007/s10714-010-1110-5)
253. J. M. Pino, J. M. Dreiling, C. Figgatt, J. P. Gaebler, S. A. Moses, M. S. Allman, C. H. Baldwin, M. Foss-Feig, D. Hayes, K. Mayer, C. Ryan-Anderson and B. Neyenhuis. 2021. Demonstration of the trapped-ion quantum CCD computer architecture. *Nature* 592, 7853 (April 2021, 209--213). doi: [10.1038/s41586-021-03318-4](https://doi.org/10.1038/s41586-021-03318-4)
254. Lorenzo Piroli, Georgios Styliaris and J. Ignacio Cirac. 2021. Quantum Circuits Assisted by Local Operations and Classical Communication: Transformations and Phases of Matter. *Physical Review Letters* 127, 22 (November 2021, 220503). doi: [10.1103/physrevlett.127.220503](https://doi.org/10.1103/physrevlett.127.220503)
255. Detlef Plump. 1999. Term Graph Rewriting. (October 1999)
256. QIR Alliance. 2021. QIR Specification v0.1. Retrieved on 31/12/24 from <https://www.qir-alliance.org/>
257. QIR Alliance. 2023. NVIDIA Joins the QIR Alliance as the Effort Enters Year Two. Retrieved on 02/01/2025 from https://www.qir-alliance.org/posts/year_one_in_review/
258. Quantinuum. 2024. Quantinuum Unveils Accelerated Roadmap to Achieve Universal, Fully Fault-Tolerant Quantum Computing by 2030. Retrieved on 30/12/2024 from <https://www.quantinuum.com/press-releases/quantinuum-unveils-accelerated-roadmap-to-achieve-universal-fault-tolerant-quantum-computing-by-2030>
259. Marcel Quanz, Korbinian Staudacher and Karl Förlinger. 2024. Parallel Quantum Circuit Extraction from MBQC-Patterns. In *2024 IEEE International*

- Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 1078-1087. doi: [10.1109/IPDPSW63119.2024.00179](https://doi.org/10.1109/IPDPSW63119.2024.00179)
260. Md Zamilur Rahman and Jacqueline E. Rice. 2014. Templates for Positive and Negative Control Toffoli Networks
261. Md. Mazder Rahman, Gerhard W. Dueck and Anindita Banerjee. 2012. Optimization of Reversible Circuits Using Reconfigured Templates
262. Abdur Rakib and Ijaz Uddin. 2018. An Efficient Rule-Based Distributed Reasoning Framework for Resource-bounded Systems. *Mobile Networks and Applications* 24, 1 (October 2018, 82--99). doi: [10.1007/s11036-018-1140-x](https://doi.org/10.1007/s11036-018-1140-x)
263. Péter Rakyta and Zoltán Zimborás. 2022. Approaching the theoretical limit in quantum gate decomposition. *Quantum* 6 (May 2022, 710). doi: [10.22331/q-2022-05-11-710](https://doi.org/10.22331/q-2022-05-11-710)
264. Davide Rattacaso, Daniel Jaschke, Marco Ballarin, Ilaria Siloi and Simone Montangero. 2024. Quantum circuit compilation with quantum computers. arXiv: [2408.00077](https://arxiv.org/abs/2408.00077) [quant-ph]
265. Robert Raussendorf and Hans J. Briegel. 2001. A One-Way Quantum Computer. *Physical Review Letters* 86, 22 (May 2001, 5188--5191). doi: [10.1103/PhysRevLett.86.5188](https://doi.org/10.1103/PhysRevLett.86.5188)
266. Xuguang Ren and Junhu Wang. 2016. Multi-Query Optimization for Subgraph Isomorphism Search. *Proceedings of the VLDB Endowment* 10, 3 (November 2016, 121--132). doi: [10.14778/3021924.3021929](https://doi.org/10.14778/3021924.3021929)
267. Arend Rensink. 2004. The GROOVE Simulator: A Tool for State Space Generation. In *Applications of Graph Transformations with Industrial Relevance*. Springer Berlin Heidelberg, 479--485. doi: [10.1007/978-3-540-25959-6_40](https://doi.org/10.1007/978-3-540-25959-6_40)
268. Arend Rensink, Ákos Schmidt and Dániel Varró. 2004. Model Checking Graph Transformations: A Comparison of Two Approaches. In *Graph Transformations. ICGT 2004*. Springer Berlin Heidelberg, 226--241. doi: [10.1007/978-3-540-30203-2_17](https://doi.org/10.1007/978-3-540-30203-2_17)
269. River Riddle. 2021. PDDL: a new declarative rewrite frontend for MLIR. (November 2021). Retrieved on 13/01/2025 (RFC on Discourse) from <https://discourse.llvm.org/t/rfc-pddl-a-new-declarative-rewrite-frontend-for-mlir/4798>
270. Francisco Rios and Peter Selinger. 2018. A Categorical Model for a Quantum Circuit Description Language (Extended Abstract). *Electronic Proceedings in Theoretical Computer Science* 266 (February 2018, 164--178). doi: [10.4204/eptcs.266.11](https://doi.org/10.4204/eptcs.266.11)
271. B. K. Rosen, M. N. Wegman and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '88*. ACM Press, 12--27. doi: [10.1145/73560.73562](https://doi.org/10.1145/73560.73562)
272. Damian Rovara, Lukas Burgholzer and Robert Wille. 2024. A Framework for Debugging Quantum Programs. arXiv: [2412.12269](https://arxiv.org/abs/2412.12269) [quant-ph]
273. Shibdas Roy, Dipankar Home, Guruprasad Kar and Archan S. Majumda. 2011. Towards Normal Forms for GHZW Calculus. In *AIP Conference Proceedings*. AIP, 112--119. doi: [10.1063/1.3635852](https://doi.org/10.1063/1.3635852)
274. Grzegorz Rozenberg. 1997. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific
275. C. Ryan-Anderson, J. G. Bohnet, K. Lee, D. Gresh, A. Hankin, J. P. Gaebler, D. Francois, A. Chernoguzov, D. Lucchetti, N. C. Brown, T. M. Gatterman, S. K. Halit, K. Gilmore, J. A. Gerber, B. Neyenhuis, D. Hayes and R. P. Stutz. 2021. Realization of Real-Time Fault-Tolerant Quantum Error Correction. *Physical Review X* 11, 4 (December 2021, 041058). doi: [10.1103/physrevx.11.041058](https://doi.org/10.1103/physrevx.11.041058)

276. Mehdi Saeedi and Igor L. Markov. 2013. Synthesis and optimization of reversible circuits—a survey. *ACM Computing Surveys* 45, 2 (February 2013, 1--34). doi: [10.1145/2431211.2431220](https://doi.org/10.1145/2431211.2431220)
277. Duncan Sands. 2011. Super-optimizing LLVM IR. (November 2011). Retrieved on 13/01/2025 (LLVM Developer's meeting) from http://llvm.org/devmtg/2011-11/Sands_Super-optimizingLLVMIR.pdf
278. Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja and John Regehr. 2017. Souper: A Synthesizing Superoptimizer. *CoRR* abs/1711.04422
279. Jay D. Sau, Roman M. Lutchyn, Sumanta Tewari and S. Das Sarma. 2010. Generic New Platform for Topological Quantum Computation Using Semiconductor Heterostructures. *Physical Review Letters* 104, 4 (January 2010, 040502). doi: [10.1103/physrevlett.104.040502](https://doi.org/10.1103/physrevlett.104.040502)
280. Christian Scheideler. 2018. Probability amplification. Retrieved lecture notes, online, visited 30/12/2024 from https://cs.uni-paderborn.de/fileadmin-eim/informatik/fg/ti/Lehre/SS_2018/AA/lecture_5.pdf
281. Albert T. Schmitz, Nicolas P. D. Sawaya, Sonika Johri and A. Y. Matsuura. 2024. Graph Optimization Perspective for Low-Depth Trotter-Suzuki Decomposition. *Physical Review A* 109, 4 (April 2024, 042418). doi: [10.1103/PhysRevA.109.042418](https://doi.org/10.1103/PhysRevA.109.042418)
282. Sarah Schneider, Lukas Burgholzer and Robert Wille. 2023. A SAT Encoding for Optimal Clifford Circuit Synthesis. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, January 2023. IEEE. doi: [10.1145/3566097.3567929](https://doi.org/10.1145/3566097.3567929)
283. Erwin Schrödinger. 1935. Die gegenwärtige Situation in der Quantenmechanik. *Naturwissenschaften*
284. Stefan Schwoon. 2013. Efficient verification of sequential and concurrent systems. PhD Thesis. École normale supérieure de Cachan-ENS Cachan
285. Nathan O. Scott and Gerhard W. Dueck. 2008. Pairwise decomposition of toffoli gates in a quantum circuit. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, May 2008. ACM, 231--236. doi: [10.1145/1366110.1366168](https://doi.org/10.1145/1366110.1366168)
286. Roberto Sebastiani and Patrick Trentin. 2015. OptiMathSAT: A Tool for Optimization Modulo Theories. In *Computer Aided Verification*. Springer International Publishing, 447--454. doi: [10.1007/978-3-319-21690-4_27](https://doi.org/10.1007/978-3-319-21690-4_27)
287. Michal Sedláč and Martin Plesch. 2008. Towards optimization of quantum circuits. *Open Physics* 6, 1 (March 2008, 128--134). doi: [10.2478/s11534-008-0039-8](https://doi.org/10.2478/s11534-008-0039-8)
288. Peter Selinger. 2013. Quantum circuits of T-depth one. *Physical Review A* 87, 4 (April 2013, 042302). doi: [10.1103/physreva.87.042302](https://doi.org/10.1103/physreva.87.042302)
289. P. Selinger. 2010. A Survey of Graphical Languages for Monoidal Categories
290. Timos K. Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems* 13, 1 (March 1988, 23--52). doi: [10.1145/42201.42203](https://doi.org/10.1145/42201.42203)
291. V.V. Shende, A.K. Prasad, I.L. Markov and J.P. Hayes. 2002. Reversible logic circuit synthesis. In *IEEE/ACM International Conference on Computer Aided Design, 2002*, November 2002. IEEE, 353--360. doi: [10.1109/iccad.2002.1167558](https://doi.org/10.1109/iccad.2002.1167558)
292. Vivek V. Shende and Igor L. Markov. 2008. On the CNOT-cost of TOFFOLI gates. arXiv: [0803.2316](https://arxiv.org/abs/0803.2316) [quant-ph]
293. Will Simmons. 2021. Relating Measurement Patterns to Circuits via Pauli Flow. *Electronic Proceedings in Theoretical Computer Science* 343 (September 2021, 50--101). doi: [10.4204/eptcs.343.4](https://doi.org/10.4204/eptcs.343.4)
294. Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington and Ross Duncan. 2020. t|ket>: a retargetable compiler for NISQ

- devices. *Quantum Science and Technology* 6, 1 (November 2020, 014003). doi: [10.1088/2058-9565/ab8e92](https://doi.org/10.1088/2058-9565/ab8e92)
295. Ethan Smith, Marc Grau Davis, Jeffrey Larson, Ed Younis, Lindsay Bassman Oftelie, Wim Lavrijsen and Costin Iancu. 2023. LEAP: Scaling Numerical Optimization Based Synthesis Using an Incremental Approach. *ACM Transactions on Quantum Computing* 4, 1 (February 2023, 1--23). doi: [10.1145/3548693](https://doi.org/10.1145/3548693)
296. Kevin C. Smith, Abid Khan, Bryan K. Clark, S.M. Girvin and Tzu-Chieh Wei. 2024. Constant-Depth Preparation of Matrix Product States with Adaptive Quantum Circuits. *PRX Quantum* 5, 3 (September 2024, 030344). doi: [10.1103/prxquantum.5.030344](https://doi.org/10.1103/prxquantum.5.030344)
297. Michael W Sneddon, James R Faeder and Thierry Emonet. 2010. Efficient modeling, simulation and coarse-graining of biological complexity with Nfsim. *Nature Methods* 8, 2 (December 2010, 177--183). doi: [10.1038/nmeth.1546](https://doi.org/10.1038/nmeth.1546)
298. M. Soeken, Z. Sasanian, R. Wille, D. M. Miller and R. Drechsler. 2012. Optimizing the Mapping of Reversible Circuits to Four-Valued Quantum Gate Circuits. In *2012 IEEE 42nd International Symposium on Multiple-Valued Logic*, May 2012. IEEE, 173--178. doi: [10.1109/ismvl.2012.64](https://doi.org/10.1109/ismvl.2012.64)
299. Richard P. Stanley. 2015. *Catalan Numbers*. Cambridge University Press. doi: [10.1017/CBO9781139871495](https://doi.org/10.1017/CBO9781139871495)
300. Damian S. Steiger, Thomas Häner and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum* 2 (January 2018, 49). doi: [10.22331/q-2018-01-31-49](https://doi.org/10.22331/q-2018-01-31-49)
301. Suzana Stojković, Radomir Stanković, Claudio Moraga and Milena Stanković. 2019. Reversible Circuits Synthesis from Functional Decision Diagrams by using Node Dependency Matrices. *Journal of Circuits, Systems and Computers* 29, 05 (August 2019, 2050079). doi: [10.1142/s0218126620500796](https://doi.org/10.1142/s0218126620500796)
302. Xiaoming Sun, Guojing Tian, Shuai Yang, Pei Yuan and Shengyu Zhang. 2023. Asymptotically Optimal Circuit Depth for Quantum State Preparation and General Unitary Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 10 (October 2023, 3301--3314). doi: [10.1109/TCAD.2023.3244885](https://doi.org/10.1109/TCAD.2023.3244885)
303. Matthew Sutcliffe and Aleks Kissinger. 2025. Fast classical simulation of quantum circuits via parametric rewriting in the ZX-calculus. arXiv: [2403.06777](https://arxiv.org/abs/2403.06777) [quant-ph]
304. Michelle Wynne Sze, Yao Tang, Silas Dilkes, David Muñoz Ramo, Ross Duncan and Nathan Fitzpatrick. 2025. Hamiltonian dynamics simulation using linear combination of unitaries on an ion trap quantum computer. arXiv: [2501.18515](https://arxiv.org/abs/2501.18515) [quant-ph]
305. J. I. Cirac, T. Pellizzari and P. Zoller. 1995. Decoherence, Continuous Observation, and Quantum Computing: A Cavity QED Model. *Physical Review Letters* 75, 21 (November 1995, 3788--3791). doi: [10.1103/PhysRevLett.75.3788](https://doi.org/10.1103/PhysRevLett.75.3788)
306. Andrew S. Tanenbaum, Hans van Staveren and Johan W. Stevenson. 1982. Using Peephole Optimization on Intermediate Code. *ACM Transactions on Programming Languages and Systems* 4, 1 (January 1982, 21--36). doi: [10.1145/357153.357155](https://doi.org/10.1145/357153.357155)
307. Ross Tate, Michael Stepp, Zachary Tatlock and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. *ACM SIGPLAN Notices* 44, 1 (January 2009, 264--276). doi: [10.1145/1594834.1480915](https://doi.org/10.1145/1594834.1480915)
308. Robert R. Tucci. 2005. An Introduction to Cartan's KAK Decomposition for QC Programmers. arXiv: [quant-ph/0507171](https://arxiv.org/abs/quant-ph/0507171) [quant-ph]

309. Ijaz Uddin, Hafiz Mahfooz Ul Haque, Abdur Rakib and Mohamad Rafi Segi Rahmat. 2016. Resource-Bounded Context-Aware Applications: A Survey and Early Experiment
310. Yosuke Ueno, Masaaki Kondo, Masamitsu Tanaka, Yasunari Suzuki and Yutaka Tabuchi. 2021. QECOL: On-Line Quantum Error Correction with a Superconducting Decoder for Surface Code. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, December 2021. IEEE, 451--456. doi: [10.1109/dac18074.2021.9586326](https://doi.org/10.1109/dac18074.2021.9586326)
311. Valve Corporation. 2024. Steam Hardware & Software Survey: December 2024. (December 2024). Retrieved on 30/01/2025 from <https://store.steam-powered.com/hwsurvey/processormfg/>
312. Vivien Vandaele, Simon Martiel and Timothée Goubault de Brugière. 2022. Phase polynomials synthesis algorithms for NISQ architectures and beyond. *Quantum Science and Technology* 7, 4 (Septempter 2022, 045027). doi: [10.1088/2058-9565/ac5a0e](https://doi.org/10.1088/2058-9565/ac5a0e)
313. Gergely Varró and Frederik Deckwerth. 2013. A Rete Network Construction Algorithm for Incremental Pattern Matching
314. Rakesh M. Verma. 1995. Transformations and confluence for rewrite systems. *Theoretical Computer Science* 152, 2 (December 1995, 269--283). doi: [10.1016/0304-3975\(94\)00255-0](https://doi.org/10.1016/0304-3975(94)00255-0)
315. Hao Wang, Erjia Xiao, Songhuan He, Zhongyi Ni, Lingfeng Zhang, Xiaokun Zhan, Yifei Cui, Jinguo Liu, Cheng Wang, Zhongrui Wang and Renjing Xu. 2024. CIM-Based Parallel Fully FFNN Surface Code High-Level Decoder for Quantum Error Correction. arXiv: [2411.18090](https://arxiv.org/abs/2411.18090) [cs.AR]
316. Andrew Shell Waterman. 2016. Design of the RISC-V Instruction Set Architecture. PhD Thesis. University of Berkeley
317. Jens H. Weber. 2017. GRAPE – A Graph Rewriting and Persistence Engine. In *Graph Transformation*. Springer International Publishing, 209--220. doi: [10.1007/978-3-319-61470-0_13](https://doi.org/10.1007/978-3-319-61470-0_13)
318. Jens H. Weber. 2022. Tool Support for Functional Graph Rewriting with Persistent Data Structures - GrapeVine. In *Graph Transformation. ICGT 2022*. Springer International Publishing, 195--206. doi: [10.1007/978-3-031-09843-7_11](https://doi.org/10.1007/978-3-031-09843-7_11)
319. Mathias Weiden, Ed Younis, Justin Kalloor, John Kubiawicz and Costin Iancu. 2023. Improving Quantum Circuit Synthesis with Machine Learning. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE. doi: [10.1109/QCE57702.2023.00093](https://doi.org/10.1109/QCE57702.2023.00093)
320. John van de Wetering. 2020. ZX-calculus for the working quantum computer scientist. arXiv: [2012.13966](https://arxiv.org/abs/2012.13966) [quant-ph]
321. John van de Wetering, Richie Yeung, Tuomas Laakkonen and Aleks Kissinger. 2024. Optimal compilation of parametrised quantum circuits. arXiv: [2401.12877](https://arxiv.org/abs/2401.12877) [quant-ph]
322. John van de Wetering and Matt Amy. 2024. Optimising quantum circuits is generally hard. arXiv: [2310.05958](https://arxiv.org/abs/2310.05958) [quant-ph]
323. Deborah L. Whitfield and Mary Lou Soffa. 1997. An approach for exploring code improving transformations. *ACM Transactions on Programming Languages and Systems* 19, 6 (November 1997, 1053--1084). doi: [10.1145/267959.267960](https://doi.org/10.1145/267959.267960)
324. Robert Wille and Rolf Drechsler. 2010. Effect of BDD Optimization on Synthesis of Reversible and Quantum Logic. *Electronic Notes in Theoretical Computer Science* 253, 6 (March 2010, 57--70). doi: [10.1016/j.entcs.2010.02.006](https://doi.org/10.1016/j.entcs.2010.02.006)

325. Max Willsey. 2021. Practical and Flexible Equality Saturation. PhD Thesis. University of Washington
326. Max Willsey. 2025. egg: egraphs good. Retrieved on 14/01/2025 (code repository) from <https://github.com/egraps-good/egg>
327. Paul Wilson and Fabio Zanasi. 2022. The Cost of Compositionality: A High-Performance Implementation of String Diagram Composition. *Electronic Proceedings in Theoretical Computer Science* 372 (November 2022, 262--275). doi: [10.4204/eptcs.372.19](https://doi.org/10.4204/eptcs.372.19)
328. David Windler, Qunsheng Huang, Arianne Meijer-van de Griend and Richie Yeung. 2024. Architecture-Aware Synthesis of Stabilizer Circuits from Clifford Tableaus. arXiv: [2309.08972](https://arxiv.org/abs/2309.08972) [quant-ph]
329. Glynn Winskel. 1987. Event structures
330. W. K. Wootters and W. H. Zurek. 1982. A single quantum cannot be cloned. *Nature* 299, 5886 (October 1982, 802--803). doi: [10.1038/299802a0](https://doi.org/10.1038/299802a0)
331. Xin-Chuan Wu, Marc Grau Davis, Frederic T. Chong and Costin Iancu. 2020. QGo: Scalable Quantum Circuit Optimization Using Automated Synthesis. arXiv: [2012.09835](https://arxiv.org/abs/2012.09835) [quant-ph]
332. Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar and Zhihao Jia. 2022. Quartz: Superoptimization of Quantum Circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, June 2022. Association for Computing Machinery, 625--640. doi: [10.1145/3519939.3523433](https://doi.org/10.1145/3519939.3523433)
333. Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu and Aws Albarghouthi. 2023. Synthesizing Quantum-Circuit Optimizers. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023, 835--859). doi: [10.1145/3591254](https://doi.org/10.1145/3591254)
334. Jin Yang, Michael I. Monine, James R. Faeder and William S. Hlavacek. 2008. Kinetic Monte Carlo method for rule-based modeling of biochemical networks. *Physical Review E* 78, 3 (September 2008, 031910). doi: [10.1103/physreve.78.031910](https://doi.org/10.1103/physreve.78.031910)
335. Yichen Yang, Mangpo Phitchaya Phothilimtha, Yisu Remy Wang, Max Willsey, Sudip Roy and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. CoRR abs/2101.01332. doi: [10.48550/ARXIV.2101.01332](https://arxiv.org/abs/2101.01332)
336. Richie Yeung, Konstantinos Meichanetzidis, Alexandre Krajenbrink and François Charton. 2024. Teaching small transformers to rewrite ZX diagrams. *MathAI submission*
337. Ed Younis and Costin Iancu. 2022. Quantum Circuit Optimization and Transpilation via Parameterized Circuit Instantiation. arXiv: [2206.07885](https://arxiv.org/abs/2206.07885) [quant-ph]
338. Ed Younis, Koushik Sen, Katherine Yelick and Costin Iancu. 2021. QFAST: Conflating Search and Numerical Optimization for Scalable Quantum Circuit Synthesis. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, October 2021. IEEE, 232--243. doi: [10.1109/QCE52317.2021.00041](https://doi.org/10.1109/QCE52317.2021.00041)
339. Yuxuan Zhao, Qi Sun, Zhuolun He, Yang Bai and Bei Yu. 2023. AutoGraph: Optimizing DNN Computation Graph for Parallel GPU Kernel Execution. *Proceedings of the AAAI Conference on Artificial Intelligence* 37, 9 (June 2023, 11354--11362). doi: [10.1609/aaai.v37i9.26343](https://doi.org/10.1609/aaai.v37i9.26343)
340. Edward Zulkoski. 2018. Understanding and Enhancing CDCL-based SAT Solvers. PhD Thesis. University of Waterloo