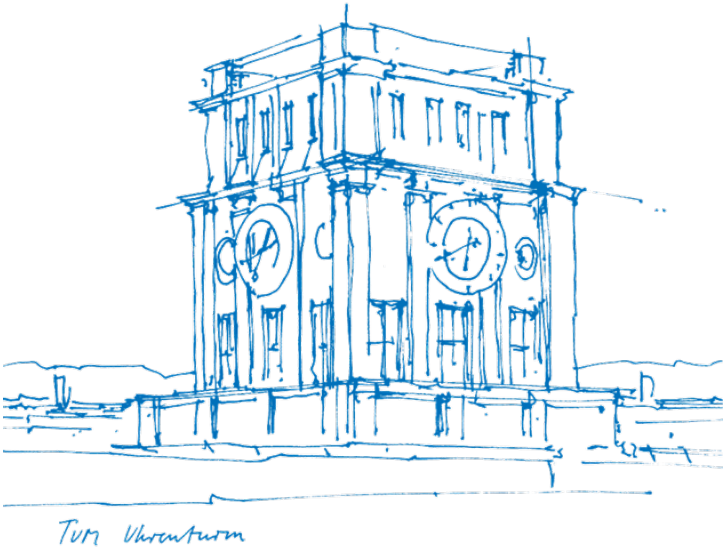
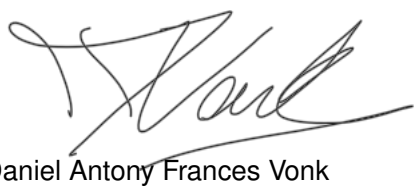
Chair for Distributed Systems and Operating Systems
School of Computing, Information and Technology
Technical University of Munich

A Formal Verification Model and Language for the Correctness of Programs on Neutral-Atom Quantum Computers

Daniel Antony Frances Vonk



I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.



Munich, 19.10.2025 Daniel Antony Frances Vonk

A Formal Verification Model and Language for the Correctness of Programs on Neutral-Atom Quantum Computers

Daniel Antony Frances Vonk

Technical University of Munich

A thesis submitted in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

at the School of Computing, Information and Technology.

October 2025

Examiner: Prof. Pramod Bhatotia
Supervisor: Francisco Romão MSc.

Contents

1 Introduction

Quantum physics and thus quantum information science is sometimes presented as an exotic world, where counterintuitive properties, such as entanglement or tunnelling, make such systems incomprehensible to humans. However, the properties behind quantum computing are well-defined and rigorous and contemporary research in quantum computers, such as in the area of *neutral-atom quantum computing*, shows that feasible and useful systems are not only in the realm of possibility but also soon, practicality. As quantum computers move from specialised, one-off physics experiments to reusable machines, many of the engineering challenges from traditional computing resurface. For example, in traditional computing, the problem of CPU utilisation in the presence of high memory latency is a constraint for which programs must be optimised and for which hardware must be specifically architected. One such architectural choice is to allow multi-threading, where several threads of execution can be run in parallel on a CPU as long as they do not contend for resources on the CPU. As quantum computing progresses and becomes a general-purpose computing platform, these architectural and compiler-specific choices will become more pertinent, just as they have in classical computing.

However, to first understand the process of quantum computing, one should start from a foundational level. Computing relies on storing information in discrete states for digit representation. In the earliest days of computing history, these could be the two states of a thermionic valve, that is either current flowing or no current flowing, but in modern computing, these states are usually represented using transistor logic, where a voltage defined by the common rail is on and a voltage of ground is off; but there is in principle, no limit to the number of discrete states a computer could use and indeed ternary computers, that is those relying on three states (-1, 0, +1), have been built, though they are more error-prone than the simpler two state solution and have thus been consigned to history. These states correspond to the *bits* (or *digits* for a generic radix) in a computer and for a computer with n bits, this leaves it with a possible 2^n possible states. Though the computer can represent only one of these at any single point in time.

Both the formal and practical models that underpin the evolution of these states are, of course, well-known. In the formal sense, these states correspond to the contents of the tape in a finite Turing Machine. The contents of this tape is transformed by a state transition function and this provides the computing power for the device. In practical models of classical machines, such as the ones used today in everyday devices, the state is contained in the RAM, whose contents are modified by a stream of instructions which are processed on the CPU. This remains much the same in quantum computing, though a qubit is rather a superposition over two bases in a Hilbert space rather than a single value. Specifically, the state of a single qubit is written as $|\psi\rangle$ and is a *waverform* which assigns an amplitude to each basis of the Hilbert space, which are $|0\rangle$ and $|1\rangle$, in this instance. To evolve the state, one uses *unitary* transformation matrices on this state, written as $\langle A|\psi\rangle$ and this provides the computational power of the device. Because the state is a *superposition* and not simply a linear combination, alternate paths in

the configuration space can interact with each other through the wave's phase and it is this ability which makes quantum computing more powerful than classical computing for several well-known problems, such as integer factorisation, through Shor's algorithm.

Despite the formal existence of mathematical models for quantum computers spanning back decades and many quantum algorithms having been written, it remains a challenge to actually implement these devices, with each quantum platform having its own distinct challenges. DiVincenzo formulated the necessary conditions for implementing a quantum computer shortly before the millennium [1] and this has guided physical research in the subsequent decades. The requirements are:

1. A scalable physical system with well-characterised qubits.
2. The ability to initialise the state of the qubits to a simple fiducial state.
3. Long relevant quantum coherence times.
4. A "universal" set of quantum gates.
5. A qubit-specific measurement capability.

As stated, several competing platforms exist in current research. Therefore, these conditions provide a benchmark to which quantum computing platforms can be evaluated. Condition (1) emphasises the necessity of a scalable system as solving real-world problems with a quantum computer requires hundreds to thousands of qubits. Condition (2) is important because without well-defined initialisation (analogous to resetting all bits to 0 in a classical computer), it is not possible to encode the input to an algorithm. Condition (3) sets a time "budget" before the quantum state is lost to the environment. All algorithmic work needs to be done within this time hence a longer coherence time is better. Condition (4) means the architecture can approximate any unitary transformation, which is necessary to run general algorithms. Condition (5) says that it must be possible to read the result of a quantum computation such that the other qubits in the system are not disturbed.

The most mature quantum computing platform are the *superconducting* quantum computers, under which the devices developed by IBM, such as *IBM Q System One* or *IBM Heron* are perhaps the best well-known. These devices cool electrical circuits to cryogenic temperatures. Below a critical temperature, causing certain materials to exhibit zero electrical resistance. This, for example, means that no energy is dissipated as heat (as dictated by $P = I^2 R$), which helps to maintain long coherence times and furthermore, also allows representing qubits through anharmonic oscillation, where the lowest two levels of oscillation act as these basis states. Within these states, operations can then be driven by specific microwave pulses. For example, a pulse resonant with the transition frequency ω_{01} implements a *flip* from $|0\rangle$ to $|1\rangle$ for the qubit.

The advantages of this platform are that they can apply these pulses very fast (on the order of tens of nanoseconds) and scale up to hundreds of qubits. In fact, IBM's latest quantum computer, IBM Heron, supports 156 qubits [2]. However, as stated, they must be cooled to milli-kelvin temperatures, which is expensive and complex. Moreover, running user-defined circuits on them requires a relatively complex compilation process as the architecture of the qubits (the *connectivity*) is fixed and pre-defined.

Another popular platform are the *trapped ion* quantum computers. In this model, ions, such as Yb-171+, are confined in place by electromagnetic Paul or Penning traps. The basis states $|0\rangle$ and $|1\rangle$ are encoded as stable internal states of the ions, separated by an energy splitting $\hbar\omega_{01}$. Just as previously, pulsing a laser resonant with this transition can, amongst other things, flip the state of the qubit from $|0\rangle$ to $|1\rangle$, which allows single-qubit unitary transformations to be performed. Because these atoms are able to oscillate in their trap, their motion is quantised (phonons) and this allows a laser pulse to address their shared motional modes, which is used to implement multi-qubit gates.

Trapped ion quantum computers have the distinction of having exceptionally long coherence times, i.e. on the order of seconds to minutes and far higher than superconducting quantum computers. Their single and multi-qubit gates can also be applied reliably (with an error rate of less than 1%). However, scaling these systems is challenging due to the fragility of large ion chains. Currently, leading systems like *Quantinuum H2* have achieved a maximum of 56 qubits [3].

Another leading platform, which is the focus of this thesis, are *neutral atom* quantum computers (NAQCs). These are built upon the principle of atoms which have the same number of protons as electrons, that is to say neutrally charged or *neutral atoms* (NA). Examples of such atoms include Rb or Yb. In this manner, they share some commonality with trapped ion quantum computers, but because of their neutrality, their implementation is conceptually much simpler. For example, neutral atoms can be trapped using only *optical tweezers*. These tweezers rely on lasers emanating light to keep atoms in place and this is possible again due to neutrality. By contrast, a charged atom would need to resort to using electrodes. Another subsequent advantage of neutrality is that the atoms interact only very weakly with extrinsic fields, which is a major source of decoherence. In fact, neutral atoms exhibit some of the highest coherence times of quantum platforms. For example, *Atom Computing*, a startup involved in NA quantum computing, reported coherence times of up to 40 seconds.

The perhaps most interesting property of neutral atom quantum computers is their scalability. The aforementioned vendor Atom Computing boasts of a NAQC which hosts over *1180* qubits. Likewise, the vendor *PASQAL* currently produces a 324 qubit machine with plans to extend to over 1000. These platforms are then a promising solution for DiVincenzo's first criterion.

These large qubit numbers are possible due to the use of optical tweezers. In this method, a laser beam is split using a lens and each maxima of the wave becomes a trap for an atom. The marginal cost to further subdividing the beam is very small and hence it becomes possible to create large grids of traps. Neutral atoms are all identical to each other therefore these atoms are able to be placed in any of these many traps. These traps can even be dynamically reconfigured, which allows the operator to change the grid topology as well as move atoms around on the grid.

The *hyperfine ground states* of the neutral atoms are used to represent $|0\rangle$ and $|1\rangle$. Just as for the other platforms, neutral atoms use microwave or laser pulses to drive Rabi oscillations between the two states to create the necessary rotation. Multi-qubit gates are implemented using the *Rydberg blockade* effect. This is an effect which momentarily puts an atom in a highly excited electronic state which is far above either $|0\rangle$ or $|1\rangle$ and is known as a *Rydberg state*. If an

atom is in such a state, then it blockades neighbouring atoms from moving into a higher state. This conditional phase effect between two atoms can be used to implement various multi-qubit gates.

These gates can also be performed relatively fast, on the order of 100ns - 1μs though their error rate of approximately 0.05 is still higher than superconducting QCs, for example. Nevertheless, NAQCs present a compelling platform that meets many of the criteria DiVincenzo posed in his work and this is perhaps a reason for their recent popularity in the research space.

For a quantum computer of any kind, let alone a NAQC, to be useful, it must be possible to translate circuit descriptions into machine-specific operations for the quantum computer. In the case of neutral atom computers, it must translate a circuit into a sequence of atom movements and laser/microwave pulses. These compiled operations should be a correct representation of the program and the program should be run as fast as possible. A compiler could use optimisations in the compiler, such as fusing or reordering gates, to further increase performance.

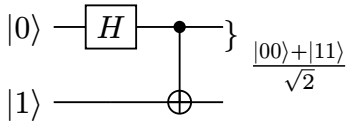


Figure 1: A simple circuit and its effect upon the 0 qubit.

Once a program has been compiled to the target hardware, it can be executed by transferring the resulting instructions to the quantum computer's controller, which steers the actual physical hardware such as the trap lasers and Rydberg lasers. However, some circuits require significant numbers of qubits to perform their algorithms and as NAQCs scale to hundreds or even thousands of qubits, one might imagine that it becomes equally trivial to make use of the extra qubits. Unfortunately, this is not the case as even small error rates in gate applications exponentiate to create large losses in fidelity. For example, a highly-nested circuit with a depth of 1000 qubits would only have a probability $0.99^{1000} \approx 0.00004$ of running without an error. This is for all intents and purposes a complete barrier to running such a circuit on current neutral atom hardware. Another simultaneous issue with neutral atom quantum computing is their complex initialisation procedure. Although this is not covered under the second condition in DiVincenzo's criteria, initialisation should preferably be fast as possible as it must be performed for every circuit execution on the quantum computer. In the case of NAQCs, this procedure involves loading the neutral atoms into a vacuum chamber, imaging the chamber using a camera to ascertain their exact locations, then sorting them into their correct trap locations and finally verifying this through another camera image. This process can take tens of milliseconds and presents a large fixed-cost overhead to executing a circuit [4], [5].

These limitations make it challenging to run real-world circuits on neutral-atom quantum computers, even if their hardware theoretically makes them well-suited to the problem. However, techniques such as multi-programming, namely the process of executing multiple circuits in parallel on the neutral atom grid have been shown to partly mitigate some of these issues by amortising the initialisation overhead over multiple circuits and also increasing utilisation of the QPU's grid space. One such example of a compiler that does this is MultiQ [6].

However, as compilers increase in complexity, it becomes more likely that bugs appear in their generated code. These could either be obvious bugs, like discontinuities in the path of an atom on the grid or even subtler ones, such as timing errors or resource scheduling contention. Therefore, this thesis seeks to implement a verification system to analyse the generated output of these compilers to detect such bugs. In order to have a general method for neutral-atom compilers in general, the verifier must also introduce a common intermediate language or IR which compilers can target. This should be flexible enough that it can be used in a variety of different neutral-atom quantum computers and be flexible enough to allow for future modification, given the frequently changing nature of quantum computing research.

2 Background

This chapter introduces the necessary mathematical background on quantum computing as well as a short description on the functionality and implementation of neutral atom quantum computers. Compilers for quantum computers are then introduced along with their standard structure, including a description of the standard textual descriptions of circuit diagrams (the input) and mid-level IR (the output). To conclude, there is a short background on sheaves and category theory, both of which are central tools to the verification model presented in this thesis.

2.1 Quantum Computing Basics

Just as in classical computing, the process of computing using quantum states can be formalised using Turing machines [7]. Formally, a single-tape *quantum Turing machine* with a finite set Q of states and a finite alphabet Σ is given by a transition function:

$$\delta : \Sigma \times Q \times \Sigma \times Q \times \{\uparrow, \downarrow, \rightarrow\} \rightarrow \mathbb{C}_{[0,1]}$$

The fundamental difference to classical Turing machines is that δ provides an *amplitude* to each transition instead of simply the next state. In this way, if c_0 is the starting *configuration*, all of its successor configurations are $c_1, ..., c_k$, and the amplitude α_i is assigned to transition to configuration c_i, then:

$$\sum_{i=1}^{k} |\alpha_i|^2 = 1$$

Therefore, this value $|\alpha_i|^2$ can be seen as the *probability* of transitioning from c_0 to c_i and it forms a tree structure, where each node is a configuration and has successor configurations weighted by the amplitude of δ. However, as the value α_i is an amplitude and not a probability, it is possible that

$$\left| \sum_{i=1}^{k} \alpha_i \right|^2 > \sum_{i=1}^{k} |\alpha_i|^2$$

holds and this would be an example of *constructive interference*. It may also be the case that there is *destructive interference* at one level of the tree, in which case the $>$ in the formula would be strictly inverted. There could also be two equal successor configurations α_i and α_j which destructively interfere so that their sum implies probability 0, i.e. despite there existing two paths two paths to this configuration, the probability of obtaining the result is strangely 0. This counterintuitive property is a used as a basic strategy in quantum algorithms: correct answers are made to have large probabilities through constructive interference whilst incorrect answers are made to have small probabilities through the use of destructive interference.

Through this quantum Turing machine model, it becomes apparent that the theoretical notion of computability, in the Church-Turing thesis sense, still holds under the presence of quantum

mechanics. They are able to compute the exact same set of partial recursive functions (i.e. enumerate the recursively enumerable languages) that normal Turing machines can. However, the efficiency of how quantum Turing machines can compute them is markedly different and they define many new complexity classes (e.g. BQP, QMA etc.), though this is out of the scope for this thesis.

As a segue to the quantum mechanics interpretation of computation, note that in a quantum Turing machine, the set of all possible configurations $\mathcal{C}$ of this Turing machine can be seen as a vector space. Specifically, to each configuration $c \in \mathcal{C}$, a basis vector $|c\rangle$ is associated. These bases form an orthonormal set if we interpret an inner product as the following:

$$\langle c|c'\rangle = \begin{cases} 1 \text{ if } c = c' \\ 0 \text{ otherwise} \end{cases}$$

The state of the quantum Turing machine is then represented as a linear combination over the configurations $c \in \mathcal{C}$, as determined by the transition function δ:

$$|\psi\rangle = \sum_{c \in \mathcal{C}} \alpha_c |c\rangle$$

where $\alpha_c \in \mathbb{C}$ is the corresponding amplitude to each configuration, which has evolved over time. The set of all possible such states that have finite ℓ_2 norm can then be enumerated:

$$\mathcal{H} = \left\{ \sum_{c \in \mathcal{C}} \alpha_c |c\rangle : \sum_{c \in \mathcal{C}} |\alpha_c|^2 < \infty \right\}$$

$\mathcal{H}$ is an example of a *Hilbert space*. Formally speaking, a Hilbert space is the space of all *complex valued functions* on a countable set which are bounded by the ℓ_2 norm (i.e. $< \infty$). This structure is central to the study of quantum mechanics as it serves as a description of quantum states and models their evolution [7].

From the above definition of the inner product, a *norm* is induced:

$$\||\psi\rangle\|^2 = \langle\psi|\psi\rangle = \sum_{c \in \mathcal{C}} |\alpha_c|^2$$

Due to the Born rule, an important requirement is that the norm of these states always remains equal to 1, i.e. $\|\psi\| = 1$ (notably the linear algebra equivalent to the sum of squares probability definition previously given at the beginning of this chapter). The state of the QTM can be evolved using the transition function δ, which induces a linear operator U on $\mathcal{H}$:

$$U|c\rangle = \sum_{c' \in \mathcal{C}} \delta(c, c')|c'\rangle$$

An important caveat is that U must be a unitary operator as these are exactly the complex-valued operators that preserve norm (i.e $\|U|\psi\rangle\| = \|\psi\|$) [8].

This construction shows that computation on quantum Turing machines reduces down to Hilbert spaces and unitary evolutions thereof. In fact, quantum Turing machines are mostly useful as a theoretical model of computation for comparison to classical computing. The

Turing machine model of discrete states with tapes whose heads move back and forth isn't a close analogue to the operation of physical implementations of quantum computers. Instead, real devices, such as superconducting qubits or neutral atom quantum computers are usually directly described using quantum states of atoms or other quantum particules in a Hilbert space and their evolution described by unitary operators.

Specifically, the state of a *qubit* is described by a Hilbert space $\mathcal{H} = \mathbb{C}^2$, which has a standard orthonormal basis:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

These bases $|0\rangle, |1\rangle$ correspond directly to a quantum state such as the polarisation of a photon or the energy level of an atom. The state of a qubit is then a superposition of these two basis states, subject to the normalisation requirement:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \text{ where } \alpha, \beta \in \mathbb{C} \text{ and } |\alpha|^2 + |\beta|^2 = 1$$

Naturally, a quantum system of only one qubit is not very useful. Fortunately, Hilbert spaces can be composed using the *tensor product*. As such, a n qubit Hilbert space can be created by considering the orthonormal basis states $\mathcal{B}_1, ..., \mathcal{B}_n$ of each of the n qubits, then:

$$\mathcal{B}_1 \otimes ... \otimes \mathcal{B}_n = \otimes_{i=1}^{n} \mathcal{B}_i = \{x_1 \otimes ... \otimes x_k : x_i \in \mathcal{B}_i\}$$

is the orthonormal basis for the new Hilbert space $\mathcal{H} = \otimes_{i=1}^{n} \mathcal{H}_i$, for which $\mathcal{H} \cong \mathbb{C}^{2^n}$ holds.

The perhaps most common transformation one can make on a product state would be to apply a Hadamard transformation H on a 2-qubit system. This single-qubit unitary transformation, or gate, maps the basis states into equal superpositions. As H is applied on one qubit only, the second qubit is multiplied with I. For this, one would start with the zero representation:

$$|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

Note that $|00\rangle \neq (0000)^T$ due to this being an invalid state due to its norm. Then one applies the tensor product between H and I:

$$H \otimes I = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

This results in the state of the first qubit being altered to an equal superposition of its bases:

$$(H \otimes I)|00\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$$

This transformation is the first step for many quantum algorithms, such as Grover's search or Shor's factoring algororithm as it introduces quantum parallelism: by creating a superposition of all possible inputs, it allows a unitary transformation to "act" on every different branch of the input in parallel.

This example applied the transformation $H \otimes I|00\rangle$ and only once. However, as stated, any transformation that preserves the norm of a state $|\psi\rangle$ is possible. Over the complex numbers, these transformations are exactly the unitary transformations (note that over the reals, these are exactly the orthogonal transformations). These transformations can be used to evolve the state in a continuous fashion, which means one can write the state as an evolution over time:

$$|\psi(t)\rangle = U(t)|\psi(0)\rangle \text{ where } U(t)^\dagger U(t) = I$$

Assuming additionally that the evolution is strongly continuous in time (i.e. has only the parameter t and is a unitary group), then through the use of *Stone's theorem* (a standard result in functional analysis), $U(t)$ has the form:

$$U(t) = \exp\left(-\frac{iHt}{\hbar}\right)$$

for some hermitian matrix H (not the Hadamard!). Letting $|\psi(t)\rangle = U(t)|\psi(0)\rangle$, one can then use the standard derivative over time of this function to derive the famous *Schrödinger's equation*, which governs how quantum states evolve [9]:

$$i\hbar\frac{d}{\mathrm{d}t}|\psi(t)\rangle = H|\psi(t)\rangle$$

This connection explains why discrete unitary transformations on quantum states are permissible, as they can be seen as a short, behaved segments of the Hamiltonian evolution of a quantum space. Some quantum computers do in fact use continuous time evolution under a Hamiltonian, such as D-Wave's quantum annealing [10] solution. In this model, there is no "program" to be run on the computer as all of the information is encoded by choosing the Hamiltonian. However, most platforms, such as neutral atom or superconducting quantum computers, prefer to abstract away from continuous dynamics and use discrete unitary gates. This is known as the *circuit model* of quantum computing.

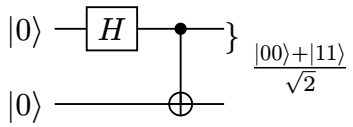


Figure 2: An example of the circuit model: A Hadamard gate is applied to the first qubit of a 2-qubit system.

In this model, individual qubits are represented as wires and their connections, the gates, are unitary transformations on these qubits. Just as in classical computing, there exist small sets of universal gates, such as Hadamard, T and CNOT, which can approximate any unitary gate (to an arbitrary precision). One crucial difference, though, is that quantum gates are *reversible*, and algorithms often exploit this to "uncompute" temporary values [11].

Assuming now that the state has been fully evolved, whether it be by continuous evolution or a sequence of discrete unitary transformations, one would like to be able to read the state in order to receive the output of an algorithm. This is where quantum computing again differs significantly from classical computing: in classical computing, one can measure a state at any time, however, in the Copenhagen interpretation of quantum mechanics, quantum properties

are not determined except when they are measured and it's only when they are measured that they take a classical value [7]. To do this, one usually defines the *computational basis* $\{|x\rangle : x \in \{0,1\}^n\}$ (all bit strings of length n). The probability of observing an x from this set then is $p(x) = |\langle x|\psi\rangle|^2$ as dictated by the *Born rule*. After this *observation*, the quantum state $|\psi\rangle$ "collapses" to x and no further measurements are possible.

2.2 Neutral Atom Quantum Computers

The primary quantum computing platform considered in this thesis is the *neutral atom* quantum computing platform. Although there are several other platforms which could have been chosen for this implementation of this verifier model, neutral atoms present an interesting challenge for verification as they are a highly flexible architecture but still have many physics-informed properties which can be statically checked [12]. However, in order to first understand the types of properties that can be guaranteed through the formal verification of the IR on neutral atom quantum computers, it helps to have a thorough understanding of the machines themselves.

At the heart of a neutral atom QPU lies a stainless steel vacuum chamber, measuring approximately 20-40cm in each dimension, which stores the atoms on which the computer operates. The collision rate of these atoms follows the formula

$$\Gamma = n\sigma v$$

where n is the background gas density, $\sigma \approx 10^{-14}\ \mathrm{cm}^2$ is the scattering cross-section and v is the thermal velocity of the background molecules [13]. As such it is paramount that the vaccum chamber be operated at extremely low pressures, usually below 10^{-10} mbar and known as an ultra-high vacuum (UHV). This ensures that $\Gamma \ll 1$ Hz, meaning that background gas molecules rarely disturb the atoms.

As stated in the introduction, not just any atoms may be used in the chamber, but rather *neutral* atoms, such as rubidium (^{87}Rb), strontium (^{87}Sr) or caesium (^{133}Cs). These atoms are procured from an atomic source, such as a *dispenser oven*. These ovens heat alkali metals (e.g. of rubidium) in a sealed reservoir, causing them to evaporate into a cloud of neutral atoms, which is then piped into the chamber [14]. However, upon entry, these atoms have thermal velocities of *hundreds* of metres per second and therefore are far too fast to be trapped directly.

In order to slow down atoms, a multi-step process, known as *magneto-optical trapping*, or MOT, is followed. The first cooling stage uses what is known as *Doppler cooling* and is itself an interesting application of quantum physics (by utilising discrete atomic energy levels). This process uses three pairs of red-detuned laser beams and a *quadrupole* magnetic field, which are configured so that the beams of the lasers all intersect one another at the centre of the magnetic coil [12].

Atoms absorb light most strongly when the light frequency (say ω_L) matches their internal transition frequency ω_0. However, if an atom is moving at velocity v along the laser beam's direction, then the atom experiences a *Doppler-shifted* frequency

$$\omega' = \omega_L\left(1 - \frac{v}{c}\right) \approx \omega_L - kv \text{ where } k = \frac{2\pi}{\lambda}$$

Thus, the frequency of the laser can be set to slightly below resonance (hence the red-detuned frequency in the setup) so that "stationary" atoms do not absorb much light but for atoms moving towards the beam, an added kv term is added to the laser frequency, making it very close to the resonant frequency. Photons have momentum and when an atom absorbs one, then due to conservation of momentum, it immediately receives an impulse of $\hbar k$ in the direction of the laser beam. Over several repeated cycles, this process can slow a rubidium-87 atom to approximately 146 μK (velocities on the order of cm/s) in a few milliseconds [15].

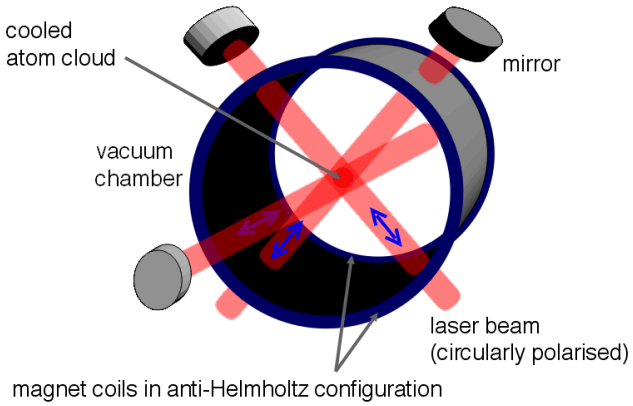


Figure 3: A cross-section of the magneto-optical trap, where three laser and a magnetic field are used to slow atoms down to millikelvin temperatures.

However, the MOT does not just employ Doppler cooling to produce *cold atoms* for use in the chamber. It also makes use of the quadrupole magnetic field, which has hitherto not been discussed but is also important: in the centre of the trap, the quadrupole magnetic field $\boldsymbol{B}(\boldsymbol{r}) = 0$ but increases linearly with distance away from the centre. This causes the atomic resonance frequency to shift via the *Zeeman effect*. Although the specifics aren't discussed here, this effect ensure that as atoms move further away from the trap centre, the laser beam pointing towards the centre is shifted closer to resonance (and the opposite holds for the converse direction) [16]. This creates a restoring force and atoms are thus moved back to the centre. In total, the two techniques combine to ensure that there is an approximately 1 mm cloud of cold neutral atoms available for pickup by the subsequent *tweezers* to use as qubits in the main part of the chamber and subsequently, the MOT system is disabled.

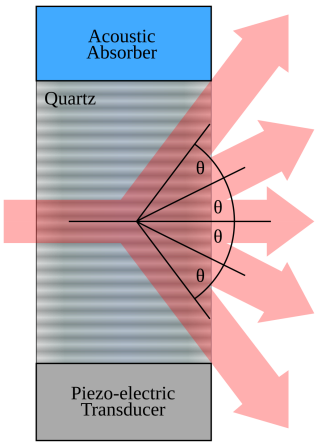


Figure 4: An acousto-optical deflector block where the incoming laser beam is split and deflected by the diffraction grating caused by the acoustic waves.

Of relevance now is the process by which cold atoms are moved from this cloud and then held in specific, stable positions in the chamber through the use of the optical tweezers. This undertaking is carried out by two separate systems in the QPU. The first, namely the *acousto-optic deflectors* (AODs), are responsible for the *dynamic* movement of the atoms in the chamber. These solid-state optical devices sit outside of the vacuum chamber and their beams shine through the windows of the chamber to make contact with the atoms. As shown in Figure 4, these devices consist of a few centimetre large rectangular block of quartz crystal onto which is glued a piezoelectric transducer, which is able to convert RF signals, typically of frequency $f \approx 100$ MHz, into an acoustic wave, which then travels through the crystal and sets up a diffusion grating. Multiple frequencies can be driven simultaneously. On the other axis, a laser of wavelength λ is then directed through the crystal. Due to the grating, this beam is split and deflected by an angle $\theta = \frac{\lambda f}{v_s}$, where v_s is the speed of sound in the crystal, i.e. a constant determined by the choice of material.

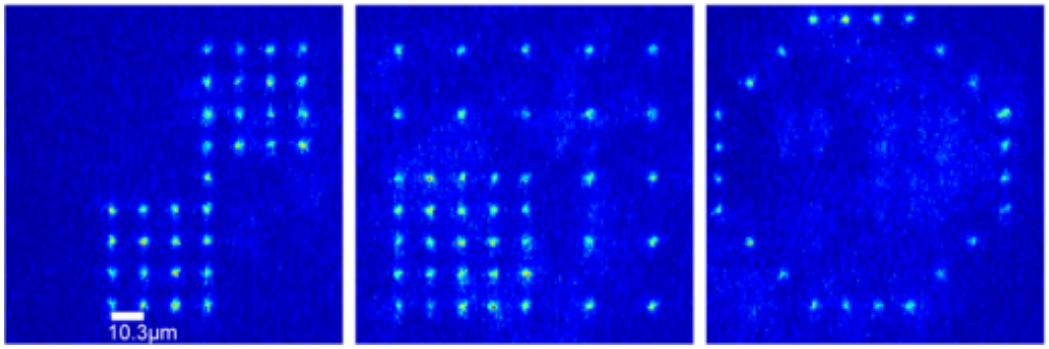


Figure 5: A neutral atom grid can be formed into various topologies by manipulating the driving frequencies of the AOD. Credit: Schlosser et al. [15]

After the beam has been split by the AOD, it enters a lens which converts the beams into linear displacements along one axis of the chamber. If another AOD is used on another side of the chamber, then a two-dimensional grid of beams is formed whose spacing can be controlled by the angle θ, which is in turn controlled by the driving frequency f. At the intersection of each of these beams, the laser intensity is the highest and this causes an *optical potential well* (specifically through the *AC Stark shift*) at these points, attracting the slowly but still randomly moving neutral atoms towards them. If two or more atoms enter the same trap location, they are attracted to each other, collide and both are promptly ejected from the trap, leaving every trap with exactly zero or one atoms. Due to this random motion followed by potential collisions, the tweezer locations are not guaranteed to fill uniformly: the probability of one location becoming filled is only 40-60%. Therefore, another set of *rearrangement tweezers* are typically used to pickup peripheral atoms and move them into empty locations after the initial fill to create a densely packed grid, as shown in Figure 6.

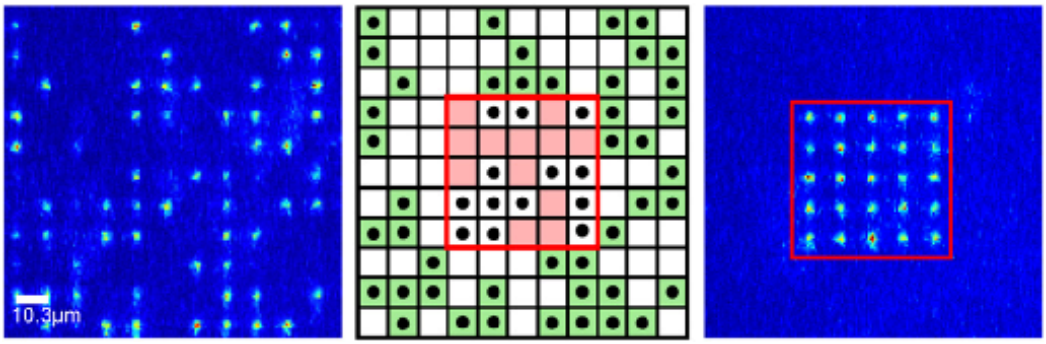


Figure 6: A grid of partially filled tweezer sites is seen on the left hand side. This is converted to a computer representation and an additional rearrangement tweezer moves atoms into a smaller but uniformly filled neutral atom grid. Credit: Schlosser et al. [15]

Although AODs work well for moving atoms and assembling grids, they are limited by the amount of RF bandwidth and laser power available and as such can only operate on tens of atoms at once, which makes them unsuitable for holding thousands of atoms statically in a neutral atom grid. To solve this problem, neutral atom quantum computers typically also use *static* traps, which are effective at holding large numbers of atoms in place for long periods of time.

One popular type of static trap is the *spatial light modulator* (SLM) trap [17]. This device consists of a grid of pixels of liquid-crystal cells on a reflective backpane. Voltages can be applied to each individual cell, which changes the orientation of the liquid-crystal molecules inside each pixel and hence changes the refractive index of the cell. This makes it extremely flexible and can represent many different trap topologies. A laser is then manipulated so that it shines uniformly onto the liquid-crystal pixels and the differing refractive index of each cell causes the beam to undergo a phase shift. A lens is then used to convert this beam into an arbitrary intensity distribution in the focal plane (on which reside the atoms in the vacuum chamber) to create the trap locations.

It is furthermore worth mentioning that neutral atom quantum computers are a highly flexible platform and the design and choice of specific optical traps also maintain this distinction. For example, besides AODs and SLMs, there also exist *microlens arrays* (MLAs), which are passive optics that generate large, fixed grids of traps and are even being scaled into three dimensions in state of the art research, using the *Talbot* effect [18].

The systems discussed until now provide the ability to produce atoms suitable for representing qubits, moving them into a specific topology and then using longer-duration traps to keep them in place. However, there is still no way to actually compute using these atoms. In the sense of Section 2.1, there is neither a mapping from physical atoms and their intrinsic quantum states to Hilbert space representation nor is there a way to implement unitary transformations on the state of these qubits.

In principle, every neutral atom has an infinite-dimension Hilbert space of quantum states. This is due to the fact that every atom has states described by a quantum number n and its n^2 orbital states. As $n \in \mathbb{N}$, there is no upper bound to the number of concrete states which could be used to define basis vectors in the Hilbert space (it is a countably infinite set). However, for the purposes of quantum computing, two basis vectors $|0\rangle$ and $|1\rangle$ are sufficient

to represent the bits 0 and 1. In $^{87}\mathrm{Rb}$, the basis $|0\rangle$ corresponds to a *hyperfine ground state* $|0\rangle \equiv |F = 1, m_F = 0\rangle$ and $|1\rangle \equiv |F = 2, m_F = 0\rangle$. Note that these are both in the same ground electronic state but they differ in hyperfine quantum numbers [16]. This provides several advantages; though chief among them is that both $|0\rangle$ and $|1\rangle$ are in the electronic ground state, meaning that they both do not decay radiatively unlike a higher electronic state, which would decay in nanoseconds.

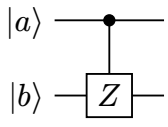
The next necessary component is the ability to bring the qubit state $|\psi\rangle$ into a superposition of the two basis states $|0\rangle$ and $|1\rangle$. In quantum physics, this can be done by applying a resonant electromagnetic field that couples the two hyperfine states, namely using *Rabi oscillations* [16]. This field is typically driven by either a microwave or *Raman laser* with Rabi frequency Ω sitting outside of the vacuum chamber. This causes the atom to undergo Rabi oscillations which causes the quibit to rotate according to the equation

$$|\psi(t)\rangle = \cos\left(\frac{\Omega t}{2}\right)|0\rangle - ie^{i\varphi}\sin\left(\frac{\Omega t}{2}\right)|1\rangle$$

for a time t and phase φ. By setting t to various values, one is able to achieve specific rotations of the qubit's state. For example, by setting $t := \frac{\pi}{\Omega}$, a so-called π-pulse is achieved, which fully transfers the qubit from $|0\rangle$ to $|1\rangle$. Similarly, by setting $t := \frac{\pi}{2\Omega}$, one achieves a $\frac{\pi}{2}$-pulse which creates an equal superposition of $|0\rangle$ and $|1\rangle$ similar to the Hadamard operation.

This allows for the application of any single-qubit gate on a qubit. However, there is still no means to create a product space of two qubits allowing for the representation of e.g. $|00\rangle$ or $|10\rangle$, which would in turn allow for two-qubit gates to be performed. The dominant method in the current research literature for solving this problem is by using the *Rydberg effect*, which allows for multi-gate application and is used by several commercial vendors including QuEra [5] and Pasqal [19].

A *Rydberg state* of an atom is a short-lived, highly excited state $|r\rangle$, meaning the outer electron is very far from the nucleus. This makes it easily polarised, giving it a large dipole moment. Two neighbouring Rydberg atoms thus interact strongly via dipole-dipole or *van der Waals* forces. The effect of this is that if one atom is in a Rydberg state $|r\rangle$, then it forms a *blockade radius*, inside of which other atoms cannot also be excited from $|1\rangle \to |r\rangle$. This control-inhibit effect is used in a similar way to *latches* in digital electronics: if the latch is active (control is 1), then the input signal is blocked otherwise if the latch control is 0, then the input signal propagates. However, note that in the Rydberg blockade, if the state is $|01\rangle$, then the target qubit will be excited to $|r\rangle$ and then subsequently de-excite to $|1\rangle$ but with a *phase* of -1, making the overall change $|01\rangle \to -|01\rangle$. In the other cases, the state remains the same. Usually, a Z gate is then applied on the *target* qubit, to change all states where the target is $|1\rangle$ to $-|1\rangle$, e.g. $|01\rangle \to -|01\rangle, |11\rangle \to -|11\rangle$. This then creates the *canonical CZ gate*.



$$\mathrm{CZ} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Figure 7: A CZ gate in circuit representation and the equivalent unitary transformation it performs, represented as a matrix over a 2-qubit Hilbert space.

A set of gates is said to be *universal* if any unitary operation on n qubits can be approximated to arbitrary accuracy using those gates [20]. This is similar to the functionally complete set of operators in Boolean algebra. It is known that any single qubit gate and one entangling two-qubit gate (e.g. the CZ) gate forms a universal set. This means that using the two methods introduced here, neutral atom quantum computers can approximate any quantum circuit.

The figures Figure 5 and Figure 6 show actual images of the neutral atoms trapped in a grid. Indeed, this is the primary way in which measurements of the qubits are taken as well [18]. Specifically, a detection laser is used whose frequency is coupled to $|1\rangle$, so that it cyclically transitions $|1\rangle \to |e\rangle$ for an excited state of the neutral atom. The atom then falls back to $|1\rangle$ emitting photons at a specific rate, which are picked up by a camera (usually a charge-coupled device (CCD)) as bright spots on the image. Note that this $|1\rangle \to |e\rangle$ cycling transition causes entanglement with the emitted photons and hence "collapses" the state (i.e. it is a true projective measurement). The probability of obtaining a dark qubit "patch" is hence $p(0) = \langle\psi|M_0|\psi\rangle = |\alpha|^2$ whilst the probability of a bright "patch" is $p(1) = \langle\psi|M_1|\psi\rangle = |\beta|^2$.

This concludes the discussion on the basics of the physical design of neutral atom quantum computers. As stated previously, it should be emphasised that neutral atom quantum computing is more of a "toolbox" than a specific, fixed architecture. Many subsystems in this design can be changed depending on the aims of the experiment in hand.

2.3 Quantum Compilers

In Section 2.2, the basic physical design of neutral atom quantum computers was discussed. This included how qubits are represented as trapped neutral atoms as well as how unitary transformations can be implemented as Rabi or Rydberg pulses on upon them. However, until now, there was only an ad-hoc discussion of how these operations are produced from circuit representations. Naturally, this does not suffice for general-purpose quantum computing and therefore *quantum compilers* exist. These programs systematically translate source code in the form of circuit representations into sequences of machine-specific operating instructions for the neutral atom quantum computer. Ideally, these compilers should also be able to perform optimisations on the circuits to maximise runtime performance while retaining semantic equivalence with the original circuit.

All compilers are programs which convert syntax from one language into another. In the case of quantum compilers, they take circuit descriptions as input, usually in the QASM format, parse it, and output an intermediate representation (IR), which can be executed by a controller chip on a quantum computer. The QASM format is a simple text-based description of a quantum algorithm which is modelled on assembly language (ASM) for classical computers. In the file, one defines a set of quantum registers (`qreg`) and a set of classical registers (`creg`). Upon these registers, gates are performed sequentially by indexing into these registers, e.g. `h q[0]` applies the Hadamard gate to the first qubit in the register. Most compilers, such as ZAC, will parse this file and then directly convert it into a directed acyclic graph (DAG) representation.

```
OPENQASM 2.0;
qreg q[10];
creg c[10];
h q[0];
cx q[0], q[1];
measure q -> c;
```

Listing 1: A small example of a quantum circuit written in the QASM format.

One notes that this is a very generic representation of a quantum circuit and does not represent a program that can be ipso facto executed on a specific kind of quantum computer, such as superconducting or neutral atom. There is no information regarding how the registers are assigned to physical quantum states nor is there any information regarding the timing of these operations and which operations, such as laser pulses, are even necessary. These are all the tasks of a quantum compiler to ascertain.

Before any compilation can be done, compilers must define the *architecture* of the QPU, as this is one area in which neutral atom QPUs are effectively arbitrarily configurable. To this end, most state of the art compilers, such as ZAC [21] or MultiQ [6], and likewise the target machines, employ the concept of *zones*. These are areas of the QPU which have specific roles and those of primary importance are the *storage* zone, the *entanglement* zone and the *read-out* zone.

The storage zone is a location in which qubits are stored while they are idling, mimicking the RAM of a classical computer. It is also possible to apply single-qubit gates to the atoms while they are in the storage zone. However, it is usually *not* possible to apply any multi-qubit gates to atoms in this zone, e.g. by using the Rydberg laser. This is because the topology of the storage zone is usually laid out so that the trap locations are far enough removed from each other so that they are always outside the Rydberg radius of each other. From a storage point of view, this is beneficial as atoms are less subject to noise or *cross-talk* when they are sufficiently separated.

By contrast, the entanglement zone usually consists of pairs of traps which are closely spaced together, such that both atoms are inside each others' Rydberg radius. This makes it possible to apply multi-qubit gates and indeed the Rydberg laser is usually positioned so that it only shines over this portion of the QPU, while leaving the others non-illuminated, meaning there is less potential for decoherence on the atoms in storage. Finally, when a qubit's state becomes fully evolved, it can be taken to the read-out zone for measurement. Usually, this is only done at the end of a circuit's execution, however, there is current research into *mid-circuit measurement*, which promises to increase performance and reduce the effects of noise by reading a classical value from a qubit as soon as it is ready.

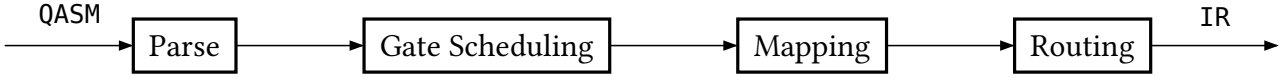


Figure 8: The compilation flow of a typical quantum circuit compiler for neutral-atoms QPUs. A textual representation (in this case QASM) is inputted by the user and then parsed by the compiler. The circuit is then turned into a sequence of gate operations, its qubits mapped to physical atoms and these atoms routed on the grid, so that they are in the right place when the gates are performed. This information is then encoded in hardware-specific IR instructions.

Once an architecture has been defined, an abstract representation of the QPU can be formed by the compiler and the program can go through the compilation pipeline with regard to this represenation. The first step in the piepline is *scheduling*, which decides *when* each gate will be executed while respecting its logical dependencies. Here, the circuit must first be resynthesised into a *native gate set*, which for neutral atom QPUs is $\{\mathrm{CZ}, \mathrm{U3}\}$, and is always possible as this set is universal. After this, the circuit is usually decomposed into *layers*, which represent units of execution (of one and two qubit gates) that could, *in principle*, be paralellised, i.e. they operate on disjunct sets of qubits. It is often most practical to model a layer as a set of two-qubit gates (2Q) appended with a sequence of 1Q gates that must be performed directly after this 2Q gate.

The order in which the gates are decomposed into layers is dictated by the *scheduling strategy*. The most generic way to schedule gates is the "as soon as possible" (ASAP) approach, where gates are places into the earliest possible layer given their data dependencies. This technique has the advantage that it minimises the overall circuit depth but it ignores physical movement overheads as it relies solely on local circuit-level information. Other compilers support more advanced methods to schedule gates. For example, graph colouring can be used to find the largest subset of gates which can be scheduled together (e.g. because they don't share a qubit) in much the same way as *register allocation* operates in classical compilers. Finally, compilers such as ZAC, further build on this idea with *reuse-aware scheduling* by further identifying schedules which reduce atom movement time[1].

The next stage is *placement*, where each qubit is assigned (or "mapped") a physical atom and location (i.e. coordinate) in the grid for each layer according to the hardware topology. This problem resembles a travelling salesman problem (TSP), where each qubit needs to make a tour from the storage zone, to its entanglement partner in the entanglement zone, and then back again for every layer. However, in TSP, there is always a single tour, whereas in the placement algorithm, there could be arbitrarily many tours, as they could be punctuated with static ("no-op") assignments in the entanglement zone if the qubit is reused directly in the next layer and moreover, there is no requirement that an atom end up in the same location that it started: it suffices that the location be unoccupied.

Placement is especially important as suboptimal mappings can result in reduced fidelity and longer execution times due to unecessarily long movement operations. As hinted at, this is a type of NP-complete combinatorial optimisation problem and hence is usually framed a multi-objective cost minimisation problem with two parts: the first is to find a good initial assignment of qubits to atoms, whereas the second is to find locations to move these atoms, in the entanglement zone and storage zone, such that the overall cost is minimised.

Initial placement, which usually corresponds directly to an outputted `init` instruction in IR, is the mapping of a logical qubit to its physical neutral atom. An overall minimal cost placement is then the assignment that minimises the total cost of all future shuttling operations from storage zone to entanglement zone. An exact solution to this placement could be found by an SMT solver but this is expensive (as NP-complete) and would rely on subsequent placements in future layers. Therefore, many compilers, such as ZAC, will use heuristics (e.g. *simulated*

[1]Reuse aware scheduling, as implemented by ZAC, isn't technically a pure scheduling algorithm as it also relies on later routing information.

annealing) and iteratively improve the solution. In ZAC, the cost of a 2Q gate for a site ω is approximated as

$$\text{cost}(g, \omega, M) = \begin{cases} \sqrt{d(\omega, m_q)} + \sqrt{d(\omega, m_{q'})} & \text{if } y_q \neq y_{q'} \\ \max\left(\sqrt{d(\omega, m_q)}, \sqrt{d(\omega, m_{q'})}\right) & \text{if } y_q = y_{q'} \end{cases}$$

where M is the current mapping and $y_q, y_{q'}$ are qubit locations. Defining a cost function for every operation means that one can find a random initial placement and iteratively refine it using simulated annealing: perform a random peturbation step and if it improves the total cost, apply the step with probability proportional to the current temperature.

After the mapping stage, the circuit is now fully represented as a sequence of movements to and from the entanglement zone on qubits. However, this logical representation is still far too abstract to be run on an actual QPU. For example, although which movements to make have been defined at every layer, there is still no notion of *how* they should be made, i.e. how the AOD laser should be manipulated to pickup and move atoms. These tasks are low-level hardware manipulations, which would be considered part of the *back-end* of a classical compiler. However, in quantum compilers, it is usually referred to as the *routing* stage.

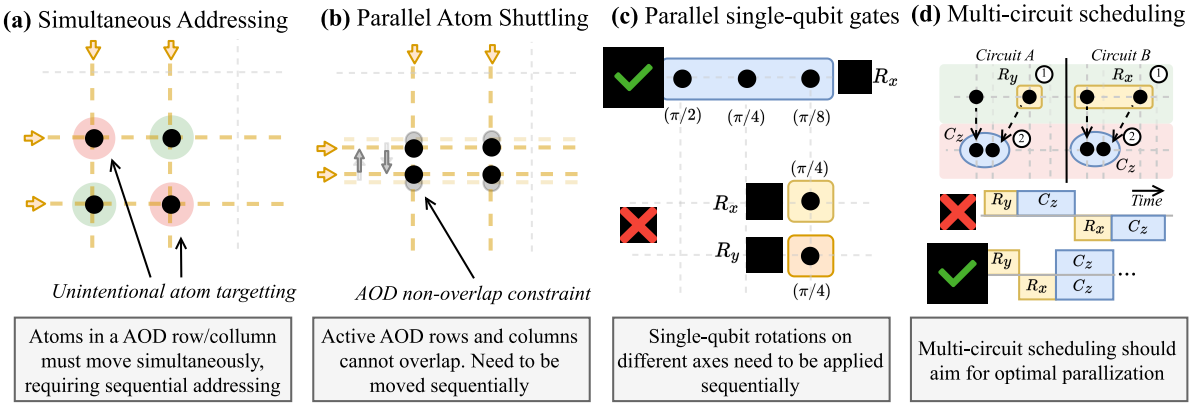


Figure 9: A series of potentially erroneous operations on the AOD hardware that could be outputted by a compiler. Source: [6]

A particularly important step in the routing stage is to convert the qubit mappings from pairs of start and end locations into actual instructions on the QPU. For each layer of execution, the naive way to do this would be to serially shuttle every atom from the storage zone to its location in the entanglement zone. Naturally this is a poor usage of the AOD laser, which is capable of moving many atoms at once. Compilers therefore need to have conflict solvers to find out the maximum number of qubits they can move per AOD laser usage while maintaining correctness (see Figure 9). Implementation bugs in this solver could lead to incorrect instructions being outputted to the IR.

Once all of the operations required to implement the circuit have been determined, there still remains the issue of when to assign each operation its "slice" of the global resources on the system, e.g. the AOD laser or Rydberg pulse laser. Doing this incorrectly could result in timing bugs, where two operations' usage of a resource overlap in time and thus cause contention or a deadlock.

The final stage of a compiler outputs the operations into a format readable by the controller of the actual QPU. Several formats exist, though there is currently no agreement on any standard format, especially in neutral atom QPUs. For example, the ZAC compiler uses a format called *ZAIR*. This format is based on JSON and encodes instructions as items in an array. This has the advantage of being flexible and extensible at the cost of formal specification of the language.

```
{"type": "init", "id": 0, "begin_time": 0, "end_time": 0, "init_locs": [[0, 0,
5, 9], [1, 0, 5, 1]},
    ...
{"type": "1qGate", "unitary": "u3", "id": 1, "locs": [[2, 0, 5, 3], [4, 0, 5,
0]],"dependency": {"qubit": [0]}, "begin_time": 0, "end_time": 2.5},
```

Listing 2 show an example of the format. Here, an `init` instruction is used to tie the locations of qubits to coordinates on the grids. The qubit identifiers implcitly correspond to the indices of the `init_locs` array. An example of a `1qgate` operation is then given, which applies a unitary pulse to all qubits at the locations in `locs`. An important feature of ZAIR is that all operations have begin and end times to ensure proper scheduling of the resources and furthermore, dependencies are tracked through the `dependency` list. This means that the `1qgate` operation can't be performed until its dependencies have completed.

2.4 Category Theory

One of the primary mathematical structures used in the formal verification part of this thesis are *categories*. Categories are a collection of objects which have maps defined between them. They are useful as a very general framework for defining diverse mathematical structures, e.g. sets, graphs and vector spaces, in a uniform manner [22]. The following section gives a short summary of the categorical structures used in the thesis.

Definition 2.4.1 *(Category)* A category $\mathcal{C}$ consists of a collection of objects $\mathrm{Ob}(\mathcal{C})$. For any two objects $A, B \in \mathrm{Ob}(\mathcal{C})$, there exists a class of *morphisms* $\mathrm{Hom}_{\mathcal{C}}(A, B)$ such that the following holds:

1. for each $A \in \mathrm{Ob}(\mathcal{C})$, an *identity morphism* $\mathrm{id}_A \in \mathrm{Hom}_{\mathcal{C}}(A, A)$ exists;
2. for each triple of objects (A, B, C), the *composition law* $\mathrm{Hom}_{\mathcal{C}}(B, C) \times \mathrm{Hom}_{\mathcal{C}}(A, B) \to \mathrm{Hom}_{\mathcal{C}}(A, C)$ holds.

For the two objects A, B, an individual morphism $f : A \to B \in \mathrm{Hom}_{\mathcal{C}}(A, B)$ maps between the two objects in the category and hence is also referred to as an arrow. However, for f to be a valid morphism, the following axioms must hold:

1. **Associativity**: Let $C, D \in \mathrm{Ob}(\mathcal{C})$ and $h : C \to D$ additionally be given, then $h \circ (g \circ f) = (h \circ g) \circ f$ holds.
2. **Existence of Identities**: $f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f$ holds.

One relevant property is that a category $\mathcal{C}$ can be "reversed" by transforming all morphisms $f : A \to B$ to $f' : B \to A$. This still preserves all of the category properties and creates what is known as the *opposite* category $\mathcal{C}^{\mathrm{Op}}$.

It is also possible to define maps from one category to another such that they preserve structure (i.e. a category homomorphism) and these are known specifically as *functors* in category theory.

Definition 2.4.2 *(Functor)* A functor $F : \mathcal{C} \to \mathcal{D}$ between two categories $\mathcal{C}, \mathcal{D}$ consists of:
1. an assignment of objects $A \to F(A)$;
2. an assignment of morphisms $f : A \to B \mapsto F(f) : F(A) \to F(B)$.

However, the functor F must be structured preserving:
1. **Preservation of Identities**: $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$ holds;
2. **Preservation of Composition**: $F(f \circ g) = F(f) \circ F(g)$ holds.

Functors themselves can also be transformed by maps. For example, if F and G are functors both from $\mathcal{C}$ to $\mathcal{D}$, then a *natural transformation* transforms F into G in a way which is compatible with the morphisms of $\mathcal{C}$.

Definition 2.4.3 *(Natural Transformation)* Let $F, G : \mathcal{C} \to \mathcal{D}$ be functors. A natural transformation $\eta : F \Rightarrow G$ then assigns to each object $c \in \mathrm{Ob}(\mathcal{C})$ a morphism $\eta_c : F(c) \to G(c)$ in $\mathcal{D}$. For every morphism $f : c \to c'$ in $\mathcal{C}$, the *naturality condition* must hold, i.e. $G(f) \circ \eta_c = \eta_{c'} \circ F(f)$. This is usually stated as a commutative diagram:

$$\begin{array}{ccc} F(c) & \xrightarrow{\eta_c} & G(c) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(c') & \xrightarrow{\eta_{c'}} & G(c') \end{array}$$

In category theory, *universal constructions* are of particular importance because they specify an object not by its internal structure but by its role in relation to other objects [22]. In the context of this formal verification model, the most commonly used constructions are *products* and *pullbacks*, which both "combine" objects together so that they remain consistent.

Definition 2.4.4 *(Product)* In a category $\mathcal{C}$, for two objects $A, B \in \mathrm{Ob}(\mathcal{C})$, a product is an object $A \times B$, which is equipped with two projections $\pi_A : A \times B \to A$, $\pi_B : A \times B \to B$.

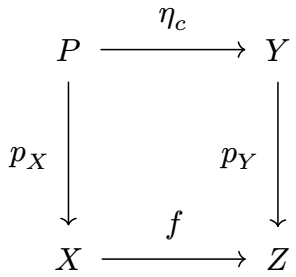
The product object must also follow the *universal property*: For any object X with maps $f : X \to A$ and $g : X \to B$, there exists a unique arrow $\langle f, g \rangle : X \to A \times B$ such that

$$\pi_A \circ \langle f, g \rangle = f, \pi_B \circ \langle f, g \rangle = g$$

which ensures that $A \times B$ is the canonical construction and not just a "random" object with two maps out of it.

However, a product on its own doesn't enforce any notion of compatibility between its two components A and B: these objects just exist side-by-side, like the cartesian product in **Set**. For modelling formal verification tasks, it often happens that two components need to be combined in such a way that they satisfy some compatibility condition or "interface"; this is modelled as a pullback [23].

Definition 2.4.5 *(Pullback)* For a category $\mathcal{C}$ and two morphisms $f : X \to Z$, $g : Y \to Z$, a pullback (also known as a fibre product) is an object $P \in \mathcal{C}$ and two morphisms $p_X : P \to X, p_Y : P \to Y$ such that the diagram commutes:



The universal property must also hold: for any object Q with morphisms $q_X : Q \to X, q_Y : Q \to Y$ that also satisfies the commutative diagram, then there always exists a unique arrow $u : Q \to P$ such that

$$p_X \circ u = q_X, p_Y \circ u = q_Y$$

, that is it *always* factors through the commutative square.

2.5 Sheaves

In Section 2.4, the basic concepts of category theory were introduced and they will be employed throughout the method chapters in the construction of the formal model of the neutral atom QPU. The most important structure for this task, however, is the *sheaf*, and this structure is significantly more complex than the previous ones and as such is introduced in its own section.

The motivation for sheaves is that in many systems, information is only available locally. For example, this local information could be sensor readings at a particular location in an otherwise larger machine. The local information *can* be verified against constraints on the local part of a subsystem, though ultimately we are actually interested in checking that the system is globally correct across all sensor readings and subsystems. To establish this correctness, one needs to know how local information can be combined into a globally consistent view. Where no information overlaps, this is trivial, but if there is a concept of overlap, then the data need to agree on this overlap and be "glued" together. This precise problem has already long been encountered in algebraic geometry and topology, where there is a need to track local solutions to equations across open subsets of space and hence the required structure has already been formalised, namely sheaves, which are a rigorous framework for describing how local data can fit together to form global data [24]. Sheaves will form the backbone of the formal verification aspect of this thesis, where the correctness of a program's runtime will be expressed as the existence of a global section that witnesses compatibility across all checked local sections.

To start with, we discuss the simpler concept of *presheaves*, which can be thought of the "raw" data structure in which the local data is defined.

Definition 2.5.1 *(Presheaf)* Let X be a topological space, for example $\mathbb{R}_{\geq 0}$, then the *open category* $\mathcal{O}(X)$ is a category which has open subsets of X as its objects $\mathrm{Ob}(X)$ with a unique morphism $U \to V$ when $U \subseteq V$. Then a functor

$$F : \mathcal{O}(X)^{\mathrm{op}} \to \mathbf{Set}$$

is a $\mathbf{Set}$ *presheaf* on X.

To each set $U \subseteq X$, the presheaf assigns a set $F(U)$, which is known as the *local section* of data over U. To each inclusion $V \subseteq U$, a *restriction map*

$$\rho_{U,V} : F(U) \to F(V)$$

specifies how the data on a larger local section can be restricted into a smaller section. As the presheaf is a functor, the restriction maps must satisfy the laws of

1. **Identity**: $\rho_{U,U} = \mathrm{id}_{F(U)}$;
2. **Composition**: if $W \subseteq V \subseteq U$ then $\rho_{V,W} \circ \rho_{U,V} = \rho_{U,W}$.

However, presheavs on their own do not guarantee the global correctness of data, but rather just that local data can be assigned to any arbitrary section. For this, one requires the *sheaf axiom*, which is defined on *open covers*. On a topological space X, an open cover of $U \subseteq X$ is a collection of open sets $\{U_i\}_{i \in I}$ such that

$$U = \bigcup_{i \in I} U_i$$

and is a way of decomposing a larger region into smaller overlapping regions.

Definition 2.5.2 *(Sheaf Axiom)* A presheaf F is a sheaf if, for every open cover $U = \bigcup_{i \in I} U_i$, the axioms

1. **Locality**: if $s, t \in F(U)$ satisfy $\rho_{U,U_i}(t)$ for all i then $s = t$;
2. **Gluing**: if $s_i \in F(U_i)$ are sections such that for all i, j then

   $$\rho_{U_i, U_i \cap U_j}(s_i) = \rho_{U_j, U_i \cap U_j}(s_j)$$

   then there exists a *unique* $s \in F(U)$ with $\rho_{U,U_i}(s_j)$.

hold. Although these axioms seem somewhat complex, their intuitive meaning is quite clear. They state that if local data match on overlaps, then there is exactly one way to glue them into a global object.

With the two definitions Definition 2.5.1 and Definition 2.5.2, the definition of a sheaf is apprent. It is precisely those presheaves that satisfy the sheaf axiom. From the presheaf definition, the ability to functorially assign data to sections is provided and from the sheaf axiom, it is ensured that all of these assignments can be glued together to build a valid global section.

However, not all presheaves satisfy the gluing axion, either because no global section *exists* or it is not *unique*. For example, $F(U) := \{f : U \to \mathbb{R} \mid f \text{ is bounded on } U\}$ is a presheaf with standard domain function restriction as its restriction map. On an open cover of $\mathbb{R}$, where each $U_n = (-n, n), n \in \mathbb{N}$ is a nested interval, the function $f_n(x) = x$ is bounded and each section agrees on overlaps. A global section would contain a function $f \in F(\mathbb{R})$, but the only possible candidate is $f(x) = x$, which is unbounded, hence no global section exists, making F not a sheaf.

2.6 Formal Methods

Formal methods are a collection of techniques for specifying and verifying software and hardware systems in a mathematically rigorous way. The core idea in this field is to model

systems and their behaviour in mathemtical formalism and then use tools from mathematics such as logics to prove that software implementations satisfy these specifications. This is different to software testing, which only can detect bugs in certain scenarios, whereas formal methods aim to guarantee that a software system is correct under all possible executions [25], [26].

There are verious possible approaches that can be applied to the verification of software. As a first example, *model checking* is a popular technique whereby a system is represented as a formal model, usually a *Kripke structure*. This structure is a tuple (S, S_0, R, L) where S is a set of states, $S_0 \subseteq S$ is the set of initial states, R is a transition relation and L is a labelling function [27]. This can be used to model a simple program

```
x := 0;
while (x < 2) {
  x := x + 1;
}
```

where the state would consist of the program counter and value `x` (i.e. tuples in $\mathbb{N} \times \mathbb{N}$), the initial state would be $(0, 0)$ and the transition relation would describe how the program executes depending on the current state. On these structures, one can define a temporal logic, such as LTL, and then use this to encode a specification of correctness properties, such as $\psi := \square(x \geq 0)$ ("x is never negative").

However, for real systems with e.g. many variables and threads of execution, this approach can quickly become intractable due to *space-state explosion*, namely that the set of states that needs to be checked often grows close to the powerset (i.e. all possible 2^n states), though this probably would not be an issue in this thesis. However, the inherently discrete nature of states in Kripke structures in general does not map cleanly to the hardware of neutral atom QPUs: neutral atom grids consist of continuous motions of atoms and precisely defined laser pulse timings, which makes it unclear as to what level of fidelity these states (e.g. qubit positions, beam intensities, trap positions) should be as including too many physical properties unnecessarily grows the state space but too little makes it underspecified.

Instead of relying on model checking, which verifies a program after it has been compiled, one could prove instead that the compiler itself only produces valid and correct code. This ensures that the compiler itself never introduces bugs or errors into the code as it translates it into a lower-level representation. This requires building formal semantics of both the source and target languages of the compiler and then proving that the compiler, which is itself built from many small passes (parsing, optimisation, register allocation etc.), is formally correct at every step. These proofs are usually written in a proof assistant such as Coq or Isabelle/HOL and technique has been shown to be successful in a verifying large-scale, optimising C compiler [28], though this was a very large undertaking, taking years of development time and thousands of lines of proofs.

Model checking therefore excels as a highly automated method of proving correctness of *specific* properties against a system or program, though it struggles with highly intricate systems due to state-space explosion. Verified compilers, by contrast, are able to universally

prove correctness across *all* compiled programs with little computational overhead once built but this requires a large up-front cost in development time to create such a compiler.

3 Motivation

In Section 2, the principles of operation of neutral atom quantum computers were introduced. They were presented as highly valuable platforms in the search for general-purpose due to their high scalability and inherent flexibility. Indeed, several vendors of these machines are now producing them commercially. It was also seen that these machines operate as a sophisticated interaction of various subsystems, such as the magneto-optical traps, which are initially used to cool the atoms down for use, to the spatial light modulator (SLM) traps, which fix thousands of atoms statically in the neutral atom grid and finally to the acousto-optical deflectors (AODs), which are able to shuttle the atoms to and forth in the grid. These subsystems all require precise instructions from a controller in order to function cohesively as a quantum computer. Hence, correctness failures at the controller-software boundary could invalidate an entire computation on the hardware or potentially even damage the hardware itself.

Therefore, the use of verification is highly beneficial in these systems as a way to prevent erroneous computation and to protect the hardware from deleterious operations. However, thus far, there has been little to no study into methods for such static verification for neutral atom quantum computers: existing compilers have only employed ad-hoc methods to check their output, such as simulation passes to check for the simple property-based bugs in their the IR outputs, and these typically do not model the neutral atom QPU in any significant detail. We therefore identify this discrepancy as a *research gap* and seek to employ richer techniques from the formal methods literature to resolve it.

However, the use of formal methods in this domain presents some challenges. Specifically, in Section 2.6, it was seen that finding a suitable abstract hardware model on which to perform formal checking is not trivial. Model checking is promising because it is a highly automated approach to verifying systems against abstract models. However, they encode state in discrete Kripke structures, which is not a natural abstraction for neutral atom grids, which operate on continuous motions of atoms and precise laser timing operations. Although more advanced logics such as *metric temporal logic* (MTL) exist, which could, in principle, model these systems, they still rely on discretising the underlying state, which can lead to complex models and hence possible state-space explosion [29].

Furthermore, rather than viewing neutral atom QPUs as a singular entity, they should instead be viewed as a "toolbox" of systems which can be altered to suit the needs of the experiment at hand. This raises the question of how *modularity* can be designed into a formal verification system. In the case of model checking, new systems are principally integrated into the existing verification design by extending the state definition and transition functions. This means that all properties and exploration heuristics must be extended to handle this new state, which could be an invasive process depending on the additional new complexity.

The hypothesis of this thesis is that by developing a certificate-based abstract machine checker for the correctness of neutral atom compilers based on sheaves, both of these issues could

be solved. As sheaves stem from algebraic topology, they are naturally suited for modelling continuous geometric properties like atom movements, rather than an "awkward" grafting on discrete space states as in model checking. Furthermore, the other principal aim is to develop a system capable of evolving as neutral atom quantum computers do themselves. This requires a verification system that is composable, namely that adding new subsystem checks doesn't break existing checks and that where they interact, this can be easily expressed. Indeed, by using the tools of category theory, such as pullbacks, sheaves are again naturally suited to this.

In order to evaluate the hypothesis quantitatively and conclude this section, a formal *research statement* is given: how can a *modular* formal verification system for the hardware and semantic correctness of neutral atom quantum compilers be developed, such that the resulting system retains a high-degree of runtime performance?

4 Overview

The basic theory behind neutral atom quantum computers was introduced in Section 2. It was seen that the model provides a highly flexible "toolbox" of techniques for achieving quantum computations on up to thousands of qubits, which makes the platform highly important for the search to develop general purpose quantum computing. Neutral atom QPUs are also considerably less sensitive to noise than other approaches such as superconducting QPUs. However, as a system consisting of many inter-related parts, it is highly important that they are operated in a correct manner otherwise, for example, small incosistencies in timing of operations could lead to incorrect computation on the device or even device faults. This problem naturally also exists in classical computing and has led to the development of *formal methods*, which is a field of study involves seeking to rigorously and abstractly model a device and then prove the a program can never enter such an erroneous state on this model. Techniques from this field are often used in compilers to verify the output of a transformation pass is correct or at least doesn't contain any obvious bugs.

In terms of research gaps, the field of compilers for quantum computers and therefore also the application of formal methods to neutral atom quantum computing is still in its infancy and as such, there have been no attempts to use techniques from formal methods to verify the correctness of compiler output in a principled manner: most compilers use ad-hoc methods such as simulators but without considering the formal semantics of the compiler's IR output. Secondly, we see a need for an IR which is able to adapt to the flexible nature of neutral atom quantum computers, by supporting *modularity*. Although there are projects which strive for this aim, such as by defining an IR using the MLIR system from LLVM, none of them explicitly support neutral atom hardware and are aimed at much higher-level program definitions, which we consider a distinct disadvantage for the hardware-guided optimisation of programs.

To alleviate these problems, the thesis proposes two contributions. In Section 5, a novel, modular instruction set (IR) is defined for neutral atom QPUs called *AtomGuard*. This IR is able to simulataneously capture both high-level and low-level properties of the program on neutral atom hardware. It does this by encoding a program in high-level, portable operations which are then *lowered* to sequences of micro-operations. These micro-operations are hardware specific and at a low enough level of abstraction to express the physical properties of hardware, such as the frequency of Rabi pulses emitted onto the neutral atom QPU grid.

The second contribution is given in Section 6. Despite a modular IR being useful on its own, this does not solve the problem of incorrect compiler passes creating bugs in compiled programs. To this end, a *certificate-based formal correctness framework* for the AtomGuard is developed. This framework is able to statically check a compiled program against an abstract machine to guarantee hardware safety and correctness properties hold on the program. Moreover, it is able to check for semantic equivalence between an input circuit to the compiler and the compiled output using ZX rewriting. A description of how these theoretical constructs can be implemented is given, which describes the architecture of such a verifier using the theorem

prover *Lean*. Simiarly, it discusses how existing compilers, such as ZAC, can be modified to compile to AtomGuard using an extra transcompilation step.

The penultimate chapter, Section 7, discusses the current landscape of compilers and formal methods for neutral atom QPUs and contrasts the work of this thesis with other research projects. Section 8 concludes this thesis with a reiteration of the most important parts of the method as well as lessons learnt in this thesis for future work in this field.

5 The AtomGuard ISA

Section 3 motivated the need for a hardware-agnostic intermediate representation (IR) language for neutral atom quantum computers. As IR languages are outputted by programs for use by other programs, brevity is not a requirement of these languages. Rather they should be written at a low level so that they are able to directly capture physical control operations over the hardware. As neutral atom QPUs are machines which operate directly on the "physics level", the semantics and correctness of programs that run on them rely on the correct operation of continuous-time, continuous-space control of atoms, traps and associated lasers. This therefore demands a *physics-level* IR that is able to expose these concepts of time, space and individual devices (e.g. lasers) as first-class primitives, whilst maintaining portability over multiple machines and keeping gate-level semantics to allow for semantic verification.

In order to achieve this goal, the AtomGuard ISA follows a layered structure where the language is split into three levels of abstraction. The most abstract part of the language is the *portable layer*, which is a minimal vocabulary that encodes the algorithmic plan of a compiler in generic, neutral-atom physics terms. This includes operations such as initialising a qubit or performing an atom movement from one zone to another.

However, the portable layer only describes what operations are to occur on the hardware, not how they are to occur. These specifics require information on the hardware itself, for which the *schema* exists: this section of the IR declares the device and its capabilities. Once the model of the underlying hardware is known, hardware-specific *dialects* can be written. These contain *micro-ops* which describe in detail how devices should be manipulated to achieve an operation. A portable operation can then be *lowered* to a sequence of these micro-ops through a mapping defined by the compiler.

5.1 Schema

The schema section of the IR defines the relevant physical properties of the target neutral atom QPU. It is not part of the compiled program itself but rather an input to both the compiler and the verifier. We inductively build up an abstract syntax for the schema using algebraic data types (ADT) constructed from records (i.e. products) and sum types as well as standard constructions like list, optional and map over primitive types like $\mathbb{N}$ or `Vec2`. The formal language for the schema is then the set of all values of the top-level `Scheme` ADT interpreted as JSON strings.

Firstly, the *GridSpec* record in Table 1, is used to define the size of a grid of neutral atoms. It is used, for example, in *SLMSpec* (Table 2) to define the SLM arrays of a zone, which are themselves defined in Table 3. Besides zones, the ADT describes which devices (in Table 6) of the neutral atom QPU, which allows for customising the system by e.g. attaching AODs (Table 4) or other devices such as Talbot effect tweezers (Table 5). A `Scheme` is then a record `Scheme(DevicesSpec)`. A full example of a schema file is given in the appendix, Listing 6.

Field	Type	Purpose
`rows`	$\mathbb{N}$	Number of grid rows
`cols`	$\mathbb{N}$	Number of grid columns
`site_separation`	`Vec2`	Spacing between grid elements in $(\Delta x, \Delta y)$.
`origin`	`Vec2`	World coordinates of the $(0, 0)$ grid point

Table 1: The `GridSpec` record is built using a data constructor of these fields.

Field	Type	Purpose
`id`	`str`	A unique (within a zone) name for a SLM array
`grid`	`GridSpec`	The physical grid lattice for this SLM

Table 2: The `SLMSpec` is effectively a named grid array

Field	Type	Purpose
`id`	`str`	A name for a zone, e.g. `ent0`
`type`	`enum`	The type of zone. Either `storage`, `entanglement` or `readout`
`slms`	`list<SLMSpec>`	A list of SLM arrays in the zone.
`geometry`	`Rect2D`	A rectangle in world coordinates defining the physical size of the zone
`rydberg_extent`	`list<Rect2D>`	The areas in the zone illuminated by the Rydberg beam(s).

Table 3: `ZoneSpec` defines a zone.

Field	Type	Purpose
`id`	`str`	A name for an AOD, e.g. `aod0`
`rows`	$\mathbb{N}$	Number of manipulatable rows
`cols`	$\mathbb{N}$	Number of manipulatable columns
`site_separation`	`Vec2`	Spacing between grid elements in $(\Delta x, \Delta y)$.
`rf_freq_range_MHz`	$(\mathbb{R}, \mathbb{R})$	The usable RF range of the AOD
`max_ampl`	$\mathbb{R}_{\geq 0}$	The upper amplitude bound

Table 4: `AODSpec` defines an AOD.

Field	Type	Purpose
`id`	`str`	A name for a tweezer, e.g. `tw0`
`patterns`	`list<str>`	List of phase masks
`swap_latency_us`	$\mathbb{R}_{\geq 0}$	The latency to switch phase masks.
`grid`	`GridSpec`	The pattern grid

Table 5: `TalbotSpec` defines a tweezer using the Talbot effect, as an example of an additional subsystem for the verifier.

Field	Type	Purpose
`clocks`	`list<` $(\texttt{str}, \mathbb{R}_{>0})$`>`	The clocks for the system, e.g. to measure begin and end times of operations.
`zones`	`list<ZoneSpec>`	Zones in the system
`aods`	`list<AODSpec>`	AOD devices
`geometry`	`Rect2D`	Defines the overall bounds on the world coordinate system of the QPU

Table 6: `DevicesSpec` defines the devices attached to the system.

5.2 Portable Operations

Now that the schema has been defined, it is possible to formalise the actual compiled instructions of the IR in more detail. The highest layer of abstraction in the IR is the *portable* layer, which describes what actions are performed but without going into precise, machine-specific detail. This corresponds to roughly the mapping stage in existing compilers such as ZAC.

Therefore, the instruction set is relatively simple, consisting only of the fundamental tasks in a neutral atom QPU. Namely, it is able to initialise qubits, move them on the grid, apply single-qubit pulses or Rydberg pulses, perform measurements and finally implement barriers.

Constructor	Field	Type	Purpose
`init`			Initialises a qubit to a blank state and bind it to a location `loc`
	`q`	$\mathbb{N}$	The ID of the qubit
	`loc`	`Loc`	The binding location
`rearrange`			A batch move of qubits from begin sites to end sites.
	`begin_locs`	`list<Loc>`	The starting location for each qubit (ordered)
	`end_locs`	`list<Loc>`	The end location for each qubit (ordered)
	`method`	`optional<Device>`	An optional reference to the specific transport device, e.g. `aod0`
`rydberg`			A singlular entanglement stage over a zone
	`zone`	`Zone`	The zone to pulse
	`pairs`	`optional<list<(Q,Q)>>`	A list of pairs of qubits in the zone
`1q`			A unary qubit pulse
	`target`	`list<Q>`	The targets for the 1Q pulse
	`basis`	`Basis`	The Z/X/Y basis to pulse
`barrier`			Implements a fence to ensure correct ordering

Table 7: The constructors for the `PortableOperation` abstract data type.

As with all other instructions, once the portable operation record has been constructed, it is tagged with an `id` and the operations on which it depends, so that a final portable instruction, as outputted by a compiler, looks like:

```
{
  "id": 2, "type": "rearrange", "depends": [1],
  "begin_locs": [{ "qubit":"q0", "zone":"store0", "array":0, "row":12,
"col":3 }, ... ],
  "end_locs":   [{ "qubit":"q0", "zone":"ent0",  "array":1, "row":0,
"col":0 }, ... ],
  "transport": "aod0"
}
```

5.3 Backend Dialects

The portable operations described in the previous chapter provide an algorithmic view of the operations ocurring on the neutral atom hardware. However, they are agnostic to the actual physical device which carry them out and therefore, extra device-specific instructions must be provided to supplement them. In the ISA, this process is called *lowering*.

A lowering from a high-level instruction to a low-level one is performed in the ISA by attaching a sequence of *micro-ops* to the high-level operation. However, as one of the goals of the ISA is modularity, a singular formal language for this task would be insufficient. Rather each control device, such as an AOD or a Rydberg pulse receives its own *dialect*. Dialects are an approach to building reusable and extensible compilers by building self-contained name-spaces of operations, types and attributes with interfaces into them [30]. Although a compiler operating on multiple levels of IR is not a new concept (GCC does this for example), the use of co-existing IR as a first-class feature, with a conversion framework between them was pioneered in the MLIR project. Using this framework as inspiration, AtomGuard defines the dialects `aod/*` for AOD operations, `slm/*` for managing the SLM device, `rabi/*` for Rabi pulse operations, `rydberg/*` for operations using the Rydberg laser. The intention is that further dialects could then be added depending on the needs of the experimental setup, for example, if one wanted to introduce a new type of tweezer, then a new dialect could be written for it and the compiler could attach appropriate lowerings to this language. Crucially, the high-level algorithmic description of the circuit remains in the portable layer and hence is unchanged.

We provide an example of such a lowering in the Appendix, Listing 7.

Constructor	Field	Type	Purpose
`aod/activate`			Activate the AOD laser at a specific intersection point
	`tone_id`	$\mathbb{N}$	Identifier for the tone (one mobile trap)
	`fx`	$\mathbb{R}_{>0}$	RF frequency for x-axis deflection
	`fy`	$\mathbb{R}_{>0}$	RF frequency for y-axis deflection
	`power`	$\mathbb{R}_{\geq 0}$	RF drive amplitude
`aod/deactivate`			Deactivate the AOD laser
	`tone_id`	$\mathbb{N}$	The tone identifier to deactivate
`aod/ampl_ramp`			Modify the ramping frequency in order to move an atom
	`tone_id`	$\mathbb{N}$	Identifier for the tone
	`p0`	$\mathbb{R}$	Initial amplitude
	`p1`	$\mathbb{N}$	Final amplitude
	`duration`	$\mathbb{N}$	Duration in clock ticks.
`aod/chirp`			A controlled frequency sweep for accelerating/decelerating an atom.
	`tone_id`	$\mathbb{N}$	Identifier for the tone
	`from`	$\mathbb{R} \times \mathbb{R}$	Starting frequencies (x,y)
	`to`	$\mathbb{R} \times \mathbb{R}$	Final frequencies (x,y)
	`duration`	$\mathbb{N}$	Duration in clock ticks

Table 8: These operations define the controls for the physical AOD device.

Constructor	Field	Type	Purpose
`slm/activate`			Activate the AOD laser at a specific intersection point
	`row`		The row of the location to activate
	`col`		The column of the location to activate
	`duration`	$\mathbb{R}_{\geq 0}$	The duration of ramp-up time
`slm/deactivate`			Deactivate the AOD laser at a specific intersection point
	`row`		The row of the location to deactivate
	`col`		The column of the location to deactivate

Table 9: These operations define the controls for the physical SLM device.

Constructor	Field	Type	Purpose
`rydberg/pulse`			Pulse the Rydberg laser
	`rabi_rads`	$\mathbb{R}$	Rydberg pulse in rad/s
	`delta_Hz`	$\mathbb{R}_{\geq 0}$	Offset frequency
	`duration`	$\mathbb{N}$	Duration of the pulse in clock ticks

Table 10: The Rydberg operation—with just one operation—is conceptually simple.

Constructor	Field	Type	Purpose
`rabi/pulse`			A global Raman pulse
	`rabi_rads`	$\mathbb{R}$	Rabi pulse in rad/s
	`delta_Hz`	$\mathbb{R}_{\geq 0}$	Offset frequency
	`phase`	$\mathbb{R}_{\geq 0}$	Phase in radians
	`duration`	$\mathbb{N}$	Duration of the pulse in clock ticks

Table 11: The Rabi dialect likewise has only one operation and is applied globally on the grid.

6 An Abstract Model of Neutral-Atom Quantum Computers

In computing, the goal of formal methods is to assign programs, which are mathematically speaking, nothing other than elements of a set of strings $L \subseteq \Sigma^*$, precise mathematical objects representing their meanings. These objects can be assigned with respect to a fixed abstract machine, which allows rigorous guarantees about properties such as correctness, safety and performance to be given regarding their execution on this machine. This assignment of *semantics* to a program can be done in many different ways; examples of which include *denotational*, *operational* or *game-theoretic* semantics. The specific methodological choice often depends on how "easy" it makes it to show the properties one wants to prove.

In denotational semantics, one assigns every program a mathematical object, such as a function or relation, which directly captures its meaning instead of defining its actions on an abstract hardware model. As a program is typically built up of syntax recursively, this likewise transforms every statement or expression of the program into an appropriate mathematical object. For example, in an imperative language, state is usually considered a mapping from the set of program's variables (or memory locations) to a set of values. As such, a command like `x := x + 1`, would be semantically interpreted as an endomorphism on the set of states:

$$[\![x := x + 1]\!] : \text{State} \to \text{State}$$

To give a specific example of a mapping of more complex constructions, one can consider *if statements*, which might have the following semantic function:

$$[\![\text{if } b \text{ then } c_1 \text{ else } c_2]\!] = \begin{cases} [\![c_1]\!](s) \text{ if } [\![b]\!]_B(s) = \text{true} \\ [\![c_2]\!](s) \text{ otherwise} \end{cases}$$

Importantly, the interpretation for *if* recursively calls $[\![\cdot]\!]$ in order to evaluate the meaning of both the if guard and its two branches. This recursion allows us to build a semantic interpretation function over the whole program through repeated function calls on repeatedly smaller segments of the program.

However, as stated, denotational semantics map directly into mathematical objects and hence make it difficult to reason about the properties of the actual machine on which the program is run. The denotational semantics do not describe the resource usage, time scheduling or register allocations of the hardware. In this thesis, we are interested in ensuring not only that the compiled program retains the original semantics of the input circuit when it is run on the specific quantum computer, but also that the compiled program can plausibly be run on an actual machine in the first place. This means that it must adhere to the physical constraints imposed by the subsystems of the quantum computer such as its AOD lasers or Rydberg pulse laser.

But first back to semantics: as denotational semantics do not expose the right level of abstraction for our purposes, one might consider operational semantics, which *do* consider the abstract machine. In these semantics, one defines the meaning of a program by *how* it is

executed in a step-wise manner on this formal machine [31]. The most common operational approach are *big-step* semantics, where a *relation* is defined between program, initial state and final state (intermediate states of computation are not visible in the semantic model). For each command c in a program, $(c, s) \Rightarrow t$ denotes the big-step from initial state s to final state t. One can then define the semantics of the programming language based on these rules. The following deduction rules show the semantics of the if statement:

$$\frac{\text{bval}(b, s) \qquad (c_1, s) \Rightarrow t}{(\text{if } b \text{ then } c_1 \text{ else } c_2, s) \Rightarrow t}\,\text{IfTrue}$$

$$\frac{\neg\ \text{bval}(b, s) \qquad (c_2, s) \Rightarrow t}{(\text{if } b \text{ then } c_1 \text{ else } c_2, s) \Rightarrow t}\,\text{IfFalse}$$

Figure 12: The semantics of the *if* statement in a simple imperative language can be expressed using a natural deduction tree. There are two variations. If b evaluates to true (through the function bval) on the starting state s, then IfTrue states that the execution terminates in the same t in which the command c_1 terminates if it starts in s. Likewise, IfTrue does the same for the command c_2 in the false case.

The only actual computation that occurred in the rules in Figure 12 is the evaluation of the if guard using the `bval` function: the rest were just inductive definitions. `bval` takes as arguments both the boolean expression b and the state s. In this case, the b is an inductively defined expression over the usual set of boolean operators $\{\neg, \wedge, \vee\}$ and the state s is simply a mapping from variables to their (integer) values.

A sensible definition for `bval` would not be difficult to imagine: it would lookup the known values for variables in s and apply the standard definitions of $\neg, \wedge, \vee$ etc. as required to produce a final boolean output. Though the complexity of this function really depends on how complex the abstract machine model (and s) is. One could imagine an arbitrarily complicated state object which affects the computation of this value in subtle ways, such as including cache behaviour or register allocation.

Until now, only the semantics of a simple, high-level imperative language on a basic stack machine has been discussed. However, in this thesis, we are seeking to analyse the properties of a much more low-level language, namely the IR, on a more complex abstract machine, namely the model of a neutral atom quantum computer. Fortunately, operational semantics can be made as concrete or abstract as desired, so long as the big-step predicate $(c, s) \Rightarrow t$ remains well-defined.

To illustrate this, one can start by defining a state s, which represents the state of the neutral atom quantum computer over all time-steps. Although we will not formalise it precisely for this example, it roughly corresponds to the neutral atom grid and all atom positions along with their current trajectories at any time. The function $\text{pos} : \text{Atom} \times \text{Time} \to \mathbb{R}^2$ will be used to retrieve the current position of an atom at time t.

Now consider a simplified version of the move instruction defined in Section 5, $\text{move}(q_i, [x, y], [z, w], t_{\text{start}}, t_{\text{end}})$. This instruction moves the qubit q_i from its start position at $[x, y]$ to $[z, w]$ in the time interval $[t_{\text{start}}, t_{\text{end}})$. For this instruction, there are several safety

properties that we want to check for the command to be considered valid on the machine. These include making sure that the movement operation is not scheduled to move too fast as this could cause the atom to be lost and become decoherent. Likewise, two movement operations should not be scheduled on the same qubit at the same time. All of these properties could be encoded as big-step semantic rules, as shown in Figure 13. It would then be possible to continue to define the semantics of the IR for each subsystem of hardware, where each invalid use of the machine by the formal language would cause the big-step predicate to end up in a new error state, such as `Fault_LaserOverlap` for invalid Rabi pulses, which pulse the same qubit with two different frequencies, or `Fault_IdleTooLong` if the IR sequence causes a qubit to remain idle for longer than its coherence time.

$$\frac{t_{\text{start}} < t_{\text{end}} \qquad \text{pos}(q_0, t_{\text{start}}) = [x, y] \qquad \frac{|[z,w]-[x,y]|}{t_{\text{start}}-t_{\text{end}}} \leq v_{\max}}{(\text{move}(q_0, [x, y], [z, w], t_{\text{start}}, t_{\text{end}}), s) \Rightarrow \text{add\_path}(s, q_0, [x, y], [z, w], t_{\text{start}}, t_{\text{end}})}\ \text{MoveOK}$$

$$\frac{t_{\text{start}} < t_{\text{end}} \quad \text{pos}(q_0, t_{\text{start}}) = [x, y] \quad \frac{|[z,w]-[x,y]|}{t_{\text{start}}-t_{\text{end}}} > v_{\max}}{(\text{move}(q_0, [x, y], [z, w], t_{\text{start}}, t_{\text{end}}), s) \Rightarrow \text{Fault\_TooFast}}\ \text{Move2Fast}$$

$$\frac{\begin{array}{c} t_{\text{start}} < t_{\text{end}} \quad \text{pos}(q_0, t_{\text{start}}) = [x, y] \quad \frac{|[z,w]-[x,y]|}{t_{\text{start}}-t_{\text{end}}} \leq v_{\max} \\ q_0 \text{ scheduled already in } s \text{ at } [t_{\text{start}}, t_{\text{end}}) \end{array}}{(\text{move}(q_0, [x, y], [z, w], t_{\text{start}}, t_{\text{end}}), s) \Rightarrow \text{Fault\_MoveConflict}}\ \text{MoveConflict}$$

Figure 13: Possible inference rules for the big-step semantics of the move command in the ISA. The first inference rule represents a correct application of the `move` command as the qubit is in fact at the correct starting position, the time interval is valid and the move velocity is scheduled to be within the velocity limit. In this case, the final state is the initial state with the new path included. All of the other inference rules represent cases where an incorrect pre-condition holds and hence they all end up in special fault states. Note that the "faulty" inference rules could be left out but this would result in the logic becoming *incomplete* and therefore the semantics a *partial* function.

If one were to continue to assign big-step semantics to every syntactic construct in the IR, then it becomes possible to check the validity of entire compiled programs through an operational interpreter. Specifically, one could write an interpreter ("the verifier") that steps through entry of the compiled IR (known as the *certificate*). For each entry, the big-step judgement either returns $(c, s) \Rightarrow t$ (i.e. success) or $(c, s) \Rightarrow \text{Fault\_X}$, which denotes that the IR has been incorrectly compiled.

However, this approach has some obvious methodological shortcomings. Firstly, the approach presented thus far is effectively a simulator of the IR. This means that the entire IR must be "run" sequentially on the machine to detect errors. As neutral atom quantum computers become increasingly more complex, such as by incorporating thousands of qubits, possibly on 3D planes, the big-step approach scales poorly due to the large global state space. Secondly, the big-step predicate itself must be written so that it contains all of the side conditions for the correctness and safety of the machine, which makes its *modularity* limited.

By contrast, the approach that will be presented in the subsequent parts of this chapter allows for a modular implementation of correctness checking of the IR using *sheaves* and can be seen as a variation of denotational semantics that can consider arbitrary machine models. In addition, sheaves allow for the separation of the checking of various subsystems into their own structures and later to "merge" them together using the tools of category theory. This technique will be used to split the neutral atom quantum computer into several different *subsystems*, each of which retain some degree of independence, allowing for a far simpler expression of the overall model of the neutral atom quantum computer.

6.1 Primer: The Atom Transport Subsystem

In order to consider how formal verification will work for AtomGuard, a first example of a subsystem based on moves will be given. This subsystem is responsible for moving atoms around on the neutral atom grid so that they are in the correct places to apply 1Q and 2Q pulses for gate application. For example, atoms must always be moved into the entanglement zone to apply Rydberg pulses and after which, they should be moved back to an appropriate place in the storage zone. We note that the actual subsystem will be based around each dialect of the IR, but this "fictional" subsystem fully explains the modelling process using sheaves and hence can be used as a "recipe" for all further subsystems.

The movement of atoms on a neutral atom grid can be considered as a *continuous dynamical system* [32]. As shown in the paper, continuous dynamical systems are modelled as a tuple $F = \big(X, f_{\text{dyn}}, f_{\text{rdt}}, X_0\big)$. Here, X is a smooth manifold (the input space), $f_{\text{dyn}} : I \times X \to TX$ are the dynamics of the system, I is the input space and TX is the tangent bundle. $f_{\text{rdt}} : S \to O$ ("readout map") is a smooth map where O is the output space and $X_0 \subseteq X$ is the set of initial states.

This system can also be presented as an ordinary differential equation:

$$\dot{x} = f_{\text{dyn}}(x, u) \quad \text{for} \quad u \in I, x \in X, x_0 \in X_0$$

$$y = f_{\text{rdt}}(x) \quad \text{for} \quad y \in O$$

As shown in the paper, a sheaf $\tilde{S} \in \widetilde{\mathbf{Int}}$ can be associated to F by the construction:

$$S(l) = \big\{(u, x) : [0, l] \to I \times X \mid u, x \text{ are smooth and } \dot{x} = f_{\text{dyn}}(x, u)\big\}$$

This makes the sections of the sheaf $\tilde{S}$ solutions to the given differential equations for a certain input and initial condition $x_0 \in X_0$. Discrete time equations can likewise be modelled if the time-step is embedded. This approach can be repurposed for our uses: the dynamics of atom movement on the NAQC grid can be modelled using a simple two-dimensional kinematics-inspired ODE. Specifically, let the state $X = \mathbb{R}^2 \times \mathbb{R}^2$, i.e. (position, velocity). Then the input space I becomes the control signals from the IR (i.e. the `move` instructions). The dynamics f_{dyn} of the system are captured by the motion laws:

$$\dot{x}(t) = v(t)$$

$$\|\dot{x}_i(t)\| \leq v_{\max} \text{ for all } t$$

Finally, the readout map f_{rdt} simply outputs the current coordinates of the qubit and the initial conditions X_0 are the starting coordinates as provided by the `init` IR command. Using the construction shown above, this system can be "sheafified" to turn it from a differential equation model into a compositional-friendly object suitable for this thesis' verification model.

Specifically, we call this construction the *movement sheaf* $\widetilde{M}_i$ over the atom i's trajectory. For each $l \in \text{Int}$, it has the form:

$$\widetilde{M}(l) = \{x : [0, l] \to \mathbb{R}^2 \mid x \text{ is } C^1, \|\dot{x}(t)\| \leq v_{\max} \forall t \in [0, l]\}$$

Note that local sections of $\widetilde{M}$ are trajectories $x : [0, l] \to \mathbb{R}^2$, which might seem incompatible with our intended notion of local sections corresponding to `move` commands over their time intervals $[t_0, t_1)$. This is solved by storing an additional morphism $\text{Tr}_{\text{begin}} : l \to L$, where L is the total runtime of the IR on the machine. This morphism offsets the local section into its correct place in the global timeline.

This completes the design for individual atom movements on the machine. During runtime, the verifier would parse the IR for individual `move` instructions and each of these would correspond to a local section of the sheaf $\widetilde{M}_i$. If there is no `move` instruction active during this time, a constant local section of velocity 0 would be created. Each local section can then be checked *in isolation* for the physical correctness properties, e.g. the velocity bound. Due to the isolation and specifically unlike the operational interpreter, this could be done in parallel thus offering a path for better runtime performance. Then, by the sheaf axiom, if all of these local sections agree on their overlaps, then they glue uniquely into a global section over the whole IR runtime $[0, L]$. This global section is the *certificate* that the entire runtime adheres to the correctness properties that were originally defined under the ODE rules.

However, the reader will have noticed that the properties encoded by the movement sheaves $\widetilde{M}_i$ are not nearly as complete as the preceding inference rules for the operational semantics. Currently, $\widetilde{M}_i$ only checks for properties local to that individual atom as it has no notion of other atoms to which to compare itself. This severely limits the expresibility of correctness checks which can be performed and is inadequate for the purposes of the verifier. Therefore, a further construction must be designed to allow cross-atom properties to be represented and checked.

To start with, we note that, it is always possible to build the *product sheaf* $\widetilde{M}_q \times \widetilde{M}_{q'}$, which provides a means for checking the compatibility between all pairs of atom trajectories. However, leaving the construction as simply the product is insufficient. This would allow all possible (smooth, restricted velocity) atom trajectories to be valid. However, neutral atom quantum computers impose constraints on the parallel movement of atoms. The most notable of which is the "no AOD laser collision" constraint. Therefore, a mechanism for filtering out these invalid configurations must be devised.

The first step is to enrich the domain with the additional information provided by the `move` commands. In $\widetilde{M}_i$, local sections are not aware of the start and end positions of their movement commands, despite this information being provided by the IR command. Therefore, a new sheaf $E \in \mathbf{Int}$ is created, whose objects are $E(l) := \mathbb{R}^2 \times \mathbb{R}^2$, which holds the pair `(src, dst)` for each move command. The restriction map is defined analogously to $\widetilde{M}$, where it simply

retains the source and destination information for a smaller time subinterval. The sheaf E is not yet associated with M: this step is done later through a subsequent pullback.

As with the product $\widetilde{M}_i \times \widetilde{M}_j$ (for $i \neq j$), building the product $E \times E$ would include colliding movement operations. Therefore, a predicate `compatible2D` is created to filter out these invalid operations. This is a function $c : (\mathbb{R}^2 \times \mathbb{R}^2) \times (\mathbb{R}^2 \times \mathbb{R}^2) \to \mathbf{Bool}$[2] and which has been written in Lean code in Listing 5.

```
@[simp] def compatible2D (a b : E) : Prop :=
  -- row check
  !(a.src.x = b.src.x ∧ a.dst.x ≠ b.dst.x)
  ∧ !(a.dst.x   = b.dst.x   ∧ a.src.x ≠ b.src.x)
  ∧ !(a.src.x < b.src.x ∧ a.dst.x   ≥ b.dst.x)
  ∧ !(a.src.x > b.src.x ∧ a.dst.x   ≤ b.dst.x)
  -- column check
  ∧ !(a.src.y = b.src.y ∧ a.dst.y ≠ b.dst.y)
  ∧ !(a.dst.y   = b.dst.y   ∧ a.src.y ≠ b.src.y)
  ∧ !(a.src.y < b.src.y ∧ a.dst.y ≥ b.dst.y)
  ∧ !(a.src.y > b.src.y ∧ a.dst.y ≤ b.dst.y)
```

Listing 5: The `compatible2D` predicate can be expressed naturally in code in a language such as Lean: two operations are compatible if their trajectories do not cross, which can also be thought of as preserving the relative order in each coordinate under the `move` operation.

Using this function, a sub-sheaf called $\widetilde{\mathbf{Safe}} \subseteq \tilde{E} \times \tilde{E}$ can be created by forming the pullback along the `true` truth value in $\widetilde{\mathbf{Bool}}$. Specifically, this is exactly the subsheaf that causes the diagram Figure 14 to commute. This is equivalent to saying that it on any interval $l \in \mathbf{Int}$, it contains exactly the sections of $\tilde{E} \times \tilde{E}$ which cause the predicate c to return `true`.

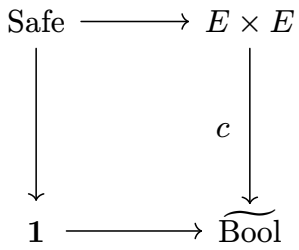


Figure 14: The sub-sheaf $\widetilde{\mathbf{Safe}}$ is formed by forming the pullback along true. Note that $\mathbf{1} \to \widetilde{\mathbf{Bool}}$ picks `true` everywhere.

It still remains to associate each local movement section with its corresponding local section in $\tilde{E}$. It's not possible to simply use the start and end positions $x(0)$ and $x(l)$ of each local section of $\widetilde{M}$ as this doesn't hold under the restriction map which maps into subintervals. Instead, when parsing the IR, we create an additional labelling object (specifically a *subobject*) $\mathrm{Lab}_q \hookrightarrow M_q \times E$. For an interval l, Lab_q simply contains the pairs (x, e) where e is the `(src,dest)` pair from $E(l)$ that is assigned to $M_q(l)$. For any restriction of $M_q(l)$, it likewise receives the same e. Just as before, we can use this object to perform a pullback on exactly the pairs of trajectories that are compatible under c. This becomes the pullback $\mathrm{Pull}_{q,r}$, which takes a pair of trajectories from $M_q \times M_r$, labels them via Lab_q, Lab_r and then filters out the pairs

[2] This is technically known as a classifying morphism in category theory.

whose labels remain in Safe. Because this is described in pure categorical terms, it can seem convoluted, but the Lean implementation can be simplified significantly.

$$\begin{array}{ccc} \mathrm{Pull}_{q,r} & \longrightarrow & \widetilde{M}_q \times \widetilde{M}_r \\ \downarrow & & \downarrow \\ & \mathrm{Lab}_q \times \mathrm{Lab}_r & \\ \downarrow & & \downarrow \\ \mathrm{Safe} & \xrightarrow{m} & \tilde{E} \times \tilde{E} \end{array}$$

Figure 15: Forming the pullback $\mathrm{Pull}_{q,r}$ causes both paths of the diagram to commute (become "equal").

Note that although only the pairwise case was discussed here, primarily due to the `compatible2D` function being pairwise defined, this construction generalises to any finite family of sheaves as all finite limits and products exist in $\mathbf{Shv}\ (\mathrm{Int})$ as it is a *topos* [23] and so it works on a neutral atom QPU of any finite number of qubits.

6.2 Abstract Machine Datum Definitions

Section 6.1 gave an introduction to the application of sheaves to verifying the movement of a set of atoms on the neutral-atom grid over time. It did this by starting with a sheaf $M : \mathbf{Int}^{\mathrm{op}} \to \mathrm{Set}$, which represented the ODE-based trajectories (i.e. position and velocity) of a single atom over time. A second sheaf, $E(l) = \mathbb{R}^2 \times \mathbb{R}^2$ was introduced to encode the `(src,dest)` pairs of the original `move` operations and this was filtered using a morphism $\mathrm{compatible2D} : E \times E \to \widetilde{\mathbf{Bool}}$ to ensure only geometrically correct pairs were in the subobject Safe. An additional labelling object `Lab` was then introduced to "connect" the M trajectories with their `move` operation variables, so that only the trajectories whose endpoint labels lie in `Safe` are considered as valid sections for the final sheaf.

In the AtomGuard verifier, there are many subsystems, and through modularity, more could always in principle be added. Therefore, this method of "manually" constructing sheaves is fairly cumbersome. Fortunately, A. Speranzon, D. I. Spivak, and S. Varadarajan [32] present an idea which generalises the method we presented earlier and which they call a *hybrid sheaf datum*. In their method, instead of just having one discrete state, for example, either laser active or not, the datum can support any number of discrete "modes" and these are able to interact with the continuous part (i.e. the trajectory in the previous section) in a principled manner. This is modelled in two parts. Firstly, the reflexive graph G represents each discrete state and has labelled transitions between them. $V \in \widetilde{\mathbf{Int}}$ is an interval sheaf representing the continuous data. A *realisation functor* then turns this tuple (G, V) into a proper interval sheaf whose sections pair the continuous data with valid discrete sequences (see paper for exact details on its construction). Each section of this sheaf then becomes a time-indexed trajectory whose continuous local sections are "labelled" by the discrete modes.

This structure can then be used to model the denotational semantics of both the high-level and low-level languages of the AtomGuard ISA, as each of these is essentially made up of instructions which operate on the neutral-atom state for some duration and this naturally corresponds to a discrete "mode" and associated continuous data over some time interval. Although this provides a convenient way to represent "hybrid" state, the additional and note-

worthy advantage of sheaf theory is that these hybrid data can always be composed through pullbacks. This allows for the application of additional constraints across different sheaves, thus satisfying the modularity requirement set out in the design goals.

The first datum we describe is for the high-level "portable" layer. We denote this construction as $\widetilde{H}_{\text{port}}(q) = H(V, G)$. In this structure, the discrete state G is relatively simple: if the atom has been moved to an entanglement zone, then a rydberg pulse must be applied on it at least once. We note that one could theoretically move an atom from the storage zone to the entanglement zone and then subsequently not apply any Rydberg pulse without actually causing incorrectness, however, this would cause a large fidelity loss due to the move operations, so in practice, this operation would never be scheduled by any compiler. For other operations, such as `measure` or `barrier`, these can be scheduled arbitrarily from the storage zone, as shown in Figure 16.

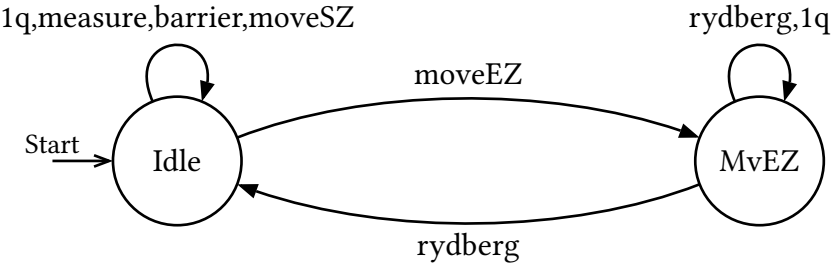


Figure 16: The reflexive graph for the sheaf $\widetilde{H}_{\text{port}}$.

Each hybrid datum also has a continuous structure V. In the case of the high-level layer, this sheaf is not concerned with the actual physical properties of the machine, so V consists only of the variables needed to execute each instruction. Specifically, on a time interval $[0, \ell]$, we use the following continuous data:

- $V_{\text{move}}(\ell) := \{(q, \text{src}, \text{dst})\}$
- $V_{\text{ryd}}(\ell) := \{(\{q\}, \text{zone})\}$
- $V_{\text{1q}}(\ell) := \{(q, \text{gate})\}$
- $V_{\text{meas}}(\ell) := \{(q, \text{basis})\}$
- $V_{\text{barrier}}(\ell) = \emptyset$

These can then be combined $V = V_{\text{move}} \uplus V_{\text{ryd}} \uplus V_{\text{1q}} \uplus V_{\text{meas}} \uplus V_{\text{barrier}}$ so that all necessary data for every operation is tracked. By building $\widetilde{H}_{\text{port}}$ in this manner, we have now encoded some of the basic operational semantics for the high-level layer, for example, that instructions cannot overlap in time or that Rydberg pulses can only be applied to qubits in the entanglement zone. Naturally, these are far from the only correctness predicates which must be encoded and in particular, *predicates*, which operate on sets of qubits rather than just individual ones are also important. This will be re-visited when the necessary tools, namely composition and products are introduced.

Underneath the portable layer, the ISA consists of subsystems which implement high-level instructions in terms of their own instructions for the actual physical devices of the neutral-atom QPU. In Table 8, a small dialect was given to control the AOD laser, which is responsible for moving atoms from one location to another on the grid.

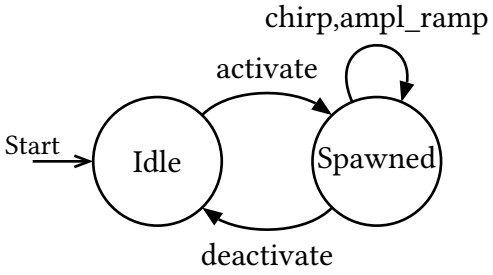


Figure 17: The reflexive graph for the sheaf $\widetilde{H}_{\mathrm{AOD}}$.

Just as previously, the correctness properties and semantics of this dialect can be stated as a sheaf datum $\widetilde{H}_{\mathrm{AOD}} = H(G, V)$. Figure 17 shows the required mode configurations for this language, namely a `activate` instruction must first be applied to "claim" the AOD and only then can the operations `chirp` and `ampl_ramp`, which cause atom movement, actually be applied.

Although the G component of $\widetilde{H}_{\mathrm{AOD}}$ operates similarly in principle to the previous sheaf, V is comparatively more complex due to the extra control signals of the subsystem and the trajectory of the atom playing an important role in the correctness of the instructions. Therefore, $V = \left(f_x, f_y, a_x, a_y, \dot{f}, \ddot{f}\right)$ includes the signals:

- Frequencies per tone $f_x, f_y \in C^2([0, \ell] \to \mathbb{R})$
- Amplitudes per tone $a_x, a_y \in C^1\left([0, \ell] \to \mathbb{R}_{\geq 0}\right)$
- The velocity and acceleration (derived through proportionality to frequency): $\left\|\dot{f}\right\|, \left\|\ddot{f}\right\|$

This already provides enough information to ensure additional local predicates remain true: in order to avoid "losing" an atom and hence causing decoherence, the AOD laser can only move atoms up to a certain velocity and acceleration, which is encoded by a predicate P where $\forall t \in [0, \ell] : \left\|\dot{f}(t)\right\| \leq v_{\max}$ and $\left\|\ddot{f}(t)\right\| \leq a_{\max}$. Formally and using this predicate, we form the subsheaf $\chi_P\left(\widetilde{H}_{\mathrm{AOD}}\right)$, where χ_P is the classifying morphism of P on $\widetilde{H}_{\mathrm{AOD}}$ (see Section 6.1), to express these semantics for the AOD dialect.

The subsequent dialect, the SLM dialect, is intended for activating and deactivating the SLM traps. This language, as defined in Table 9, is almost trivial: it controls either enabling or disabling the SLM trap at a specific coordinate. Therefore, this abstract machine is just "event-driven" and doesn't rely on any continuous dynamics (like the ODE in the AOD subsystem) and hence no V is strictly needed, so we take $V := \mathbf{1}$, that is the sheaf with strictly one section over every interval. However, if additional local checks on the semantics are needed in the future, these could trivially be added to V, exhibiting another advantage of the sheaf-based approach.

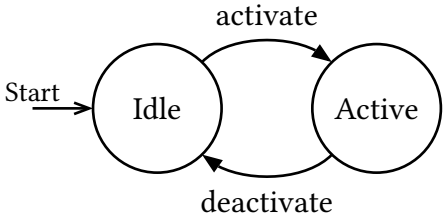


Figure 18: The reflexive graph for the sheaf $\widetilde{H}_{\mathrm{SLM}}$.

In a similar manner, both the Rydberg dialect and the Rabi dialect are simple subsystems defined by one event: their `pulse` function. Therefore, both components of (V, G) are trivial.

These sheaves can be seen as "carriers" for the pulse instructions and become more useful when composed with other sheaves to express more complex multi-subsystem constraints.

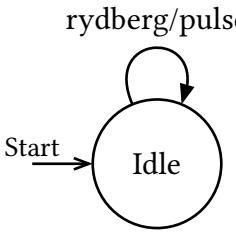


Figure 19: The reflexive graph for the sheaf $\widetilde{H}_{\text{Ryd}}$.

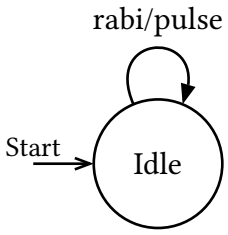


Figure 20: The reflexive graph for the sheaf $\widetilde{H}_{\text{Rabi}}$.

6.3 Sheaf-Theoretic Interfaces for AtomGuard Dialects

The preceding chapter defined the base sheaves in the AtomGuard verifier. These sheaves hold the instructions for each of the dialects and maintain some correctness by checking that basic, local predicates always hold through defining what can and cannot constitute a section of these sheaves. However, many correctness properties of the machine span more than one subsystem. For example, maintaining that an atom cannot be pulsed by the Rabi laser while it is being moved from the storage zone to the entanglement zone requires interacting with multiple sheaves, which has not been possible until now. In this chapter, we build up compositions of these base sheaves through the standard categorical tools outlined in Section 2.4, which allows for the expression of these predicates while maintaining modularity.

To start with, we define several correctness checks (called *contracts* in the literature [32]) first in prose and then proceed to formalise them for use in the sheaf system:

1. **No pulses during movement**: When a movement operation is active for a zone, there should be no Rabi pulse on this qubit nor a Rydberg pulse on the zone.
2. **Rydberg pulse after entanglement move**: If there is a move operation into an entanglement zone, this zone must be Rydberg pulsed at least once thereafter.
3. **AOD-SLM handoff**: Once the AOD subsystem has moved an atom and become deactivated, the SLM subsystem should become activate to trap the atom.
4. **1Q/Rydberg Exclusivity**: The Rabi subsystem and the Rydberg subsystem do not overlap temporally in the same location.
5. **Barrier Enforcement**: All parallel operations complete before a barrier operation.

It is clearly seen that all of these contracts span over multiple sheaves. For example, 1. requires knowing whether the AOD system is active for a particular zone and whether either the Rydberg or Rabi subsystem is firing. Similarly, 3. requires knowledge of the activities of both the AOD subsystem and the SLM subsystem. The necessary data to implement these predicates can be found in the hybrid sheaves of each subsystem and to access this data in a uniform manner, *interfaces* are defined on each of the $\widetilde{H}_d$ dialect sheaves. An interface $\mathbf{Obs}_d$ is a small sheaf, which for each dialect, contains the variables which are externally interpretable over each time interval (i.e. the "outputs" of each abstract machine). In order to implement the first contract, the necessary observable variables are:

- $\text{Active}_{\text{AOD}}(t) \in \{0, 1\}$ and $\text{Zones}_{\text{AOD}}(t) \subseteq \mathcal{Z}$ in $\mathbf{Obs}_{\text{AOD}}$, which describe whether the AOD laser is active and in which zones.

- $\text{Active}_{\text{Rabi}}(t) \in \{0, 1\}$ in $\mathbf{Obs}_{\text{Rabi}}$, which describes whether the global Rabi pulse is being fired at time t.
- $\text{Active}_{\text{Ryd}}(t) \in \{0, 1\}$ in $\mathbf{Obs}_{\text{Ryd}}$ and $\text{Zone}_{\text{Ryd}} \in \mathcal{Z}$, which describes whether the Rydberg pulse is being fired at time t and which (fixed) zone this device was configured for in the architecture.

For both the Rabi and Rydberg sheaves, these variables are trivial to compute as they relate to the discrete state of the sheaves and are implemented through the projections $\pi_{\text{Rabi}} : \widetilde{H}_{\text{Rabi}} \to \mathbf{Obs}_{\text{Rabi}}$ and $\pi_{\text{Ryd}} : \widetilde{H}_{\text{Ryd}} \to \mathbf{Obs}_{\text{Ryd}}$ respectively. $\mathbf{Obs}_{\text{AOD}}$ is more complex as it requires receiving information from the portable layer, namely the `src` and `dst` of the `rearrange` instructions so that the active zones can be computed. This requires an additional mapping $W : \widetilde{H}_{\text{port}} \to \mathbf{Obs}_{\text{AOD}}$, which collects the zones used by each move operation active in each local section, in addition to the projection π_{AOD}. Once the interfaces are defined, it is possible to define the contract over

$$\mathbf{Obs}_{\text{AOD}} \times \mathbf{Obs}_{\text{Ryd}} \times \mathbf{Obs}_{\text{Rabi}}$$

In fact, this is a simple morphism onto $\widetilde{\mathbf{Bool}}$, just as `comptatible2D` previously. We define $c_1 : \mathbf{Obs}_{\text{AOD}} \times \mathbf{Obs}_{\text{Ryd}} \times \mathbf{Obs}_{\text{Rabi}} \to \widetilde{\mathbf{Bool}}$ as

$$\neg\big(\text{Active}_{\text{AOD}}(t) \wedge \big(\text{Active}_{\text{Rabi}}(t) \vee \big(\text{Active}_{\text{Ryd}}(t) \wedge \text{Zone}_{\text{Ryd}} \in \text{Zones}_{\text{AOD}}(t)\big)\big)\big)$$

Just as in Figure 15, we can then enforce this contract on the dialect sheaves $\widetilde{H}_d$ by the fibre pullback on `true` to get $C_1 \hookrightarrow \prod_d \widetilde{H}_d$. In fact, assuming we have the other additional contracts $c_2, ..., c_5$, we can do this on all of the contracts to get $C_1, ..., C_5$. The final sheaf for the system is then $\widetilde{H}_{\text{QPU}} = \lim\big(\prod_d H_d \leftarrow C_1, ..., C_5\big)$ or stated equivalently, the intersection (product) of the dialect sheaves and $C_1, ..., C_5$.

This construction allows us to add any new contract to the abstract machine without much hassle: simply create a new contract morphism c_i over the observed variables from any of the dialect sheaves, form the pullback C_i and intersect it with $\widetilde{H}_{\text{QPU}}$ to add this constraint to each "top-level" section of the machine and hence the valid IR semantics.

6.4 Preservation of Circuit Semantics

We now make a departure from the sheaf-based modelling of subsystems, which is designed to ensure hardware correctness, to focus on modelling the semantics of the input circuit and compiled programs. Our aim is to ensure these semantics are preserved under compilation. Note that the condition we are looking for is *functional equivalence*, i.e. that both the input circuit and the compiled program implement the same unitary transformation over time (modulo phase differences), that is to say $U_{\text{comp}} = e^{i\varphi} U_{\text{in}}$ for a factor $\varphi \in \mathbb{C}$. Exact circuit diagram equivalence would not be sufficient as it is perfectly valid for the compiler to modify the program in order to optimise the circuit's runtime, e.g. by swapping gate orderings or even merging gates together.

One tool for checking functional equivalence between circuits is the ZX-calculus [33]. The ZX-calculus is a universal notation for capturing processes on quantum states in a graphical manner [34] and one can consider it as an extension of the basic gate and wire diagrams (i.e.

symmetric monoidal categories) with, among other things, additional rewrite rules (axioms) allowing for the simplification of the diagram.

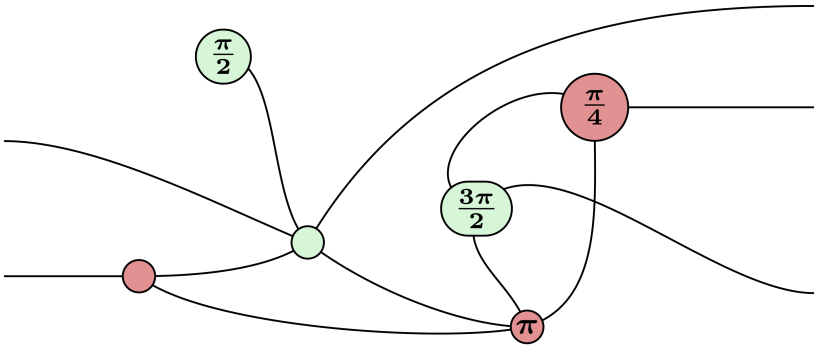


Figure 21: A ZX-diagram. Source: [35]

Specifically, each ZX graph is composed of *spiders* (nodes) of two types: Z-spiders (drawn in green) and X-spiders (drawn in red) connected by wires, where the top side is the output of a spider. Each spider is annotated with a phase angle (given in radians modulo 2π). A Z-spider, with m inputs and n outputs with angle α represents the linear mapping

$$|0\rangle^{\otimes m} \mapsto |0\rangle^{\otimes n}, |1\rangle^{\otimes m} \mapsto e^{i\alpha}|1\rangle^{\otimes n}$$

i.e. it "copies" $|0\rangle$ but transforms $|1\rangle$ by applying the phase α only if all of its incoming wires are $|1\rangle$. For the X-spider, defined similarly, the transformation is

$$|+\rangle^{\otimes m} \mapsto |+\rangle^{\otimes n}, |-\rangle^{\otimes m} \mapsto e^{i\alpha}|-\rangle^{\otimes n}$$

i.e. it does the same, but operates instead on the X basis, so applies the phase only if all inputs are $|-\rangle$. Although it may seen arbitrary upon first glance, this Z and X spider formalism allows for any quantum gate to be described as combinations of them. Furthermore, as this system is a *calculus*, it allows for rewrite rules to be performed. For example, two spiders of the same colour can be merged together with their phases summed (*spider fusion*) or additionally, a Z spider can be brought in front of a X spider if the X spider's phase is then inverted (*n-commutation*). Many, more complex rewrite rules also exist for various use-cases [34].

This formalism can be used to check the functional equivalence of two circuits through a multi-step algorithm. Firstly, both the input circuit and the compiled circuit must be converted to ZX diagrams, which we denote as D_S (source) and D_C (compiled). We then form the inverse D_C^{-1} of D_C by "flipping" the inputs and outputs and negating all of the phases ($\alpha \mapsto -\alpha$) in the spiders. Then, the two diagrams are combined $D' = D_S \circ D_C^{-1}$ by attaching the output of each wire of D_S to the input of each wire of D_C^{-1} in order to represent the unitary transformation $U_S U_C^\dagger$. The previously presented rewrite rules can then be performed on this circuit, in principle arbitrarily or using an automated strategy such as *full reduce* from PyZX [36], to simplify the circuit as far as possible. If the circuit can be reduced to the identity diagram (i.e. an empty diagram or trivial wires) or a set of disconnected spiders representing global phase, then we can deduce $U_S U_C^\dagger = e^{i\varphi} I$ and as such, the compiled circuit is functionally equivalent to the source circuit.

As ZX is *complete* over the fragment with gate set $\{\mathrm{U3}, \mathrm{CZ}\}$ (i.e. the native gate set for neutral atom QPUs), it is always possible to perform this process (a terminating proof derivation

always exists) and find if two circuits are unitary-equivalent. Furthermore, ZX-calculus is *sound* and therefore this semantic equivalence check is guaranteed to be correct [34].

The obviously missing part of the algorithm discussed until now is the actual construction of the ZX-diagrams from either the source circuit or the AtomGuard IR. For source circuits, which we assume to already be using the native gate set of $\{\mathrm{U3}, \mathrm{CZ}\}$, the process is simple. Each qubit in the circuit receives its own wire, running from left to right in the diagram. Then, the gates are processed in linear order. If a U3 gate is encountered, then it is decomposed into its rotations

$$\mathrm{U3}(\theta, \varphi, \lambda) = R_z(\varphi) R_x(\theta) R_z(\lambda)$$

Therefore, in the ZX-representation, the $R_z(\alpha)$ rotation becomes a Z-spider with phase α and likewise the $R_x(\beta)$ rotation becomes an X-spider with phase β. If a CZ gate is encountered (see Figure 7 for definition), then the corresponding wires of the qubits are attached to one Z-spider with phase π because, as stated in Section 2, the CZ gate acts as identity on all states other than $|11\rangle$, which it multiplies by -1.

The more intricate translation into a ZX-diagram is the compiled circuit, which must recover the algorithmic information about the circuit from lines of IR. However, because of the design of the AtomGuard portable dialect, it is not especially complex either as all necessary information is contained in it. As a reminder of the definition of the dialect in Section 5.2, the layer consists of six instructions:

- `init`: Establish a mapping from a logical qubit to a physical atom at a grid location.
- `rearrange`: Shuttle an atom from one location to another.
- `1q`: Perform a U3 pulse.
- `rydberg`: Perform a Rydberg pulse.
- `measure`: Perform a measurement on a specific grid location.
- `barrier`: A logical fence to be used as a synchronisation primitive.

We note that `barrier` and `rearrange` instructions are not relevant to the semantics of the circuit, so these can be ignored. Furthermore, the `rydberg` instruction can logically be be seen as a `CZ` gate, so long as the two atoms participating in the entanglement are both present in the entanglement zone and within the common Rydberg radius. However, this geometric precondition is already checked by the static analysis provided by the hardware correctness verifier, so this is always the case.

With the remaining instructions, it is possible to build an *interpretation function* from IR to ZX-diagrams. Although ZX-diagrams are often given in a strictly graphical manner, their theory is rooted in category theory and in particular, they are examples of free dagger symmetric monoidal categories (†-SMCs) [37]. Loosely speaking, symmetric monoidal categories (SMC) are categories where the objects can be combined together using the symmetric binary operation $\otimes$ ("tensor product"). They are used to encode the ideas of *serial* composition and *parallel* composition. A common example of an SMC is $(\mathrm{Set}, \times, \mathbf{1})$, which operates on set objects and whose morphisms are functions between sets. The tensor product is defined as $A \otimes B := A \times B$. Its unit object is $\mathbf{1} = \{\star\}$. The category is monoidal because of the associativity and symmetry of $\times$ as well as the existence of a unit $\mathbf{1}$. SMCs already provide enough structure to

model plain circuits that operate only forward in time and suffice for our purposes. However, in actuality, quantum circuits are reversible: in particular, every gate g additionally has an adjoint $g^\dagger$ and the tensor product is correspondingly adapted to preserve this.

The first step is to create an SMC called **IR** from the instruction stream. Here, the objects are the natural numbers $0, 1, 2, \ldots$, which are interpreted as "wire types", e.g. the object 0 is "no wires" while 2 is "two qubit wires". Each IR instruction is a *morphism generator* for **IR**, i.e. a morphism from a specific object to another specific object:

- $\text{init} : 0 \to 1$
- $\text{1q}(\theta, \varphi, \lambda) : 1 \to 1$
- $\text{rydberg} : 2 \to 2$
- $\text{rearrange}, \text{measure}, \text{barrier} : \text{id}$

Note that the parameters to be kept are in the morphism, but are not shown here. As this is an SMC category, morphisms can be composed either by sequential composition ($\circ$) or parallel composition ($\otimes$). In the IR, parallel operations are always "compounded" into a single instruction with multiple qubit arguments, so this is trivial to detect. For example, the (simplified) IR output `init(...); 1q(...)` is interpreted as the composition $0 \overset{\text{init}}{\to} 1 \overset{\text{1q(...)}}{\to} 1$. This "chain" of compositions can then be used to build the whole circuit.

However, as shown above, ZX-diagrams only have two generators: the Z- and X-spiders. All linear maps (H, CZ, CNOT etc.) are ultimately derived from these two. Therefore, we must create a mapping from **IR** to ZX, i.e. the interpretation function (functor) $[\![\cdot]\!] : \mathbf{IR} \to \text{ZX}$:

For every object $n \in \mathbb{N}$, we simply let $[\![n]\!] = n$. Whereas, for arrows, we simply "rename or expand" the morphisms (the wiring itself is unchanged):

$$[\![x]\!] = \begin{cases} |0\rangle & \text{if } x = \text{init} \\ Z(\varphi) \circ X(\theta) \circ Z(\lambda) & \text{if } x = \text{1q}(\theta, \varphi, \lambda) \\ Z(\pi) & \text{if } x = \text{CZ} \end{cases}$$

Although we do not formally prove the correctness of the functor, it is trivial to show this when relying on the correctness of the $\text{1q}(\theta, \varphi, \lambda) = Z(\varphi) \circ X(\theta) \circ Z(\lambda)$ algebraic equivalence. This completes the transformation from instruction stream to ZX-diagram and allows us to check for the correctness of the compiler transformation on the circuit via the circuit equivalence check.

6.5 Summary

This chapter started with introducing different types of semantic models for programming languages. Specifically, denotational semantics and operational semantics were both introduced in the context of classical programming languages. However, the AtomGuard ISA is quite different to these classical languages in two important ways. Firstly, it spans multiple layers of abstraction through the use of dialects. Secondly, its operations include continuous time signals of actual hardware operations rather than typical algorithmic language constructs like "if … then … else …". Therefore, the verification model for AtomGuard is likewise different.

The first part in the verification model was to establish an abstract model of the machine. In principle, this could either be done at a very concrete level, such as a simulator of the QPU, or

a very abstract level, such as a joint Hilbert space over the atoms. In this thesis, sheaves were chosen to model the machine and denotational semantics of AtomGuard as they are naturally suited to hybrid systems, which combine continuous signals (such as movements of atoms) with discrete modes (such as individual instructions). They also have the advantage of modularity as the model can be simply extended using pullbacks while maintaining correctness, which means that no further lemmas or induction proof steps have to be carried out, as would be the case in a simulator-based approach.

It was also shown that in contrast to most classical languages, AtomGuard has the advantage that its IR can still be interpreted in the source language (the circuit-level) due to the split between the portable (algorithmic) layer and the subsystem dialects. This means that we were able to directly prove compiler equivalence using the tools of the ZX-calculus, where otherwise, we would have had to show equivalence on the abstract machine level, which is more complex.

7 Related Work

We now compare the work done in this thesis with the current field of research in quantum compilers and associated formal methods work. In this chapter, we are particularly interested in contrasting our work with how other IRs are designed, how well they provide modularity as well as the extent to which formal semantics are defined on them. We also briefly describe other methods for verifying the correctness of compiled quantum programs and the extent to which they compete with the sheaf-based model presented here.

Firstly, this thesis contributed a novel IR for neutral atom QPUs called AtomGuard. The main feature of this IR is the ability to specify both high-level, platform agnostic algorithms through the portable layer plus detailed hardware information through the lowerings. The approach of using this high-level to low-level translation (dialects) was pioneered by the MLIR project and there have been several attempts to bring this into the domain of quantum computing. For example, A. McCaskey and T. Nguyen [38] introduce a quantum dialect directly for LLVM's MLIR project. The dialect contains operations for qubit allocation (`qalloc`, `dealloc`), extraction (`qextract`) and instruction invocation (`inst`). This dialect is then progressively lowered into standard LLVM IR using the rules defined by Microsoft's QIR standard [39]. Various languages, such as OpenQASM and Q#, can be compiled into this quantum dialect and then further into the QIR-compliant LLVM IR, which is then to be used as a global intermediate represenation amongst all quantum computing vendors (IBM, QuEra, Rigetti, etc.). The authors argue that their dialect, which sits in the middle of the source language and QIR, provides a modular layer on which to build further language features (e.g. `for` loops in OpenQASM 3.0) without having to modify the underlying QIR.

Although this technique is certainly useful for quantum language designers, neither it nor QIR target the same level of granularity as AtomGuard: QIR is deliberately a hardware-agnostic standard which contains no notion of physical implementation details, such as pulsing lasers, shuttling or even hardware topology definitions. These details are left to the runtime implementations of the QPU or further vendor-specific compilation stages. Furthermore, the modularity in AtomGuard is intended to allow for expansion of the machine through additional devices like new types of laser pulses, which is different to the idea of modularity to allow for additional high-level language constructs. As AtomGuard explicitly targets solely neutral atom QPUs, its high-level IR (the portable layer) is very expressive in terms of algorithmic details and allows the entire semantics of the cirucit to be recovered in a simple and principled manner. However, in parallel, it is able to richly express low-level physical details, such as the frequency and duration of an AOD laser pulse. This design means it spans a much larger range of abstraction compared with other IRs, which, amongst other things, means that comparably many types of optimisations can be performed on it. We note that the idea of defining IRs that lower to QIR specifically with the ability to host optimisations has also been explored in the literature [40], though again, this work relies primarily on circuit-level optimisations and not physics-level optimisations.

However, none of the works we have discussed until now have handled verification. K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks [41], however, introduce a novel IR and optimiser named *SQIR*. This paper defines a small language (DSL) which lives inside the *Coq* proof assistant, which resembles the form of the OpenQASM language; for example, `H q; X q` defines a Hadamard gate followed by a Pauli-X gate on a qubit `q`. Importantly, they define formal denotational semantics on this language, where, for example, `H`, `X` are denoted by their unitary matrices and the sequence $[\![c_1; c_2]\!] = [\![c_2]\!] \cdot [\![c_1]\!]$. Programs are considered *equivalent* if their denotational matrices are the same. The second important part of the paper is their introduction of the optimiser *VOQC*, likewise written directly in Coq. VOQC contains functions which are able to transform (optimise) SQIR circuits, such as `cancel_two` or `merge_rotations` in order to increase performance by minimising the number of operations. Crucially, for each translation, they prove the theorem $\forall c.[\![c]\!] = [\![\mathrm{opt}(c)]\!]$, where c is a source program and $\mathrm{opt}(c)$ is the optimised version. These optimised circuits can then be exported into a language such as `OpenQASM` for further compilation, but with the guarantee that the optimisations have been correctly performed.

Whilst this approach is an excellent way to verify the correctness of compiler passes, unlike AtomGuard, it has no means to formally verify the actual target-executable program output of a compiler as it merely verifies circuit-level transformations. Through the modular, layered structure of AtomGuard, it is able to express both circuit-level semantics as well as target-level semantics and can verify the correctness of both. Therefore, we see this thesis as a strict extension of the functionality provided by SQIR.

We further note that the application of sheaves to the verification of compiler output is novel. However, their use in the general modelling of abstract machines, specifically as a means of guaranteeing that local to global correctness holds, has been recently explored in other projects. For example, R. Short *et al.* [42] explored their use in delay tolerant networks for satellite communication.

8 Conclusion

To bring this work to its conclusion, we first shortly reiterate the goals of the thesis. Namely, in Section 3, we noted that there exists a gap in the research regarding low-level IRs: many neutral-atom compilers target a high-level IR which disregards important aspects of the machine, such as pulse timings. However, simply targeting a low-level IR instead makes it more difficult to ensure that the algorithmic properties of the program are preserved under compilation because these data become "hidden" in the various signals and operations on the individual components of the QPU. Therefore, a hybrid model was suggested, whereby a high-level dialect describes "what" the machine should do, whilst low-level dialects, which operate on the continuous control signals of the actual hardware, provide the notion of "how" these operations are to be performed. The design of AtomGuard also allowed for these dialects to be extended or supplemented with entire new dialects as each high-level operation can be mapped or "lowered" to a dialect arbitrarily, thus facilitating one of the key goals of the project, namely modularity. In Section 7, we noted that this idea of IR dialects has been applied to quantum computing already, for example in QIR project, however, these projects aim to uniformly treat different quantum computing architectures as a platform for optimisations or cloud orchestration, whereas AtomGuard is designed to provide an abstraction over low-level, neutral-atom QPU specific subsystems for the purpose of verification and hardware extensibility in research environments.

As the AtomGuard ISA allows for the expression of quantum circuits in a wide range of abstraction whilst keeping modularity, this makes it more of a challenge to verify, which was the other main goal of the thesis. In Section 2, we noted that subtle errors in timing or ordering of operations could lead to entirely false computations on the device, which may or may not be detected by the end-user, therefore having a way to define correctness of the machine is highly important. In this thesis, we bridged two areas of research, namely formal methods and Category theory, by showing a novel way of defining the denotational semantics of a programming language using Sheaf theory, which has hitherto only been used to define the operation of safety systems like the *traffic collision avoidance system* (TCAS) in aviation [32] and fault-tolerant networking [42]. These denotational semantics are particularly suited to the verification of IRs in quantum computing because they naturally reflect the continuous nature of the control signals of low-level subsystems in neutral-atom QPUs, such as the AOD subsystem or Rabi pulse subsystem. Moreover, we sought to provide a modular way to verify IR in a mathematically precise manner, where a researcher could include a new subsystem to the IR and define its correctness properties, without having to, for example, re-prove induction steps. The model presented in Section 6 does just this: using the tools of Category theory, such as pullbacks and limits, it becomes trivial to compose the abstract machine with additional correctness constraints. From an end-user's perspective, this is much simpler than defining additional inference rules in operational semantics and then extending the case distinctions of an induction proof (as well as proving additional lemmas if required) for a formally-correct simulator. We note that it is also superior to ad-hoc solutions because the sheaf theorem gives us a mathematical guarantee of the correctness of our language, not just a "best-effort" run on

a simulator, as exists in some other compilers, though further work should be done to extend the abstract model to include more "realism", perhaps by including a noise model or fidelity analysis. Finally, the use of dialects again proved to be auspicious as formally showing the preservation of semantics did not even need to be done on the sheaf level, rather the tools of the ZX-calculus provided all of the tools required on the circuit-level.

We also believe that the tools developed in this thesis, particularly the sheaf-theoretic models of QPUs, are well-suited to further the development of quantum software verification and compilers in general. For example, the use of sheaf-theoretic denotational semantics could be used in the development of programming languages for analogue quantum computers, which remain quite rudimentary, as many of them control laser signals through the use of imperative commands embedded in Python, which does not give strict guarantees about the correctness of these operations. When combined with additional tools from, for example, type theory, such as effects or linear typing, this could improve the developer experience in writing the code while also providing rigorous correctness guarantees.

In any case, it is clear that the field of formal methods and verification will remain an important area of research as the development of practical quantum computing marches on and this work has demonstrated that multi-dialect IRs and novel forms of semantics could play a role in it.

9 Appendix

```
{
  "devices": {
    "clocks": [{ "name": "sys", "freq_hz": 1e6 }],
    "aods": [{
      "id": "aod0",
      "rows": 10,
      "cols": 10,
      "site_separation": 2,
      "rf_freq_range_MHz": [60,180]
    }],
    "zones": [
      {
        "id": "store0",
        "type": "storage",
        "geometry": { "offset": { "x": 0, "y": 0 }, "dimension": { "x": 30, "y": 30 } },
        "slms": [{
          "id": "slm0",
          "grid": {
            "rows": 10, "cols": 10,
            "site_separation": { "x": 3, "y": 3 },
            "origin": { "x": 0, "y": 0 }
          }
        }]}, {
        "id": "ent0",
        "type": "entanglement",
        "geometry": { "offset": { "x": -22, "y": 42 }, "dimension": { "x": 60, "y": 72 } },
        "slms": [
          {
            "id": "slm1",
            "grid": {
              "rows": 3, "cols": 7,
              "site_separation": { "x": 12, "y": 10 },
              "origin": { "x": -22, "y": 42 }
            }
          },
        ],
        "rydberg_extents": [
          { "offset": { "x": -22, "y": 33 }, "dimension": { "x": 60, "y": 39 } }]
      }],
    "geometry": { "offset": { "x": -32, "y": 0 }, "dimension": { "x": 92, "y": 72 } } }
  }
}
```

Listing 6: An example schema for a “generic” neutral atom QPU.

```
  {
    "lowerings": [{
      "id": 30,
      "dialect": "aod",
      "ops": [
      { "op": "aod/activate",   "tone_id": 0, "fx": 98.0,  "fy": 12.5, "power": 0.20 },
      { "op": "aod/activate",   "tone_id": 1, "fx": 101.0, "fy": 12.5, "power": 0.20 },
      { "op": "aod/chirp",      "tone_id": 0, "from": [98.0, 12.5],  "to": [108.0, 14.0],
"duration": 60000 },
      { "op": "aod/chirp",      "tone_id": 1, "from": [101.0, 12.5], "to": [111.0, 14.0],
"duration": 60000 },
      { "op": "aod/ampl_ramp",  "tone_id": 0, "p0": 0.20, "p1": 0.00, "duration": 10000 },
      { "op": "aod/ampl_ramp",  "tone_id": 1, "p0": 0.20, "p1": 0.00, "duration": 10000 },
      { "op": "aod/deactivate", "tone_id": 0 },
      { "op": "aod/deactivate", "tone_id": 1 }
],
      "contracts": [
        "compatible2D",
        "vmax",
      ]
    }]
  }
```

Listing 7: A lowering of a rearrange instruction to the AOD dialect.

Bibliography

[1] I. Georgescu, 'The DiVincenzo criteria 20 years on', *Nature Reviews. Physics*, vol. 2, no. 12, p. 666, 2020.

[2] IBM, 'IBM Quantum Documentation'. [Online]. Available: https://quantum.cloud.ibm.com/docs/en/guides/processor-types

[3] Quantinuum, 'Quantinuum's H-Series hits 56 physical qubits that are all-to-all connected, and departs the era of classical simulation'. [Online]. Available: https://www.quantinuum.com/blog/quantinuums-h-series-hits-56-physical-qubits-that-are-all-to-all-connected-and-departs-the-era-of-classical-simulation

[4] C. Sheng *et al.*, 'Efficient preparation of two-dimensional defect-free atom arrays with near-fewest sorting-atom moves', *Phys. Rev. Res.*, vol. 3, no. 2, p. 23008, Apr. 2021, doi: 10.1103/PhysRevResearch.3.023008.

[5] J. Wurtz *et al.*, 'Aquila: QuEra's 256-qubit neutral-atom quantum computer'. 2023. doi: https://doi.org/10.48550/arXiv.2306.11727.

[6] F. Romão and D. Vonk, 'MultiQ: A Compiler-Controller Co-Design for Neutral Atom Quantum Architectures', *TBD*, 2025.

[7] J. Gruska and others, *Quantum computing*, vol. 2005. McGraw-Hill London, 1999.

[8] S. Axler, *Linear algebra done right*. Springer Nature, 2024.

[9] J. J. Sakurai and J. Napolitano, *Modern quantum mechanics*. Cambridge University Press, 2020.

[10] H. Munoz-Bauza and D. Lidar, 'Scaling advantage in approximate optimization with quantum annealing', *Physical Review Letters*, vol. 134, no. 16, p. 160601, 2025.

[11] T. Toffoli, 'Reversible computing', in *International colloquium on automata, languages, and programming*, 1980, pp. 632–644.

[12] K. Wintersperger *et al.*, 'Neutral atom quantum computing hardware: performance and end-user perspective', *EPJ Quantum Technology*, vol. 10, no. 1, p. 32, 2023.

[13] LibreTexts, 'LibreTexts: Physical Chemistry'. [Online]. Available: https://chem.libretexts.org/

[14] J. Fortagh, H. Ott, A. Grossmann, and C. Zimmermann, 'Miniaturized magnetic guide for neutral atoms', *Applied Physics B*, vol. 70, no. 5, pp. 701–708, 2000.

[15] M. Schlosser, D. O. De Mello, D. Schäffner, T. Preuschoff, L. Kohfahl, and G. Birkl, 'Assembled arrays of Rydberg-interacting atoms', *Journal of Physics B: Atomic, Molecular and Optical Physics*, vol. 53, no. 14, p. 144001, 2020.

[16] C. J. Foot, *Atomic physics*, vol. 7. Oxford university press, 2005.

[17] F. Nogrette *et al.*, 'Single-Atom Trapping in Holographic 2D Arrays of Microtraps with Arbitrary Geometries', *Phys. Rev. X*, vol. 4, no. 2, p. 21034, May 2014, doi: 10.1103/PhysRevX.4.021034.

[18] M. Schlosser *et al.*, 'Scalable multilayer architecture of assembled single-atom qubit arrays in a three-dimensional Talbot tweezer lattice', *Physical review letters*, vol. 130, no. 18, p. 180601, 2023.

[19] J. Park *et al.*, 'Rydberg-atom experiment for the integer factorization problem', *Physical Review Research*, vol. 6, no. 2, p. 23241, 2024.

[20] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, vol. 2. Cambridge University Press, 2001.

[21] W.-H. Lin, D. B. Tan, and J. Cong, 'Reuse-aware compilation for zoned quantum architectures based on neutral atoms', in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 127–142.

[22] B. Fong and D. I. Spivak, 'Seven sketches in compositionality: An invitation to applied category theory', *arXiv preprint arXiv:1803.05316*, 2018.

[23] S. Mac Lane, *Categories for the working mathematician*, vol. 5. Springer Science & Business Media, 1998.

[24] J. M. Curry, *Sheaves, cosheaves and applications*. University of Pennsylvania, 2014.

[25] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[26] H. A. Gabbar, *Modern formal methods and applications*. Springer Science & Business Media, 2006.

[27] E. Grädel, 'Mathematische Logik', *Vorlesungsmanuskript, RWTH Aachen*, 2011.

[28] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, 'CompCert-a formally verified optimizing compiler', in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[29] J. Ouaknine and J. Worrell, 'Some recent results in metric temporal logic', in *International Conference on Formal Modeling and Analysis of Timed Systems*, 2008, pp. 1–13.

[30] C. Lattner *et al.*, 'MLIR: A Compiler Infrastructure for the End of Moore's Law'. [Online]. Available: https://arxiv.org/abs/2002.11054

[31] T. Nipkow and G. Klein, *Concrete semantics: with Isabelle/HOL*. Springer, 2014.

[32] A. Speranzon, D. I. Spivak, and S. Varadarajan, 'Abstraction, composition and contracts: A sheaf theoretic approach', *arXiv preprint arXiv:1802.03080*, 2018.

[33] T. Peham, L. Burgholzer, and R. Wille, 'Equivalence checking of quantum circuits with the ZX-calculus', *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 12, no. 3, pp. 662–675, 2022.

[34] B. Coecke, 'Basic ZX-calculus for students and professionals', *arXiv preprint arXiv:2303.03163*, 2023.

[35] Wikipedia:Jonnie102, 'ZX Diagram'. [Online]. Available: https://en.wikipedia.org/wiki/ZX-calculus/media/File:Zx-diagram-example.svg

[36] A. Kissinger and J. van de Wetering, 'PyZX: Large Scale Automated Diagrammatic Reasoning', in *Proceedings of 16th International Conference on Quantum Physics and Logic, Chapman University, Orange, CA, USA., 10-14 June 2019*, B. Coecke and M. Leifer, Eds., in Electronic Proceedings in Theoretical Computer Science, vol. 318. Open Publishing Association, 2020, pp. 229–241. doi: 10.4204/EPTCS.318.14.

[37] T. Carette, E. Jeandel, S. Perdrix, and R. Vilmart, 'Completeness of graphical languages for mixed state quantum mechanics', *ACM Transactions on Quantum Computing*, vol. 2, no. 4, pp. 1–28, 2021.

[38] A. McCaskey and T. Nguyen, 'A MLIR dialect for quantum assembly languages', in *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, 2021, pp. 255–264.

[39] M. Alan Geller, 'Introducing Quantum Intermediate Representation (QIR)'. [Online]. Available: https://quantum.microsoft.com/en-us/insights/blogs/qir/introducing-quantum-intermediate-representation-qir

[40] D. Ittah, T. Häner, V. Kliuchnikov, and T. Hoefler, 'QIRO: A static single assignment-based quantum program representation for optimization', *ACM Transactions on Quantum Computing*, vol. 3, no. 3, pp. 1–32, 2022.

[41] K. Hietala, R. Rand, S.-H. Hung, X. Wu, and M. Hicks, 'A verified optimizer for quantum circuits', *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[42] R. Short *et al.*, 'Sheaf theoretic models for routing in delay tolerant networks', in *2022 IEEE Aerospace Conference (AERO)*, 2022, pp. 1–19.