A high-level quantum programming language with dynamical types to represent Hilbert space bases

Mads J. Damgaard

June 8, 2020

Abstract

In this bachelor's project, a high-level quantum programming language for hybrid computers is presented. The language, which has yet to be named, uses dynamical types to track how basis states transform between different Hilbert space bases throughout the execution of a function. These types include a special Boolean type to represent the z-basis of single qubits, which makes it easy for both the compiler and any human reader to see when a function can be implemented as a classical procedure as well as a quantum procedure. The types furthermore help to increase the safety of the language and to make it easier to reason about the semantics of programs. This is aided by the fact that only variables of the (Boolean) z-basis type can be used as condition variables in the basic if statements of the language, which ensures that these statements are analogous to classical if statements.

1 Introduction

Quantum computing is a field in rapid development. As quantum computer technology develops more and more and quantum computers become more available, there will increasingly be a need for quantum programming languages that are easy to use.

Existing programming languages typically have a clear distinction between what is the classical part and what is the quantum part of a program. The paradigm here is to have a quantum language for building quantum circuits embedded in a classical language. But if the choice whether to compile and run a certain subroutine of a program as either a full quantum procedure, a full classical procedure or as quantum procedures embedded in classical control is all left to the compiler, the task of optimizing the program to fit the hardware of the users can be taken away from the programmer. This would also means that the same program can be exported and run on different machines. The ability to run programs on different kinds of quantum hardware is especially important with a technology were so much development is expected to happen.

1.1 Problem formulation

This project seeks to develop a hybrid quantum programming language with a high-level syntax where programs are written using the same syntax regardless of whether they are quantum algorithms or classical algorithms, or if they can be implemented as both. Whether an algorithm is run on quantum hardware or classical hardware should then be left for the compiler to decide and should not be a concern for the programmer. The language should at least have a subset that has the appearance of a classical programming language so that a programmer is able to write classical-like subroutines in a classical-like syntax.

1.2 Contents of this report

For the report of this project, we will first go through a short introduction to the quantum mechanics needed to understand subject of quantum computing and then briefly list some existing languages. We will then analyse how we can solve the problem formulated above in terms of a high-level quantum program language. The language solution that I have developed for this project will then be explained. First, I will try to give an overview of the central concepts of the language and state the grammar of the language. We will then look at the type checking rules that I have worked to formulate for this project. These type checking rules will hopefully aid the understanding of the different features of the language. We will then go through some common examples known from the field of quantum computing as a way to evaluate the language. It will thus not be an actual test of the language per say, but it will give us a chance discuss some benefits and shortcomings of the language. Finally, we will see a summarised discussion of the language before concluding the report and describe the work left for the future regarding this project.

2 Background Knowledge

2.1 Quantum mechanics and quantum computing

A quantum computer is a computer that makes computations on carefully controlled quantum mechanical systems. In order to understand quantum computers in great detail, one would therefore need an introduction to the field of quantum mechanics. For our purposes, however, such a full introduction is not necessary as we will only really need to see how to treat special quantum mechanical states called *spinors*. The reason why these states are important is that they can be used much like bits for a quantum computer. These quantum mechanical bits, which are called *qubits* for short, are much analogous to classical bits in that the can only have two distinct values, when we query them for their value. We will get back to these qubits and spinors in a moment, once we have introduced the most basic concepts of quantum mechanics.

In a nutshell, quantum mechanics treats particles of a system, not as point-like objects, which is common for classical physics, but as waves. A quantum mechanical system is thus described by a *wave function*, which is a complex-valued function over all possible configurations of a system and whose absolute square describes the probability function for finding the system in each of the possible configurations. The wave function is therefore also sometimes referred to as the *state* of the system. A system of a single particle in three dimensions would as an example be described by a wave function over $\mathbb{R}^3$ since the possible configurations are all the positions it can be in in $\mathbb{R}^3$. For two particles, the wave function will be a function over $\mathbb{R}^6$ since now the configurations are any pair of positions, both in $\mathbb{R}^3$. The dynamics of such systems are then described by linear (probability preserving) wave equations. For the interested reader, in case of particles moving in an n-dimentional space such as these, the relevant wave equation is called the Schrödinger equation.[1] In the standard interpretation of quantum mechanics, the wave function then follows the relevant equation until we decide to make a *measurement*, in which case the wave function *collapses* according its aforementioned derived probability function. When we measure particles to be in a certain configuration, the wave function thus collapses into a single peak around the coordinates of this configuration. It is not a particularly realistic example but for simplicity, we can imagine an array of traps being activated, which then traps all n particles of a given system. In this case, the probability of finding the particles in one trap configuration would be exactly that of the wave function's absolute square integrated over the volume of the specific trap configuration (in the $3n$-dimensional configuration space).

[1]See for instance Griffiths [2] for an introduction to the Schrödinger equation.

This is how quantum mechanics works for continuous systems of particles. We can also have systems where there are only a finite amount of discrete configurations. A simple example of such a system is a trapped ion, which is located in exactly one position, but which has a so-called *spin*. This spin can take a finite amount of discrete values. The reason why is is called a spin is that it is analogous to an angular momentum (i.e. a rotational movement) of the ion. The spin of an ion produces a magnetic field (analogous to a spinning ball carrying a electrical charge) and this magnetic field can be used to interact with the ion and to measure which way it spins. Let us assume that the ion in this example has two spin-configurations, which is not an unrealistic assumption in practise. The spin can then either be *spin up* or *spin down* when measured along an axis, where up and down refers to the direction of the angular momentum. This means that the ion can only spin one way or the other around the chosen axis for a measurement. The wave function of such a system is then no longer a continuous function, as there are only two distinct configurations. The wave function, being complex-valued function over the configuration space, will thus simply be function over a binary set and can be represented with just two complex numbers. For discrete systems such as these, we therefore usually use the term state rather than wave function, as it is no longer a continuous function. We can then represent a general state of such binary system by

$$|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \tag{1}$$

where α and β are two complex numbers, and where we use the bra-ket notation, also called the Dirac notation[2], when we write the state as $|\psi\rangle$. We call these kinds of states spinor states, or spinors for short. In terms of the probability distribution, $|\alpha|^2$ will then be the probability of measuring the spin as being spin up, and $|\beta|^2$ will be the probability of measuring the spin as being spin down. The two absolute squares therefore always sum to 1 for realistic states.

As was mentioned, the equations determining the dynamics of a quantum system are always probability-preserving. Another very central part of quantum mechanics is that the equations are always linear. This means (by definition) that any sum of individual solutions to the equations will also be a solution itself. A reader familiar with vector spaces will therefore be able to see that if we treat $|\psi\rangle$ as a vector in a two-dimensional vector space, any function that sends $|\psi\rangle$ into its corresponding time-evolved form must be a linear function. Furthermore, if we take the vector space to be an inner product space with inner product $\langle\psi|\phi\rangle$ (using bra-ket notation) given by

$$\langle\psi|\phi\rangle = \alpha^*\gamma + \beta^*\delta, \qquad |\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \quad |\phi\rangle = \begin{pmatrix} \gamma \\ \delta \end{pmatrix}, \tag{2}$$

we see that the time-evolution function should also be unitary in order to be probability-preserving. The reason for this is that the probabilities derived from $|\psi\rangle$ sum to $|\alpha|^2 + |\beta|^2 = \langle\psi|\psi\rangle$, and because unitary functions are defined by being norm-preserving. The fact that we can treat the state of a quantum system as vector in a vector space and represent any time-evolution with a unitary function is a very useful fact, and it applies, not just for these spinor states, but for all quantum systems (including the continuous ones from before). Since quantum mechanics does not deal with incomplete vector spaces, the inner product vector spaces are always what is called *Hilbert spaces*, and are typically referred to as such.

The linear functions we see in literature on quantum mechanics typically take the form of operators (unary and typically right-associative), and when we deal with a discrete and finite system such as this spinor system, we can then represent such linear operators by matrices. For instance, some of the most common unitary matrices we run into when dealing with single-spinor

[2]See Griffiths [2].

systems, apart from the identity matrix, are the so-called Pauli matrices[3], which are given by

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{3}$$

These operators are equivalent of rotating a spinor 180° (or π radians) respectively around the x-axis, the y-axis and the z-axis. This is given that we use the standard convention of taking the basis vectors,

$$|\uparrow\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |\downarrow\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{4}$$

to respectively be a state with 100 % probability of being measured us spin up along the z-axis and a state with 100 % probability of being measured us spin down along the z-axis. Note that both σ_x and σ_y has the effect of flipping $|\uparrow\rangle$ and $|\downarrow\rangle$, which is what we would expect, and note also σ_y also has the effect of adding a complex phase factor to the state in both cases. While σ_z does not flip any of the two basis states, it also has the effect of adding a phase factor, namely -1, to the state when the state is a spin down state. It is a general fact that that a rotation operation can have the effect of adding complex phase factors to the states.

The fact the the Pauli matrices can be seen as three-dimensional rotations does not just apply for these special matrices, but applies for all unitary matrices on this single-spinor vector space. General rotations around the x-, y- and z-axes with an angle of θ can be represented respectively by the matrices[4]

$$R_x(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}, \quad R_y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}, \quad R_z(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 \\ 0 & e^{i\theta/2} \end{pmatrix}. \tag{5}$$

While a spinor can only be pointing either up or down along an axis once we measure it, the full wave function (before it collapses) actually points in a specific three-dimensional direction. If we thus start out with a $|\uparrow\rangle$ state, rotate it with the matrices such as the ones above, and measure it along an axis that is rotated in exactly same way compared to the z-axis, the spinor state will then always be measured as spin up along that axis. Since we can get any arbitrary state by such a rotation, a spinor can thus always be thought of as pointing in a specific direction.

Another matrix operator which is important for quantum computers is the Hadamard matrix given by

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{6}$$

It corresponds to a 180° rotation around an axis in the z-x-plane lying diagonally between the z- and the x-axis. It is useful since it transforms states back and forth between the usual basis, which we can also call the z-basis, and the basis where the basis vectors point along the x-axis, i.e. the x-basis.

Thus far, the only difference between a classical spinning ball and a quantum mechanical spinor, is that we cannot generally determine which way a spinor points if we are given an unknown state. We can only measure it once along an axis of our choosing and after we have done so, the wave function will have collapsed and will thus be pointing along exactly that axis. We thus only get a binary answer to our query and there is no way to know if we chose the right axis. While it is true that single-spinor states has this simple classical interpretation, things get more interesting when we add more spinors to our system. In our example this would amount to adding more trapped

[3]See Nielsen and Chuang [1] or Griffiths [2].
[4]See Nielsen and Chuang [1].

ions to our system. If we look at a system of two such ions, there are now a total of four different spin configurations of the two ions. The wave function should thus be a complex-valued function over a set of four distinct elements, and in terms of a vector in a complex vector space, it should now be a four-dimensional vector. We can for instance choose to represent the basis vectors $|{\uparrow\uparrow}\rangle$, $|{\uparrow\downarrow}\rangle$, $|{\downarrow\uparrow}\rangle$ and $|{\downarrow\downarrow}\rangle$ by

$$|{\uparrow\uparrow}\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \quad |{\uparrow\downarrow}\rangle = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \quad |{\downarrow\uparrow}\rangle = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \quad |{\downarrow\downarrow}\rangle = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}, \tag{7}$$

such that a general vector will be given by

$$|\psi\rangle = \alpha\,|{\uparrow\uparrow}\rangle + \beta\,|{\uparrow\downarrow}\rangle + \gamma\,|{\downarrow\uparrow}\rangle + \delta\,|{\downarrow\downarrow}\rangle = \begin{pmatrix}\alpha\\\beta\\\gamma\\\delta\end{pmatrix}. \tag{8}$$

If we for instance want to rotate either one of the spinors according the the Hadamard matrix, the relevant matrices for this two-spinor system would then be given by

$$H_1 = \frac{1}{\sqrt{2}}\begin{pmatrix}1&0&1&0\\0&1&0&1\\1&0&-1&0\\0&1&0&-1\end{pmatrix}, \quad H_2 = \begin{pmatrix}1&0&0&0\\0&-1&0&0\\0&0&1&0\\0&0&0&-1\end{pmatrix}, \tag{9}$$

where H_1 rotates the first spinor and H_2 rotates the second spinor. We can see how these matrices work as we want them to by multiplying them to the basis vectors above. The reader might also want to check that they are unitary, which, since they are Hermitian, can be done by multiplying them with themselves and check that the result is the identity matrix.

In order to have useful quantum computers, we also need operations that cannot be constructed simply from rotations individual spinors. When these spinor states are used as the qubits of a quantum computer, this would otherwise be analogous to having only unary, single-bit operations for our computer. We need at least some binary operations as well. It is not important for our purposes to know how this is achieved physically, say for system of trapped ions, but we only need to know that it can be done. In general, it is possible to construct any operation, also typically called a *gate* in terms of quantum computers, that makes any of the single-qubit rotations we have seen so far (within the precision of what is practically doable) conditioned on what states other qubits are in. We can for instance have a gate that makes a Hadamard operation on one qubit only in the part of the vector space where the other qubit is spin down. For our two-qubit system, such operations would be given by

$$H_1^c = \frac{1}{\sqrt{2}}\begin{pmatrix}\sqrt{2}&0&0&0\\0&1&0&1\\0&0&\sqrt{2}&0\\0&1&0&-1\end{pmatrix}, \quad H_2^c = \begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&-1\end{pmatrix}, \tag{10}$$

where H_1^c rotates the first qubit depending on what state the second qubit is in, and vice versa for H_2^c. We have used the superscript of c here, to denote *controlled.* This is the common terminology for quantum gates and we thus say that gates can be controlled by certain qubits, when the is gate only carried out in the part of the vector space where these qubits are spin down. By taking our $|{\uparrow}\rangle$ and $|{\downarrow}\rangle$ states to be the qubit states that respectively represent the binary values 0 and 1, we see that such operations are analogous to conditional operations, which only does non-trivial work on its target bits if the condition bits all have values of 1. It is indeed the convention to have $|\uparrow\rangle$ and $|\downarrow\rangle$ represent said values and they are therefore commonly renamed as $|0\rangle$ and $|1\rangle$ respectively.

The most common controlled gate we see in the field of quantum computers is the CNOT gate, which is given by

$$C_1 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}, \quad C_2 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{11}$$

depending on which qubit is the control qubit and which is the target qubit. We see that this is just a controlled σ_x gate. The reason why it is called a CNOT gate is because of its neat classical interpretation: the target bit is negated if and only if the control bit is a 1. When looking on the basis states in the basis where all spinors are either spin up or spin down, which is often called the *computational basis*, this is indeed exactly what happens. The only difference is that in a quantum system we can also have so-called *superpositions* of these basis vectors, i.e. vectors that are (non-trivial) linear combinations of the basis vectors. For these superposition states, the action of C_1 and C_2 are then the unitary operation according to eq. (11). A reader familiar with reversible computing might know that such controlled not gates are all we need to be able to construct any kind of classical procedure. Since the quantum operators are unitary, any quantum computation will always be reversible, at least until we start measuring qubits and causing a collapse of the wave function. A quantum computer can thus, when it is able to execute CNOT gates, make all the same computations that a reversible program can,[5] but is also able to make the same computations on input states that are superpositions of the computational basis vectors.

We can now get a sense of why quantum computers have an interesting potential. With quantum gates we are able to mimic classical operations, but on bits that, in a sense, are able to have several states at once. Our qubits can thus be in classical-like states when they state vectors point along a basis vector (of the computational basis) and they can also be in states that are superpositions of the basis vectors. For a qubit string of n qubits, there are 2^n distinct configurations, and the overall state can thus be a superposition of 2^n computational basis states. Since the gates act linearly on such states, the result when a gate acts on a superposition state will just be the same as the sum of the results when it acts on each individual basis vector of the linear combination. So for a gate that implements a classical-like operation, when it acts on a superposition state of all 2^n basis states (which can easily be obtained just by acting on each of the n qubits with a Hadamard gate), we can therefore in a sense think of the gate as making 2^n individual computations at the same time! If we for example have 16 qubits, the quantum gate can in this case be thought of as making 65,536 computations at the same time, and this number grows exponentially in the number of qubits. Since we cannot measure the whole wave function at once, but can only make one measurement of each of the qubits, we are not able to get the answer of all these individual computations at once. But for problems where we look for a relatively compact answer compared to how many computations on average are needed to solve the problem, quantum computers might have a potential to be able to solve such problems efficiently. An example of a well-known problem that quantum computers have been proven (at least theoretically) to be able to solve more efficiently than the known classical algorithms is the problem of breaking an RSA encryption. For this example, the quantum algorithm, which is called Shor's algorithm[6], is actually able to make an exponential speedup of the running time compared to the best known classical algorithms.

[5]At least unless the reversible program uses control such as while loops with no upper bound on the running-time, which is actually allowed for classical reversible languages even though it means that the algorithms cannot always be implemented as physically reversible procedures, even in theory.

[6]See Nielsen and Chuang [1].

Shor's algorithm make use of a so-called *quantum Fourier transform* as well as a *phase estimation* procedure. It will not be required as background knowledge for this project how the phase estimation procedure works, nor will a detailed explanation for the quantum Fourier transform be necesarry. The interested reader can for instance see Nielsen and Chuang [1] for an explanation of both. In terms of the quantum Fourier transform, all we need to know here is that it transforms states back and fourth from a computational basis and a Fourier transformed basis, where each basis vector has its qubit spinors lying in the x-y-plane such that the difference of the orientation of two neighboring spinors are the same for all neighboring pairs.

A quantum computer consists of hardware in the form of a controlled quantum system. If we stick to the example above this would be the trapped ions. Such systems have to be cooled down to very low temperatures in order for them not to lose *coherence*. When we talk about a system maintaining its coherence, we mean the wave function is not disturbed by interactions with the environment. We will also be using the term in relation to when qubits are measured in a way that unintentionally collapses part of the wave function essential to the relevant quantum computation. A good example where this might happen is when *ancilla qubits* are used for a computation but is not correctly uncomputed. Ancilla bits are known from reversible computing and ancilla qubits are exactly analogous. They are thus used to aid certain computations as a sort of local variables, which should then always be restored to their initial state after a computation. Of course, we can always "restore" ancilla qubits by measuring them and preparing them once again in their initial states, but if the ancilla qubits was not correctly uncomputed the resulting wave function collapse might interfere with the states of other qubits as well. When a measurement of one qubit state cause a wave function collaps that also interferes with another qubit, we say that the two qubits were *entangled*. Maybe the reader will have heard of this term in relation to quantum mechanics. For the interested reader, two states are formally said to be entangled if their combined wave function cannot be separated as a vector into a tensor product of two state vectors. An example of a seperable vector is $|\psi\rangle \propto |\uparrow\uparrow\rangle + |\uparrow\downarrow\rangle$ as $|\psi\rangle$ here is proportional to the tensor product of $|\uparrow\rangle \otimes (|\uparrow\rangle + |\downarrow\rangle)$, and an example of an entangled state is $|\psi\rangle \propto |\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle$, which cannot be written as a tensor product. In the first example, measuring either of the two spinor states will have no effect on the probability distribution of the other state. In the second example, however, if we for instance measure the first spinor as spin up, the probability distribution for a measurement of the second spinor will then also collapse into being 100 % for measuring it as spin down and 0 % for measuring it as spin up. Hopefully, the reader has now gotten at least some understanding of what coherence and entanglement means, and we can move on.

A quantum computer must then have some built-in gates that is can implement on its quantum data, i.e. its qubits, and also be able to execute these gates in different orders according to a program. A program of quantum gates are typically referred to as a *quantum circuit*. Such circuits are often depicted in literature as diagrams where the qubits run on wires from left to right through a sequence of gates.[7] For a reader interested in the field of quantum computers, these circuit diagrams are a very useful tool to depict and understand quantum algorithms on a low level. We will not really be needing these diagrams for this project, however. The reason is that once we begin to see quantum algorithms, we will already be seeing them written up as programs in a high-level language and circuit diagrams are essentially just programs of a lower-level language. Showing the algorithms as diagrams would therefore only aid the reader when said reader is already familiar with quantum circuit diagrams.

Apart from the *quantum hardware*, it is also possible for a quantum computer to have classical hardware, which can be used to compile quantum circuits and also to make other computations on classical hardware. We can thus have hybrid computers which utilizes both quantum and classical hardware. Since classical hardware is expected to be more efficient for certain kinds of computations (e.g. for computations that uses large amounts of data), at least for a long time to

[7]See for instance Nielsen and Chuang [1].

come, this would be beneficial for complicated programs that rely on several different subroutines. At the same time, many quantum algorithms, such as Shor's algorithm relies on quantum data being measured and new circuits being generated each time whenever until the right encryption key is found. A computer that utilizes algorithms such as Shor's algorithm as subroutines must therefore be such hybrids.

2.2 Existing quantum programming languages

Even though we will not be discussing existing languages much in this report, it is still worthwhile to take a brief moment to look at an overview of the different kinds of existing languages.

The main jumping-off point for the language of this project has been Quipper, which is described in Green et al. [3] and [4]. It is a higher-order quantum programming language embedded in Haskell and is used to build quantum circuits. Such quantum circuits can have both qubit wires and bit wires. These wires are represented in Quipper by the types `Qubit` and `Bit`. The standard functions then take these wire types as input and build circuits that operate on the wires. A circuit then (typically) has outgoing wires, which can have more circuits added to them afterwards. This way, the functions can be used to build complicated circuits in a modular way. There is also a third type called `Bool`, which can be used to specify how circuits should be built. These Boolean wires can potentially come from measurement of a previous circuit, which is how Quipper allow for circuits to be generated dynamically depending on previous output. While Quipper does separate classical data from quantum mechanical data, and classical functions from quantum mechanical ones, it is possible to define a function classically and then transform it to a quantum mechanical one with the `classical_to_reversible` operator.

Some other programming languages include QASM and Quil, which are assembly languages designed for quantum hardware.[8] While QASM is an instruction set only for quantum hardware, Quil also includes classical instructions and can thus be used for hybrid computers.

There are also many procedural quantum languages with a higher-level syntax for classical control. An example is QCL.[9] Here quantum operators are free to be defined from complex (floating point) numbers representing matrix indices. Another example is QPL[10], which also have high-level syntax to define classical control.

Then there are functional quantum programming languages such as QML, which is described in Altenkirch and Grattage [5]. This language is interesting in the context of this project since it uses quantum mechanical if expressions to define controlled operations. These if expressions operate on any qubit variable, however, which means that you generally have to think about them in a quantum mechanical way. This is opposed to the quantum programming language described in this project where there always is a sound interpretation that sees the condition variables as Booleans for the function in which the if statement is used.

There are also other embedded languages designed to build quantum procedures as well such as SDKs for high-level languages. There is for example ProjectQ[11], which is a SDK for Python. Like the previously mentioned languages, ProjectQ is also used to define operations on qubit data, and does not distinguish between different kinds of qubits. What I have tried to do with the language of this project, as we will see below, is to differentiate between qubits via types that denote what states the qubits are in. The reasoning behind this is that qubit states can be very different and can range from having very simple and separable states that can almost be thought of as classical states, and having complicated entangled states that are impossible to make sense of in a classical way. None of the languages I have seen in literature tries to do this. The closest we seem to come to features that makes it possible to sometimes think of a qubit in a more down-to-earth way are

[8]See Cross et al. [8] for a description of QASM and Smith et al. [9] for a description of Quil.
[9]See Ömer [6].
[10]See Selinger [7].
[11]See Steiger et al. [10] for a description.

when languages include features to convert classical procedures to quantum procedures, such as Quipper has with the `classical_to_reversible` operator.

I will end this section here. For a more inclusive overview of different kinds of quantum languages see for instance Gay [11].

3 Problem analysis

We have seen how it is possible to have quantum gates that implements conditional operations, i.e. with the so-called controlled gates. Since our goal is to have a classical-like syntax at least as a part the language, it would be beneficial to be able to write these controlled gates as if statements, where the conditional variables are the control qubits. Since we want to have the same syntax regardless of whether an if statement is used for a quantum algorithm or a classical algorithm, the if statements should look the same regardless of whether the condition variables are classical Boolean types or qubits. But an important issue that we then need to consider, is the fact that controlled gates only have the classical interpretation of a conditional operation when the control qubits are basis vectors of the computational basis. If their states on the other hand is a superposition of these basis vectors, the picture loses its simple interpretation. In fact, when we use a controlled gate with control qubits in such a superposition, the action of the gate can sometimes just as well be thought of as an operation *on* the control qubits and controlled by the target qubits. This goes against the usual understanding of an if statement where the condition variables always remain unchanged by the statement (at least if there is no reference of them inside the body of the if statement).

In order to make sure that the if statements do not change their condition variables, we should therefore include a type which represent the computational basis for qubit states, which is the z-basis for spinors with our conventions. When conditional variables have this type, we will then know that the if statements behave in a sensible way. By calling this type `bool`, we also make sure that the program can have a classical-like appearance for algorithms that can be run on classical hardware. We should therefore take the type `bool` to represent either a classical Boolean variable or a qubit with a state of the z-basis. The programmer can then interpret this type either of these two ways depending on what makes the most sense in the context. And at the same time, the compiler can also choose to interpret the type either of the ways, so to speak, when aiming to compile the program in the most efficient way possible.

Now, since we cannot utilize the potential of quantum hardware if all qubits have to be in the computational basis all the time, we also need other types than `bool`. The straightforward option is then to include a type such as `qbit` to a qubit with any state and not limited to states of the computational basis. We can, however, also include other types such as for instance `xspin` and `yspin` (which will be used for this language solution) to denote the other two common single-spinor bases, i.e. the x-basis and the y-basis. This would be beneficial since gates, such as for instance the Hadamard gate, can transform states from basis vectors of one basis into basis vectors of another. If we for instance have `xspin` as a type and use a Hadamard gate on a variable of this type, then the variable gets a state in the z-basis and can thus be thought of as a `bool` variable. Once it becomes a `bool` variable, we can once again use it as a conditional variable for if statements. For this reason, it will be beneficial to dynamically keep track of the types of variables.

Using different types also has the usual benefit of being able to reason about the correctness of a program. A good example where is an algorithm such as the one seen in Draper [13], where a the number represented by a state of the computational basis is added to a state in the Fourier transformed basis. The semantics of this algorithm is only meaningful (at least for most cases) if the two involved states start out in those bases. By giving the function that implements such algorithm the appropriate input types, we can make the language safer to use. To see this, let us look at a function that implements this algorithm, call it `AddToFourierNumber`. We will see below that its type can be written as `Int -> FTInt* -> ()` in the language solution of this project, when `Int` and `FTInt` are custom types defined to respectively denote computational and Fourier transformed integers. We will thus be taking the `*` qualifier as a way to signal that side-effects can happen for the relevant input. If not for these types, a programmer might then accidentally take the function on two `FTInts` instead of an `Int` and a `FTInt` if he or she forgets its intended domain. Even though this would technically be possible given that the `Int` and the `FTInt` types have the same amount of qubits, the output would probably not be meaningful at all compared to what the programmer expected. But by making sure that such function calls result the static check of the program to fail, we thus shield the programmer from such an error. We will see below, that we will not let the program fail immediately when these types are violated, just in case that the programmer actually *did* want to take the quantum algorithm on two `FTInts`. Instead such a call will result the variables getting the more relaxed `qbit` type, but this will still yield a failure to compile the program, if the programmer did not expect to generate these `qbit` types.

For the language developed in this project, I have therefore decided to keep track of the dynamic types via the type signatures of functions. The input types and output types are then taken to represent bases of the Hilbert space with the exception of some more relaxed types that can be used when we lose track of the bases. The function should then signal that if the input as a basis vector of the advertised input basis then the output is a basis vector of the advertised output basis (unless we lose track of them). We will see how this works in detail below. A function that only has `bool` types as input and output types will therefore have a classical interpretation, unless it is a special irreversible function. This should make it easy, both for any human reader of the program as well as for the compiler, to see that such functions can be implemented both as a classical procedure and a quantum procedure whenever they are taken on Boolean input. A compiler can for instance decide to only initialize Boolean variables as qubits once these are transformed to other bases and/or entangled with other qubits. If a Boolean qubit is measured before any such transformations, it can then be replaced by a classical bit altogether.

4 The language design

In this section, I will give a detailed explanation of the quantum programming langauge that I have designed for this project. I will start by briefly introducing the core concepts of the language and give some code examples so that the reader can get a basic understanding of it. I will then go on to define the full grammar of the language and explain the remaining features not covered in the introduction. The type checking rules will then be stated in the next section. These further restrict the language. In order to be able to write up and reason about these rules, we will have to consider the semantics of the relevant expressions or statements carefully. The type checking rules will thus not just serve as a tool for implementing the language in further work but will also serve to give the reader a detailed understanding of the language. If some of the concepts are a bit quickly explained in this section, it might therefore help to read about the same features in the next section as well to get a better understanding of them.

4.1 Introduction to the syntax and the basic concepts

The language of this project has some basic similarities with Quipper. Like Quipper, it uses Haskell-like type signatures to declare the type of the functions. These type signatures can have functional types both as input and as output. Despite this, the language uses a procedural syntax within the function definitions, which is another thing is has in common with Quipper. The language thus both have functional and procedural elements in it.

4.1.1 Functions and data types

Whereas the basic functions of Quipper take input representing wires in a circuit and returns new circuits, the basic functions of this language should be thought of as taking quantum states as input and outputting new quantum states, either via the return value or via side-effects on the input parameters. Such side-effects are only allowed to occur if the input type has the `*` type qualifier, in which case it can be thought of as a pointer to a quantum state. The return type is always either a type with no additional qualifier or the unit type, which is represented by `()`.

To give an example of a function declaration and definition, the well-known CNOT gate can be implemented by the following function.

```
CNOT :: bool -> bool* -> ()
CNOT a b {
    if (a) {
        Not b;
    }
}
```

We see that `bool` is a basic, built-in type of the language. It represents states which are either $|\uparrow\rangle_z$ or $|\downarrow\rangle_z$, as can be guessed from the conventional definition of the CNOT gate. We can also see in this example that the language includes if statements. The quantum mechanical semantics of such if statements are that the combined gate of the inner statements is controlled by the qubit variable of the condition. When the condition variable has the type `bool`, we see that we can think of these if statements classically as well. The function `Not` is then a built-in function equivalent to Pauli's σ_x which flips states in the z-basis. The built-in type signature of `Not` is `Not :: bool* -> ()`. It has no return value but takes an input parameter in the form of a pointer to a state in the z-basis, which means that it is allowed the change the state at this address. Only changes where the state remains in the z-basis (i.e. remains a `bool`) is allowed, however.

Types such as `bool` can be thought of as representing a certain basis of a Hilbert space. In the case of `bool`, it is the z-spinor basis of the single-qubit Hilbert space. The other built-in basis types are `xspin` and `yspin`, which represents to x-spinor basis and the y-spinor basis respectively.

The reason why the `bool` type is called as such, and not just `zspin` similarly to the rest, is clear when looking at the above example. We see that the whole definition of `CNOT` looks like a definition of a classical function. This is not by accident, but a key part of the language. The goal is to make an abstraction away from the quantum circuits and give the programmer the opportunity to think of the functions classically all the way up until they are used on quantum mechanical states that are superpositions of the z-basis states. The compiler is then, as mentioned, free to choose whether to run a classical-like function as either a quantum circuit or a classical procedure whenever both are possible, i.e. when the function is taken on Boolean input.

The language also includes other common statements of classical, procedural languages such as if-else statements, for loops and even while loops. The conditions used in the if and the if-else statements can also be compound and use logical operators `||`, `&&` and `!`, which operate on single-qubit variables.

4.1.2 Basic operations for creating, swapping and consuming variables

As a more complicated example of a classical function implemented in this language, let us try to implement the ripple-carry adder of Cuccaro et al. [12]. This will give us the opportunity to look at many other features of the language, including the ability to lump garbage return values together with the `+ garbage` type qualifier, while still being able to uncompute said garbage afterwards using special *temp* and *restore* statements. Here, temp of course is short for temporary. We will first need, however, to define a majority function, and this will give us the opportunity to see some more basic operations on variables.

The function `majority` has to return the majority of three Boolean input values. I will tend to use lower camel case for functions that return garbage and/or return a non-unit value and use upper camel case for the ones that only return a unit value and no garbage. A rather verbose, but instructive, version of `majority` can be defined as follows.

```
majority :: ()bool* -> ()bool* -> ()bool* -> bool + garbage
majority a b c_in {
    // declare aliases xor_ab for b and xor_ac for c.
    b    -> xor_ab;
    c_in -> xor_ac;
    // compute xors.
    if (a) {
        xor_ab <- Not . b;
        xor_ac <- Not . c_in;
    }
    // declare alias maj for a.
    a -> maj;
    // compute majority.
    if (xor_ab && xor_ac) {
        maj <- Not . a;
    }
    // discard xor_ab and xor_ac (i.e. b and c_in).
    discard xor_ab;
    discard xor_ac;
    // return the majority (consuming the input parameter).
    return maj;
}
```

The first important thing to note is that the input type of `()bool*` in the type signature means that the parameter in question comes into the function as a `bool*` variable and ends up as a unit type variable, which means that the variable is consumed by the function. When variables are consumed, they can either be used for the return value or they can go to adding garbage to the output. The `->` operator as seen in the first lines is used to create aliases of variables. It can be read as "becomes" and is a way for the programmer to be able to use instructive variable names when the variables change during the procedure despite having a limited amount of variables available. This is especially beneficial functions without the garbage qualifiers as these cannot initialise any new variables themselves. The `->` operator is thus only used to change the appearance of programs. The `<-` operator, which appears inside the two if statements, is more conventional and is somewhat similar to the one found in Quipper. In this language it can only be used, however, when all variables in the expression on the left-hand side (which can also be a variable tuple) are either uninitialized or are consumed in the expression on the right-hand side of the operator. The `<-` operator can be used to make swap operations between variables. An important point to make here is that all swap operations can be implemented as quantum gates, which ensures that if statements always will have a quantum mechanical implementation, even when variables are swapped around in the body. The dot operator as seen in `Not .` changes the type of the function to the left of it, such that it now consumes its input parameters and add them as a return value. It also uncurries higher-order functions. In the case of `Not .`, the function type is sent from `bool* -> ()` into `()bool* -> bool`. The `discard` function is used at the end to discard `b` and `c_in`. It is only needed for input parameter such as these, as any local data variable will be discarded automatically at the end of the relevant block. Whether discarded manually or automatically this will always add garbage to the output (making the `+ garbage` qualifier required), except if an uninitialized variable is discarded automatically. We could have chosen to not discard `b` and `c_in` and instead have their input types be `bool*`, in which case we would not have needed the garbage qualifier. But since we will only need the returned majority, it simplifies the function's API and also gives us the opportunity to look at garbage generation and uncomputation.

The above version of `majority` was a verbose but instructive implementation. Before we move on, let us briefly look at some other, shorter versions. By using CNOT from before, we can define it as follows.

```
majority :: ()bool* -> ()bool* -> ()bool* -> bool + garbage
majority a b c_in {
    bool g1 = CNOT a . b;
    bool g2 = CNOT a . c_in;
    if (g1 && g2) {
         Not a;
    }
    return a;
}
```

Here, g1 and g2 are declared within the block of the function body and is therefore automatically discarded. We see that new variables can be declared and initialized in much the same way as in C-like languages. Note that the dot operator can operate on CNOT a since this has the type of a function.

By defining a function to implement the well-known Toffoli gate[12], which is just a CNOT gate with two control qubits, call it Tof, we can also make the following implementation.

```
majority :: ()bool* -> ()bool* -> ()bool* -> bool + garbage
majority a b c_in {
    bool[2] (g1, g2);
    a -> maj;
    (g1, g2, maj) <- Tof . (CNOT a . b, CNOT a . c_in, a)
    return maj;
}
```

This example shows both how bool[2] is syntactic sugar for the product type (bool, bool), using a Haskell-like syntax for product types, and also shows how the dot operator uncurries higher-order functions. The expressions inside tuples or in function calls are actually evaluated right-to-left as opposed to the more normal convention. The reasoning behind this will be more clear when we see how the dot operator can be used to string functions together and make them behave just like quantum operators with the standard notation, binding to the right. If we for instance had changed the first expression in the tuple from the left from (CNOT a . b) to (CNOT a . c_in), the program would therefore fail at exactly this point since c_in will already have been consumed in the second expression in the tuple. Its dynamical type will thus have turned into () and (CNOT a . c_in) will fail as CNOT a . expects a bool. Changing the same expression to (CNOT b . a) would also fail since this would make both the first and the last expression in the tuple be based on a. We can however still use a as an input parameter in the two first expressions from the left as long as its state is not changed. This means that the input type cannot have the * qualifier, which is true for the first input of CNOT. The right-to-left evaluation applies both for tuples and for function calls. The rules for this will be explained in detail below.

4.1.3 Temporary computations

Let us now move on to look at how to implement the ripple-carry adder. If we stick to the description of Cuccaro et al. [12], it should take two qubit arrays plus an additional qubit as input and output one unchanged array (in the z-basis), one array containing the sum and then the overall carry-out XORed with the additional qubit input. The function that implements this gate can be defined as follows when the qubit arrays are four qubits long.

```
RippleCarryAdder :: bool[4] -> bool[4]* -> bool* -> ()
RippleCarryAdder a[0:3] b[0:3] z {
    // (s[0], s[1], s[2], s[3]) will end up as the sum (mod 16).
    b[0:3] -> s[0:3];
```

[12]See Nielsen and Chuang [1].

```
    // z' will end up as z ^ c_out.
    z -> z';
    // compute each carry-out progressively and then compute the
    // sum in reverse while restoring each necessary carry-outs on
    // the way.
    bool[4] c[0:3];
    temp c[0] <- new false;
        temp c[1] <- majority a[0] b[0] c[0];
            temp c[2] <- majority a[1], b[1], c[1];
                temp c[3] <- majority a[2], b[2], c[2];
                    temp c[4] <- majority a[3], b[3], c[3];
                        z' <- CNOT c[4] . z;
                    restore;
                    s[3] <- CNOT c[3] . CNOT a[3] . b[3];
                restore;
                s[2] <- CNOT c[2] . CNOT a[2] . b[3];
            restore;
            s[1] <- CNOT c[1] . CNOT a[1] . b[1];
        restore;
        s[0] <- CNOT c[0] . CNOT a[0] . b[0];
    restore;
}
```

The important thing to note here is that `temp` takes a statement (possibly a block statement), executes this statement and pushes it to a temp stack. A restore statement then runs the last statement from the top of the temp stack in reverse to restore all the original types and values. Garbage generated in the temp statement can also be uncomputed this way, but measurements are not allowed to take place in the statement. Since temp statements guarantees to restore the data variables in the future, they are allowed to make changes even to variables without the `*` qualifier. However, every variable used in a temp statement loses its `*` qualifier afterwards until the relevant restore statement is reached. Otherwise the restore statement would not be able to restore the variables correctly. But if there are nested temp statements (before the outer restore), then these are still free to change the variables that has lost their `*` qualifier. We can therefore see how we are able to keep on taking `majority` on both the `a`s and the `c`s, even though the `a`s come into the function body without the `*` qualifier and even though the `c`s loses it at the end of the previous temp statement in each case. When the `c`s are finally restored back to being uninitialized, they do not, as was mentioned, add garbage to the output when the block ends. All garbage is therefore uncomputed in `RippleCarryAdder`.

Some other things to note about the above implementation are first of all the fact that we have used `new false` to generate the ancilla qubit, `c[0]`, which is used for the 0-valued carry-in. We have also used syntactic sugar again, this time for the data variables as well. A single number in square brackets to the right of a variable can be regarded simply as a surname, i.e. as just a part of the variable's name. Then when two numbers appear with a `:` between them, such as `a[0:3]`, the expression can be exchanged for a tuple containing the whole range of numbers, in this case `(a[0], a[1], a[2], a[3])`. The numbers inside the square brackets are not restricted to constant numbers but can also be number variables and even compound arithmetic expressions.

The above implementation also gives an example of how the dot operator can be used to string functions together, such as the partially applied CNOT functions in this case. This is done by having the dot operator be overloaded as a binary, right-associative operator (with higher precedence) as well as the unary postfix operator. The semantics of e_2 `.` e_1 is then defined as being the same as (e_2 `.`) e_1 for any expressions e_1 and e_2. The action of the dot operator is therefore still to always change the type of the function just to the left of it.

4.1.4 Generic functions

As was mentioned, the language also contains for loops, which is a useful feature for functions such as `RippleCarryAdder`. It is even more useful if we want to define a generic `RippleCarryAdder` that uses a variable number to decide the length of the input tuples and to change to implementation accordingly. This is indeed possible, and we can define it by the following, where I have chosen to shift the index range to reduce the need for arithmetic operations inside the square brackets.

```
RippleCarryAdder<n> :: bool[n] -> bool[n]* -> bool* -> ()
RippleCarryAdder<n> a[1:n] b[1:n] z {
    b[1:n] -> s[1:n];
    z -> z';
    bool[n] c[1:n];
    temp c[1] = new false;
        for i = 1 to n {
            temp c[i + 1] = majority a[i] b[i] c[i];
        }{
            z' <- CNOT c[n] . z;
        }
        for i = n downto 1 {
            restore;
            s[i] <- CNOT c[i] . CNOT a[i] . b[i];
        }
    restore;
}
```

Here, I have included the extra block statement in the middle to make the three-part computation look more connected.

The possibility for generic functions such as these in the language is not just useful for defining functions on a whole range of input types at once. They also serve as a way to make it easier to build the control flow of a program in a modular way, without having to define the whole control flow in a single main function. The generic functions (or *variable functions* if one will) can thus use their number variables, not just in for loops, but also in special if statements that uses relational operators between numbers. They can also pass the numbers on in nested function calls including recursive calls to themselves. Functions are also allowed to measure data variables to generate new (generally random) numbers, as long as this is advertised by its type signature. The way to declare that a function makes measurements of quantum data along the way is by using `-->` as the last arrow of the type signature. This arrow can be understood as signaling that it the function is irreversible.

A compiler is not required to check validity of all possible variations of variable functions such as the one above when it is defined. The same applies for functions that uses measurements. A compiler is only required to check the constant functions statically, before any execution takes place.

As the language is intended for hybrid quantum and classical computers, we can think of all statements as being run and checked by a classical processor, using the type checking rules described below, while communicating to the quantum hardware what circuit operations needs to be done whenever the checks succeed. In this interpretation, we can think of the function calls as being implemented by a classical call stack. When we talk about reversibility, we therefore only talk about the quantum data, i.e. the fundamental data variables, and whether or not quantum mechanical coherence is maintained for these. While this is useful as a standard interpretation of the semantics of the language, the compiler is of course free to implement the programs in any way it "sees fit" as long as the outcome is the same.

Since I have no knowledge about whether the quantum hardware will be able to compete with the speed of the classical hardware in the future, I have not decided to put any upper (or lower) bounds for the number type. If it turns out that the classical hardware will be much faster, it would be beneficial to set the upper bound relatively high, if any bound is set at all. These bounds are therefore left to the compilers of the future to decide (when and if this particular language is implemented) by adding warnings when too large numbers are generated.

4.1.5 The built-in functions and their type signatures

By going through and dissecting the examples above, we have already covered most of the basic concepts of the language. What is left now is just to introduce some more concepts about the type signatures, as well as introducing the rest of the built-in gates/functions. We have already seen how the type of a data variable might change dynamically in a program, namely that it can turn into the unit type. This type represent that the variable is used up and no longer holds any meaningful data. Variables can also have their types changed into other types, as long as they have the same amount of underlying qubits and have the same structure. A `(bool, (bool, bool))` variable can for example be turned into a `(xspin, (yspin, zspin))` variable. Such type changes are associated with (unitary) transformations of the states held by the variables. They are shown in the type signature of a function by having the resulting type in brackets to the left of the initial type. This syntax is somewhat similar to the syntax of casting in C-like languages, but with a type on the right-hand side. The syntax can be seen to fit the general right-to-left theme of the language.

As an example of such type changes, the built-in function `Had` that implements the Hadamard gate has the following type signatures.

```
Had :: (xspin*)bool* -> ()
Had :: (bool*)xspin* -> ()
Had :: ~yspin* -> ()
```

The two first type signatures shows the fact that the Hadamard gate transforms states in the z-basis into states in the x-basis and vice versa. For states in the y-basis, it flips the state and changes the phase by $\pi/2$ and $-\pi/2$ respectively for the two basis states. Since the outgoing type is therefore still `yspin`, we can omit the brackets in front, but we need the tilde (∼) in front to advertise the phase changes. Formally, a tilde in front of an input type in a type signature states that there can potentially be a non-zero phase change to the overall output state and that this phase might depend on the initial basis state held by the relevant input variable. This means first of all that when no tildes appear, we know that the overall phase does not change relative to the bases of the input and output types. And if tildes appear on some input types, we at least know that the phase of the output states do not vary as long as we keep the states of the relevant input parameters constant. The tilde symbol can be read as "proportional to" when reading the type signature out loud. It can also be combined with an input type that changes type after the call. The tilde is still kept in front for an input type such as `∼(xspin*)bool*` even though we need to distribute it on each basis type, when reading it out loud. The input type `∼(xspin*)bool*` can thus be correctly understood and read as "a pointer to a state proportional to a bool (or z-spinor), which turns into a pointer to a state proportional to an x-spinor after the function call." Without the tilde, we can read it as just "a pointer a bool (or z-spinor), which turns into a pointer to an x-spinor after the function call."

The above example shows why it is beneficial to use Haskell-like function declarations in this language as it makes is easy to declare multiple type signatures for the same function. Most of the other built-in functions also have several type signatures. The other built-in functions are `Px`, `Py` and `Pz`, which implement the Pauli gates, and generic functions `Rx<n>`, `Ry<n>` and `Rz<n>`, which implement a rotation around the relevant axis. Following the convention used in Nielsen and Chuang [1], I take this rotation to be $2\pi/2^n$ radians.

One might note that `Px` implements an equivalent gate as `Not`. I have included `Px` in the language as well both for symmetry reasons and because it allows us to have the type signature of `Not` be only the classical-like type signature, which is the following.

```
Not :: bool* -> ()
```

The function Px on the other hand is granted all the type signatures as follow.

```
Px :: bool* -> ()
Px :: ~xspin -> ()
Px :: ~yspin* -> ()
```

Similar type signatures are given to Py and Pz but of course with the types cyclically exchanged. For Rx<n>, Ry<n> and Rz<n> we can not generally keep track of the bases except for the stationary one in each case. In case of Rz<n> as an example, where the z-spinor basis is the only stationary one, the type signature is therefore just the following one.

```
Rz<n> :: ~bool -> ()
```

The type signatures of Rx<n> and Ry<n> are similar but of course with types xspin and yspin instead.

There is one more built-in type in the language. It is the type qbit, and unlike the three others, it does not represent a basis set of a Hilbert space. We can instead often think of it as representing the full single-qubit Hilbert space. This interpretation is not always sufficient, however, as a variable that has obtained the qbit type can also be entangled with any other qbit variable, and also with any garbage produced. It does not appear without the * qualifier, so we can simple think of it as just a qubit pointer with no restriction on the state of that qubit. In practise, the qbit type is used to signal whenever the programmer decides to lose track of the Hilbert space basis of certain data variables. This happens when the relevant basis has not been defined (perhaps yet) as a type, and/or when the programmer does not find it helpful to keep track of it. The programmer are of course free to declare other types derived from the built-in ones. We will see an example of this just below.

4.1.6 Spanning types and custom types

To show an example where some of these types other than bool is used, and where a new type is declared, let us define a function to implement a generic quantum Fourier transform such as described by Nielsen and Chuang [1]. We can start by defining a function to rotate a spinor in the x-y plane by π times the floating point number $(0 \leq a < 1)$ held by an array of n bool qubits. This function, call it RotateByFloat<n>, can be defined as follows.

```
RotateByFloat<n> :: bool[n] -> (qbit*)xspin* -> ()
RotateByFloat<n> a[1:n] x {
    for i = 1 to n {
        if (a[i]) {
            Rz<i + 1> a0;
        }
    }
}
```

We can then use this function to implement the quantum Fourier transform by the following program code, where we also declare the generic Fourier transformed integer as a generic type.

```
QFT<n> :: (qbit[n]*)bool[n]* -> ()
QFT<n> a[1:n] {
    for i = 1 to n - 1 {
        Had a[i];
        RotateByFloat<n - i> a[i + 1 : n] a[i];
    }
    Had a[n];
}

type FTInt<n> := QFT<n> . bool[n]
```

Even though we chose to lose track of the Hilbert space basis along the way, we still know that the `QFT<n>` must implement a unitary transform and we should therefore be able to define a new Hilbert space basis given by its transform of a previously defined basis. In the last line, we thus define `FTInt<n>` to be the type that represents the basis obtained by transforming the computational basis of an n-qubit Hilbert space with the newly defined `QFT<n>` gate. The type declaration not only records the declared type but also records that the function `QFT<n> .` is given the type of `()bool[n]* -> FTInt<n>*` in addition to its former types. It is also records that the `FTInt<n>` basis spans the same Hilbert space as `bool[n]`. We can say that the span of the basis vectors of `FTInt<n>` is equal to the n-qubit Hilbert space and write this as span(`FTInt<n>`) = `qbit[n]`.

Let us now finally get back to the if statements and see what happens when viewing them quantum mechanically and why it is important to be able to keep track of the Hilbert space bases of the variables via their types. The important point is that while the syntax of an if statement suggest that nothing happens to the conditional variables, this is only a meaningful description of what happens if the conditional variables are in the computational state. If not, one can in some cases just as well argue that the conditional variables are the ones that are operated on, conditioned on states of the variables in the body statement, and not the other way around. For this reason, it is made sure that if all conditional variables must have the type `bool`. Any variable that has its type changed inside the body statement will also obtain the corresponding spanning type afterwards. The reason for this is that we cannot generally know the state of the condition variables and therefore cannot keep track of any types that changes conditioned on them. Additionally, phase changes are only allowed if all conditional variables have the $\sim$ qualifier to their in-going type.

Function calls make sure that all the variables involved gets their corresponding spanning type after a call if any of their input has the wrong basis type (except only for unchanged input types appearing before the first offending variable). The reason for this is again, similarly to the if statement, that the type signatures only really makes sense in a classical way of thinking about them if all input are of the correct basis (with no superpositions relative to it). Once the input basis is violated for just one input, the function can no longer be interpreted in any classical way. Of course if the input type is, say, `FTInt<4>`, we have already moved into quantum territory. But since any non-superposition state of such a basis is always only one well-defined transformation away from a non-superposition state of the computational basis, I would argue that one can still think of such functions in a classical way, at least when viewing them from the outside. By keeping track of the Hilbert space bases and making it clear whenever untracked superposition states are created, we therefore allow the programmer to safely make these classical abstractions when defining classical-like subroutines of a quantum program.

Functions are only allowed to be taken on other input types than what appears in their type signature, if these types are constructed with the same overall tuple structure (and thus spans the same Hilbert space). Otherwise, the function call will simply fail immediately.

4.1.7 Functions as input

The language also allows for higher-order functions that have function types as input types. There is a restriction here, however, as opposed to conventional functional languages which is that only function variables can be passed as input in function calls. They can in other words not be in the form of compound expressions. This is to eliminate the otherwise real possibility of unwanted aliasing in function calls. The function variable can be of a global function such as the ones we have seen so far, or it can be of a locally defined functions, the syntax of which will be stated below. Since these local functions, potentially defined from compound expressions, might implicitly work on live variables, it has to be checked that they do not change the types of these variables. It also has to be checked that none of the involved variables are used afterwards in the same local block.

An example of a higher-order function with function type input can be seen in section 6 below in the form of a function to implement an (un-optimized) phase estimation procedure.

4.2 Grammar

In this section, the full grammar of the language is introduced. The fundamental grammar used inside function definitions is given by the following.

$$
\begin{aligned}
\mathit{Type} &::= \mathit{Type}' \;\texttt{->}\; \mathit{Type} \mid \mathit{Type}' \;(\texttt{->} \mid \texttt{-->})\; \overline{\mathit{Type}} \;(\texttt{+ garbage})?\\
\mathit{Type}' &::= \sim \mathit{Type}' \mid \texttt{(}\; \overline{\mathit{Type}} \;\texttt{*}\; \texttt{)}\; \overline{\mathit{Type}} \;\texttt{*} \mid \texttt{( )}\; \overline{\mathit{Type}} \;\texttt{*} \mid \overline{\mathit{Type}} \;\texttt{*} \mid \overline{\mathit{Type}}\\
\overline{\mathit{Type}} &::= \texttt{( )} \mid \texttt{(}\; \overline{\mathit{Type}} \;(\texttt{,}\; \overline{\mathit{Type}})^* \;\texttt{)} \mid \overline{\mathit{Type}} \;\texttt{[}\; \mathit{Num} \;\texttt{]} \mid \widehat{\mathit{Type}}\\
\widehat{\mathit{Type}} &::= \texttt{bool} \mid \texttt{xspin} \mid \texttt{yspin} \mid \texttt{qbit} \mid \mathit{id} \;(\texttt{<}\; \mathit{Num} \;(\texttt{,}\; \mathit{Num})^* \;\texttt{>})?\\
\mathit{Stat} &::= \texttt{;} \mid \texttt{\{}\; \mathit{Stat}^* \;\texttt{\}} \mid \mathit{Exp} \;\texttt{;} \mid \overline{\mathit{Type}} \;\; \overline{\mathit{Exp}} \;(\texttt{=}\; \mathit{Exp})? \;\texttt{;} \mid \overline{\mathit{Exp}} \;\texttt{<-}\; \mathit{Exp} \mid\\
&\qquad \texttt{return}\; \mathit{Exp} \;\texttt{;} \mid \texttt{return}\; \mathit{Num} \;\texttt{;} \mid \texttt{discard}\; \mathit{Var} \;\texttt{;} \mid \texttt{temp}\; \mathit{Stat} \mid\\
&\qquad \texttt{restore ;} \mid \texttt{if (}\; \mathit{Cond} \;\texttt{)}\; \mathit{Stat} \;(\texttt{else}\; \mathit{Stat})? \mid\\
&\qquad \texttt{fun}\; \mathit{id} \;(\texttt{(}\; \overline{\mathit{Exp}} \;\texttt{:}\; \mathit{Type} \;\texttt{)})^* \;\texttt{=}\; \mathit{Exp} \;\texttt{;} \mid \texttt{let}\; \mathit{id} \;\texttt{=}\; \mathit{Num} \;\texttt{in}\; \mathit{Stat} \mid\\
&\qquad \texttt{let}\; \mathit{id} \;\texttt{=}\; \mathit{Prop} \;\texttt{in}\; \mathit{Stat} \mid \texttt{if (}\; \mathit{Prop} \;\texttt{)}\; \mathit{Stat} \;(\texttt{else}\; \mathit{Stat})? \mid\\
&\qquad \texttt{for}\; \mathit{id} \;\texttt{=}\; \mathit{Num} \;(\texttt{to} \mid \texttt{downto})\; \mathit{Num}\; \mathit{Stat} \mid \texttt{while (}\; \mathit{Var} \;\texttt{)}\; \mathit{Stat} \mid\\
&\qquad \texttt{assert}\; \overline{\mathit{Exp}} \;\texttt{of}\; \mathit{Type} \;\texttt{;} \mid \texttt{assume}\; \overline{\mathit{Exp}} \;\texttt{of}\; \mathit{Type} \;\texttt{;} \mid\\
&\qquad \texttt{assume}\; \overline{\mathit{Exp}} \;\texttt{=}\; (\mathit{Num} \mid \mathit{Prop}) \;\texttt{;}\\
\mathit{Exp} &::= \mathit{Exp}' \;\texttt{.}\; \mathit{Exp} \mid \mathit{Exp}' \;\texttt{.} \mid \mathit{Exp}'\\
\mathit{Exp}' &::= \mathit{Exp}'\; \mathit{Exp}'' \mid \mathit{Exp}''\\
\mathit{Exp}'' &::= \texttt{(}\; \mathit{Exp} \;(\texttt{,}\; \mathit{Exp})^* \;\texttt{)} \mid \overline{\mathit{Exp}} \;\texttt{->}\; \overline{\mathit{Exp}} \mid \overline{\mathit{Exp}} \mid \texttt{new}\; \mathit{Prop} \;(\texttt{[}\; \mathit{Num} \;\texttt{]})? \mid\\
&\qquad \texttt{new}\; \mathit{Num} \mid \mathit{id} \;(\texttt{<}\; \mathit{Num} \;(\texttt{,}\; \mathit{Num})^* \;\texttt{>})? \;\texttt{!}? \mid \texttt{Had} \mid \texttt{Not} \mid \texttt{Px} \mid \texttt{Py} \mid\\
&\qquad \texttt{Pz} \mid \texttt{Rx <}\; \mathit{Num} \;\texttt{>} \mid \texttt{Ry <}\; \mathit{Num} \;\texttt{>} \mid \texttt{Rz <}\; \mathit{Num} \;\texttt{>}\\
\overline{\mathit{Exp}} &::= \texttt{( )} \mid \texttt{(}\; \overline{\mathit{Exp}} \;(\texttt{,}\; \overline{\mathit{Exp}})^* \;\texttt{)} \mid \mathit{Var} \mid \mathit{id} \;\texttt{[}\; \mathit{Num} \;\texttt{:}\; \mathit{Num} \;\texttt{]}\\
\mathit{Cond} &::= \mathit{Cond}' \;(\texttt{||}\; \mathit{Cond}')^* \mid \texttt{(}\; \mathit{Cond} \;\texttt{)}\\
\mathit{Cond}' &::= \mathit{Cond}'' \;(\texttt{\&\&}\; \mathit{Cond}'')^* \mid \texttt{(}\; \mathit{Cond}' \;\texttt{)}\\
\mathit{Cond}'' &::= \mathit{Var} \mid \texttt{!}\; \mathit{Var} \mid \texttt{all}\; \overline{\mathit{Exp}} \mid \texttt{(}\; \mathit{Cond}'' \;\texttt{)}\\
\mathit{Num} &::= \mathit{Num} \;(\texttt{+} \mid \texttt{-})\; \mathit{Num}' \mid \mathit{Num}'\\
\mathit{Num}' &::= \mathit{Num}' \;(\texttt{*} \mid \texttt{/} \mid \texttt{\%})\; \mathit{Num}'' \mid \mathit{Num}''\\
\mathit{Num}'' &::= \texttt{(}\; \mathit{Num} \;\texttt{)} \mid \texttt{sqrt (}\; \mathit{Num} \;\texttt{)} \mid \texttt{measure}\; \mathit{Exp} \mid \texttt{(num)}\; \mathit{Prop} \mid n \mid \mathit{id}\\
\mathit{Prop} &::= \mathit{Prop} \;\texttt{||}\; \mathit{Prop}' \mid \mathit{Prop}'\\
\mathit{Prop}' &::= \mathit{Prop}' \;\texttt{\&\&}\; \mathit{Prop}'' \mid \mathit{Prop}''\\
\mathit{Prop}'' &::= \mathit{Num} \;(\texttt{==} \mid \texttt{!=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{>=})\; \mathit{Num} \mid \texttt{!}\; \mathit{Prop}''' \mid \mathit{Prop}'''\\
\mathit{Prop}''' &::= \texttt{(}\; \mathit{Prop} \;\texttt{)} \mid \texttt{true} \mid \texttt{false} \mid \texttt{measure}\; \mathit{Exp}\\
\mathit{Var} &::= \mathit{id} \mid \mathit{id} \;\texttt{[}\; \mathit{Num}\texttt{]}
\end{aligned}
\tag{12}
$$

The reason that I have chosen to put accents over some of the non-terminals is that I will use these below in the type checking rules. When for instance a $\bar{t}$ appears, it is assumed that the syntax tree which it represents can been parsed as a production from the $\overline{\mathit{Type}}$ non-terminal, and so on. I have also made use of regular expression syntax in some cases and thus used the raised symbols $^+$ and * for when there are one or more repetitions or zero or more repetitions respectively and the ? symbol when the occurrence is optional. I have also used the | symbol inside brackets when there are multiple choices. All the lexemes of the language are in a monospaced font, including brackets. The non-monospaced brackets are part of the regular expression syntax. This should not be too confusing as these always have either $^+$, * or ? right after them or include a series of options separated by $|$. Lastly, I have used n and id to represent the lexeme domains of constant numbers and of variable identifiers respectively.

We have already seen much of this syntax in the previous section. One of the features we have not yet seen is the while statement, which takes a single-qubit variable, measures it and initializes it to `true` ($|1\rangle$) at the beginning of each loop. It then only executes the loop body if said variable was measured to be `true`. When the variable is measured to `false` the loop stop (if it has even begun) and the variable is then initialized to `false` ($|0\rangle$).

We also have the assert statement, which helps the users to keep track of types explicitly, and the assume statements, which can be used by expert users to manually terminate variables and to overwrite their types, if they are willing to rely on proofs outside of what the language guarantees via its type system.

The language can be seen to include syntax to handle numbers and propositions used for the control flow of programs. New numbers and propositions can be generated dynamically by measuring data variables, using the `measure` keyword, which also has the effect of consuming said variables. At the same time, new data variables (or anonymous data values) can also be created from numbers and propositions. The expression `new` $P[n]$, where P is a proposition initializes n `bool*` qubits in the `false` ($|0\rangle$) state. The expression `new` N, where N is a number that evaluates to a non-negative n, can be used to initialise a whole `bool`$[n]$ tuple at once according the binary, unsigned representation of n. The `return` N statement is primarily meant for the main function and will fail for all other functions. By letting the program try to evaluate N first, however, this can also serve as a way of letting a function fail with an error message.

A final part of the language we have not yet seen is the `!` operator, which is used to reverse a function. It can only be taken on function variables. This makes sure that no variable have its type changed implicitly by a call to the function. This is true even for variables of local functions as these are not allowed to do so. A reversed function will simply run the statements in reverse from the bottom to the top of the underlying global function. As we will see below, expert users can also declare the reverse algorithm themselves for a global function.

The reader might note that the conditional expressions have been restricted so that all recursion is eliminated from the productions, except for the possibility to add brackets around terms. The brackets can thus only be used to emphasize the precedence order and cannot change the semantics in any way. The reason for this is to make sure that the number of ancilla qubits needed to implement the condition does not explode compared to the size of the condition expression. With this restriction, the circuits implementing the conditional will be straightforward to construct and will need no more than $\lg n$ ancilla qubits, where n is the number of expressions in the outer disjunction. This restriction could also change for a future version, however.

The `all` $\overline{Exp}$ condition is included to open up for the possibility of having if statements with a variable number of condition variables. The expression can then be a tuple of any size, as long as each of the tuple elements still has a Boolean type. For a general tuple $(v_1, \ldots, v_n)$ where the vs are Boolean varaibles, `all` $(v_1, \ldots, v_n)$ is then equivalent to v_1 `&&` $\ldots$ `&&` v_n.

Comments are not taken to be part of the grammar, but should have been removed by the lexer, along with any whitespace between the lexemes once these have been identified. As was seen in the examples previously, the comments follow the syntax of C and languages like it. Lines are thus only parsed until `//` occurs and all text between (and including) a `/*` and the first appearance of `*/` after it is removed as a comment.

Now that the inner grammar is defined, we can define the rest of the programming language by the following grammar.

$$\begin{aligned}
\mathit{Prog} &::= \mathit{Prog}'\ \mathit{Prog} \mid \epsilon \\
\mathit{Prog}' &::= \mathit{FunDec} \mid \mathit{FunDef} \mid \mathit{TypeDef} \mid \mathit{TypeEq} \mid \mathit{NumDef} \mid \mathit{MainDef} \\
\mathit{FunDec} &::= \mathit{id}\ (\texttt{<}\ \mathit{id}\ (\texttt{,}\ \mathit{id})^*\ \texttt{>})?\ (\texttt{::} \mid \texttt{+:})\ \mathit{Type} \\
\mathit{FunDef} &::= \mathit{id}\ \big(\texttt{<}\ \mathit{id}\ (\texttt{,}\ \mathit{id})^*\ \texttt{>}\big)?\ \overline{\mathit{Exp}}^{+}\ \mathit{Stat}\ \big(\mathit{id}\ (\texttt{<}\ \mathit{id}\ (\texttt{,}\ \mathit{id})^*\ \texttt{>})?\ \texttt{!}\ \overline{\mathit{Exp}}^{+}\ \mathit{Stat}\big)? \\
\mathit{TypeDef} &::= \texttt{type}\ \mathit{id}\ (\texttt{<}\ \mathit{id}\ (\texttt{,}\ \mathit{id})^*\ \texttt{>})?\ \texttt{:=}\ \mathit{Exp}\ \texttt{.}\ \overline{\mathit{Type}} \\
\mathit{TypeEq} &::= \texttt{assume}\ \overline{\mathit{Type}}\ \texttt{:=}\ \overline{\mathit{Type}} \\
\mathit{NumDef} &::= \texttt{num}\ \mathit{id}\ \texttt{=}\ \mathit{Num} \\
\mathit{MainDef} &::= \texttt{num main (}\ \big(\texttt{num}\ \mathit{id}\ (\texttt{,}\ \texttt{num}\ \mathit{id})^*\big)?\ \texttt{)}\ \mathit{Stat}
\end{aligned} \tag{13}$$

Here, *Prog* is the non-terminal from which a program is parsed.

We have already seen function declaration, but not seen the `+:` lexeme. This is used to signal that the programmer has allowed him- or herself to make assumptions about the types. This is not a central feature but is just there to allow expert users more control over the program on a low level.

The whole last optional expression for the production of the function definition is also a bonus feature more than a central feature. It gives expert users the opportunity to define the reverse procedure for a function themselves. This can be used when the programmer has defined almost-reversible procedures and wants to use them as subroutines in temp statements. An illuminating but unlikely example is the teleportation circuit[13] in which no quantum coherence is lost even though measurements are used for it. Such a procedure clearly has an inverse procedure but without this extra feature, the programmer would not be able to use it in a temp statement (or to take the inverse directly with the `!` postfix operator). Another example would be a higher-order function to implement error correction around error-prone functions. This is also somewhat of an unlikely example as well, since this would most likely be implemented automatically on a low level for a specific piece of quantum hardware. But it is still a good example to illustrate why the feature might be useful.

For the production of the type definition above, we use an *Exp* non-terminal. Since no data variable can be live in the global scope, the only expressions that can succeed are expressions consisting solely of function variables and number variables. A type can thus not be defined conditioned on any data variables.

The type equation declaration is used to join two types that are defined via different functions but which the (expert) user knows represent the same Hilbert space basis.

The main function is much similar to the other functions but can only take and return numbers, since no data variables can be live in the global scope as mentioned. Since there is no upper bound on the number type by default, the input numbers can potentially be used encode string input as well as numerical input.

5 Type checking rules

In this section, we will go through the type checking rules. There are a lot of important restrictions on the language that are worthwhile to discuss and the type checking rules will state these restrictions.

I will use the well-known notation for rules of inference[14], where the premises are written above an inference line, which are then needed to be fulfilled in order to prove the conclusion on

[13]See for instance Nielsen and Chuang [1] or Green et al. [4].

[14] I was introduced to the notation in a course on Programming Language Design on DIKU. It can also be seen for instance in Yokoyama et al. [14]. An introduction to the notation is found in Harper [15].

the bottom. I will sometimes have several conclusions under the inference line for a rule as a way to reuse premises. In terms of type checking, these rules state the steps needed to check if an expression or statement succeeds, as well as what types can be inferred from them and what changes to the environment is produced if they succeed. Any free variable appearing in a rule will have an implicit universal quantification outside of the whole rule. There is in other words the freedom to search for any object in the relevant domain when wanting to make a rule to succeed to prove a certain conclusion. The domains of the variables are denoted by the symbols used for them. For instance, t will as mentioned be a variable in the domain of terms parsed as $Type$.

The terms on the left-hand side of the turnstile symbol ($\vdash$) represent the hypotheses. In the context of conventional first-order logic, these would be propositional formulas, but here, we allow ourselves to use mathematical terms instead and let the propositions be implicitly understood. We will for instance often use the symbol τ on the left-hand side of the turnstile symbols to denote an environment of variables and their types. The implicit proposition is then that τ is the environment of the program at the point of the expressions or statements appearing on the right-hand side of the turnstile symbol. For expressions, we will also need a hypothesis about what variables are already used as input parameters to the relevant function call in order to prevent aliased parameters. For this we will use a set typically denoted by γ and write the full hypothesis as τ, γ. We will see how this works shortly.

As I have come to understand, it might not be conventional to formulate inference rules the way that I have formulated them below as they leave a lot of mathematical derivations needed to use the rules as being implicit.[15] The rules below should not be seen collectively as a direct algorithm for making the type checking, but should instead be seen just as outlining the type checking algorithm by making up the most important axioms of a mathematical model where the true propositions define all the programs that are valid.

I will violate the conventions of first-order logic a bit, as I will often put a quantification around a whole premise, or even several premises. Since these are *sequents* and since sequents are not typically regarded as propositional formulas themselves, as I understand, this is unconventional. This will, however, be useful when the hypotheses need to depend on the parameter of the quantifier. If we wanted to reformulate this in more conventional first-order logic, all we would need to do is to pull the hypothesis into the formula on the right-hand side of the turnstile symbol, i.e. the consequent, and write it up explicitly as a propositional formula. This would then allow us to put the quantification around the resulting consequent and the formula would thus no longer violate the standard conventions.

When we get to while loops and measurements, I will violate the conventions even further by writing $(\mathcal{U}_a \to \mathcal{U}_a)$ as a hypothesis, which states that the execution of the statement takes the universe from one wave function and into a new (collapsed) wave function. It might seem a bit far-fetched at first, but it will useful to keep track of what happens for these statements and expression. If we were to be consistent, however, we would then need to carry this kind of hypothesis around for all rules where measurements can happen. Instead, we simply let this be implicit for all other rules and only use the notation for the most relevant ones.

Note that since measurements can happen during the program, it would not make sense to require all these type checks to happen statically. The same applies when the generic function get to complicated. For simplicity, we can just assume that all type checks happen dynamically when thinking about the rules.

[15]According to my supervisor, it is more common to have rules where you can directly build an inference tree from them by matching premises directly to conclusions. My rules are often too complicated to be able to do so directly.

5.1 Expressions

There is a lot we need to ensure in order to check the expressions in this language. This is due to the fact that we have dynamic types as well as the fact that we need to prevent aliased input for function calls. In the following, I will first go through the rules surrounding function calls since these are central to the language. This will bring us through much of the new notation which I have needed to introduce in order to state the type checking rules compactly. Afterwards, we should therefore be able to go through the rest of the rules more quickly.

5.1.1 Function calls

Let us begin with some examples of what need to by checked for function calls. First of all, we need to make sure that variables are not repeated in a call. A simple example in this regard would be `Tof a b a`, where `Tof` has the same type as above, namely `bool -> bool -> bool* -> ()`. If we assume that all the variables have type `bool*` accoriding to the environment, τ, the expression will fail at the leftmost `a`. The way this happens in the procedure described by the type checking rule is by first evaluating $e_2\, e_1$ with $e_2 =$ (`Tof a b`) and $e_1 =$ `a`. Since e_1 is based on `a` we add `a` to the γ set. This set comes in as empty since we assume that there are no more input expressions to the right of `a`. $e_2 =$ (`Tof a b`) is now checked with an incoming γ equal to $\{\texttt{a}\}$. Our expression is again checked as $e_2'\, e_1'$, now with $e_2' =$ (`Tof a`) and $e_1' =$ `b`. Since `b` is not in $\gamma = \{\texttt{a}\}$, this also succeeds and `b` is then added to γ. Lastly, $e_2''\, e_1'' =$ `Tof a` is evaluated but now, `a` is already in $\gamma = \{\texttt{a}, \texttt{b}\}$, which prevents the rule from succeeding. A similar example where `a` is replaced here by a variable `c` of the same type, the rule would succeed, as `c` would not be in $\gamma = \{\texttt{a}, \texttt{b}\}$. In this case, we would reach `Tof` and its type would be looked up in the incoming environment, τ. Going backwards out of the expression, the types obtained at each step is then matched with the function on the left, and since all three variables have a valid type for the call, the type check of the outer function call would succeed.

If a certain input is a tuple of variables, then all variables are added to γ. But, if an input expression is a function call then no variables are added to γ. The reason for this is that functions can only return anonymous variables and their return value can therefore never not be aliased with anything else. Even if a function uses the input variable as part of its return value, then that function would have to consume said variable, giving it the unit type, in which case any further appearance of the variable in the overall function would fail anyway due to a type mismatch. For the same reason, if the input is a mixed tuple of both variables and function calls, only the variables are added to γ.

Whenever we have a nested function call inside an outer call, we need a fresh, empty γ to check the nested call. The reason for this is that such nested functions are always evaluated before the function call in which they are nested. For a nested function, we therefore only need to ensure that is does contain any repeated variables within itself and do not need to consider its context.

We also need to make sure that once a variable is used in a function call, we do not change its state further in expressions to the left of it, as this would make for some cryptic semantics. This is ensured by the type checking rule by removing the `*` qualifier of the variable, if there is one, in context of the function to the left of it. To clarify, the `*` qualifier is only removed from said context and is thus still kept outside of the function call. As an example, a function call e_2 `(a, b)` will checked by removing any `*` qualifiers of `a` and `b` in the context of the type check for e_2. In the rule below, we use $[\lfloor e_1 \rfloor : \mathbf{c}]$ to implement the removal of the `*` qualifiers, where $\lfloor e_1 \rfloor$ is the set of variables of e_1 with all function calls removed and where $\mathbf{c}$ can be seen is a inverse qualifier of `*` (the $\mathbf{c}$ symbol is chosen since the variables can then only be used as *control* input). The notation of which will be explained further below.

With all this in mind, we are now ready to state the type checking rule for function calls. It can be states as

$$\frac{\begin{array}{c}\tau, \emptyset \vdash e_1 \hookrightarrow \Delta\tau_1 \\ \tau \vdash \gamma' = \lfloor e_1 \rfloor \diamond \gamma \quad \vdash \tau' = [\,\lfloor e_1 \rfloor : \mathbf{c}\,] \circ \Delta\tau_1 \circ \tau \\ \tau', \gamma' \vdash\ e_2 :\ t_1 \texttt{ -> } t_2 + \Delta g \ \hookrightarrow\ \Delta\tau_2 \\ \tau \vdash t_1 \bullet e_1 \hookrightarrow \Delta\tau_3 \quad \vdash \Delta\tau = \Delta g \circ \Delta\tau_3 \circ \Delta\tau_2 \circ \Delta\tau_1\end{array}}{\tau, \gamma \vdash\ e_2\ e_1 : t_2 \hookrightarrow \Delta\tau}. \tag{14}$$

Looking at the conclusion below the inference line, we see that the rule is used to check a function call on the form $e_2\ e_1$. The symbol e is always used, when the domain of the variable is the domain of expressions, i.e. when they are parsed from the *Exp* non-terminal of the grammar. A proposition on the form $e : t$ is used to say that e has the type t. Variables that uses the symbol t comes, as mentioned, from the *Type* non-terminal. The hooked arrow ($\hookrightarrow$) is used to state that the expression produces a change to the environment, which is sent along to the next expression (from right to left, since this is the order of evaluation), and also sent on to the next statement of the program in the end. A proposition on the form $e \hookrightarrow \Delta\tau$ states that e produces a change to the environment $\Delta\tau$. Since we will often need to use both $:$ and $\hookrightarrow$ at the same time, I have let $e : t \hookrightarrow \Delta\tau$ be a shorthand for writing $e : t \wedge e \hookrightarrow \Delta\tau$. On the left-hand side of the turnstile symbol we have τ, which is the environment coming into $e_2\ e_1$, and we have γ, which is the set of data variables already used to the left of $e_2\ e_1$ if $e_2\ e_1$ is itself the left-hand expression (i.e. the function) of a function call. The first premise at the top then states that e_1 produces the environment $\Delta\tau_1$. As was mentioned before, we have to give it a fresh, empty γ, which is why the γ in the hypothesis is set as $\emptyset$. The next premise is used to check that e_1 is not based on any variable found in γ. The term $\lfloor e_1 \rfloor$ is thus the set of all variables in e_1 when removing all function calls, as we will see below. I have used the diamond symbol ($\diamond$) to represent a union that is only partially defined when the two sets are disjoint. If $\gamma' = \lfloor e_1 \rfloor \diamond \gamma$ can be proved, it therefore means that $\lfloor e_1 \rfloor$ and γ are disjoint. We will also assume that the diamond operator sees any aliased variables in the relevant τ as being equal. In the third premise, we see the symbol $\circ$ for the first time, which I have taken to be an operator that adds an environment change on the left to an environment on the right. These changes will generally be written as variables using $\Delta\tau$, or Δg when it comes to garbage (or measurements), and for specific environment changes, they will be as proposition-like terms enclosed in square brackets such as with $[\,\lfloor e_1 \rfloor : \mathbf{c}\,]$. An operation $\Delta\tau \circ \tau$ thus adds $\Delta\tau$ to the environment τ to produce a new environment. We will also be using $\circ$ as an associative operator between environment changes, as seen already in the last premise of this rule. It is then defined by $(\Delta\tau_2 \circ \Delta\tau_1) \circ \tau = \Delta\tau_2 \circ \Delta\tau_1 \circ \tau$, which of course again is equal to $\Delta\tau_1 \circ (\Delta\tau_2 \circ \tau)$. The third premise thus defines τ' to be the environment with which we have to check e_2. It first gets the type changes of e_1 and then also gets the environment change that all variables e_1 is based on loses their `*` qualifier (but only in the context of e_2). The fourth premise then uses the new τ' and γ' in its hypothesis to make the type check for e_2 in this new context, and to check that its resulting type is of the form $e_2 :\ t_1 \texttt{ -> } t_2 + \Delta g$. Here, Δg is taken to include garbage if the function in question has the `+ garbage` overall qualifier and is at the same time taken to include a measurement if it has `-->` as its last arrow. This only applies when t_2 is not itself a function. Otherwise Δg will include no garbage or measurement. We thus have

$$\begin{aligned} t_1 \texttt{ -> } t_2 \texttt{ -> } t_3 &= t_1 \texttt{ -> } t_2 \texttt{ -> } t_3 + [\,], \\ t_1 \texttt{ --> } t_2 + \Delta g &= t_1 \texttt{ -> } t_2 + ([\text{measurement}] \circ \Delta g). \end{aligned} \tag{15}$$

Here, $[\,]$ is an empty environment change and [measurement] records that a measurement has taken place in the environment. Additionally we have here that Δg is set as [garbage] in the end if the `+ garbage` qualifier is there, and it is set to $[\,]$ if it is not.

We have yet to elaborate on the last two premises of the above rule. The second last premise is used to generate the environment change advertised by the input type. For instance, if $t_1 =$

(xspin $*$) bool $*$ and e_1 is a variable v with an infered type of v : bool $*$, then $\Delta\tau_3$ should be $[v : \text{xspin}\,*]$. I therefore take the $\bullet$ to be an operator, with its own rules stated below, that checks whether the expression on its right-hand side matches the input type on its left-hand side, and produces the relevant environment change upon success. I will state these rules shortly. The last premise then finally defines the full environment change $\Delta\tau$ for the function call to send onwards. One last thing we have ensure is that the return value is given the `*` qualifier in case the return type is a data type (i.e. a non-function type). To ensure this we simply state that the `*` qualifier should initially be granted to the return type of all functions. We can state this by having

$$t_1 \texttt{ -> } \bar{t}_2 + \Delta g = t_1 \texttt{ -> } \bar{t}_2 * + \Delta g. \tag{16}$$

Here, $\bar{t}$, as mentioned, always represent a type parsed from the $\overline{Type}$ non-terminal. By solving the problem this way, it technically means that there are two options for rule (14). Since it is always the optimal choice to grant to type the `*` qualifier, however, we are free to assume that this always happens in the context of constructing a proof. But when implementing a compiler, one needs to remember to add the qualifier every time it is possible in order to avoid dead ends.

This almost completes the explanation for rule (14). We still need to define $\lfloor e \rfloor$ and $t \bullet e$, however, as well as explain what it means when a whole set like $\lfloor e_1 \rfloor$ is given a qualifier (like $\mathbf{c}$).

Let us start with $\lfloor e \rfloor$. We will also need a similar operation, $\|e\|$, to get all variables in e, including the ones appearing in function calls. The can therefore define this at the same time. We have to use rules, and not just equations, to define $\lfloor e \rfloor$ and $\|e\|$ since their results depend on what data variables are defined in τ. We can define the two operation by the following set of rules.

$$\frac{}{\tau \vdash\ \lfloor e_1\, e_2 \rfloor = \lfloor e_2 \texttt{ . } e_1 \rfloor = \lfloor e \texttt{ .} \rfloor = \lfloor e \texttt{ !} \rfloor = \lfloor \texttt{new}\ p \rfloor = \lfloor \texttt{new}\ p[N] \rfloor = \lfloor \texttt{new}\ N \rfloor = \emptyset},$$
$$\tau \vdash\ \|\texttt{new}\ p\| = \|\texttt{new}\ p[N]\| = \|\texttt{new}\ N\| = \emptyset$$
$$\tau \vdash\ \|e \texttt{ .}\| = \|e \texttt{ !}\| = \|e\|$$
$$\tau \vdash \|e_1\, e_2\| = \|e_2 \texttt{ . } e_1\| = \|e_2\| \cup \|e_1\|$$

$$\frac{\tau \vdash v : \bar{t}}{\tau \vdash \lfloor v \rfloor = \{v\} \quad \tau \vdash \|v\| = \{v\}}, \qquad \frac{\tau \vdash id : t_1 \texttt{ -> } t_2 + \Delta g}{\tau \vdash \lfloor id \rfloor = \emptyset \quad \tau \vdash \|id\| = \emptyset}, \tag{17}$$

$$\frac{\begin{array}{c}\forall_{i \in \{1,\ldots,n\}} \left[\, \tau \vdash \lfloor e_i \rfloor = \gamma_i \,\right] \\ \tau \vdash \gamma = \gamma_n \diamond \cdots \diamond \gamma_1\end{array}}{\tau \vdash \lfloor (e_n, \ldots, e_1) \rfloor = \gamma}, \qquad \frac{\begin{array}{c}\forall_{i \in \{1,\ldots,n\}} \left[\, \tau \vdash \|e_i\| = \nu_i \,\right] \\ \tau \vdash \nu = \nu_n \cup \cdots \cup \nu_1\end{array}}{\tau \vdash \|(e_n, \ldots, e_1)\| = \nu}.$$

These rules define $\lfloor e \rfloor$ and $\|e\|$ for almost all expressions (they will also be defined for $id[N : M]$ later on). We see that the symbol v is used in the second rule and that id is used in the third rule. v represents a term parsed as the Var non-terminal and id still represents the domain of identifiers. Since we are only collecting data variables, the third rules states the the function variables are not included. We see that the fourth and fifth rule have the quantification around the premise, which we have already talked about. The line with the quantification can be exchanged for n premises where i runs through the relevant range. We can think of it here almost like a for loop. Note that we have used γ as the symbol for the $\lfloor e \rfloor$ sets and used ν for the (more inclusive) $\|e\|$ sets. We will stick to this trend in the following. In the conclusion of the two rules (the fourth and fifth rule), we also see $(e_n, \ldots, e_1)$. Here, n does not need to be strictly larger then 1 but can also be equal to 1, and even 0, in which cases the tuple is just equal to (e_1) or $()$ respectively. Note that I have used the $\diamond$ operators again for the fourth rule, which means that $\lfloor e \rfloor$ is not defined if e is a tuple based on repeated variables.

The rules for the $\bullet$ operator are more interesting. For data type inputs, they are given by

$$\begin{gathered}\frac{\tau \vdash e : \bar{t}}{\tau \vdash \bar{t} \bullet e \hookrightarrow [\,]}, \qquad \frac{\begin{gathered}\tau \vdash \|e\| = \nu \\ \forall_{v \in \nu} \left[\tau \vdash \ v : \sim \right] \quad \tau \vdash t \bullet e \hookrightarrow \Delta\tau\end{gathered}}{\tau \vdash \sim t \bullet e \hookrightarrow [\nu : \sim] \circ \Delta\tau}, \\ \frac{\tau \vdash \lfloor e \rfloor = \gamma \qquad \forall_{v \in \gamma} \left[\tau \vdash \ v : * \right] \qquad \tau \vdash e : \bar{t}^{in}}{\begin{gathered}\tau \vdash (\bar{t}^{out} *) \, \bar{t}^{in} * \bullet e \hookrightarrow [e : \bar{t}^{out} *] \quad \tau \vdash (\,) \, \bar{t}^{in} * \bullet e \hookrightarrow [\gamma : (\,)] \\ \tau \vdash \text{span}(\bar{t}^{in}) * \bullet e \hookrightarrow [e : \text{span}(\bar{t}^{in}) *] \quad \tau \vdash (\,) \, \text{span}(\bar{t}^{in}) * \bullet e \hookrightarrow [\gamma : (\,)]\end{gathered}}.\end{gathered} \tag{18}$$

The first rule here states that no changes happen to the environment if the input type is just a $\bar{t}$ type with no $*$ after it, as long as e can be proven to have that type. An important thing to state here, is that we assume, that $e : \bar{t}*$ always also infer $e : \bar{t}$. The same applies for $\sim$, such that $e : \sim \bar{t}$ also infers $e : \bar{t}$. Additionally, when e is a tuple, we can also remove any inner $*$ or $\sim$ qualifiers in the same way. We will use this later on as well. The second rule shows what needs to be checked when the input type has the $\sim$ symbol in front. The $\sim$ qualifier, when it is attached to the type of a data variables, tells us that we are allowed to make operations for said data variable even if the overall phase gained by the operation depend on the state of the variable. Since this exact thing might be case when the input type of a function has the $\sim$ symbol in front, we need to check that variables in e, i.e. in the set ν, all have the $\sim$ qualifier attached to them (which they can only get back from the type signature of the function being defined). Once this is checked, the third premise checks that the type check also succeeds without the $\sim$ and passes the relevant $\Delta\tau$ on. I have included $[\nu : \sim]$ in the conclusion to make sure, that none of the variables in ν loses their $\sim$ qualifier. The third rule takes care of the rest, whenever there is a $*$ at the end of an input type, and the variable is either consumed or have its type changed. In case their is no explicit outgoing type, i.e. no brackets to the left, this is the same as if the same type is repeated in the brackets. We thus have for input types that

$$\bar{t} * = (\bar{t} *) \, \bar{t} * \tag{19}$$

for any $\bar{t}$. This case is therefore also treated by the two conclusions on the left of the rule. The two conclusions on the right shows that all variables that e is based on are consumed. This is what is denoted by $[\gamma : (\,)]$. Note that we again have used $v : *$ similarly to how $v : \sim$ was used in the second rule, namely as to say that v has the $*$ qualifier without concerning ourselves about the rest of its type. The first line of conclusions in rule three is trivial enough but the conclusions on the second line need some more explanation. These define what happens when the input type is spanning. We see that regardless of whether $\bar{t}^{in}$ was spanning to begin with, the rule will succeed if the input type is a spanning type, as long as it is equal to span($\bar{t}^{in}$), i.e. as long as it has the same underlying tuple structure as $\bar{t}^{in}$. As mentioned, I use the word spanning, since the types can be thought of as subsets of different Hilbert spaces. The important thing in our case, however, is not what we call it, but how it is defined. It is defined by

$$\begin{aligned} \text{span}((\,)) &= (\,), \\ \text{span}(\bar{t} *) &= \text{span}(\bar{t}), \\ \text{span}(\sim t) &= \text{span}(t), \\ \text{span}(\text{new } t) &= \text{span}(t), \\ \text{span}((\bar{t}^{out} *) \, \bar{t}^{in} *) &= \text{span}(\bar{t}^{in}), \\ \text{span}((\,) \, \bar{t}^{in} *) &= (\,) \, \text{span}(\bar{t}^{in}), \\ \text{span}((t_1, \ldots, t_n)) &= (\text{span}(t_1), \ldots, \text{span}(t_n)), \end{aligned} \tag{20}$$

and by its action on the basic types, which is given by

$$\text{span}(\text{bool}) = \text{span}(\text{xspin}) = \text{span}(\text{yspin}) = \text{span}(\text{qbit}) = \text{qbit}. \tag{21}$$

We will get back to the 'new' qualifier later. Furthermore, we have that for any function, we can infer for a more relaxed function type of that function. The function type is relaxed by changing all the involved types with their corresponding spanning type, except maybe for some initial, unchanged input types. We can define this by the rule,

$$\frac{\tau \vdash e : t_1 \texttt{ -> } t_2 + \Delta g \hookrightarrow [\,]}{\tau \vdash e : \text{ Span}(t_1 \texttt{ -> } t_2 + \Delta g)}, \tag{22}$$

where the function Span is defined by relations

$$\begin{aligned} \text{Span}(t_1 \texttt{ -> } t_2 + \Delta g) &= \text{span}(t_1) * \texttt{ -> } \text{Span}(t_2 + \Delta g), \\ \text{Span}(\bar{t} + \Delta g) &= \text{span}(\bar{t}) + \Delta g. \end{aligned} \tag{23}$$

If we start by looking at what this rule says when e is a function variable, we can see that the rule allows us to change all input types to their spanning types as well as the return type. When type checking a function variable, no environment change is produced, which is why the rule succeed in this case. We will not be stating a rule for the fact that function variables produce no environment change. We will simply take this as a basic assumption. With the way we have implemented the function call, where non-empty environment changes are generated for each input type with a `*` qualifier, and empty type changes are generated only for input types without it, we see that the rule also works when e is a partial function call that has not changed any of its input. As an example, we can infer the type `qbit* -> ()` for the expression `CNOT a`, where `a` is a Boolean variable and `CNOT` is defined as above. If `a` is not a Boolean, we instead have to infer the type `qbit* -> qbit* -> ()` for `CNOT` itself to be able to make a successful type check. For functions with several input types without the `*` qualifier at the end, we can see that the rule also works, if we assume $[\,] \circ [\,] = [\,]$, which we do.

It is worth noticing that the above rule can lead to several outcomes when type checking a function call where some of the input variables have the wrong basis type. The type checking should always try to change as few input types to their spanning types as possible, since this would be optimal in terms of checking a program. Note that if we implement the type checking for function calls in the same way as rule (14) suggests, the type matching happens at the end when going back out of the recursive checks. The type matching thus happens left to right at the end. Once the first input occurs that has a wrong input basis type, we therefore only need to potentially backtrack a little to get to the first time an environment change was produced and then use rule (22) here to change the rest of the types from there.

This is what makes the functions defined by a program useful for quantum algorithms, namely the fact that they can be taken on all kind of input states in the Hilbert space on not just the non-superposition state of the input bases. Whenever an input of a function call has the same underlying tuple structure but not not necessarily match the correct input basis types, we can use the function's more relaxed type according to rules (22) and eq. (23) instead. As seen in the last line of conclusions in the third rule of rules (18), the spanning input types will change the input variable(s) to its/their spanning type. And as seen in relations (23), the returned type will also be a spanning type.

We have now managed to go through the rules regarding the type changes generated by function calls when it comes to data variables. Before we state the rules for function types as input, let us take a moment to define some relations for the environment changes. We have already used some of this notation as we have seen environment changes on the form $[\nu : \odot]$, where ν is some set and $\odot$ is some type qualifier or the unit type. We have also seen an environment change on the form

$[e : t]$ in rules (18) (third rule), where e might be a tuple expression. The relations regarding the former case can be stated as

$$\begin{aligned}
[\{v_1,\ldots,v_n\} : *\,] &= [v_1 : *\,] \circ \cdots \circ [v_n : *\,] \circ [\,],\\
[\{v_1,\ldots,v_n\} : \mathbf{c}\,] &= [v_1 : \mathbf{c}\,] \circ \cdots \circ [v_n : \mathbf{c}\,] \circ [\,],\\
[\{v_1,\ldots,v_n\} : \sim] &= [v_1 : \sim] \circ \cdots \circ [v_n : \sim] \circ [\,],\\
[\{v_1,\ldots,v_n\} : \nsim] &= [v_1 : \nsim] \circ \cdots \circ [v_n : \nsim] \circ [\,],\\
[\{v_1,\ldots,v_n\} : (\,)] &= [v_1 : (\,)] \circ \cdots \circ [v_n : (\,)] \circ [\,],\\
[\{v_1,\ldots,v_n\} : \text{new}\,] &= [v_1 : \text{new}\,] \circ \cdots \circ [v_n : \text{new}\,] \circ [\,],\\
[\{v_1,\ldots,v_n\} : \text{none}\,] &= [v_1 : \text{none}\,] \circ \cdots \circ [v_n : \text{none}\,] \circ [\,],\\
[\{v_1,\ldots,v_n\} : \text{undef}\,] &= [v_1 : \text{undef}\,] \circ \cdots \circ [v_n : \text{undef}\,] \circ [\,],\\
[\{v_1,\ldots,v_n\} : \text{spanning}\,] &= [v_1 : \text{spanning}\,] \circ \cdots \circ [v_n : \text{spanning}\,] \circ [\,].
\end{aligned} \tag{24}$$

These relations show all the type qualifiers we will use. So far, we have only seen examples with **c**, $\sim$ and ( ). Then we have $*$, which is similar to $\sim$ and give the variables in question the * qualifier. We also have $\nsim$, which removes the $\sim$ qualifier just like **c** removes the * qualifier. The qualifier 'new' is used to specify that a variable is uninitialized, despite the recorded type that goes with it. The qualifier 'none' is used for variables when an appearance of the variable should result in a failure in the context. For statements, we also use 'none' by the way, but this is to state that the expression has no return statement in it. The qualifier 'undef' is used to remove the variable from the a scope, which means it can be declared again. Lastly, 'spanning' simply turns the variable's types into their corresponding spanning types. Note that we have included an empty environment change, $[\,]$, at the end to take care of case when $\{v_1,\ldots,v_n\}$ is empty (and $n = 0$ in other words).

When we have an expression on the form $[e : t]$, we have to give all the variables of e the corresponding type of t (having the same tuple structure as e), except that we ignore all function calls. This is once again due to the fact that the returned types will be anonymous and we will therefore not need to consider them in terms of the environment. The notation can thus be defined by

$$\begin{aligned}
[(e_1,\ldots,e_n) : (t_1,\ldots,t_n)\,] &= [e_1 : t_1\,] \circ \cdots \circ [e_n : t_n\,] \circ [\,],\\
[e_2 \,.\, e_1 : t\,] &= [e_2\ e_1 : t\,] = [\,]
\end{aligned} \tag{25}$$

in conjunction with the relations

$$\begin{aligned}
(t_1 *, \ldots, t_n *) &= (t_1,\ldots,t_n) *,\\
(\sim t_1, \ldots, \sim t_n) &= \sim (t_1,\ldots,t_n),\\
(\,) &= (\,) * .
\end{aligned} \tag{26}$$

This is what was used in the third rule of rules (18) to distribute $\bar{t}^{\,out}$ and $\text{span}(\bar{t}^{\,in})$ in e when this is a tuple.

With all this out of the way, let us now define the rules for the $\bullet$ operator when the input type is a function type. These rules can be stated as

$$\frac{\tau \vdash id : t_1 \texttt{->}\ t_2 + \Delta g}{\tau \vdash (t_1 \texttt{->}\ t_2 + \Delta g) \bullet id \hookrightarrow [\,]}, \qquad \frac{\forall_{i\in\{1,\ldots,m\}} \left[\, \tau \vdash N_i := n_i \,\right] \quad \tau \vdash id \texttt{<} n_1,\ldots,n_m \texttt{>} : t_1 \texttt{->}\ t_2 + \Delta g}{\tau \vdash (t_1 \texttt{->}\ t_2 + \Delta g) \bullet id \texttt{<} N_1,\ldots,N_m \texttt{>} \hookrightarrow [\,]}. \tag{27}$$

The important thing to note here is the fact that the $\bullet$ operator is only defined when the input is a function variable id or a generic function variable with numbers attached to it, which has the general form $id \texttt{<} N_1,\ldots,N_m \texttt{>}$. The first rule then simply states that if the function variable can be inferred to have the correct type then the $\bullet$ operator succeeds with no environment changes.

For the second rule, we see that it includes some new notation. First of all, I use capital letters N, M, K and L for Num terms and use lower-case letters n, m, k and l only for constant numbers. I define proposition on the form $N := n$ to state that N evaluates to n (in the context of the relevant environment). N is evaluated by looking up the values recorded for any variable numbers in it and by computing the result of any arithmetic operation. As we will see later, I will also use $[id := n]$ as notation for recording that id has the value n. With this in mind we see that the rule first computes the values for each N_i and then look up the type of $id\texttt{<}n_1, \dots, n_m\texttt{>}$ in τ. This might by the way be checked by compiling the generic function on demand, if the compiler was not able to do this statically. Similarly to the first rule, the $\bullet$ operator then succeeds with no environment changes, if said type matches the incoming one from the $\bullet$ operation.

We have now finally explained all the notation surrounding rule (14) for function calls. In the rest of this section, we will state some rules that are relevant for function call, namely the rule for what is needed when the input is a tuple, and also the rules for what happens when the `.` or the `!` operators are used.

The rule for type checking tuples follow much the same pattern as rule (14). Whereas we for function calls needed to pass γ to the left-most (function) expression, and needed to overwrite the $*$ qualifiers in its context, we now just need to do the same for each expression from right to left in the tuple. We thus need to pass an expanding γ along from right to left to make sure that the tuple is not based on repeated variables. Similarly as with the function call, we also need to make sure that variables do not have their types changed to the right of them in the tuple once they are used. The type checking rule for tuples can thus be stated as

$$\frac{\begin{array}{c} \vdash \gamma_0 = \gamma \quad \vdash \tau_0 = \tau \\ \forall_{i \in \{1,\dots,n\}} \left[\, \tau_{i-1}, \gamma_{i-1} \vdash e_i : t_i \hookrightarrow \Delta\tau_i \,\right] \\ \forall_{i \in \{1,\dots,n\}} \left[\, \tau_{i-1} \vdash \ \gamma_i = \lfloor e_i \rfloor \diamond \gamma_{i-1} \ \wedge \ \tau_i = [\lfloor e_i \rfloor : \mathbf{c}] \circ \Delta\tau_i \circ \tau_{i-1} \,\right] \\ \vdash \Delta\tau = \Delta\tau_n \circ \cdots \circ \Delta\tau_1 \circ [\,] \end{array}}{\tau, \gamma \vdash (e_n, \dots, e_1) : (t_n, \dots, t_1) \hookrightarrow \Delta\tau}. \tag{28}$$

Here, we have again used the universal quantifiers to implement what can be regarded as a for loop. Note that this "for loop" includes both the second and the third line since the first line has variables that are defined in "loop iterations" of the third line (and vice versa). The second line defines t_i and $\Delta\tau_i$ for each i in the context of τ_{i-1} and γ_{i-1} and the third line shows how each τ_i and γ_i is generated. Again the partially defined $\diamond$ ensures that all $\lfloor e_i \rfloor$ sets are disjoint. Note how each $\Delta\tau_i$ is used to decide the new environment of the expressions the the left of where they are coming from. When defining the full $\Delta\tau$ on the last premise line, I have again made sure to include a $[\,]$ at the end in case $n = 0$ and the tuples are empty. When the tuple includes only expression and is equal to just (e_1), the expression should not be evaluated as a tuple but should be evaluated and type checked in the same way as e_1 without the brackets. We can ensure this simply by stating the relation that

$$(\bar{t}) = \bar{t}. \tag{29}$$

This relation also applies in general for types.

We can now move on to the dot operator. We need both a rule to equate $e_2 \,.\, e_1$ with $(e_2 \,.)\, e_1$ and a rule for what happens for expressions on the form $(e \,.)$. These rules can be stated as

$$\frac{\tau, \gamma \vdash \ (e_2 \,.)\ e_1 : t \hookrightarrow \Delta\tau}{\tau, \gamma \vdash \ e_2 \,.\, e_1 : t \hookrightarrow \Delta\tau}, \qquad \frac{\begin{array}{c} \tau, \gamma \vdash \ e : t \hookrightarrow \Delta\tau \\ \vdash t' = \text{returnOutput}(\text{uncurry}(t)) \end{array}}{\tau, \gamma \vdash e \ . \ : t' \ \hookrightarrow \ \Delta\tau}. \tag{30}$$

The first rule tell us that we can use the result coming from type checking $(e_2 \,.)\, e_1$ when we need to type check $e_2 \,.\, e_1$. The second rule tells us, that we can simple type check e as if the dot operator was not there and then change the resulting type afterwards. To change the type, I first use a function, 'uncurry', to uncurry the function type and then a function, 'returnOutput', to move the uncurried input tuple to the left of the return arrow and consuming it on the left by adding ( ) to the type. The function 'uncurry' can be defined by

$$\begin{gathered}\text{uncurry}\big((t_1^{out}*)\, t_1^{in}* \texttt{->} \dots \texttt{->} (t_n^{out}*)\, t_n^{in}* \texttt{->} \bar{t}^{ret} + \Delta g\big) = \\ \big((t_1^{out},\dots,t_n^{out})*\big)\,(t_1^{in},\dots,t_n^{in})* \texttt{->} \bar{t}^{ret} + \Delta g, \\ \text{uncurry}\big(\sim(t_1^{out}*)\, t_1^{in}* \texttt{->} \dots \texttt{->} \sim(t_n^{out}*)\, t_n^{in}* \texttt{->} \bar{t}^{ret} + \Delta g\big) = \\ \big(\sim(t_1^{out},\dots,t_n^{out})*\big)\,(t_1^{in},\dots,t_n^{in})* \texttt{->} \bar{t}^{ret} + \Delta g, \\ \text{uncurry}\big(t_1 \texttt{->} \dots \texttt{->} \sim t_i \texttt{->} \dots \texttt{->} t_n \texttt{->} \bar{t}^{ret} + \Delta g\big) = \\ \text{uncurry}\big(\sim t_1 \texttt{->} \dots \texttt{->} \sim t_i \texttt{->} \dots \texttt{->} \sim t_n \texttt{->} \bar{t}^{ret} + \Delta g\big).\end{gathered} \tag{31}$$

Here, we have also made sure with the second and third equation that $\sim$ are propagated out and kept on the whole input type, if it appears in just one place (or more). When using the dot operator me might therefore loose some information about on what input the phase change depend. The i in the last equation is thus assumed to be anything in the range $1 \leq i \leq n$. The next function, returnOutput, has a more simple definition. It is defined by

$$\begin{gathered}\text{returnOutput}\big((t^{out}*)\, t^{in}* \texttt{->} (\,) + \Delta g\big) = (\,)\, t^{in}* \texttt{->} t^{out} + \Delta g, \\ \text{returnOutput}\big(\sim(t^{out}*)\, t^{in}* \texttt{->} (\,) + \Delta g\big) = \sim (\,)\, t^{in}* \texttt{->} t^{out} + \Delta g.\end{gathered} \tag{32}$$

With these definitions in place, we now finally have defined the correct way to handle the dot operator.

For the `!` operator, we can define it in much the same way as the dot operator. First we define a rule to state that we can again just get the type of e first and then reverse this type afterward. The rule can be stated as

$$\frac{\begin{gathered}\tau \vdash e : t \\ \vdash t' = \text{reverse}(t)\end{gathered}}{\tau \vdash e\,\texttt{!} : t' \hookrightarrow \Delta\tau}. \tag{33}$$

Even though the `!` operator can only be taken on function variables, we do not need to account for this here as this is already taken care of by the grammar. We define the reverse function used by the following when there is no return type of the function is the unit type. We have

$$\begin{gathered}\text{reverse}\big((t_1^{out}*)\, t_1^{in}* \texttt{->} \dots \texttt{->} (t_n^{out}*)\, t_n^{in}* \texttt{->} (\,) + [\,]\big) = \\ (t_1^{in}*)\, t_1^{out}* \texttt{->} \dots \texttt{->} (t_n^{in}*)\, t_n^{out}* \texttt{->} (\,) + [\,].\end{gathered} \tag{34}$$

Note that Δg has to be equal to $[\,]$ in order for the reverse to be defined. We can even add a rule to reverse functions with a non-unit return type and with exactly one consumed input type. For this we can use there following rule. We have

$$\frac{\tau \vdash \text{span}(\bar{t}_i) = \text{span}(\bar{t}^{ret})}{\begin{gathered}\tau \vdash \text{reverse}\big((t_1^{out}*)\, t_1^{in}* \texttt{->} \dots \texttt{->} (\,)\, \bar{t}_i* \texttt{->} \dots \texttt{->} (t_n^{out}*)\, t_n^{in}* \texttt{->} \bar{t}^{ret} + [\,]\big) = \\ (t_1^{in}*)\, t_1^{out}* \texttt{->} \dots \texttt{->} (\,)\, \bar{t}^{ret}* \texttt{->} \dots \texttt{->} (t_n^{in}*)\, t_n^{out}* \texttt{->} \bar{t}_i + [\,]\end{gathered}}. \tag{35}$$

Note how we have made sure from this rule, that the consumed input goes entirely into creating the return value since they have the spanning type, and since there is no garbage created and no measurement happening.

I have chosen not to state explicitly what happens when some input has the $\sim$ qualifier in the above rules and equations for the reverse function. Since we have already seen what happens for the dot operator, I will just state that a similar thing happens for reverse. When $\sim$ appears on any of the input types, we thus have to distribute it on all other input types as well when reversing the function.

5.1.2 Other expressions

We will now go through the remaining forms of expressions of the language. These mainly have to with the numbers and propositions of the language. There is also the `->` expressions which records aliases for variables. We will start by stating the rules for such `->` expressions.

The `->` can only be used between two variables or between two variable tuples of the same structure. We can define this by the following two rules. The two rules are stated as

$$\frac{\begin{array}{c}\tau,\gamma \vdash\ v_1 \in \|\tau\|_{vars}\ \wedge\ v_1 : t \\ \tau \vdash\ v_2 \notin \|\tau\|_{vars}\ \vee\ v_1 := v_2\end{array}}{\tau,\gamma \vdash\ v_1\ \texttt{->}\ v_2\ :\ t\ \hookrightarrow\ [v_2 := v_1]_{local}},$$

$$\frac{\begin{array}{c}\vdash \tau_1 = \tau \\ \forall_{i\in\{1,\dots,n\}}\left[\,\tau_i,\gamma \vdash\ \bar{e}_1^i\ \texttt{->}\ \bar{e}_2^i \hookrightarrow \Delta\tau_i\ \wedge\ \tau_{i+1} = \Delta\tau_i \circ \tau_i\,\right] \\ \tau,\gamma \vdash\ (\bar{e}_1^1,\dots,\bar{e}_1^n) : t \\ \vdash \Delta\tau = \Delta\tau_1 \circ \dots \circ \Delta\tau_n \circ [\,]\end{array}}{\tau,\gamma \vdash\ (\bar{e}_1^1,\dots,\bar{e}_1^n)\ \texttt{->}\ (\bar{e}_2^1,\dots,\bar{e}_2^n) : t \hookrightarrow \Delta\tau}. \tag{36}$$

The first rule describes what should happen when there is only one variables on either side of `->`. Here, we have used $\|\tau\|_{vars}$ to denote all variables defined in τ. First premise thus makes sure that v_1 exists as a variable in τ and defines its type as t, which will be the resulting type of the whole expression. The second line makes sure that v_2 is either undefined or is already an alias of v_1. If the premises succeed, the conclusion then additionally states that $[v_2 := v_1]$ should be added to the environment, which is taken to mean that v_2 is an alias of v_1. Whenever v_2 is then used in the context of the resulting environment, it should then be replaced by v_1, including when computing an $\lfloor e \rfloor$ expression and so on. The *local* subscript is used to make the alias restricted to the local scope. This is ensured by having all local environment changes removed at the end of a block statement, as we will see later on. The second rule simply uses the first rule and itself to recursively define what happens for variable tuples. Nothing special happens in this rule that we have not already seen, except for the fact that we have used a superscripts instead of just subscript to denote the individual expressions.

The `new` keyword is used to initialize (quantum) data variables. Both propositions and numbers can be used to initialize variables. In a similar way to the numbers, we will use capital letters P and Q for any *Prop* term and use lower-case letters p and q for constant propositions, which are either `true` or `false`. The rules for the three `new` expressions can be stated as

$$\frac{\tau \vdash P := p \hookrightarrow \Delta\tau}{\tau \vdash \texttt{new}\ P : \text{bool} \hookrightarrow \Delta\tau}, \qquad \frac{\begin{array}{c}\tau \vdash N := n \hookrightarrow \Delta\tau \\ \vdash 0 \le n < 2^m\end{array}}{\tau \vdash \texttt{new}\ N : \text{bool}[m] \hookrightarrow \Delta\tau},$$

$$\frac{\begin{array}{c}\tau \vdash N := n \hookrightarrow \Delta\tau_1 \\ \Delta\tau_1 \circ \tau \vdash P := p \hookrightarrow \Delta\tau_2\end{array}}{\tau \vdash \texttt{new}\ P[N] : \text{bool}[n] \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1}. \tag{37}$$

The reason why $\Delta\tau$s are included here is that *Num* and *Prop* terms might include measurements, in which case the relevant data variables are consumed and [measurement] is added to the environment as well. We will see how this happens below. The expression of the first rule can be seen to initialize a Boolean variable. This variable should of course be initialized according to the computed p. The second rule initialized an m-tuple of Boolean variables. These variables should be initialized according to the binary representation of n, which is why we make sure to check that $0 \le n < 2^m$. For this rule, m is not necessarily given explicitly by the context. For this version of the language, we will assume that the compiler can always guess m from the context. How the compiler should guess m will, however, be left for future work to decide. The third rule shows that a `new` $P[N]$ expression is used to initialize an n-tuple, where N evaluates to n. These should all be initialized with the same value according to p. Note how we ensure that N and P do not measure any of the same variables (if they measure any) by including $\Delta\tau_1$ in the hypothesis of the second premise.

The reader might note that we used the shorthand for writing the array-like tuple types in two of the above rules. In order to define this more formally, we should state a rule for such terms. We can state this rule simply as

$$\frac{\begin{array}{c}\tau \vdash t = (\overbrace{\bar{t},\, \ldots\, ,\bar{t}}^{n}) \\ \tau \vdash N := n \hookrightarrow [\,]\end{array}}{\tau \vdash \bar{t}\,[N] = t}. \tag{38}$$

This rule will be used every time we have to deal with terms on the form $\bar{t}\,[N]$ in our type checking rules. An important thing to note here, is that we do *not* allow measurements to happen inside N for such type terms. With this restriction in place, the rule is all we need to add to the type checking rules in order to be able to deal with $\bar{t}\,[N]$ expressions.

For our compactly written tuples of the form $id\,[M : N]$, we need the following rule. We have that

$$\begin{array}{c}\dfrac{\begin{array}{c}\tau \vdash N := n \hookrightarrow \Delta\tau_1 \\ \Delta\tau_1 \circ \tau \vdash M := m \hookrightarrow \Delta\tau_2 \\ \vdash \Delta\tau = \Delta\tau_2 \circ \Delta\tau_1 \\ \Delta\tau \circ \tau, \gamma \vdash (id\,[n],\, \ldots,\, id\,[m]) : t\end{array}}{\tau, \gamma \vdash id\,[M : N] : t \hookrightarrow \Delta\tau}. \\ \tau \vdash \lfloor id\,[M : N] \rfloor = \{id\,[n],\, \ldots,\, id\,[m]\} \\ \tau \vdash \| id\,[M : N] \| = \{id\,[n],\, \ldots,\, id\,[m]\}\end{array} \tag{39}$$

Again, we have made sure that M does not measure any variables measured by N, by adding $\Delta\tau_1$ to the environment for the second premise. When we have obtained our n and m, we can simply get the type of $(id\,[n],\, \ldots,\, id\,[m])$ from the resulting environment (for which no further environment change will be produced) and use it as our resulting type for $id\,[M : N]$. We have also included two more conclusion to make sure that $\lfloor e \rfloor$ and $\|e\|$ are also defined for $id\,[M : N]$ expressions.

Finally, we need to state how generic functions has their types inferred. For this we can use a rule stated as

$$\frac{\begin{array}{c}\vdash \tau_1 = \tau \\ \forall_{i \in \{1,\ldots,m\}} \left[\, \tau_i \vdash N_i := n_i \hookrightarrow \Delta\tau_i \;\wedge\; \tau_{i+1} = \Delta\tau_i \circ \tau_i \,\right] \\ \tau \vdash id\,\texttt{<}\, n_m, \ldots, n_1 \,\texttt{>} : t \\ \vdash \Delta\tau = \Delta\tau_m \circ \cdots \circ \Delta\tau_1 \circ [\,]\end{array}}{\tau \vdash id\,\texttt{<}\, N_m, \ldots, N_1 \,\texttt{>} : t \hookrightarrow \Delta\tau}. \tag{40}$$

This rule follows the same procedure as the previous one but with a whole series of numbers in this case. First each number is evaluated and then $id\,\texttt{<}\, n_m, \ldots, n_1 \,\texttt{>}$ is looked up in τ to get the resulting type of $id\,\texttt{<}\, N_m, \ldots, N_1 \,\texttt{>}$. Note that the changes of the $\Delta\tau$s are not needed in this case to look up $id\,\texttt{<}\, n_m, \ldots, n_1 \,\texttt{>}$ since they only interfere with the data variables. As mentioned before, $id\,\texttt{<}\, n_m, \ldots, n_1 \,\texttt{>}$ has not necessarily been compiled at the time $id\,\texttt{<}\, N_m, \ldots, N_1 \,\texttt{>}$ is encountered in the program, but can then be checked and compiled on demand in this case.

5.2 Numbers and propositions

We will now state the rules for the non-trivial number and proposition terms. Most of the arithmetic operations and relational operations are trivial since there is no upper bound on the number types and since the proposition types are strictly binary. The square root function is non-trivial since we have no floats in the language and have to round its return up or down. We will take it here to round it down. The `/` and `%` operators are also non-trivial in the sense that they can be defined several ways when it comes to negative integers. The question is whether we should round up or down for these and what sign the resulting integer number should have. What convention we choose for this language is not that important, the reason being that negative numbers do not play a big role as numbers will mainly be used in the context of determining loop bounds and tuple lengths. Since the inner syntax is made to be close to the syntax of C, however, it might be a good idea as to simply use the conventions of C in order not to surprise programmers used to C-like programming languages.

In terms of the logical operators, there is a non-trivial thing we need to decide. Since the proposition terms might produce changes to the environment, we should make sure that they short-circuit. However, since all other evaluations happen right-to-left, we should also let this be the case for the logical operators for the sake of consistency. We can state this right-to-left short-circuiting by the rules

$$\frac{\begin{array}{c}\tau \vdash P := \text{true} \hookrightarrow \Delta\tau_1 \\ \tau \vdash Q := q \hookrightarrow \Delta\tau_2\end{array}}{\begin{array}{c}\tau \vdash Q \texttt{ || } P := \text{true} \hookrightarrow \Delta\tau_1 \\ \tau \vdash Q \texttt{ \&\& } P := q \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1\end{array}}, \qquad \frac{\begin{array}{c}\tau \vdash P := \text{false} \hookrightarrow \Delta\tau_1 \\ \tau \vdash Q := q \hookrightarrow \Delta\tau_2\end{array}}{\begin{array}{c}\tau \vdash Q \texttt{ || } P := q \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1 \\ \tau \vdash Q \texttt{ \&\& } P := \text{false} \hookrightarrow \Delta\tau_1\end{array}}. \tag{41}$$

It should be noted here that when $\Delta\tau_2$ is not used in the desired conclusion, none of the potential measurements should be done for Q.

With the `measure` keyword a program can measure qubits dynamically and record the outcome as numbers or propositions, which can then be used to determine the control flow of the program going forwards. We can state the two rules needed as follows.

$$\frac{\begin{array}{c}\vdash P = \texttt{measure}\ e \\ \tau, \gamma \vdash\ e : \bar{t}* \hookrightarrow \Delta\tau_1 \\ \tau \vdash \text{span}(\bar{t}) = \text{qbit} \\ (\mathcal{U}_1 \to \mathcal{U}_2) \vdash e \xrightarrow{measure} p \\ \tau \vdash \Delta\tau_2 = [\lfloor e \rfloor : \text{undef}] \circ [\text{measurement}]\end{array}}{(\mathcal{U}_1 \to \mathcal{U}_2), \tau \vdash P := p \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1}, \qquad \frac{\begin{array}{c}\vdash N = \texttt{measure}\ e \\ \tau, \gamma \vdash\ e : \bar{t}* \hookrightarrow \Delta\tau_1 \\ \tau \vdash \text{span}(\bar{t}) = \text{qbit}[m] \\ (\mathcal{U}_1 \to \mathcal{U}_2) \vdash e \xrightarrow{measure} n \\ \tau \vdash \Delta\tau_2 = [\lfloor e \rfloor : \text{undef}] \circ [\text{measurement}]\end{array}}{(\mathcal{U}_1 \to \mathcal{U}_2), \tau \vdash N := n \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1}. \tag{42}$$

Here, we have used the mentioned $(\mathcal{U}_a \to \mathcal{U}_b)$ notation for the first time, and we have also used $e \xrightarrow{measure} n$ to denote that e is measured as bit sequence which represents the positive number n in the (unsigned) binary representation. The only difference between the two rules is for the measured proposition, e has to be a single-qubit type, whereas for a measured number, e can be any (array-like) m-tuple of single-qbit types. Note that the type of e has to have the `*` qualifier coming into the measurement, and that the variables that e is based on is set as undefined after the measurement. The reason why the type is set to 'undef' and not to ( ) is to open up for the possibility of reusing the variable names afterwards when declaring new variables. The parenthetical hypothesis of $(\mathcal{U}_1 \to \mathcal{U}_2)$ is not so important here but will be beneficial when we get to the while loop. As mentioned, it is meant to be understood as stating that the wave function of the universe starts out as $\mathcal{U}_1$ before the measurement and collapses into $\mathcal{U}_2$ afterwards. This is just my solution to the problem of describing a truly random process in terms of first-order logic; we just pretend that we know how the wave function of the universe will collapse given a measurement and that we are able to use this knowledge to "compute" p or n.

Note that there is an ambiguity in the language when an expression has the form `new measure` e since term `measure` e here can be parsed as both a Num or a $Prop$. For single-qubit types, this gives the same result, however. If we therefore simply choose to always parse the term as a Num for such an expression, it is ensured that the expression succeed regardless of whether e has a tuple type or a single-qubit type. This can be implemented by always trying to parse an expression as `new` Num before trying to parse it as `new` $Prop$.

The language also supports generating a number from a proposition by "casting" it as a number. The following rule can be used to implement the type check for this feature. We have

$$\frac{\begin{array}{c}\tau \vdash P := p \hookrightarrow \Delta\tau \\ \tau \vdash\ p = \text{true} \,\vee\, n = 0 \\ \tau \vdash\ p = \text{false} \,\vee\, n = 1\end{array}}{\tau \vdash (\texttt{num})\ P := n \hookrightarrow \Delta\tau}. \tag{43}$$

Note that since p has to be either true or false, n will have to be either 1 if p is true or 0 if p is false. I will use this way of making a piecewise definition in several instances below as well. This does mean that one has to look across to connect the condition with the relevant definition, but it makes it possible to write such a piecewise definition neatly on two lines.

5.3 Statements

In this section, we will go through the type checking rules for statements. We have already introduced most of the required notation in the last section, so the reader should at this point hopefully be able to follow what is going on for each rule.

5.3.1 Basic statements

We can start by stating the rules for the three most simple kinds of statements, namely the rules

$$\frac{\vdash\ s = \texttt{;}\ \vee\ s = \{\,\}}{\tau \vdash s : \text{none} \hookrightarrow [\,]}, \qquad \frac{\begin{array}{c}\vdash s = e\ \texttt{;} \\ \tau, \emptyset \vdash\ e : (\,) \hookrightarrow \Delta\tau\end{array}}{\tau \vdash s : \text{none} \hookrightarrow \Delta\tau}. \tag{44}$$

Here, the type 'none' is as mentioned used for a statement when the statement is neither a return statement nor a block statement with a return statement in it. Note that we use s to denote the domain of statement terms. The first rule simply states that nothing happens for the skip statement or for the empty block statement, and the second rule simply states that a statement e `;` produces exactly the environment change that the expression e produces. By requiring e to evaluate to the unit type in the second rule, we ensure that no garbage is implicitly created without being bound to a variable.

When a new variable is declared, we use the rules

$$\frac{\begin{array}{c}\vdash s = \bar{t}\ v\ \texttt{;} \\ \vdash v \notin \|\tau\|_{vars}\end{array}}{\tau \vdash\ s : \text{none} \hookrightarrow [\,v : \text{new}\,] \circ [\,v : \bar{t}\,*\,]}, \qquad \frac{\begin{array}{c}\vdash s = \bar{t}\ (\bar{e}_1, \ldots, \bar{e}_n)\ \texttt{;} \\ \tau \vdash \bar{t} = (\bar{t}_1, \ldots, \bar{t}_n) \\ \forall_{i \in \{1,\ldots,n\}} \left[\,\tau \vdash (\bar{t}_i\ \bar{e}_i\ \texttt{;}) \hookrightarrow \Delta\tau_i\,\right] \\ \tau \vdash \nu = \|\Delta\tau_1\|_{vars} \diamond \cdots \diamond \|\Delta\tau_n\|_{vars} \diamond \emptyset\end{array}}{\tau \vdash\ s : \text{none} \hookrightarrow \Delta\tau_n \circ \cdots \circ \Delta\tau_1 \circ [\,]}. \tag{45}$$

The first rules states what should happen when a single variable is being declared. It is checked that the variable is not already present in $\|\tau\|_{vars}$. When a variable is declared, it initially has 'new' qualifier attached to it to denote that it has not yet been initialized. This is then removed as soon as it is initialized. Note that a new variable is always automatically given the `*` qualifier, since we should of course be free to change it afterwards. The second rule uses the first one as well as itself to recursively define what should happen for a variable tuple. Note that we have made sure via the diamond operators that no variables are repeated in the tuple.

The `<-` operator, which is used when functions return their output, as well as an easy way to swap variables, should be type checked according to

$$\frac{\begin{array}{c}\vdash s = \bar{e}_1 \texttt{ <- } e_2\ \texttt{;}\\ \tau,\emptyset \vdash \bar{e}_1 : \bar{t}_1 * \ \wedge\ \gamma_1 = \lfloor \bar{e}_1 \rfloor\\ \tau,\emptyset \vdash e_2 : \bar{t}_2 \hookrightarrow \Delta\tau' \ \wedge\ \gamma_2 = \lfloor \bar{e}_2 \rfloor\\ \tau \vdash \mathrm{span}(\bar{t}_2) = \mathrm{span}(\bar{t}_1)\\ \forall_{v\in\gamma_1\setminus\gamma_2} \left[\,\Delta\tau' \circ \tau \vdash\ v : (\,) \vee v : \mathrm{new}\,\right]\\ \vdash \Delta\tau = [\,\bar{e}_1 : \bar{t}_2 *] \circ [\,\gamma_2 : (\,)] \circ \Delta\tau'\end{array}}{\tau \vdash\ s : \mathrm{none} \hookrightarrow \Delta\tau}, \qquad \frac{\begin{array}{c}\vdash s = \bar{e}_1 \texttt{ <- } e_2\ \texttt{;}\\ \tau,\emptyset \vdash \bar{e}_1 : \sim \bar{t}_1 * \ \wedge\ \gamma_1 = \lfloor \bar{e}_1 \rfloor\\ \tau,\emptyset \vdash e_2 : \bar{t}_2 \hookrightarrow \Delta\tau' \ \wedge\ \gamma_2 = \lfloor \bar{e}_2 \rfloor\\ \tau \vdash \mathrm{span}(\bar{t}_2) = \mathrm{span}(\bar{t}_1)\\ \forall_{v\in\gamma_1\setminus\gamma_2} \left[\,\Delta\tau' \circ \tau \vdash\ v : (\,) \vee v : \mathrm{new}\,\right]\\ \vdash \Delta\tau = [\,\bar{e}_1 : \sim \bar{t}_2 *] \circ [\,\gamma_2 : (\,)] \circ \Delta\tau'\end{array}}{\tau \vdash\ s : \mathrm{none} \hookrightarrow \Delta\tau}. \tag{46}$$

The only reason to write the rule up in two versions is to make sure that any initial $\sim$ operator is maintained. The first important things to note here is that $\bar{e}_1$ has to span the same type as $\bar{e}_2$. This means that variables of $\bar{e}_1$ *cannot* come into statement as having $(\,)$ types and then be revived again. However, if variable is consumed in $\bar{e}_2$, which would be recorded in $\Delta\tau'$, then said variables can appear again in $\bar{e}_1$ and be immediately revived. This is made sure by the second last premise. Note that this premise also makes it so that uninitialized variables can be initialized by the `<-` operator. For this it is used that the span function ignores the 'new' qualifier according to eq. (20). Since $[\,\bar{e}_1 : \bar{t}_2 *]$ and $[\,\bar{e}_1 : \sim \bar{t}_2 *]$ work according to eq. (25), we see that the rule works for tuples as well as for single variables. By also adding $[\,\gamma_2 : (\,)]$ to the environment change, we ensure that all variables in $\bar{e}_2$ which are moved into new variables are consumed by the statement. To sum up the restrictions for this operator, all variables on its left-hand side must either stand by themselves on its right-hand side, outside of any function call, or if they appear as part of a function call, then they must be consumed by that function (or another one in e_2 to the left of it). This is unless they are uninitialized (new), in which case they are also free to appear on the left-hand side of the operator.

Now that we can declare and update variables, we can implement the type checking for combined declarations and initializations by the following rule. The rule states that

$$\frac{\begin{array}{c}\vdash s = \bar{t}\ \bar{e}_1 \texttt{ = } e_2\ \texttt{;}\\ \tau \vdash (\bar{t}\ \bar{e}_1\ \texttt{;}) \hookrightarrow \Delta\tau_1\\ \Delta\tau_1 \circ \tau \vdash (\bar{t}\ \bar{e}_1 \texttt{ <- } e_2\ \texttt{;}) \hookrightarrow \Delta\tau_2\end{array}}{\tau \vdash s : \mathrm{none} \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1}. \tag{47}$$

Note that because variables of e_2 are consumed when they are moved into new variables by the `<-` operator, as we have just seen, a statement such as `bool b = a;` will have the effect of consuming `a`.

Together with the rule for the discard statement, which can be trivially stated as

$$\frac{\begin{array}{c}\vdash s = \texttt{discard}\ v\ \texttt{;}\\ \vdash v \in \|\tau\|_{vars}\end{array}}{\tau \vdash s : \mathrm{none} \hookrightarrow [\mathrm{garbage}] \circ [v : (\,)]}, \tag{48}$$

we thus have defined all the rules necessary for how variables are moved around in a program. To conclude this section of the basic statements, we can also look at the rules for what happens for a return statement. These can be stated as

$$\frac{\begin{array}{c}\vdash s = \texttt{return}\ e\ \texttt{;}\\ \tau,\emptyset \vdash e : t \hookrightarrow \Delta\tau\end{array}}{\tau \vdash s : t \hookrightarrow [\,\lfloor e \rfloor : (\,)\,] \circ \Delta\tau}, \qquad \frac{\vdash s = \texttt{return}\ N\ \texttt{;}}{\tau \vdash s : \mathrm{num}}. \tag{49}$$

We see that we finally have an inferred type for a statement different from the 'none' type. This is what is carried out of the relevant function definition and matched with the declared return type. An important thing to note is that all variables of $\lfloor e \rfloor$ are consumed by a return statement. We will see below, when we get to the block statement, that a block ends its execution at a return statement, but that it is still checked afterwards what garbage is potentially generated for this block. By making sure to consume the returned variables, it is therefore made sure that these does not count towards the garbage generation. The second rule is used to return a number type (denoted by 'num') for the main function. As mentioned, this might also be used as an opportunity to make regular functions fail with an error message, without the need to introduce a new statement for this. It might be a good idea, however, to include as special statement still for this, just to make the meaning more clear when a programmer wants a function to fail in certain cases. Such statement is not included in this version of the language, however.

5.3.2 The temp and restore statements

We will now look at some of the compound statements, i.e. statement that include substatements in them. Let us begin with the rule for temp statements. This can be stated as follows.

$$\frac{\begin{array}{c} \vdash s = \texttt{temp}\, s' \; \texttt{;} \\ [\,\|\tau\|_{vars} : *\,] \circ \tau \vdash s' \hookrightarrow \Delta\tau' \\ \vdash \text{unrestored}(\Delta\tau') = 0 \\ \vdash [\text{measurement}] \notin \Delta\tau' \\ \vdash \{v_1, \ldots, v_n\} = \|s'\|_{vars} \\ \vdash \tau' = \Delta\tau' \circ [\,\|\tau\|_{vars} : *\,] \circ \tau \\ \forall_{i \in \{1,\ldots,n\}} \left[\tau' \vdash v_i : t_i\,\right] \\ \vdash \Delta\tau = \left[\,[\{v_1, \ldots, v_n\} : \mathbf{c}\,] \circ [v_1 : t_1\,] \circ \cdots \circ [v_n : t_n\,]\,\right]_{temp} \end{array}}{\tau \vdash s : \text{none} \hookrightarrow \Delta\tau}. \tag{50}$$

Here, we see the the goal is to produce an environment change with a certain *temp* subscript attached to it. Such an environment change can be seen as what is pushed to the aforementioned temp stack when reasoning about the semantics of the temp statement. These environment changes are then removed again one by one for each restore statement. Before we state the rule for the restore statement, we need to explain what happens in the premises of this rule. First of all, we see that all variables are given the * qualifier in the context in s'. This is because we can allow ourselves to change the state of variables without the * qualifier as long as this only happens temporarily. This is ensured by the fact that functions are not allowed to have unrestored temp statements. In the following premise, the function 'unrestored' is assumed to count all unrestored *temp* environment changes that contribute to the input, in this case to $\Delta\tau'$. The premise then states that this number has to be 0, which means that one cannot have unrestored temp statements nested inside s'. Since a temp statement is meant to be uncomputed, s' cannot include any measurements either. The next three premises, which ends by $\forall_{i\in\{1,\ldots,n\}} [\tau' \vdash v_i : t_i]$, is used to get the types of all the variables included in s', which (by the assumed definition of $\|s'\|_{vars}$) are the variables in the set $\|s'\|_{vars}$. By adding $[\{v_1, \ldots, v_n\} : \mathbf{c}\,]$ after the sequence of these resulting types, we make it explicit in the rule that all the variables of $\|s'\|_{vars}$ have any * qualifiers removed (temporarily) from their types after the temp statement.

We will define the rule for the restore statement by the following. We have

$$\frac{\vdash s = \texttt{restore ;}}{\tau \vdash \; s : \text{none} \hookrightarrow [\text{restore}]}. \tag{51}$$

It is then simply assumed that $[\text{restore}] \circ \tau$ will have the effect of removing the temporary environment change that was last added to the environment. If we think about an environment on the form $\tau' = \Delta\tau_n \circ \cdots \circ \Delta\tau_i \circ \cdots \circ \Delta\tau_1 \circ \tau$, where $\Delta\tau_i$ is the first environment change from the left that has the *temp* subscript attached to it, then $[\text{restore}] \circ \tau' = \Delta\tau_n \circ \cdots \circ [\,] \circ \cdots \circ \Delta\tau_1 \circ \tau$.

5.3.3 The if and if-else statements

Let us now move on to the if statement. First, let us state a separate rule (along with some auxiliary rules) to check the condition of the if statement individually. We will give this rule the responsibility to first of all make sure that no variables are repeated in the condition and that all variables have Boolean types. We will also give it the responsibility to produce the local environment change that has to be added for the context of the body statement. This environment change should first of all include the removal of all possibilities for phase changes within the body statement, unless all condition variables have the $\sim$ qualifier. The reason for this is that the $\sim$ qualifier, as we recall, states that we are allowed to make computations for which the resulting phase might be non-vanishing and might depend on the state of the relevant variable. But if the body of an if statement changes the overall phase in any way, then there certainly is a phase change that depends on the states of the condition variables. We are therefore only allowed to have phase changing computations in the body statement if all these condition variables have the $\sim$ qualifier. At the same time, our environment change for the body should also hide any variable appearing in the condition as well as hiding any uninitialized variables to make sure that no variables are initialized inside the body. Otherwise it would lead to cryptic semantics when thinking about the statement as a quantum circuit. To do all this, we can state the type checking rules for conditions as follows. We have

$$\frac{\vdash c = v \;\vee\; c = \texttt{!}\; v}{\tau \vdash \lfloor c \rfloor = \{v\}}, \qquad \frac{\tau \vdash c = \texttt{all}\;\bar{e} \;\wedge\; \lfloor \bar{e} \rfloor = \gamma}{\tau \vdash \lfloor c \rfloor = \gamma},$$

$$\frac{\begin{array}{c}\tau \vdash c = c_1 \;\texttt{\&\&}\; \ldots \;\texttt{\&\&}\; c_n \;\vee\; c = c_1 \;\texttt{||}\; \ldots \;\texttt{||}\; c_n \\ \tau \vdash \gamma = \lfloor c_1 \rfloor \diamond \cdots \diamond \lfloor c_n \rfloor\end{array}}{\tau \vdash \lfloor c \rfloor = \gamma},$$

$$\frac{\begin{array}{c}\tau \vdash \{v_1, \ldots, v_n\} = \lfloor c \rfloor \\ \forall_{i\in\{1,\ldots,n\}} \left[\, \tau \vdash v_i : \text{bool} \,\right] \\ \forall_{i\in\{1,\ldots,n\}} \left[\, \tau \vdash \quad v^i : \sim \text{bool} \;\;\vee\;\; \Delta\tau_{phase} = [\, \|\tau\|_{vars} : \not\sim \,] \,\right] \\ \exists_{i\in\{1,\ldots,n\}} \left[\, \tau \vdash \neg\,(v^i : \sim \text{bool}) \;\;\vee\;\; \Delta\tau_{phase} = [\,] \,\right] \\ \tau \vdash \nu = \{ v \in \|\tau\|_{vars} \mid v : \text{new} \}\end{array}}{\tau \vdash c \hookrightarrow [\, \gamma : \text{none} \,] \circ [\, \nu : \text{none} \,] \circ \Delta\tau_{phase}}. \tag{52}$$

Here, c of course represents the domain of *Cond* terms. With the first three (auxiliary) rules, we have defined $\lfloor c \rfloor$ to include all variables in the condition and made sure that they are not repeated. For the final rule, we first pick out the atomic condition variables, namely the vs, and then check that they all have the type bool (possibly with additional qualifiers). In the third and fourth line, we again see a piecewise definition, this time of $\Delta\tau_{phase}$, which should be equal to $[\, \|\tau\|_{vars} : \not\sim \,]$ if and only if there exist a condition variable that does not have the type $\sim$ bool. The third premise ensured that unless $\Delta\tau_{phase} = [\, \|\tau\|_{vars} : \not\sim \,]$, then all variables have to be of type $\sim$ bool, and the fourth premise ensures that unless $\Delta\tau_{phase} = [\,]$, there has to exist at least one variables which can be proven not to have the type $\sim$ bool. We thus assume that we know the basic types to be distinct so that we can prove variables not to have a certain type. In the last premise, we define ν to be all uninitialized (new) variables in τ in order to remove these in the environment produced by c. The environment changes are then collected in the conclusion.

We can now move on to the if statement itself. Its rule can be stated as

$$\frac{\begin{array}{c} \vdash s = \texttt{if}\ (c)\ s' \\ \tau \vdash c \hookrightarrow \Delta\tau_{in} \\ \Delta\tau_{in} \circ \tau \vdash\ s' : \text{none} \hookrightarrow \Delta\tau' \\ \vdash \text{unrestored}(\Delta\tau') = 0 \\ \vdash [\text{garbage}] \notin \Delta\tau' \qquad \vdash [\text{measurement}] \notin \Delta\tau' \\ \vdash \tau' = \Delta\tau' \circ \tau \\ \forall_{v \in \|\Delta\tau'\|_{vars}} \left[\tau \vdash v : t_v^{in} \qquad \tau' \vdash v : t_v^{out} \right] \\ \forall_{v \in \|\Delta\tau'\|_{vars}} \left[\tau \vdash \text{span}(t_v^{in}) = \text{span}(t_v^{out}) \right] \\ \vdash \{v_1, \dots, v_n\} = \{v \in \|\Delta\tau'\|_{vars} \mid t_v^{out} \neq t_v^{in}\} \\ \forall_{i \in \{1,\dots,n\}} \left[\vdash \Delta\tau_i = [v_i : \text{span}(t_{v_i}^{in})\,]\,\right] \\ \vdash \Delta\tau = \Delta\tau_n \circ \cdots \circ \Delta\tau_1 \circ \Delta\tau' \circ [\,] \end{array}}{\tau \vdash s : \text{none} \hookrightarrow \Delta\tau}. \tag{53}$$

Here, the first thing we do is to check the condition with the previous rule and get the environment change, $\Delta\tau_{in}$, of the environment that goes into the body statement s'. The body statement is now checked which yields us $\Delta\tau'$. We check that the type of s' is 'none' since we do not allow any return statement inside of the body of an if statement. The fourth premise then checks that there is no unrestored temporary environment changes in $\Delta\tau'$, as this would not be sensible. The fifth premise line then checks that there has been no garbage generation and no measurement in s'. Lines six through eight checks that no variable has been consumed in s'. The following premise defines the (possibly empty) set $\{v_1, \dots, v_n\}$ to be the set the set of all variables with changes recorded in $\Delta\tau'$ that do not end out as having the same types they had initially, but have their types changed in the if statement. Since this type change is dependent on the condition variables, the states about which we have no knowledge, it would no longer make sense to track one or the other basis type for such variables. We therefore lose track of their basis type, which is exactly what the spanning types are there for. The next premise therefore defines $\Delta\tau_i$ for all $1 \leq i \leq n$ to record that each v_i gets its corresponding spanning type. The last premise then collects all these $\Delta\tau$s to yield the resulting $\Delta\tau$.

We have now finally seen how the if statement works in detail. From here, it is not too hard to define what should happen for the if-else statement in terms of type checking rules. Even though it does not necessarily have to be implemented this way, we can simply define its type checking rule according to a basic implementation, where we use a temporarily computed condition variable to hold the Boolean value of what the condition evaluates to. Other implementations of the rule should then make sure to yield the same outcome. We can thus state the rule for the if-else statement as

$$\frac{\begin{array}{c} \vdash s = \texttt{if}\ (c)\ s_1\ \texttt{else}\ s_2 \\ \vdash v \notin \|\tau\|_{vars} \\ \vdash s_{temp} = \texttt{temp bool}\ v\ \texttt{= false ;} \\ \vdash s'_{temp} = \texttt{temp if}\ (c)\ \texttt{Not}\ v\ \texttt{;} \\ \vdash s_{if} = \texttt{if}\ (v)\ s_1 \qquad \vdash s_{else} = \texttt{if}\ (!\ v)\ s_2 \\ \vdash s_{rest} = \texttt{restore ;} \\ \tau \vdash \{\ s_{temp}\ \ s'_{temp}\ \ s_{if}\ \ s_{else}\ \ s_{rest}\ \ s_{rest}\ \} \hookrightarrow \Delta\tau \end{array}}{\tau \vdash s : \text{none} \hookrightarrow \Delta\tau}, \tag{54}$$

Much of this rule should speak for it self. Since we have not seen it before, the reader might note that we are able to declare and initialize a new variable in a temp statement, as seen in the third premise line, which will then be undefined once again when we reach the relevant restore statement. This is only possible when the body of the temp statement is not a block statement, however, since the new variable would otherwise not be able to escape that block. It should also be noted that while the semantics of an if-else statement is independent of the order of the if part and the else part, the program is not completely independent of this order. This is because of the fact that the outgoing types of the if part of the statement will be the incoming types of the else part. If types are changed in the if part, they will therefore come into the else part as spanning. The resulting types after the else part might therefore be different if the order of the two parts are changed.

5.3.4 Block statements

Another kind of compound statement that is very central to the language is the block statement. Its rule can be stated as

$$\frac{\begin{array}{c} \vdash s = \{s_1 \ \dots \ s_n \ s' \ s'_1 \ \dots \ s'_m\,\} \\ \vdash \tau_1 = \tau \\ \forall_{i\in\{1,\dots,n\}} \left[\,\tau_i \vdash \ s_i : \text{none} \hookrightarrow \Delta\tau_i \ \wedge \ \tau_{i+1} = \Delta\tau_i \circ \tau_i\,\right] \\ \tau_{n+1} \vdash \ s' : \tilde{t} \hookrightarrow \Delta\tau_n \\ \vdash \ \{s'_1, \dots, s'_m\} = \emptyset \ \vee \ \tilde{t} \neq \text{none} \\ \vdash \Delta\tau' = \text{unlocal}(\Delta\tau_n \circ \cdots \circ \Delta\tau_1 \circ [\,]) \\ \vdash \tau' = \Delta\tau' \circ \tau \\ \vdash \{v_1, \dots, v_k\} = \|\Delta\tau\|_{vars} \setminus \|\tau\|_{vars} \\ \begin{array}{l} \forall_{i\in\{1,\dots,k\}} \left[\tau' \vdash \quad \big(v_i : (\,) \vee v_i : \text{new}\,\big) \vee \Delta g_{new} = [\text{garbage}\,]\,\right] \\ \exists_{i\in\{1,\dots,k\}} \left[\tau' \vdash \neg \big(v_i : (\,) \vee v_i : \text{new}\,\big) \vee \Delta g_{new} = [\,]\,\right] \end{array} \\ \vdash \Delta\tau = \Delta g_{new} \circ [v_1 : \text{undef}\,] \circ \cdots \circ [v_k : \text{undef}\,] \circ \Delta\tau' \end{array}}{\tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau}. \tag{55}$$

The first thing to note is how we have ensured that s' is the last statement of the block to be evaluated. In the first five premise lines, we thus make sure that the initial n statements (where $n \geq 0$) all evaluate to 'none', and that s' is the last statement in the block if it also evaluates to 'none'. We have used $\tilde{t}$ here to represent the domain of all types plus the 'none' type. In the third line we also get the $\Delta\tau_i$ for each s_i. We then define $\Delta\tau$ to be the collected environment change inside the block, but with all local variables removed. These are the environment changes with the *local* subscript attached to them. We have already seen this for the alias variables, and in a moment, we will also see how local functions are defined. The unlocal function is thus assumed to remove said environment changes. We then get ready to discard any new variables declared inside the block. First, we define τ' as the resulting outgoing environment and also define $\{v_1, \dots, v_k\}$ to be the set of all variables declared in $\Delta\tau$ that was also live at the beginning. We then proceed to define Δg_{new} with a piecewise definition such that it is set equal to $[\text{garbage}\,]$ if and only if there exists one of the newly declared variables that have been initialized in the block statement but has not been consumed before the end. In the last premise, we make sure to make all these locally declared variables undefined once again (which means that they are free to be redeclared again afterwards) and also collect the rest of the environment changes.

5.3.5 Local functions

So far, we have not discussed the local function declaration much. It is, however, an important piece of the language since it gives is us the flexibility needed in order for higher-order functions defined over other functions to be useful. We only allow local functions to have a single type signature (apart from the ones that can be inferred from it), which is why we can use the more compact ML-like notation to declare the type of each input parameter, as seen in the first premise of rule (56) just below. Note that these parameters are optional. We see that local functions are defined from expressions. The main thing we need to ensure by the rule, is of course that all the types matches. There is another interesting thing we need to ensure as well, however, which is that every variable used to make up the expression defining the function should be removed from the environment of the relevant block. Only at end of the block in which the local function is declared, when we remove all local declarations, should we get these variables back. The reason for this is first of all that it would otherwise be impossible to check against aliasing in function calls. If we for instance defined a local function `foo` by `fun foo = CNOT a;` (with our usual definition of `CNOT`), it would be impossible to check against the meaningless expression `foo a` if we did not also remove `a` from the local scope. We also need to make sure that the expression on the right-hand side, e, is not based on any variables directly, i.e. that $\lfloor e \rfloor = \emptyset$. Otherwise, this would be another potential source of unwanted aliasing. Additionally, we need to make sure that no type is changed by e, apart from what is advertised by the types of the function parameters. The reason why we need to do this is that it does not make sense to make several calls to the local function if the variables used in e have their types changed.

With all this in mind, we can state the rule for local function declarations as follows.

$$\frac{\begin{array}{c} \vdash s = \texttt{fun}\, id\ (\bar{e}_1 : t_1)\ \ldots\ (\bar{e}_n : t_n)\ \texttt{=}\ e\ \texttt{;} \\ \vdash \|\Delta\tau_{in}\|_{all} = \bigcup_{i=1}^{n} \|\bar{e}_i\|_{ids} \\ \forall_{i\in\{1,\ldots,n\}} \left[\, \Delta\tau_{in} \circ \tau \ \vdash\ t_i \bullet \bar{e}_i \hookrightarrow \Delta\tau_i \,\right] \\ \Delta\tau_{in} \circ \tau \vdash\ \gamma = \lfloor \bar{e}_1 \rfloor \diamond \cdots \diamond \lfloor \bar{e}_n \rfloor \diamond \emptyset \ \wedge\ \lfloor e \rfloor = \emptyset \\ \Delta\tau_{in} \circ \tau, \emptyset \ \vdash\ e : t \hookrightarrow \Delta\tau \\ \vdash \Delta g = \text{getMeasured}(\Delta\tau) \circ \text{getGarbage}(\Delta\tau) \\ \vdash \Delta\tau \circ \tau = \Delta\tau_1 \circ \cdots \circ \Delta\tau_n \circ \tau \\ \tau \vdash \nu = \|e\| \setminus \|(\bar{e}_1, \ldots \bar{e}_n)\| \end{array}}{\tau \vdash\ s : \text{none} \hookrightarrow [\,\nu : \text{none}\,]_{local} \circ [\,id : t_1\ \texttt{->}\ \ldots\ \texttt{->}\ t_n\ \texttt{->}\ t + \Delta g\,]_{local}} . \tag{56}$$

Here, we have used the already defined $\bullet$ operator as a way to define $\Delta\tau_{in}$ as giving each $\bar{e}_i$ the input type according to t_i. The reason why this works is that the third premise line would fail otherwise due to the fact that the $\bullet$ operator is only partially defined. With the second premise, we also make sure that $\Delta\tau_{in}$ includes no other variables than the ones in the $\bar{e}_i$s. We thus take $\|\Delta\tau_{in}\|_{all}$ to denote all variables in $\Delta\tau_{in}$ including number, proposition and function variables, and take $\|\bar{e}_i\|_{ids}$ to denote all ids used in $\bar{e}_i$ (regardless of any environment). With the third premise, we also get the $\Delta\tau_i$s, which are all the environment changes advertised by the parameter types. The fourth premise checks that no variables are repeated inside and amongst the $\bar{e}_i$s. It also checks that $\lfloor e \rfloor$ is indeed the empty set, which, as mentioned, is required in order to prevent unwanted aliasing. We then use the $\Delta\tau_{in}$ to infer the type and environment change for e. Next, we use functions 'getGarbage' and 'getMeasured' to get any garbage and/or measurement produced in $\Delta\tau$. We have thus assumed these functions to find any instances of respectively [garbage] or [measurement] for its input and to return a Δg in accordance with what was found. In the second last premise, we check that $\Delta\tau$ is equivalent to the environment change advertised by the parameter types. It is thus assumed that we can prove the equivalence of two environments, when they infer all the same types for all the same variables. Note that this premise both makes sure that the type changes of the input parameters are as advertised and also makes sure that no variables have their types changed by e apart from the variables appearing in the $\bar{e}_i$s. The last premise collects the set of all variables of e that are not part of the parameters. This set is then removed from the local scope from that point on by adding $[\,\nu : \text{none}\,]_{local}$ to the resulting environment change of the conclusion. The conclusion also record that id has now been successfully defined as a local function.

5.3.6 The assume and assert statements

The rules for the assume and assert statements are simple enough to define. Collectively, they can be defined by

$$\frac{\vdash s = \texttt{assert}\ \bar{e}\ \texttt{of}\ t\ \texttt{;} \qquad \tau, \emptyset \vdash \bar{e} : t}{\tau \vdash s : \text{none} \hookrightarrow [\,]}, \qquad \frac{\vdash s = \texttt{assume}\ \bar{e}\ \texttt{of}\ t\ \texttt{;} \quad \tau, \emptyset \vdash \bar{e} : t' \quad \tau \vdash \text{span}(t') = \text{span}(t) \quad \tau, \gamma \vdash \text{ assume : allowed } \vee\ \bar{e} : t}{\tau \vdash\ s : \text{none} \hookrightarrow [\,\bar{e} : t\,]},$$

$$\frac{\vdash s = \texttt{assume}\ v\ \texttt{=}\ P\ \texttt{;} \quad \tau \vdash P := p \hookrightarrow \Delta\tau \quad \Delta\tau \circ \tau \vdash v : \hat{t} \quad \tau \vdash \text{ assume : allowed}}{\tau \vdash\ s : \text{none} \hookrightarrow [\,v : (\,)\,] \circ \Delta\tau}, \qquad \frac{\vdash s = \texttt{assume}\ \bar{e}\ \texttt{=}\ N\ \texttt{;} \quad \tau \vdash N := n \hookrightarrow \Delta\tau \quad \Delta\tau \circ \tau \vdash \bar{e} : \bar{t} \quad \tau \vdash\ \text{span}(\bar{t}) = \text{qbit}[m]\ \wedge\ n < 2^m \quad \tau \vdash\ \text{assume : allowed} \quad \tau \vdash \lfloor \bar{e} \rfloor = \gamma}{\tau \vdash\ s : \text{none} \hookrightarrow [\gamma : (\,)\,] \circ \Delta\tau}. \tag{57}$$

In the first rule, we see how an assert statement simply checks that the type of the expression is as stated. If the full type of $\bar{e}$ includes qualifiers not present in t, the statement still succeeds as $\bar{e} : t$ can then still be inferred. For the assume statement of the second rule, we see that it is checked that t and t' span the same Hilbert space. It is assumed that the proposition 'assume : allowed' is true if and only if the type signature being checked uses the `+:` lexeme, which is meant to allow for assumptions inside the definition. Furthermore, by including $[\ldots] \vee \bar{e} : t$, the assume statement never fails if the type is already in the assumed type, even if the normal `::` lexeme is used for the relevant type signature. The third and fourth rule show what happens when the user manually terminates a variable. Note that for both rules, it is checked that v or $\bar{e}$ is not measured in $\Delta\tau$ coming from evaluating the P or N. When assuming that $\bar{e}$ can be terminated in the state represented by n, it is also checked that the representation of n does not exceed m bits as a bit string. When evaluating P or N respectively, the obtained p or n is then recorded as the value for the relevant qubits, and the program can then reuse these qubits without having to (irreversible) prepare them as fresh qubits once again.

5.3.7 Classical control with numbers and propositions

We will now look at the special statement that uses numbers and propositions to decide the control flow of the program in a classical way. We will also start by looking at the rules for the statements to define new numbers and propositions.

The rules used when defining new numbers and propositions can be stated as

$$\frac{\vdash s = \texttt{let}\ id\ \texttt{=}\ N\ \texttt{in}\ s' \quad \vdash id \notin \|\tau\|_{vars} \quad \tau \vdash N := n \hookrightarrow \Delta\tau_1 \quad [id := n] \circ \Delta\tau_1 \circ \tau \vdash s' : \tilde{t} \hookrightarrow \Delta\tau_2}{\tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1}, \qquad \frac{\vdash s = \texttt{let}\ id\ \texttt{=}\ P\ \texttt{in}\ s' \quad \vdash id \notin \|\tau\|_{vars} \quad \tau \vdash P := p \hookrightarrow \Delta\tau_1 \quad [id := p] \circ \Delta\tau_1 \circ \tau \vdash s' : \tilde{t} \hookrightarrow \Delta\tau_2}{\tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1}. \tag{58}$$

Here, we make sure that $\Delta\tau_1$ coming from evaluating N or P is sent along when evaluating s' and when obtaining the resulting environment change. We also make sure to give id the value of n or p in the context of s'. Note that we check that id is not defined already as a data variable but that we do not care if we overwrite any previous definitions of id if id was already number or proposition variable. We are, by the way, also free to overwrite a number or proposition variable locally with a declaration of a data variable of the same name, at least as the language is in this version.

We can use propositions for classical control in the form of the special if statement, the rules for which can be defined simply as

$$\frac{\begin{array}{c}\vdash s = \texttt{if}\ (P)\ s' \\ \tau \vdash P := \text{true} \hookrightarrow \Delta\tau_1 \\ \Delta\tau \circ \tau \vdash s' : \tilde{t} \hookrightarrow \Delta\tau_2\end{array}}{\tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1}, \qquad \frac{\begin{array}{c}\vdash s = \texttt{if}\ (P)\ s' \\ \tau \vdash P := \text{false} \hookrightarrow \Delta\tau\end{array}}{\tau \vdash s : \text{none} \hookrightarrow \Delta\tau}. \tag{59}$$

Note here that the body of this kind of if statement can include a return statement, in which case $\tilde{t}$ will be different from 'none' in the first rule. The program might therefore end the execution of the relevant function depending of what P evaluates to.

For the for loop, we can define its type checking recursively. We only have to make sure that the loop stops immediately if a return statement is reached. We do so in the following three rules, which define the type checking for the for loop. We have, when the keyword `to` is used, that

$$\frac{\begin{array}{c}\vdash s = \texttt{for}\ id\ \texttt{=}\ N\ \texttt{to}\ M\ s' \\ \tau \vdash N := n \hookrightarrow \Delta\tau_1 \\ \tau \vdash M := m \hookrightarrow \Delta\tau_2 \\ \vdash m < n\end{array}}{\tau \vdash s : \text{none} \hookrightarrow \Delta\tau_2 \circ \Delta\tau_1},$$

$$\frac{\begin{array}{c}\vdash s = \texttt{for}\ id\ \texttt{=}\ N\ \texttt{to}\ M\ s' \\ \tau \vdash N := n \hookrightarrow \Delta\tau_1 \\ \tau \vdash M := m \hookrightarrow \Delta\tau_2 \\ \vdash m \geq n \\ [id := n] \circ \tau \vdash s' : \text{none} \hookrightarrow \Delta\tau_{first} \\ \tau \vdash (\texttt{for}\ id\ \texttt{=}\ (n+1)\ \texttt{to}\ M\ s') : \tilde{t} \hookrightarrow \Delta\tau_{rest}\end{array}}{\tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau_{rest} \circ \Delta\tau_{first} \circ \Delta\tau_2 \circ \Delta\tau_1}, \qquad \frac{\begin{array}{c}\vdash s = \texttt{for}\ id\ \texttt{=}\ N\ \texttt{to}\ M\ s' \\ \tau \vdash N := n \hookrightarrow \Delta\tau_1 \\ \tau \vdash M := m \hookrightarrow \Delta\tau_2 \\ \vdash m \geq n \\ [id := n] \circ \tau \vdash s' : \tilde{t} \hookrightarrow \Delta\tau_{first} \\ \tau \vdash \tilde{t} \neq \text{none}\end{array}}{\tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau_{first} \circ \Delta\tau_2 \circ \Delta\tau_1}, \tag{60}$$

and the rules for when `downto` is used can be stated in a similar way. The first rule states what happens when m is less than n, in which case all we get from the for loop are the potential measurement environment changes from evaluating N and M. In the second rule, we recursively define what should be checked for a for loop when the first loop iteration is executed with no return statements reached. We thus check s' with $id := n$ and make sure that it can be inferred to have the statement type 'none' for this rule. This gives us the environment change of the first loop iteration, $\Delta\tau_{first}$. In the last premise we recursively get $\Delta\tau_{rest}$, which is the environment change of the rest of the loop iterations. The third rule describes how the loop should be stopped once a return statement is reached and s' thus can be inferred to have a type that is distinct from the 'none' type. It follows the same procedure as the second rule but does not get $\Delta\tau_{rest}$ before stopping the loop.

5.3.8 The while loop

The while loop is a bit more interesting. It is the only strictly irreversible compound statement as it measures its condition variable before every loop iteration. It can for instance be used for a random quantum algorithm where the outcome is checked after each iteration and needs to be validated before ending the procedure. The while loop stops as soon as the condition variable is measured as false. For consistency, it should also stop if a return statement is reached just like the for loop. Going into the body statement of the while loop, the condition variable is always reinitialized as true, and when a while loop finishes by measuring the variable as false, the variable is then reinitialized as false. The rules for the while loop can be stated as

$$\frac{\begin{array}{c}\vdash s = \texttt{while}\ v\ s' \\ \vdash P = \texttt{measure}\ v \quad \tau \vdash \text{span}(v) = \text{qbit} \\ (\mathcal{U}_1 \to \mathcal{U}_2), \tau \vdash P := \text{false} \hookrightarrow \Delta\tau' \\ \vdash \Delta\tau = [v : \text{bool}\,*] \circ \Delta\tau'\end{array}}{(\mathcal{U}_1 \to \mathcal{U}_2), \tau \vdash s : \text{none} \hookrightarrow \Delta\tau},$$

$$\frac{\begin{array}{c}\vdash s = \texttt{while}\ v\ s' \\ \vdash P = \texttt{measure}\ v \quad \tau \vdash \text{span}(v) = \text{qbit} \\ (\mathcal{U}_1 \to \mathcal{U}_2), \tau \vdash P := \text{true} \hookrightarrow \Delta\tau_1 \\ \vdash \Delta\tau_{init} = [v : \text{bool}\,*] \\ (\mathcal{U}_2 \to \mathcal{U}_3), \Delta\tau_{init} \circ \tau \vdash s' : \text{none} \hookrightarrow \Delta\tau_2 \\ \vdash \Delta\tau' = \Delta\tau_2 \circ \Delta\tau_{init} \circ \Delta\tau_1 \\ (\mathcal{U}_3 \to \mathcal{U}_{fin}), \Delta\tau' \circ \tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau_{rest} \\ \vdash \Delta\tau = \Delta\tau_{rest} \circ \Delta\tau'\end{array}}{(\mathcal{U}_1 \to \mathcal{U}_{fin}), \tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau}, \qquad \frac{\begin{array}{c}\vdash s = \texttt{while}\ v\ s' \\ \vdash P = \texttt{measure}\ v \quad \tau \vdash \text{span}(v) = \text{qbit} \\ (\mathcal{U}_1 \to \mathcal{U}_2), \tau \vdash P := \text{true} \hookrightarrow \Delta\tau_1 \\ \vdash \Delta\tau_{init} = [v : \text{bool}\,*] \\ (\mathcal{U}_2 \to \mathcal{U}_3), \Delta\tau_{init} \circ \tau \vdash s' : \tilde{t} \hookrightarrow \Delta\tau_2 \\ \tau \vdash \tilde{t} \neq \text{none} \\ \vdash \Delta\tau = \Delta\tau_2 \circ \Delta\tau_{init} \circ \Delta\tau_1\end{array}}{(\mathcal{U}_1 \to \mathcal{U}_3), \tau \vdash s : \tilde{t} \hookrightarrow \Delta\tau}. \tag{61}$$

In the first rule, we define what should happen when v is measured to be false. With the third premise, we check that v is a single-qubit variable. Just as we did for the measure terms, we have again included the parenthetical $(\mathcal{U}_a \to \mathcal{U}_b)$ hypotheses. This is useful especially for the last two rules. Recall that $(\mathcal{U}_a \to \mathcal{U}_b)$ denotes a collapse of the overall wave function. Such a collapse happens at the beginning of each iteration, i.e. when we measure P, and can also happen when executing the body statement. Since P is measured to be false in this first rule, however, the body is not executed. The collected $\Delta\tau$ we need in this case is therefore just the environment coming from s', namely $\Delta\tau'$, as well is the environment change when we reinitialize v. The second rule defines the type check recursively when v is measured to be true and when no return statement is reached in s'. Here, $\mathcal{U}_1$ denotes is the initial wave function, $\mathcal{U}_2$ denotes the wave function after the initial measurement, $\mathcal{U}_3$ denotes the wave function after the first loop iteration and $\mathcal{U}_{fin}$ denotes the final wave function after the whole while loop. Recall that this notation is simply used to be able to describe a random process using inherently deterministic rules. Apart from this extra notation the procedure is almost the same as for the for loop. The only real additional difference is that here, we can simply call the whole while loop statement, s, again after each iteration has been executed, as we do not need to change any loop indices. The third rule also follows the same procedure as the corresponding rule for the for loop in order to make sure that the loop stops once a return statement has been reached.

5.4 Declarations and definitions

In this section we will see the type checking rules for function definitions (and declarations) and type declarations. The type equation, the number definition and the main function definition will not be discussed as their semantics are to simple to require further discussion.

We will start by stating the rule for the most basic function definition, when there is only one type signature and no number input. The rule follows a procedure which has overlaps with the procedure for the local functions. The rule can be stated as

$$\frac{\begin{array}{c}\vdash t = t_1 \texttt{ -> } \dots \texttt{ -> } t_n \texttt{ -> } \bar{t}^{ret} + \Delta g \\ \vdash id \notin \|\tau\|_{all} \\ \vdash \ \|\Delta\tau_{in}\|_{all} = \bigcup_{i=1}^{n} \|e_i\|_{all} \\ \forall_{i\in\{1,\dots,n\}} \left[\, \Delta\tau_{in} \circ \tau \vdash \ t_i \bullet e_i \hookrightarrow \Delta\tau_i \,\right] \\ \Delta\tau_{in} \circ \tau \vdash \gamma = \lfloor \bar{e}_1 \rfloor \diamond \cdots \diamond \lfloor \bar{e}_n \rfloor \\ \Delta\tau_{in} \circ \tau \vdash \text{assume} : a \\ \Delta\tau_{in} \circ [\, id : t\,] \circ \tau \vdash s : \bar{t}^{ret} \hookrightarrow \Delta\tau \\ \vdash \text{unrestored}(\Delta\tau) = 0 \\ \vdash \Delta g' = \text{getMeasured}(\Delta\tau) \circ \text{getGarbage}(\Delta\tau) \\ \vdash \ a = \text{allowed} \ \vee \ \Delta g = \Delta g' \\ \vdash \ a \neq \text{allowed} \ \vee \ [\text{measurement}] \notin \Delta g' \\ \vdash \Delta\tau \circ \tau = \Delta\tau_1 \circ \cdots \circ \Delta\tau_n \circ \tau \end{array}}{\tau \vdash \big(id, \langle\,\rangle, \langle a \rangle, \langle t \rangle, \langle e_1, \dots, e_n \rangle, s, \langle\,\rangle, \texttt{;}\big)_{FunDef} \hookrightarrow [\, id : t\,]}. \tag{62}$$

In the conclusion of this rule, we see a tuple with a subscript of $FunDef$. We take this tuple to represent what is parsed when parsing a function definition as well as its associated declarations. The first element, id, is the name of the function, the next element, which is empty in this case is an array of all the number ids for generic functions. Then there is an array with each of the declared types of the function followed by an array with flags denoting whether the `+:` was used or not for each type (in the same order). In this case just the one t and the one a since the rule assumes only one type signature. This is then followed by an array of the input parameters and then comes the defining expression, s. The two next elements are there for function definitions where the reversal of the function is explicitly defined. They are first the input parameter array once again (as the input parameters can be called by different names for the reversed function) and then at last the statement defining reversed function. When there is no reverse definition, we just take this statement to be the skip statement, which is why `;` is seen as the last element of the tuple. The rule starts out in a same way as rule (56) above for local function declarations, which is by defining $\Delta\tau_{in}$ to hold all the incoming types of the input parameters and by getting the advertised type changes, i.e. the $\Delta\tau_i$s, using the $\bullet$ operator. We also check that variables are not repeated inside and amongst the e_is. We then use the assume flag a to make sure that $\Delta\tau_{in}$ allows assumptions if and only if a is set as 'allowed' by the parser. When we type check s in the seventh premise, it can be seen that we allow for recursion inside the definition by including $[\, id : t\,]$ in the incoming environment. We proceed to check that there is no unrestored temp statements and that any garbage or measurements produced in s is advertised in Δg, which has been parsed according to whether the type signature includes the `+garbage` qualifier and whether it ends with a `->` arrow or a `-->` arrow. The two next premises before the last one thus makes this check, but, as we can see, only if the type signature does not allow assumptions. If on the other hand `+:` is used, the programmer is allowed to state whether a garbage and/or a measurement production is recorded by the function type or not. The last premise then finally checks that the obtained $\Delta\tau$ is equivalent with the advertised environments changes just as we saw for rule (56). If all these premises succeed, the program then records $id : t$ as the produced environment change.

When the reversal of the function is explicitly defined, and there is still only one type signature, we can use the following rule to check its type. We have

$$\frac{\begin{array}{c}\vdash a = \text{allowed}\\ \vdash t = t_1 \texttt{ -> } \dots \texttt{ -> } t_n \texttt{ -> } \bar{t}^{ret} + [\,]\\ \vdash \text{reverse}(t) = t'_1 \texttt{ -> } \dots \texttt{ -> } t'_m \texttt{ -> } \bar{t}'^{ret} + [\,]\\ \vdash t_{test} = t_1 \texttt{ -> } \dots \texttt{ -> } t_n \texttt{ --> } \bar{t}^{ret} + [\,]\\ \vdash t'_{test} = t'_1 \texttt{ -> } \dots \texttt{ -> } t'_m \texttt{ --> } \bar{t}'^{ret} + [\,]\\ \tau \vdash \big(id, \langle\,\rangle, \langle a\rangle, \langle t_{test}\rangle, \langle \bar{e}_1, \dots, \bar{e}_n\rangle, s, \langle\,\rangle, \texttt{;}\big)_{FunDef} \hookrightarrow [id : t_{test}]\\ \tau \vdash \big(id, \langle\,\rangle, \langle a\rangle, \langle t'_{test}\rangle, \langle \bar{e}'_1, \dots, \bar{e}'_m\rangle, s_{rev}, \langle\,\rangle, \texttt{;}\big)_{FunDef} \hookrightarrow [id : t'_{test}]\end{array}}{\tau \vdash \big(id, \langle\,\rangle, \langle a\rangle, \langle t\rangle, \langle \bar{e}_1, \dots, \bar{e}_n\rangle, s, \langle \bar{e}'_1, \dots, \bar{e}'_m\rangle, s_{rev}\big)_{FunDef} \hookrightarrow [id : t]}. \tag{63}$$

First of all, it is checked that the `+:` lexeme is used and that a is thus set to 'allowed'. We then define t_{test} and t'_{test} in order to use rule (62) for both of the two statements, which are s and s_{rev}. We use the `-->` arrow for t_{test} and t'_{test} since it is up to the programmer to make sure that no measurements happen in s or s_{rev}, or that if do, that it does not ruin the intended API of the function. If rule (62) can be used to make the last two premises succeed, we can thus conclude that the function definition produces $[id : t]$.

When there are several type signatures, we can just use the two previous rules repeatedly for each signature. The rule for such definitions can thus simply be stated as

$$\frac{\begin{array}{c}\vdash t = (t_1 \mid \dots \mid t_n)\\ \forall_{i \in \{1,\dots,n\}} \big[\tau \vdash \big(id, \langle\,\rangle, \langle a_i\rangle, \langle t_i\rangle, es, s, es', s_{rev}\big)_{FunDef} \hookrightarrow [id : t_i]\big]\end{array}}{\tau \vdash \big(id, \langle\,\rangle, \langle a_1, \dots, a_n\rangle, \langle t_1, \dots, t_n\rangle, es, s, es', s_{rev}\big)_{FunDef} \hookrightarrow [id : t]}. \tag{64}$$

Here, we have chosen to write the input parameter arrays more compactly as es and es' since they are only being passed along.

For generic functions, we will implement the rule by delaying the type check. An implementation of a type checker that follows these rules exactly would thus only compile generic functions when they are being called with specific input numbers. At first when encountering the function definition, all we do is therefore to record the definition in the environment of the program. The following rule states that we can do so. We have

$$\frac{\begin{array}{c}\vdash d_{fun} = \big(id, \langle id_1, \dots, id_n\rangle, as, ts, es, s, es', s_{rev}\big)_{FunDef}\\ \vdash n \geq 1 \qquad \vdash id \notin \|\tau\|_{all}\end{array}}{\tau \vdash d_{fun} \hookrightarrow [id : d_{fun}]}. \tag{65}$$

The second premise here is used to check that the function indeed has number input, and the third premise is used to check that id is not already declared above. The conclusion shows how the function definition is simply recorded for id instead of recording a specific type. It is thus understood here that $id : d_{fun}$ does not mean "id has the type d_{fun}," but instead means "id defined by d_{fun}." When we encounter id in the context of a program, we should then type check it according to the following rule, which states that

$$\frac{\begin{array}{c}\vdash d_{fun} = \big(id, \langle id_1, \dots, id_m\rangle, as, \langle t_1, \dots, t_n\rangle, es, s, es', s_{rev}\big)_{FunDef}\\ \tau \vdash id : d_{fun}\\ \forall_{i \in \{1,\dots,m\}} \big[\tau \vdash N_i := n_i \hookrightarrow [\,]\big]\\ \vdash \tau' = [id_1 := n_1] \circ \dots \circ [id_m := n_m] \circ \tau\\ \tau' \vdash \big(id, \langle\,\rangle, as, \langle t_1, \dots, t_n\rangle, es, s, es', s_{rev}\big)_{FunDef} \hookrightarrow [id : t]\end{array}}{\tau \vdash id\texttt{<}N_1 \ \dots \ N_m\texttt{>} : t}. \tag{66}$$

Here, we see how we simple evaluate each N_i and then use the previous rules to in a sense compile $id\texttt{<}N_1\ \dots\ N_m\texttt{>}$ in order to get its type. The reason why this works is that by recording that each id_i, which are the number parameters, evaluates to n_i in τ', they will do so in s and s_{rev} as well. Note that the third premise only succeeds if there is no measurement happening in any of the number terms. This means that in this version of the language, number term are not allowed to include `measure` terms when they appear as inputs to a function call.

This concludes the rules for the function declaration and definition (combined). For the type definitions we can do something similar when it comes to generic types. We will see this in a moment but first, we state the rule for non-generic types.

The rule for type definitions when there are no number input parameters can be stated as follows. We have that

$$\frac{\begin{array}{c}\vdash id \notin \|\tau\|_{all} \\ \tau \vdash e\ \texttt{.} : (\,)\,\bar{t}_1 * \texttt{->}\ \bar{t}_2 \\ \tau \vdash \text{span}(\bar{t}^{\,in}) = \text{span}(\bar{t}) \\ \vdash \Delta\tau = [e\ \texttt{.} : (\,)\,\bar{t} * \texttt{->}\ id\,*] \circ [id \subseteq \text{span}(\bar{t})]\end{array}}{\tau \vdash \big(id, \langle\,\rangle, e, \bar{t}\big)_{TypeDef} \hookrightarrow \Delta\tau}. \tag{67}$$

First, we check that the type is not already defined. Then we get the type of e `.` and make sure it has the form $(\,)\,\bar{t}_1 * \texttt{->}\ \bar{t}_2$. This means that e needs to have the type $(\,\bar{t}_2 *)\,\bar{t}_1 * \texttt{->}\ (\,)$. We then check that the original type is a valid input type for e, i.e. that it spans the same type as $\bar{t}^{\,in}$. Finally we record that the $\Delta\tau$ that goes into the conclusion consists of the following two environment changes. The first environment change records that id is now a type and also records what its spanning type is. The second environment change additionally record that e `.` now has the type $(\,)\,\bar{t} * \texttt{->}\ id\,*$. This means that when e `.` from that point on will output a type of $(\,)\,\bar{t} * \texttt{->}\ id\,*$ the type that can be inferred for it from e and from using rule (30).

Note that we have used e for the expression instead of id or $id\texttt{<}N_1\ \dots\ N_m\texttt{>}$ in our rule just above, even though e can actually only be a term of these two types (or potentially also a higher-order function call with only function inputs). Since there is never any data variables live in the global scope of a program, where type definitions take place, we do, however, not need to fear that e is an expression that includes any of these.

When a type definition is generic, we can state a similar rule as for the generic function definition. We can thus state the rule as

$$\frac{\begin{array}{c}\vdash d_{type} = \big(id, \langle id_1, \dots, id_n\rangle, e, \bar{t}\big)_{TypeDef} \\ \vdash n \geq 1 \qquad \vdash id \notin \|\tau\|_{all}\end{array}}{\tau \vdash d_{type} \hookrightarrow [(id, e\ \texttt{.}) : d_{type}]} \tag{68}$$

Here, we thus use $(id, e\ \texttt{.}) : d_{type}$ to denote that id is a type and that e `.` has a type which are both defined by d_{type}. When encountering an expression e' `.`, where e' is an expression that can be obtained by substituting all number parameters in e by number terms, we can use the following rule, given by

$$\frac{\begin{array}{c}\vdash d_{type} = \big(id, \langle id_1, \ldots, id_m\rangle, e, \bar{t}\big)_{TypeDef} \\ \tau \vdash (id, e\ \texttt{.}) : d_{fun} \\ \forall_{i\in\{1,\ldots,m\}}\, \big[\, \tau \vdash N_i := n_i \hookrightarrow [\,]\,\big] \\ \vdash e' = e[N_1/id_1, \ldots, N_m/id_m] \\ \vdash \bar{t}' = \bar{t}[n_1/id_1, \ldots, n_m/id_m] \\ \vdash \tau' = [id_1 := n_1] \circ \cdots \circ [id_m := n_m] \circ \tau \\ \tau' \vdash \big(id, \langle\,\rangle, e, \bar{t}\big)_{TypeDef} \hookrightarrow [e\ \texttt{.}: (\,)\, \bar{t}* \texttt{->}\ id*] \circ [id \subseteq \mathrm{span}(\bar{t})]\end{array}}{\begin{array}{c}\tau \vdash id \texttt{<} N_1\ \ldots\ N_m \texttt{>} \subseteq \mathrm{span}(\bar{t}') \\ \tau \vdash e'\ \texttt{.}: (\,)\, \bar{t}' * \texttt{->}\ id*\end{array}}. \tag{69}$$

This rule can also be used to infer what type $id \texttt{<} N_1\ \ldots\ N_m \texttt{>}$ spans, when such a term is encountered in the program. In the rule, we have used a notation[16] for substituting terms in a compound term, which is for instance seen in $e[N_1/id_1, \ldots, N_m/id_m]$. We thus take this to denote that each id_i is substituted by N_i in the resulting term. Here, the resulting term should then be equal to e', which is the term we have encountered in the program, when we want to use the second conclusion. We also define $\bar{t}'$ to be the type equal to $\bar{t}$ when each id_i is replaced for the constant number n_i that N_i evaluates to. The last premise works in a similar way to the last premise of rule (66), and thus uses τ', where each of the id_is are now defined to their corresponding constant number, in order to check that the resulting non-generic version of the type definition compiles in the right way.

Note again that we do not allow measurements to take place in any of the N_is, at least not in this version of the language. The programmer should therefore in general remember not to use measurement terms directly inside calls to generic functions or in generic types and should instead make any required measurements beforehand.

6 Common program examples to evaluate the language

In this section, we will test the language in a qualitative way by seeing how it works for some common algorithms. Since the project has been focused so much on writing up and explaining the intended semantics in detail via the type checking rules (in an indirect way), any implantation towards a compiler will be left for future work. To analyse the language, we will therefore instead go through examples of algorithms and discuss how well this language can be used to implement them.

For a fair analysis, we should try not to choose algorithms specifically because we expect them to work well for the language. Instead, we should try to choose a representative set of algorithms and with a good amount of variation between them. As a way to try to do this, I have chosen some of the most common algorithms that I know from literature, that are also not too complicated. The most complicated algorithm we will see below is thus Grover's algorithm. This algorithm is explained in Nielsen and Chuang [1], and so is the rest of the algorithms I have chosen. Since Nielsen and Chaung [1] is an introductory text, the algorithms should showcase several aspects of quantum computing and should thus be varied. The specific algorithms that I have chosen are the following five. First we will take a look at an implementation of the teleportation circuit, then the Deutsch–Jozsa algorithm, the Fourier transform, which we have also seen previously, the phase estimation procedure and finally Grover's algorithm.

What the algorithms do and how they work will not be explained as we go through the examples. See for instance Nielsen and Chuang [1] for a description of them. Instead we will just discuss the

[16]See Harper [15].

aspects of how well the language does in terms of implementing them.

For the teleportation circuit, we agian need the CNOT gate. We should therefore first define the function implementing it. Recall that this can be done by the following lines of code.

```
CNOT :: bool -> bool* -> ()
CNOT a b {
    if (a) {
        Not b;
    }
}
```

The teleportation circuit can then be implemented as follows.

```
Teleport +: ()bool*  -> bool
Teleport +: ()xspin* -> xspin
Teleport +: ()yspin* -> yspin
Teleport x {
    // initialize and prepare b1 and b2 in the bell_00 state.
    qbit[2] (b1, b2) = new false[2];
    Had b1;
    CNOT b1 b2;
    // entangle bell state with the input state.
    CNOT x b1;
    Had x;
    // measure x and b1 to determine gates on b2.
    let p = measure b1 in {
        if (p) {
            Px b2;
        }
    }
    let q = measure x in {
        if (q) {
            Pz b2;
        }
    }
    return b2;
}
Teleport! x {
    return Teleport x;
}
```

Here, we have chosen to give it a type signature for all the basic types, since the teleportation circuit just implements a complicated version of the identity gate. We use the `+:` lexeme to be able to define the reversal of the function at the bottom. Since the teleport circuit is its own inverse, we can just call it for the reversed implementation as well. On the positive side, we see that the classical control syntax of the last part of the implementation does well to let us be able to make the gates at the end be controlled by the measured qubits. On the negative side, we were not able the algorithm more understandable in a classical sense. This is as expected since the algorithm has no real classical analog to what is going on. There is a much more important issue, which is not directly apparent by the program, but is an issue that I only noticed after having written it. I have used `+:` to assume that the function is reversible, which it normally is. But note what happens when the function is then controlled. Since we cannot make controlled measurements, the function is then nonsensical. Of course we have already dealt with this fact by not allowing measurements and garbage production inside if statement. But by allowing users to suppress the `-->` arrow and the `+ garbage` qualifier, they might be tempted to define functions such as this one, which are nonsensical when they are controlled. There are three solutions to this. We can either remove the possibility to change the `-->` arrow and to remove the `+ garbage` qualifier as the first option, or we could just let the options remain and let the program fail measurements are controlled as the second option. The third option would be to find a way to define how a function should be controlled in a similar way as how we can now define how it is reversed. The reason why I bring the third option up, is that for an error correction procedure, there is a way to control the error-corrected gate in a meaningful way. This would, however, complicate the language further and might not be worth the cost. I will therefore suggest removing the option to change the `-->` arrow and to remove the `+ garbage` qualifier for a future version of the language.[17] I have also come to think that the option to define the reversal of the function should be removed for a future version. I still like the idea that almost-reversible gates might be assumed to be reversible. However, after having thought more about the issue, I have come to the conclusion that the assume statements of the language should already be sufficient for expert users to be able to achieve this.

Let us move on to the Deutsch–Jozsa algorithm. It is an algorithm to test a Boolean-valued function that is known to be either balanced or constant on its domain and to find out which one it is. First let us define the following function to make a series of Hadamard transformations of all its input qubits.

```
Hadamards<n> :: (xspin[n]*)bool[n]* -> ()
Hadamards<n> :: (bool[n]*)xspin[n]* -> ()
Hadamards<n> a {
    for i = 1 to n {
        Had a[i];
    }
}
```

We will not need the second type signature here. It is simply included it for the sake of symmetry. With this auxiliary function defined, we can define the following function to impliment the Deutsch–Jozsa algorithm.

```
1   isBalanced<n> :: (bool[n] -> bool) --> bool + garbage
2   isBalanced<n> f {
3       xspin[n] x[1:n] = Hadamards<n> . new 0;
4       xspin y = Had . new 1;
5       //
6       temp bool a = f x[1:n];
7       if (a) {
8           Not y;
9       }
10      restore;
11      // assert that y will that y will have qbit type by now.
12      assert y of qbit;
13      // make another Hadamard tranformations of the xs.
14      Hadamards<n> x[1:n];
15      // measure y to see if the output is 0 or not.
16      let m = measure x[1:n] in {
17          if (m == 0) {
18              return new false;
19          } else {
20              return new true;
21          }
22      }
23  }
```

With this implementation, we see that it is easy enough to manage the generic functions for the language. The temp statement provides a straightforward way to XOR the the function value coming from `f` with `y`, which we are supposed to do as part of the algorithm. A programmer would here have to reason about ancilla qubits when using a lower-level language, and would have to write detailed explanations in order for a reader to follow what is going on, even when aiming the documentation towards readers familiar with said language. Here, lines 6–10 takes care of the matter in a simple way, and it is not very difficult to see what is going on. Note that the assert

[17]The reason why I have not done so for this report, is that I think the discussion is still interesting, and I also prefer not to change the language in the process of conducting the final evaluation of it.

statement does nothing other than tell the reader that we have lost track of the type of y and that y might be entangled with other variables at that point.

The next example is the quantum Fourier transform. We have already seen how this algorithm be implemented with the following program.

```
RotateByFloat<n> :: bool[n] -> (qbit*)xspin* -> ()
RotateByFloat<n> a[1:n] x {
    for i = 1 to n {
        if (a[i]) {
            Rz<i + 1> a0;
        }
    }
}

QFT<n> :: (qbit[n]*)bool[n]* -> ()
QFT<n> a[1:n] {
    for i = 1 to n - 1 {
        Had a[i];
        RotateByFloat<n - i> a[i + 1 : n] a[i];
    }
    Had a[n];
}

type FTInt<n> := QFT<n> . bool[n]
```

This example shows how we can define new types in the language using reversible functions like QFT<n>. It also shows how auxiliary functions can be used to make the program easier to reason about. We have talked about how single spinors have a classical analog to a spinning ball in three dimensions. With this analog, RotateByFloat<n> has a simple classical analog as well as it can be interpreted as rotating its input spinor by an amount equal to the number value held by the n-bit register of the first input. This interpretation is then aided by the name we have chosen for the function, and also further aided by the type signature. The reader might agree that if we had not had the different types such as bool and xspin for the type signature, but only had for instance qbit, we would probably need a longer function name to convey the same meaning.

Moving on to the phase estimation procedure, we can implement it in a fairly simple way by using two higher-order, auxiliary functions. It should be noted that this implementation might be very ineffective and a programmer might (depending on how effective the compiler is) have to do more work to bring the running time of the procedure down. But for our purposes, it works as a good example of how the phase estimation procedure can be implemented. First we define a recursive function as follows to take a gate 2^n times on the input.

```
powerGate<n> :: (qbit[n]* -> ()) -> qbit[n]* -> ()
powerGate<n> :: f x {
    if (n == 0) {
        return f x;
    } else
        return powerGate<n-1> f . powerGate<n-1> f . x;
    }
}
```

Then we can define a function that applies the input gate a number of times equal to the integer value held by its second input. This can be defined as follows.

```
ApplyMultipleTimes<n> :: (qbit[n]* -> ()) -> bool[n] -> qbit[n]* -> ()
ApplyMultipleTimes<n> f a[1:n] x {
    for i = 0 to n-1 {
        if (a[n-i]) {
            powerGate<i> f x;
        }
    }
}
```

Finally, we can implement the phase estimation procedure as the following.

```
phaseEstimation<n> :: (qbit[n]* -> ()) -> qbit[n]* -> qbit[n]
phaseEstimation<n> f x {
    qbit[n] a = Hadamards<n> . new 0;
    ApplyMultipleTimes<n> f a x;
    QFT<n>! a;
    return a;
}
```

Here, we have let it be up to the user of the function to select an input that contains a sought-for eigenvector as part of its superposition, and we also let it be up to the user to measure the returned value afterwards. We see that the `!` operator was handy in this example to reverse the quantum Fourier transform. The use of the dot operator for `powerGate` also made it easy to string the functions together in a single expression for the recursive calls.

The last example is Grover's search algorithm, which finds the unique input for which an oracle function returns true. As part of the algorithm, we need a procedure to flip the phase of the wave function only when the input is correct. This can be implemented as follows.

```
FlipPhaseIfTrue<n> :: (bool[n] -> bool) -> ~bool[n] -> ()
FlipPhaseIfTrue<n> f a[1:n] {
    temp bool b = f a[1:n];
    if (all a[1:n]) {
        // since all a[1:n] have the ~ qualifier, we can do:
        Pz b;
    }
    restore;
}
```

Here, we see an example where the ∼ qualifier aids the understanding of the function. The input type of `bool[n]` (without any `*` qualifier) shows that the input does not change its state (when viewed the z-basis), but will only potentially have its phase changed. From the type checking rules, we can see how the type signature will succeed for the function. We will also need the following function.

```
isZero<n> :: bool[n] -> bool
isZero<n> a[1:n] {
    bool b = new false;
    temp {
        for i = 1 to n {
            Not a[i];
        }
    }
    if (all a[1:n]) {
            Not b;
    }
    restore;
    return b;
}
```

This function returns true if the input only holds qubits with the value false. We will use this function to flip the phase only for the zeroth state, which is an important part of Grover's algorithm. The full algorithm can then be implemented as follows.

```
groverSearch<n> :: (bool[n] -> bool) --> bool[n]
groverSearch<n> f {
    // prepare superposition of all numbers from 0 to 2^n-1.
    bool[n] a[1:n] = new false[n]
    Hadamards<n> a[1:n];
    // declare local function to flip the phase if input is zero.
    fun FlipPhaseIfZero = FlipPhaseIfTrue<n> isZero<n>;
    // make sqrt(n) Grover iterations.
```

```
 9      for i = 1 to sqrt(n) {
10          // use oracle function f on superposition state to flip the
11          // phase for computational states when f is true.
12          FlipPhaseIfTrue<n> f a[1:n];
13          // use FlipPhaseIfZero transformed unto the xspin[n]-
14          // basis (by having Hadamards on both sides).
15          Hadamards<n> a[1:n];
16          FlipPhaseIfZero a[1:n];
17          Hadamards<n> a[1:n];
18      }
19      // measure a[1:n] (which all now have qbit types), and use result
20      // to determine whether algorithm needs to be repeated.
21      let m = measure a[1:n] in {
22          // revive a[1:n] for this scope according to their measured
23          // values.
24          bool[n] a[1:n] = new m;
25          // measure f a[1:n] to decide if the output is correct.
26          let p = measure f a[1:n] in {
27              if (p) {
28                  return a[1:n];
29              } else {
30                  discard a[1:n];
31                  return groverSearch<n> f;
32              }
33          }
34      }
35  }
```

Here, we have expanded the algorithm to include a check afterwards to see that we have found the right input (which might not be a certainty). By including this, we also get to see an example where the same function can be used both as a quantum circuit and as a classical procedure. If we look at lines 24–33, we see that there is actually no need to initialize `a[1:n]` as a qubit register if `f` is instead implemented classically. It is not a difficult task for a compiler to see this, and the last part will therefore likely be implemented most efficiently as a classical procedure on a hybrid computer when the program is compiled. For the Grover search, however, the algorithm can only be implemented as a quantum circuit. The fact that the oracle functions is written with the same syntax regardless of whether the programmer thinks of them as quantum or classical procedures means that the compiler is free to choose this way. Note that the discard statement at the end does not actually change the semantics of the program since `a[1:n]` are local variables. It is simply used to tell the compiler, as well as any human reader, that `a[1:n]` is no longer used from that point on. Note also that we have used the fun statement to define `FlipPhaseIfZero` as a local function for the Grover search. I also feel obligated to mention that i actually only realized that a square root function might be a good idea to have for the language when i wrote this program. I therefore added it as an afterthought.[18]

7 Discussion

Throughout the report we have already discussed a couple of different points where the language might change in future versions. One of the points that has to be decided is whether to allow for measurements directly in calls to generic function or to keep it this way. It is also still a bit open how some of the arithmetic operators should be defined such as the `%` operator and `sqrt` function. There is of course also the possibility of adding more arithmetic operations, just like I added `sqrt` in order to be able to implement Grover's algorithm above, without having to write a square root function myself. At some point, however, it is better to let the users have to define

[18]This of course goes a little bit against what I have written in the previous footnote, but I see it as an important, yet simple addition to the language.

classical operations themselves as functions, so that they can be implemented both classically and quantum mechanically.

An important point to consider as well is how to prevent users from assuming that gates are reversible, which they might be on their own, when forgetting that the gate might be nonsensical when controlled by other qubits such as we saw for the teleportation circuit above. The easy thing to do is to simply remove the option to suppress the parts of the type signature that would otherwise signal that measurements take place or that garbage is generated. Another option would, as mentioned, be to include the possibility of defining what should happen when the function is controlled, but this would be too complicated in my opinion. If the option is kept to declare a function as reversible even though measurements take place, then users should at least be warned that they should only use it for algorithms where the measurements are very likely to always yield the same outcome (within the desired precision). The users can, however, still use the assume statement to terminate such qubits that are almost untangled from the rest either way. As these two different approaches should not matter to the overall precision of program, as far as I can see, it would be best in my opinion for future version just to remove the option for declaring the reversal of functions as well as for assuming no garbage and no measurements. Expert users should then still be able to rely on the assume statements to achieve what they want.

In a previous version of the language I allowed for non-Boolean types for condition variables in if statements, as long as these and all other variables in the if statements were given their corresponding spanning type afterwards. The choice to only allow Boolean types makes the semantics more clear in my opinion. It does force the programmer to write more auxiliary functions in some cases in order to control qubits in superpositions. This can be seen minus for the language, but on the other hand, such functions often has a simple interpretation, as we saw for `RotateByFloat<n>` above, and it therefore often helps the readability to separate them as individual functions, at least as far as I can see. At the same time, a programmer can quite easily define a generic, higher-order function to implement a more relaxed version of the if statement.

While we saw some good examples in the previous section of the benefits and shortcomings of the language, it is not nearly enough in terms of a usability test of the language. It is not nearly enough to have the developer of the language test it by himself, such as was the case for the previous section. Once a version of a compiler has been implemented, a real test of it should thus include other users. The experience of such users, and particularly their familiarity with quantum mechanics, might be taken to vary for such a test, but they should all have the thing in common that are not me. Apart from testing the usability of the language, the compiler should also be tested to see that programs compile as expected. Here, it is a good idea to make sure to test all individual features of the language as well making sure to test a good variety of programs that users might realisticly want to implement. Both the tests for the features, i.e. the unit tests, and the test of the realistic programs should be done both for common and for extreme cases. Any issues found in such test might simply be due to errors for the relevant compiler, but they might also uncover potential issues with the design of the language as well as with how I have planned the type checking through the type checking rules.

8 Conclusion

In this project, I have developed and introduced a design of a high-level programming language for quantum computers where quantum procedures and basic classical procedures are both written with the same syntax. The language uses dynamic types to represent different Hilbert space bases as a way to make it easier to reason about the semantics of the functions of the language and to add more safety to it. A special Boolean type is used in particular when the semantics of the relevant statements can be interpreted as both classical operations as well as quantum mechanical ones. Furthermore, when a function is seen to only use these Boolean types, it gives the compiler an easy way determine that these functions can potentially be compiled as classical procedures. In this report we have seen how the language is defined in detail and how the type checking should be implemented. We have then evaluated the language by looking at some common examples of programs. The language seemed to perform very well for these examples, at least in my opinion. The only negative thing to report thus far is that I should probably not have added the feature to define the reversal of functions and should not have added the possibility to make assumptions about the function signatures, except when using the assumption statements within a function definition.

For future work, a compiler should be implemented for the language. This compiler should implement the type checking described in this report as well as be able to translate the statements into a circuit-oriented language. This compiler can then be used to test the usability of the language as well as the correctness and soundness of how it is detailed in this project. For these tests, it might be beneficial to compare to other languages such as Quipper. Apart from this, I have also yet to find a name for the language. Finding a name for it will thus also be left up for future work.

9 Acknowledgements

I would like to thank my supervisor for this project, Michael Kirkedal Thomsen, for giving me this delightful task and for all the help during the project. I would also like to thank Robin Kaarsgaard for assisting the supervision sessions in the beginning of the project and for helping to explain the problem.

References

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th Anniversary Ed. (Cambridge University Press, 2010).

[2] D. J. Griffiths, *Introduction to Quantum Mechanics*, Second Ed. (Pearson Education Limited, 2014).

[3] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger and B. Valiron, *Quipper: A Scalable Quantum Programming Language*, arXiv:1304.3390 [cs.PL].

[4] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger and B. Valiron, *An Introduction to Quantum Programming in Quipper*, arXiv:1304.5485 [cs.PL].

[5] T. Altenkirch and J. Grattage, *A functional quantum programming language*, arXiv:quant-ph/0409065.

[6] B. Ömer, *Quantum Programming in QCL*, Retrieved from `http://tph.tuwien.ac.at/~oemer/qcl.html#doc`.

[7] P. Selinger, *Towards a Quantum Programming Language*, (Cambridge University Press, 2004).

[8] A. W. Cross, L. S. Bishop, J. A. Smolin and J. M. Gambetta, *Open Quantum Assembly Language*, arXiv:1707.03429 [quant-ph].

[9] R. S. Smith, M. J. Curtis and W. J. Zeng, *A Practical Quantum Instruction Set Architecture*, arXiv:1608.03355 [quant-ph].

[10] D. S. Steiger, T. Häner and M. Troyer, *ProjectQ: An Open Source Software Framework for Quantum Computing*, arXiv:1612.08091 [quant-ph].

[11] S. J. Gay, *Quantum Programming Languages: Survey and Bibliography* (Cambridge University Press, 2006).

[12] S. A. Cuccaro, T. G. Draper, S. A. Kutin and D. P. Moulton, *A new quantum ripple-carry addition circuit*, arXiv:quant-ph/0410184.

[13] T. G. Draper, *Addition on a Quantum Computer*, arXiv:quant-ph/0008033.

[14] T. Yokoyama, H. B. Axelsen, R. Glück, *Principles of a Reversible Programming Language*, In Proceedings of the 5th conference on Computing frontiers, 2008.

[15] R. Harper, *Practical Foundations for Programming Languages*, v1.32 of 05.15.2012.