A Dynamic Intermediate Representation for Hybrid Quantum-Classical Programs

Alex Rice
University of Edinburgh
United Kingdom
alex.rice@ed.ac.uk

Chris Heunen
University of Edinburgh
United Kingdom
chris.heunen@ed.ac.uk

Tobias Grosser
University of Cambridge
United Kingdom
tobias.grosser@cst.cam.ac.uk

Abstract

Quantum compilers typically follow the circuit model, representing programs as fixed sequences of gates. This static view breaks down in hybrid quantum-classical applications, where gate choices depend on runtime data or measurement results. We introduce a new Intermediate Representation (IR) that elevates gates to first-class values, enabling their dynamic creation, composition, and control. This unified representation allows classical computation to steer quantum behaviour, capturing phenomena including stochastic gate selection, adaptive error correction, and measurement-driven computation within a single framework. Case studies in noise modelling, randomised compilation, error correction, and measurement-based quantum computing show that our IR expresses these programs compactly and supports optimisations that were not possible in the circuit model. Evaluation on a benchmark suite of hybrid quantum-classical programs indicates that our IR represents programs compactly and facilitates compiler analysis and transformation.

1 Introduction

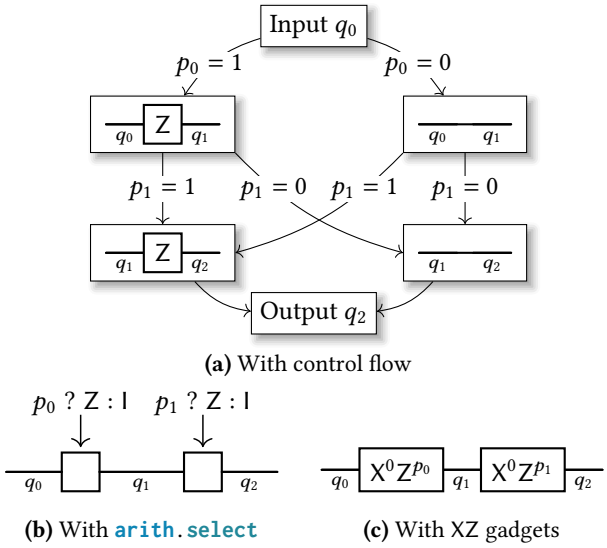
Quantum circuits are the dominant representation of quantum computations [13]. However, the circuit model is inherently static and too restrictive for many important applications, which are more naturally expressed as hybrid quantum-classical programs combining quantum operations with classical data and control flow [8, 35, 38, 40]. Established quantum toolkits have begun extending support for classical computation within quantum compilers, such as IBM's OpenQASM 3.0 [10] and Quantinuum's t|ket⟩ 2 [43], emphasising the importance of representing subcircuits that depend on external classical or probabilistic computations.

Consider noisy quantum channels, which are essential for modelling real hardware [36]. The phase-flip channel, a common form of noise, probabilistically applies a Z gate to a qubit. As the choice of gate depends on a classical random variable, the circuit model can only represent this behaviour by generating a fresh circuit for each random sample.

A more general representation of quantum computation should support optimisation across the quantum-classical boundary, be expressive enough to encode diverse hybrid techniques, and remain simple enough for efficient compilation and transformation. Unlike classical hardware—like in an ASIC or FPGA, where circuit structure is fixed at runtime [29, 44]—quantum programs are executed dynamically, allowing circuit layout to change during execution. Thus there is no fundamental reason to adhere to a static circuit model, and instead we can adopt a dynamic representation better suited for optimising hybrid programs.

In this work, we introduce *dynamic gates*, a representation for hybrid quantum-classical computation that treats gates as *runtime values*. Rather than defining a separate operation for each quantum gate, a gate value can be created, manipulated by classical code, and applied dynamically during execution.

Listing 1 illustrates this concept using two consecutive phase-flip noise channels. The key construct, `qssa.dyn_gate`,



(a) With control flow

(b) With `arith.select`

(c) With XZ gadgets

Figure 1. Graphical representations of two sequential phase flip channels using control flow in (a), and dynamic gates in (b) and (c), with the gate value being generated by `arith.select` and our XZ gadget respectively. The dataflow is vastly simplified in (b) and (c), where the quantum component retains its circuit structure. The XZ gadgets in (c) remove the need for the selection operation, allowing the dynamic gates to be fused.

Authors' Contact Information: Alex Rice, University of Edinburgh, United Kingdom, alex.rice@ed.ac.uk; Chris Heunen, University of Edinburgh, United Kingdom, chris.heunen@ed.ac.uk; Tobias Grosser, University of Cambridge, United Kingdom, tobias.grosser@cst.cam.ac.uk.

Listing 1 Our dynamic gates can represent two concatenated probabilistic phase flip channels without requiring control flow. Quantum-classical transformations can then reduce this to a single phase flip with updated probability, using the self-invertibility of the quantum Z gate.

```
%id  = gate.constant #gate.id
%z   = gate.constant #gate.z
%p_1 = prob.bernoulli 0.1
%g_1 = arith.select %p_1, %z, %id : !gate.type<1>
%q_1 = qssa.dyn_gate<%g_1> %q
%p_2 = prob.bernoulli 0.1
%g_2 = arith.select %p_2, %z, %id : !gate.type<1>
%q_2 = qssa.dyn_gate<%g_2> %q_1
```

takes as input a dynamically chosen gate value `%g`, determined by a classical selection that yields a Z gate if the random variable `%p` is true and the identity otherwise. In contrast to traditional circuits, the gate applied by `qssa.dyn_gate` is only determined at runtime.

Dynamic gates make the boundary between quantum and classical computation porous, enabling optimisations that cross this boundary while preserving the familiar circuit structure. They can be viewed as a natural generalisation of the circuit model, and can replace instances of more explicit control flow necessary in other IRs [10, 37].
Our contributions are:

- A *dynamic gate model* for tightly coupled hybrid quantum-classical programs, instantiated as a compiler Intermediate Representation (IR) that enables optimisations across the quantum-classical interface (Section 2, Section 4).
- The *XZ gadget*, a dynamic gate representing conditional combination of X and Z Pauli gates allowing an elegant way to realise optimisations (Section 4.1).
- A curated *benchmarking suite* of hybrid programs that is representative of current demands on quantum-classical interaction.

We show that this IR advances the state of the art in representing and optimising hybrid quantum-classical programs in two ways (Section 5):

- Depth: Through case studies in *randomised compilation* (Section 5.1), *quantum error correction* (Section 5.2), and *measurement-based quantum computing* (Section 5.3), we demonstrate how complex hybrid transformations become simple local rewrites.
- Breadth: Through evaluation on our new benchmark suite, covering nontrivial quantum-classical interactions and scalable program sizes, we demonstrate that our IR compactly expresses hybrid programs and enables comparative analysis across representations (Section 5.4).

2 Dynamic Gates

Designing IRs for classical computation is a well-studied problem, as is designing IRs to represent the quantum circuit model. A naive union of a classical IR and quantum circuit IR can therefore represent both quantum and classical computation, but struggles to model the interaction between the two components, which is integral for many modern quantum computing applications [8, 35, 38, 40].

An example of an optimisation that crosses the boundary between the classical and quantum components of the program is fusing consecutive quantum phase flip channels (Figure 1). In our dynamic gate model this can be represented as a single circuit, whereas it becomes a complicated branching of circuits when mixing classical and quantum IRs.

A phase flip channel acts on a single qubit, and is equivalent to conditionally applying a Z gate to the qubit with some probability p. Phase flip channels may not immediately appear to be hybrid quantum-classical programs, but representing them requires quantum operations to depend on the result of classical computation, in this case the instantiation of a random variable. Thus they demonstrate the tools we have developed well.

Performing two such channels in sequence is equivalent to performing a single one (with an updated probability): with probability $(1-p)^2$ no Z gates are applied, with probability p^2 two Z gates are applied which cancel out, and with probability $2p(1-p)$ a single Z gate is applied. Such a transformation should be simple to perform in a compiler, but note that the necessary transformation is not purely quantum or purely classical. Thus, how easy it is to perform such a simplification depends on the quantum-classical interface exposed by the IR.

One way to represent the channel is to use classical control flow, either as blocks of computation linked by branching operations, or a more structured `if` conditional statement (Figure 1a). Such an approach is not ideal for the transformation described above, as performing it would require reasoning across multiple control flow blocks. Furthermore, this representation obscures the quantum dataflow of the program, because the same qubit can be used in multiple execution branches. This complicates simple analyses. For example, to check if a qubit is used linearly, one must check that no execution trace uses the qubit twice. Verifying this property is undecidable in general and common over-approximations require global control flow analysis.

Instead, the dynamic gate model bridges the classical and quantum components of a program. The fundamental primitive is the *dynamic gate* operation. Unlike the usual representation of gates in a quantum circuit, the gate to be run is supplied by an operand, instead of a static piece of information. This input can be the result of arbitrary computation, providing the interface for classical data to affect quantum

```
%false = arith.constant false
%p_1 = prob.bernoulli 0.1
%g_1 = gate.xz %false, %p_1
%q_1 = qssa.dyn_gate<%g_1> %q
%p_2 = prob.bernoulli 0.1
%g_2 = gate.xz %false, %p_2

%q_2 = qssa.dyn_gate<%g_2> %q_1
```

(a) Double phase flip with XZ gadgets.

```
%false = arith.constant false
%p_1 = prob.bernoulli 0.1


%p_2 = prob.bernoulli 0.1
%p_3 = arith.xori %p_1, %p_2 : i1
%g   = gate.xz %false, %p_3
%q_2 = qssa.dyn_gate<%g> %q
```

(b) Double phase flip after fusion.

Figure 2. The XZ gadget allows the phase flip channel to be represented without any selection operations (a). This allows the consecutive dynamic gates to be fused, reducing the program to a single phase flip channel (b).

computation. We refer to operands that specify quantum gates as *gate values*.

Listing 1 represents two sequential phase flips in the dynamic gate model using our IR. The gate value is obtained from a selection operation on a randomly generated boolean. This shows the strength of the dynamic gate model: no complexity is added to the quantum dataflow by its interaction with classical dataflow. The output qubit from the first phase flip passes directly to the input of the second phase flip. Essentially, the quantum component of the program retains the structure of a quantum circuit (Figure 1b).

The dataflow is simpler, but two classical choices of quantum gates still need to be fused. In this small-scale example a full case analysis is possible, but would quickly become intractable in general. To remedy this we introduce the *XZ gadget*, a succinct way to represent conditional combinations of Pauli X and Z gates. So far, all gate values have been constant or classical selections between other constant gate values, but the dynamic gate model can be extended by further operations for generating gate values. Therefore the XZ gadget is implemented as a fresh operation that takes two boolean values x and z as operands and produces a gate value representing the gate $\mathsf{X}^x\mathsf{Z}^z$.

The full utility of XZ gadgets derives from the transformations which apply to it. A selection operation between two XZ gadgets can be replaced by a single XZ gadget with selection operations on each of its operands. Following this transformation with simple arithmetic simplifications reduces the double phase flip to the representation in Figure 2a, as illustrated in Figure 1c.

Moreover, XZ gadgets can be easily fused together by taking an exclusive disjunction of their parameters, which preserves the semantics of the gate value up to global phase. This reduces our example to a single phase flip channel, given in Figure 2b. As the XZ gadget considers X gates in addition to Z gates, the same reduction sequence can similarly simplify compositions of bit-flip and depolarising channels. Section 5 demonstrates that this gadget applies much more widely than the scenario presented here, because conditional Pauli operators are pervasive in quantum computing.

3 Background

Now that the key idea of the paper has been introduced, let us step back to recall relevant background. We briefly review compilation, especially in the context of the MLIR framework, and the relevant parts of quantum computation.

Quantum computing

The basic data in quantum computing is carried by *qubits*. The state of a one-qubit system is represented by a vector in $\mathbb{C}^2$. Qubits subsume bits, with 0 and 1 being represented by the unit vectors $|0\rangle := \left(\begin{smallmatrix}1\\0\end{smallmatrix}\right)$ and $|1\rangle := \left(\begin{smallmatrix}0\\1\end{smallmatrix}\right)$.

Computations on qubits are represented by *unitary* linear maps $\mathbb{C}^{2^n} \to \mathbb{C}^{2^n}$, that is, invertible matrices whose inverse is equal to their conjugate transpose. Especially, the *Pauli gates* X, Y, and Z satisfy some useful relations, such as

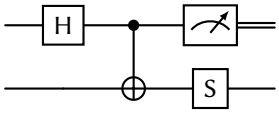
$$\mathsf{X}^2 = \mathsf{Y}^2 = \mathsf{Z}^2 = -i\mathsf{XYZ} = \mathsf{I} = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$$

and $\mathsf{ZX} = i\mathsf{Y} = -\mathsf{XZ}$.

Multiple qubits can be combined to form a register of qubits using *tensor product*: the state of an n-qubit system is represented as a vector in $\mathbb{C}^2 \otimes \cdots \otimes \mathbb{C}^2 = (\mathbb{C}^2)^{\otimes n} \simeq \mathbb{C}^{2^n}$. Important two-qubit gates are the controlled X and Z gates, denoted CX and CZ. These two gates, along with the Pauli gates, the Hadamard gate H, and the phase gate S are known as *Clifford* gates, the gates that map Pauli gates to Pauli gates under conjugation. For example, S satisfies the following equations:

$$\mathsf{ZS} = \mathsf{SZ} \qquad \mathsf{XS} = -\mathsf{SY} \qquad \mathsf{YS} = \mathsf{SX}$$

Gates can be combined into *quantum circuits*. They can be combined in parallel (with tensor products) and in sequence (with matrix multiplication). Quantum circuits form the *de facto* standard hardware-agnostic low-level language for quantum computing. They are usually drawn graphically. For example, the quantum circuit



represents a 2-qubit operation that performs an H gate on the first qubit, a CX gate on the first and second qubits, and then an S gate on the second qubit.

The last operation on the first qubit is a *measurement*, used to convert qubits into bits. This is a probabilistic operation. When measuring a qubit in state $\alpha v_1 + \beta v_2$ with respect to a basis $\{v_1, v_2\}$, the resulting bit will be 0 with probability $|\alpha|^2$, and 1 with probability $|\beta|^2$. Two states u and v are said to differ by a *global phase* if $v = e^{i\theta}u$ for some angle θ. Such states cannot be discriminated by measurement, and are hence identified.

To aid optimisation, quantum compilers often allow quantum circuits to include *quantum gadgets* in addition to gates. These gadgets typically represent commonly occurring subcircuits or patterns within circuits, but are easier to simplify than the equivalent sequences of primitive gates. A representative example are *phase gadgets* [9]; optimisation proceeds by identifying subcircuits that correspond to these phase gadgets, fusing adjacent gadgets together, and finally lowering each resulting gadget to primitive gates.

For many applications of quantum computing (see for example our case studies and benchmark suite in Section 5), we must move beyond statically defined quantum circuits to *hybrid* quantum programs. These hybrid programs include both classical and quantum computation, and can decide which quantum operations to perform at runtime, possibly depending on the results of measurements. It is not always trivial to translate optimisations from the simpler circuit setting to hybrid programs, and the hybrid setting admits new optimisations, especially those that concern both the quantum and classical components of the program.

For more information on quantum computing, we refer the reader to [33, 48].

Compilers

A compiler converts a computation from one representation to another, typically from a higher-level program to some form of machine code. The term compiler is often used to refer to an optimising compiler, which attempts to minimise various metrics of the target code, such as program size or execution time. Instead of performing a direct translation, compilers often transform a program into different IRs, each of which eases performing certain optimisations or analyses.

It is common for IRs to contain programs in Static Single-Assignment (SSA) form [41], where each identifier is assigned a value in exactly one location. This makes dataflow explicit, as it is trivial to determine the origin of any value and to obtain a list of its uses.

To put a quantum gate in SSA form, it should consume the qubits it acts on, producing new fresh qubits as output. We contrast this *value semantics* view of qubits to the *reference semantics* view of qubits commonly used in existing quantum IRs, such as QASM and QIR, where a quantum gate has no outputs and is understood to be mutating its input qubits. The value of SSA form for optimisation is immediate; consider an optimisation that cancels gates with their inverses. To apply this to a single-qubit gate U with input q, it suffices to examine the unique origin of q. Under reference semantics, q may have been mutated by any number of previously applied gates, complicating this analysis.

The IR introduced in this paper is specifically presented as a set of MLIR [28] dialects, implemented on top of xDSL [16], a Python clone of MLIR sharing the same textual representation. Dialects in MLIR allow the user to add new types and operations to craft their own intermediate representations, while still having access to core data structures and optimisation machinery.

All IR snippets in this paper follow MLIR syntax in style. Consider the following operation.

```
%0 = arith.select %p, %lhs, %rhs : !qu.bit
```

The names `arith` and `qu` are names of dialects, which contain `select` and `bit` respectively, with the exclamation mark denoting that `!qu.bit` is an MLIR type. Any symbol beginning with `%` is an SSA value, with operands to the left of the equality symbol and outputs to the right. This operation takes a boolean input `%p` and two inputs (`%lhs` and `%rhs`) of the given type, and returns the first if `%p` is true and the second otherwise.

Extra data can be attached to MLIR operations via attributes. Any symbol preceded by a `#` is an attribute, for example `#gate.id` in Listing 1.

4 A Hybrid Quantum-Classical IR

The IR represents quantum operations in SSA form, with qubits having value semantics. It is realised as a set of MLIR dialects. For completeness, we include the types, attributes, and operations used in the paper and the benchmark suite (see Section 5.4) in Appendix A.

In contrast to previous SSA-based quantum IRs [22, 31, 34], we do not introduce a new operation for each quantum gate, but instead add a single operation `qssa.gate`, for executing (static) gates. The specific gate to be run is supplied by a *gate attribute*, a static piece of compile-time data. Gate attributes specify the number of qubits they act on, enabling verification of the operands and outputs to `qssa.gate`.

Specifying gates via attributes has multiple advantages. Users can easily extend the gates available by introducing custom gates through new dialects. Gates added in this way automatically cooperate with `qssa.gate` and its transformations. Furthermore, the set of available gates can be reused between different applications. Our implementation uses this to define a `qref` dialect in parallel to the `qssa` dialect, which represents quantum circuit operations with reference semantics instead of value semantics. Available gates can be used in both dialects, and transformations are given between the two dialects without the need to inspect the gate attribute.

Table 1. Common gates as XZ or XZS gadgets. The `convert-to-xzs` compiler pass converts constant gate values of gates in the right hand column to XZ(S) gadgets. We assume `%false` holds the constant boolean false, and `%true` holds the constant true.

Gate	Gadget Representation
I	`gate.xz %false, %false`
X	`gate.xz %true, %false`
Y	`gate.xz %true, %true`
Z	`gate.xz %false, %true`
S	`gate.xzs %false, %false, %true`
$S^\dagger$	`gate.xzs %false, %true, %true`

Most importantly, the various gate attributes can be reused to build our dynamic gate infrastructure. The dynamic gate operation `qssa.dyn_gate` (and its corresponding `qref` operation) takes the gate to be executed as an operand instead of an attribute. This operand must have type `!gate.type<n>` for some n, the type of n-qubit *gate values*, runtime values storing an n-qubit quantum gate.

Gate values are primarily generated with `gate.constant`, which creates a gate value from a gate attribute. Passing a constant gate value directly into a dynamic gate operation is equivalent to applying the corresponding static gate operation, and we give this simplification as a canonicalisation pattern on each respective dynamic gate operation.

As shown in Listing 1, combining constant gate values with classical selection operations already allows more expressivity than the circuit model. Applying a classically selected gate value is equivalent to branching with an `scf.if` operation. The compiler pass `lower-dyn-gate-to-scf` performs this transformation. Further lowering to basic block control flow gives a representation that is compatible with existing representations – such as QIR [37], which inherits LLVM's control flow.

Parallel with dynamic gates, the IR contains a similar system for quantum measurements: a static measurement operation, `qssa.measure`, takes a measurement attribute; and a dynamic measurement operation, `qssa.dyn_measure`, takes a *measurement value* as an operand. In contrast to dynamic gates, a measurement attribute can be omitted, defaulting to a measurement in the computational basis.

The IR also adds tools for working with angles (represented as multiples of π quotiented by $2\pi = 0$), and operations for generating (classical) randomness.

4.1 Gadgets for X, Z, and Phase (S) Gates

Non-constant gate values can also be generated through specialised operations, extending the dynamic gate model; for example, operations such as `gate.dyn_rx` dynamically generate rotation gates parameterised by a runtime angle.

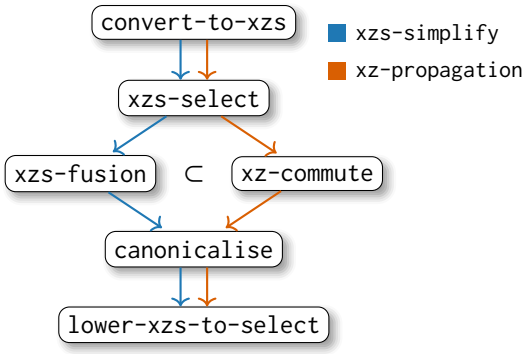


Figure 3. Optimisation pipelines for XZS gadgets. The left path, `xzs-simplify`, fuses all adjacent conditional Pauli and Phase gates. The right path, `xz-propagation`, attempts to move Pauli gates towards the end of the program while performing fusion.

We further include two more operations for generating dynamic gates, the XZS gadget `gate.xzs`, and the XZ gadget `gate.xz`. These operations (and their transformations) play a pivotal role in our case studies (see Section 5), and so we pause to discuss them in greater depth. The XZS gadget creates compositions of Pauli X, Pauli Z, and Phase gates, and its simplification, the XZ gadget, introduced in Section 2, omits the Phase component.

The XZS gadget takes three boolean values x, z, and s as input, outputting a gate value representing the composition $X^x Z^z S^s$. The XZ gadget has only two inputs for the inclusion of the Pauli X and Z gates. Various gates can be translated to XZS (or XZ) gadgets by the pass `convert-to-xzs`. These translations are given by Table 1. As we only consider gates up to global phase, we can soundly define the Pauli Y gate as the composition of Pauli X and Pauli Z gates, meaning the XZ gadget can represent all Pauli gates. With `%false` holding the constant false, we have:

`gate.xzs %x, %z, %false` ≡ `gate.xz %x, %z`

Canonicalisation converts the left expression to the right.

The inputs to XZ and XZS gadgets can be arbitrary SSA values, and not just constant values, allowing the gadget to represent conditionally applied gates, and enabling powerful optimisations. These optimisations organise into two pipelines, `xzs-simplify` and `xz-propagation` (Figure 3). Both pipelines convert (possibly conditional) Pauli and Phase gates to XZS gadgets, perform some transformation on these generated gadgets, and then lower back to selections between constant gates. The `xzs-simplify` pipeline fuses adjacent XZS gadgets, whereas the `xz-propagation` pipeline pushes XZ gadgets towards the end of the circuit while fusing them, performing Pauli propagation with conditional gates.

***Interaction with selection:* `xzs-select`.** If the inputs to an `arith.select` operation are XZS or XZ gadgets, then this selection can be converted to a single XZ(S) gadget with

parameters given by selection operations between the parameters of the original gadgets. More concretely,

```
%g_1 = gate.xzs %x_1, %z_1, %s_1
%g_2 = gate.xzs %x_2, %z_2, %s_2
%g_3 = arith.select %p, %g_1, %g_2
```

is replaced by

```
%x_3 = arith.select %p, %x_1, %x_2
%z_3 = arith.select %p, %z_1, %z_2
%s_3 = arith.select %p, %s_1, %s_2
%g_3 = gate.xzs %x_3, %z_3, %s_3
```

with similar conversions being given for XZ gadgets.

While this appears to make the program more complex, it is often the case that selections between boolean values can be trivially simplified, often to constant values, reducing the total number of operations. When preceded by the pass `convert-to-xzs` and combined with canonicalisation for `arith.select`, this pass converts any conditional Pauli or Phase gate to its canonical XZ(S) gadget representation. For an example, see Figure 2a.

Merging sequential gadgets: **`xzs-fusion`.** A key optimisation is fusing together consecutive dynamic applications of XZ(S) gadgets. Up to global phase, we have:

$$\begin{aligned} X^x Z^z S^s \circ X^{x'} Z^{z'} S^{s'} &\simeq X^x Z^z (X^{x'} Z^{s \wedge x'} S^s) Z^{z'} S^{s'} \\ &\simeq X^x X^{x'} Z^z Z^{z'} Z^{s \wedge x'} S^s S^{s'} \\ &\simeq X^{x \oplus x'} Z^{z \oplus z' \oplus (s \wedge x')} S^{s \oplus s'} \end{aligned}$$

with the first line following from the equation $SX \simeq XZS$ and a case analysis. For XZ gadgets, we have

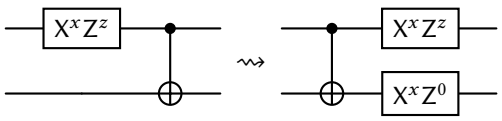
$$X^x Z^z \circ X^{x'} Z^{z'} \simeq X^{x \oplus x'} Z^{z \oplus z'}$$

These equations are implemented through pattern rewrites.

Pauli propagation: **`xz-commute`.** As the XZ gadget represents all (conditional) Pauli gates, it is an ideal tool for performing Pauli propagation, which commutes Pauli gates through other quantum gates, pushing them towards the end of the circuit. The rules for commuting an XZ gadget through a gate are specific for each gate, and are given by a "Clifford" trait on gate attributes, allowing this pass to apply to custom user-specified gates. This commutation data could be reused for other purposes, such as stabiliser simulation.

As an example, the case for the Hadamard gate is:

with X gates being converted to Z gates when commuted past the Hadamard and Z gates being converted to X gates. The case for the CX gate is slightly more complicated; an XZ gadget on the control wire of CX commutes as follows:



This case highlights a difficulty of this transformation: an individual commutation rewrite for a two qubit gate can increase the number of XZ gadgets, and a naive implementation of this compiler pass would have an exponential execution time (with respect to the length of the program).

To avoid this blow-up, we do not push each XZ gadget all the way to the end before processing the next one. Instead, a single linear sweep commutes a gadget through at most one neighbouring gate, then greedily applies `xz-fusion` rewrites whenever adjacent XZ gadgets are created. Immediately fusing new gadgets avoids uncontrolled duplication, ensuring linear runtime in the program length.

Lowering: **`lower-xzs-to-select`.** The last compiler pass we define for XZ gadgets effectively reverses the action of `convert-to-xzs` and `xzs-select` by converting each XZ(S) gadget to a selection between classical gates. This can be used as a first step to converting these gadgets to a more traditional representation when followed by compiler passes such as `lower-dyn-gate-to-scf`, which was introduced in the previous section.

5 Evaluation

We evaluate our work in two ways: application to various representative areas of quantum computing through in-depth case studies, as well as a more exhaustive benchmarking suite. The case studies are self-contained select examples of the dynamic gates model, rather than an exhaustive list of applications. Therefore we also evaluate our IR on a larger benchmarking suite drawn from practice that covers the interaction of hybrid quantum-classical computing.

5.1 Randomised Compilation

Randomised compilation [46] is a procedure for tailoring the noise of a quantum circuit. Quantum computation is probabilistic and it is often desirable to run a computation many times, taking the results of measurements as samples. Instead of running the same circuit each time, randomly generated equivalent (under noiseless execution) circuits can be run, changing the noise characteristics of the computation when the results of successive runs are averaged.

The procedure acts on a circuit in the Clifford+T gate set, and begins by partitioning the gates into *hard* gates (T, H, and CX) and *easy* gates ($\{X, Y, Z, S, S^\dagger\}$). Before each hard gate, a randomly chosen Pauli gate is applied to each of the input qubits. After the execution of the hard gate, corrective Pauli or Phase gates are applied to the output qubits, preserving the semantics of the circuit. We refer to the random gates added to the circuit (including the corrective operations) as *padding gates*. The padding gates are *easy* gates, which can be fused with other adjacent easy gates, limiting the increase in size of the circuit due to this procedure.

The ability to simplify static Pauli and phase gates, which is possible in many existing compilers and optimisers, is not

sufficient for fusing conditional padding gates. Two ways to apply this optimisation are known:

- Instantiate each iteration of the circuit with its randomly generated padding gates, and then optimise the resulting programs in existing compilers individually.
- Generate the random gates at runtime, and perform the gate merging in real time. This procedure can possibly run on the same classical control hardware used to perform error correction.

Neither option is optimal. Instantiating early duplicates work, as each iteration must go through the entire compilation pipeline. Instantiating at runtime is hardware dependent, and requires lower-level access to the control system.

The fundamental problem is that the random choice of padding gates cannot be represented as part of the quantum program, which prevents optimisation before instantiation of random variables. The dynamic gate IR solves this problem: the randomly generated padding gates can be encoded just like the phase flip channel in Section 2, and can be fused with XZS gadgets (see Section 4.1) long before any random variable has its value chosen.

The padding gates are introduced by the compiler pass `randomized-comp`. This pass matches on each hard gate, adding conditional Pauli X and Z gates before each input with a classical selection on a random variable and feeding the result to a dynamic gate. It then similarly inserts dynamic gates for the correction operations to each output. Crucially, it shares the same random variables for the padding gates before and after the operation.

Once the conditional padding gates have been added, they can be fused with the `xzs-simplify` pipeline introduced in Section 4.1. More precisely, this converts the padding and correction gates to XZS gadgets, fuses the gadgets, and then lowers back to constant gates and selections. Note that these applications need XZS gadgets; XZ gadgets are not expressive enough. Even though the gates generated before each hard operation are Pauli gates, placing an X gate before a T gate necessarily involves a Phase gate as part of the correction. It is important not to run the `xz-commute` pass in this scenario, as it would commute the padding gates through CX and H gates, cancelling them and nullifying the effect of the procedure.

We summarise two methods of performing randomised compilation in Figure 4:

- The naive pipeline represents the state of the art; n circuits are obtained by cloning the original circuit, instantiating the random variables with the `flip-coins` pass (randomly replacing each probabilistic operation by a constant), before running a pass to fuse easy gates (named `gate-fusion` in the figure) on each copy.
- The dynamic gate pipeline represents the method detailed above, running `xzs-simplify` before cloning the circuit and instantiating random variables with `flip-coins`. In this pipeline, the fusion operation is only run once, rather than once per iteration.

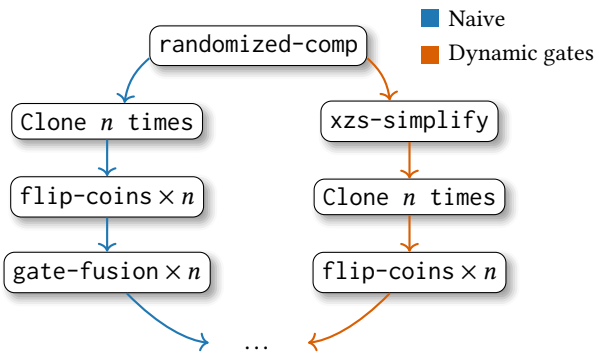


Figure 4. Two compiler pipelines for generating n circuits with the randomised compilation algorithm. The naive pipeline represents the state of the art, while the dynamic gate pipeline utilises dynamic gates to delay any cloning until after optimisation.

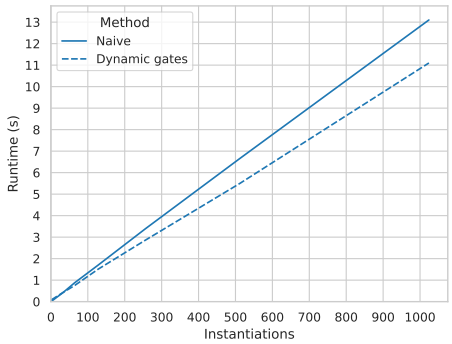


Figure 5. Time taken to generate n randomised circuits from a circuit preparing the 10-qubit GHZ state, with the naive and dynamic gate pipelines outlined in Figure 4. With greater values of n, the dynamic gate pipeline is up to 20% faster.

We benchmark the naive pipeline against the dynamic gate pipeline on a circuit preparing a 10-qubit GHZ state, displaying the results in Figure 5. As expected, the simpler naive pipeline is faster for small numbers of iterations, but is quickly outperformed by the dynamic gate pipeline, which is up to 20% faster for greater instantiation counts.

A secondary benefit of the dynamic gate approach is the size of the generated representation. If restricted to static quantum circuits, each individual instantiation must be stored separately, causing the representation size to scale linearly with the iteration count. Conversely, the dynamic gate representation (before instantiation) can be used to create an arbitrary number of samples. The overhead for this is small; the dynamic gate representation of applying randomised

compilation to the 10-qubit GHZ state requires 237 operations, whereas the static representation requires (depending on random seeding) around 50 operations per iteration, and so is already less concise at representing 5 samples.

We believe the benefits of this approach increase the longer random variables persist in the compilation pipeline. Evaluating this is out of scope, as this work considers higher-level representations of quantum programs only. The benchmark above is effectively the worst case scenario for the dynamic gate setup, where the circuit is cloned directly after fusion – and yet it still displays a moderate speedup.

5.2 Quantum Error Correction

Another way to combat noise in a computation is Quantum Error Correction (QEC); see e.g. [40] for a summary. Broadly, QEC works by encoding a logical qubit as a state over multiple physical qubits, introducing redundancy. Errors can be detected by syndrome measurements, which preserve the code-space (the states of the physical qubits corresponding to states of the logical qubit). The results of these measurements are then passed into a decoder, a classical function that determines which error has most likely occurred. The physical state can then be corrected with Pauli gates, returning it to the code-space and thereby correcting the errors.

In stabiliser codes, each syndrome measurement consists of CZ and CX gates between qubits containing the state of the logical qubit and a freshly allocated auxiliary qubit, before measuring the auxiliary qubit to get a classical bit. The decoder could in principle be any classical computation. In the case we will discuss, it takes the form of a small boolean circuit, but for production scale QEC codes it could take the form of large matrix calculations or even neural network computations on a GPU or an FPGA. Finally, the output of the decoder is fed into conditional Pauli gates, which we represent in the dynamic gate model as XZ gadgets.

To preserve a quantum state, these three phases are run in a loop. QEC protocols can only tolerate a certain number of errors while preserving the quantum state. Noise affects idle qubits, so the protocol works better when the time between successive cycles of syndrome measurements is shorter.

One method of reducing the time between syndrome measurements is to delay the corrective operations. The gates used in QEC are all part of the *Clifford group*, the gates which normalise the Pauli gates, allowing the correction operations to be propagated past the next round of syndrome measurements. The correction operations for successive cycles can be fused, reducing the number of quantum operations required. More importantly, they can be buffered: the next round of syndrome measurements can start before the decoder finishes execution. This optimisation is crucial in modern QEC workflows, including recent experiments such as those performed by Google [18], that can perform up to millions of QEC cycles a second.

Table 2. The number of operations needed to present various numbers of cycles of the perfect 5-qubit QEC code in the dynamic gate IR, both before and after application of the `xz-propagation` pipeline. The operation counts are split into quantum operations (allocations, gates, and measurements) and any other operation.

	Operations (#)			
	Before `xz-propagation`		After `xz-propagation`	
Cycles (#)	Quantum	Other	Quantum	Other
1	42	40	37	32
10	420	400	325	500
100	4200	4000	3205	5180
1000	42000	40000	32005	51980

This optimisation is performed using the `xz-propagation` pipeline. When applied to a number of consecutive QEC cycles, the pipeline first converts each corrective gate to its XZ gadget representation, then propagates these corrections to the end with an `xz-commute` pass, merging them together. This method automatically modifies the results of later syndrome measurements and automatically synthesises a function to combine the results of decoding cycles.

As a prototypical case, consider the smallest error correction code capable of correcting a single qubit error: the perfect 5-qubit code [26]. The implementation in the dynamic gate IR encodes all three phases of the error correction cycle, including the classical decoder. Each phase is visually distinct within the IR, and when the pipeline is run on two consecutive cycles, it can be easily verified by inspecting the generated IR that only one correction phase remains.

Table 2 analyses the size of IR required to represent this QEC code within the dynamic gate IR. It demonstrates that the `xz-propagation` pass reduces the number of quantum operations, at the cost of a small increase in the number of classical operations.

We further ran our pipeline on IR representing larger numbers of consecutive QEC cycles, measuring the duration of each compiler pass in the `xz-propagation` pipeline. The results, in Figure 6, provide evidence that the `xz-propagation` pipeline runs in linear time with respect to the number of operations in the input.

Although we only consider correction cycles for a QEC code above, and not any logical gates, the `xz-propagation` pipeline will commute corrective gates past transversal (Clifford) logical gates. We therefore conjecture that this pipeline could form part of a compiler which converts a circuit to its fault-tolerant implementation.

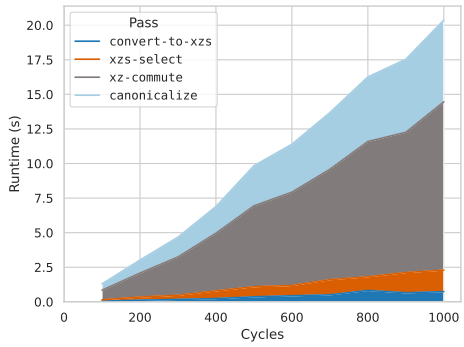


Figure 6. Duration of each pass of the `xz-propagation` pipeline when run on a varying number of cycles of the 5-qubit perfect QEC code. Each pass runs in linear time with respect to the input size.

5.3 Measurement-Based Quantum Computing

An alternative to the circuit model of quantum computing is given by Measurement-Based Quantum Computing (MBQC) [38]. This way of implementing unitary evolution begins by entangling the input qubits with fresh auxiliary qubits; the entire computation is then driven by performing measurements. As quantum measurement is probabilistic, the computation is non-deterministic. The operation can be made deterministic (unitary) again by applying corrective Pauli gates to the output qubits, conditioned on the classical outputs of the measurements.

The measurement calculus [11] gives a syntax for MBQC programs. The normal form for MBQC programs is given by *measurement patterns*. These first allocate new qubits in the $|+\rangle = \mathrm{H}|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ state, entangle them (with controlled-Z gates), perform XY-plane measurements (measurements in the basis $\frac{1}{\sqrt{2}}(|0\rangle \pm e^{i\theta}|1\rangle)$ for some angle θ), and finally correct with Pauli gates conditioned by a linear combination of the measurement results. Consider a generic rotation, given by the unitary $\mathrm{R_X}(-\phi)\mathrm{R_Z}(-\theta)\mathrm{R_X}(-\lambda)$:

$$Z_5^{s_1+s_3} X_5^{s_2+s_4} [M_4^{\phi}]^{s_1+s_3} [M_3^{\theta}]^{s_2} [M_2^{\lambda}]^{s_1} M_1^0 E_{4,5} E_{3,4} E_{2,3} E_{1,2}$$

Each operation in this pattern acts on the qubits in its subscript (with qubit 1 being the input qubit and qubit 5 being the output qubit), and s_n corresponds to the classical outcome of measuring qubit n (which is unambiguous as each qubit is measured at most once). Each $E_{n,m}$ is a CZ gate on qubits n and m, each $[M_n^{\psi}]^x$ is an XY-plane measurement of qubit n with angle $(-1)^x\psi$, and each X_n^x or Z_n^x is a Pauli X or Z gate, classically conditioned on the boolean x.

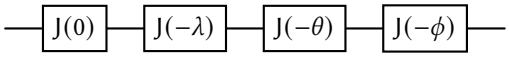
This pattern is in the standard CME (Correction-Measurement-Entanglement) form, where entanglement operations come before measurements, which in turn come before corrective gates. Any well-formed pattern can be converted to this form by a standardisation procedure [5].

```
%q2 = qu.alloc<#qubit.plus>
%q3 = qu.alloc<#qubit.plus>
%q4 = qu.alloc<#qubit.plus>
%q5 = qu.alloc<#qubit.plus>
%q1_1, %q2_1 = qssa.gate<#gate.cz> %q1, %q2
%q2_2, %q3_1 = qssa.gate<#gate.cz> %q2_1, %q3
%q3_2, %q4_1 = qssa.gate<#gate.cz> %q3_1, %q4
%q4_2, %q5_1 = qssa.gate<#gate.cz> %q4_1, %q5
%s_1   = qssa.measure<#measurement.xy<0>> %q1_1
%a2    = angle.cond_negate %s_1, %lambda
%m2    = measurement.dyn_xy<%a2>
%s_2   = qssa.dyn_measure<%m2> %q2_2
%a3    = angle.cond_negate %s_2, %theta
%m3    = measurement.dyn_xy<%a3>
%s_3   = qssa.dyn_measure<%m3> %q3_2
%z     = arith.xori %s_1, %s_3 : i1
%a4    = angle.cond_negate %z, %phi
%m4    = measurement.dyn_xy<%a4>
%s_4   = qssa.dyn_measure<%m4> %q4_2
%x     = arith.xori %s_2, %s_4 : i1
%g     = gate.xz %x, %z
%q5_2 = qssa.dyn_gate<%g> %q5_1
```

Figure 7. An MBQC program equivalent to the generic rotation $\mathrm{R_X}(-\%\mathtt{phi})\mathrm{R_Z}(-\%\mathtt{theta})\mathrm{R_X}(-\%\mathtt{lambda})$ in our dynamic gate IR, with input qubit `%q1` and output `%q5_2`. Both quantum and classical dependencies are represented explicitly as SSA values.

The dynamic gate model can be used as an alternative SSA syntax for MBQC programs. The dynamic angle on measurement operations is accommodated by the operation `measurement.dyn_xy`, conditional negation of angles is explicated into an operation, and a single dynamic XZ gadget performs the conditional X and Z operators. The translation of the pattern above can be found in Figure 7.

The IR is not only flexible enough to represent both circuit-based programs and MBQC programs, but can also convert from the former to the latter via local rewrites. The transformation takes as input a quantum circuit consisting of CZ and $\mathrm{J}(\theta)$ gates, which form a universal gate set. The generic rotation unitary $\mathrm{R_X}(-\phi)\mathrm{R_Z}(-\theta)\mathrm{R_X}(-\lambda)$ of our running example is given by the circuit:



To allow the angles λ, θ, and ϕ to be parameters to the circuit, the operation `gate.dyn_j` encodes the J gates dynamically in the IR by taking a single angle operand.

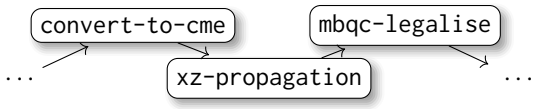
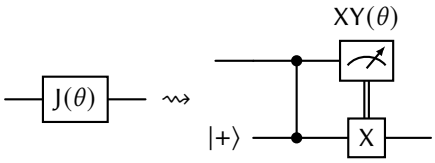


Figure 8. A compiler pipeline to convert CZ/J circuits to MBQC programs within the dynamic gate IR.

As MBQC programs already permit CZ gates, the next step is to convert each J gate to a measurement pattern:



The measurement is given by a dynamic XY-plane measurement and the conditioned X gate is given by a dynamic selection between an X gate and an identity gate. The pass `convert-to-cme` performs this pattern rewrite.

After this transformation, we already have a valid measurement pattern, but it is not necessarily in CME form. After all, converting composed J gates adds a corrective Pauli gate in the middle of the program, before other CZ gates and XY-plane measurements. The `xz-propagation` pass standardises the pattern by pushing the corrective Pauli gates to the end of the program (and then fusing them). This process introduces corrective Z gates via the commutation rules for CZ gates, and conditionally negates angles via the commutation rules for XY-plane measurements.

Note that this optimisation does not need extra "signal shifting" operations. Previous standardisation procedures for the measurement calculus needed signal shifting to track the negations of the classical measurement results. The explicit classical dataflow in our IR means this is no longer required.

Finally, the `mbqc-legalise` pass reorders the operations to result in a valid CME pattern, leaving the dataflow graph of the program unchanged. The full pipeline is summarised in Figure 8.

5.4 Benchmarking Suite

To gather more general metrics of our IR, we curated a new benchmarking suite of hybrid quantum-classical algorithms. To ensure that the programs we have picked are representative, we followed the overview of hybrid classical-quantum algorithms given by Ella et al. [14, Fig. 1]. Although this previous analysis focusses on pulse-level programs, we surveyed the papers they cited, and after filtering for gate-level algorithms arrived at the programs in Table 3.

While not a comprehensive list of all algorithms, the benchmark contains many of the constructions typically found in hybrid programs, including angle-parameterised gates, conditional gate application, and classical loops. It includes programs from different quantum computing paradigms, such as measurement driven computation, NISQ variational algorithms, and building blocks of fault-tolerant computation. We chose more focussed, concise programs, with line counts varying from 17 to 147, easing the manual implementation and tuning of these programs for other systems in the future.

Teleport: Constant-depth teleportation protocol [12, Fig. 1].
Prep: State preparation of logical plus state [17].
RUS: Three Repeat-Until-Success circuits for V_3 [1, Fig. 1].
QML: Conditional gearbox circuit from a quantum neural network [32, Fig. 1].
IPE: Iterative phase estimation [8, Fig. 1 (Bottom)].
RWPE: Random walk phase estimation [19, Algorithm 1].
MBQC-Rot: General MBQC rotation gate [38].
MBQC-CX: MBQC CX gate [38].
QEC-Adap: Adaptive QEC cycle [42, Fig. 17].
QAOA: Max-cut Quantum Approximate Optimisation Algorithm. See e.g. [6, 15].

Table 3. Hybrid Benchmark Suite. Implementations in our IR are contained in the artifact [39].

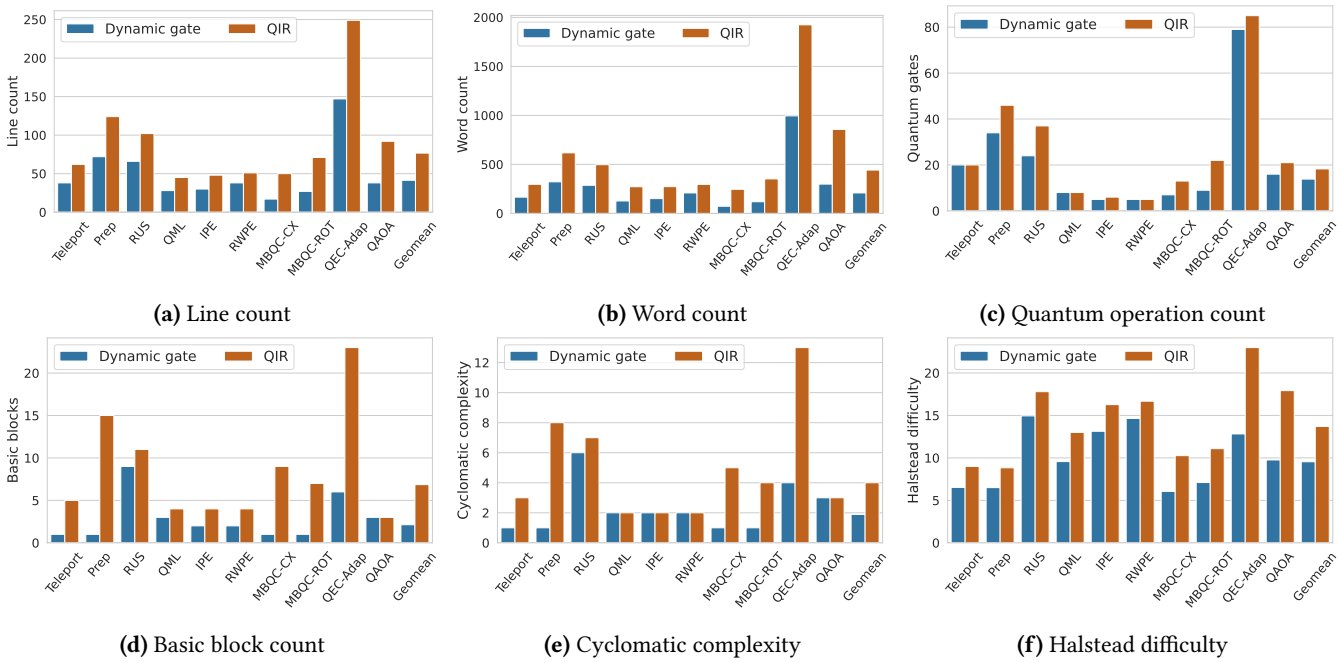
This benchmarking suite is used to compare our IR to QIR. To obtain QIR programs, we systematically lowered each program in the benchmarking suite. First, `qssa` operations were converted to their `qref` counterparts. Then, dynamic gates were lowered to structured control flow operations, which were further lowered to branching operations. Our gate operations then had to be converted to QIR style gate invocations. Finally, we used MLIR passes to reduce the classical components of the program to LLVM-IR. We lowered to the quantum instruction set supported by qir-runner [2], an official QIR alliance project.

We performed six analyses of the programs, the results of which are graphed in Figure 9. The analyses chosen require no human input and are purely numerical metrics.

The line and word counts of each program provide an initial measure of how verbose each program is, with the line count correlating strongly to the total number of operations in each program. We were further able to count the number of quantum operations in each program, which we define to be `qssa` operations for our IR, and calls to functions of the form `@__quantum__qis*` for QIR. The difference between the representations in this metric arises from the representation of conditional gates, as well as our IR allowing non-zero allocation and measurement in bases other than the Z basis.

We further included counts of the number of basic blocks in each program as well as cyclomatic complexity [30], to measure the complexity of the control flow of each program. Lastly, we include the following metric H of the complexity of a program due to Halstead [20]:

$$H = \frac{\#\text{ of unique operations}}{2} \times \frac{\text{Total }\#\text{ of operands}}{\#\text{ of unique operands}}$$



(a) Line count **(b)** Word count **(c)** Quantum operation count

(d) Basic block count **(e)** Cyclomatic complexity **(f)** Halstead difficulty

Figure 9. Numerical analysis on the benchmark suite. The Dynamic Gate IR is more concise than QIR, and is able to rely less on (unstructured) control flow operations, reducing the overall complexity of programs.

Here, we define operands as in MLIR/LLVM, and consider two operations to be equal if they are identical other than their operands (in particular letting applications of different gates be distinct operations).

Our IR outperformed QIR in every analysis. When comparing the geometric means over each benchmark (listed as the final entry in each plot of Figure 9), dynamic gates used 47% fewer lines, 52% fewer words, 24% fewer quantum operations, 69% fewer basic blocks, had 52% less cyclomatic complexity, and 30% less Halstead difficulty.

6 Related and Future Work

In this section, we analyse some of the differences between the IR presented in this paper with other previously introduced quantum IRs. We then review other work that could form the basis for different gadgets, or that we otherwise believe could be represented well by the dynamic gate framework. We follow this discussion with some further suggestions of future work and potential additions to the IR.

6.1 Related Quantum Intermediate Representations

There exist two dominant textual representations for quantum programs: OpenQASM and QIR. OpenQASM [10] closely corresponds to the representation of quantum programs in the Qiskit [23] quantum toolkit. Version 2 of OpenQASM was restricted to quantum circuits with a fixed number of qubits, though version 3 has added a limited ability to represent classical computations. Conversely, QIR [37] is built on top of LLVM [27], and inherits all its classical functionality, but quantum gates are applied via opaque function calls with little interaction between the classical and quantum components of the program. In both these formats, qubits have reference semantics, unlike the value semantics common in modern compilers.

Other quantum IRs have been proposed within the MLIR framework. McCaskey and Nguyen [31] introduce an MLIR dialect for representing quantum programs, leveraging the tooling present in MLIR to define a translation down to QIR, while remaining expressive enough to provide lowerings from other toolkits to this dialect. Like QIR, qubits in this dialect are given by reference semantics. The first MLIR quantum dialect using value semantics was QSSA [34]. QIRO [22], which is the basis of the IR used in the Catalyst compiler [21] (distributed as part of Pennylane [4]), introduces parallel quantum dialects; one with reference semantics for qubits and one with value semantics, leveraging that linearity analysis is easier within the non-SSA dialect, and optimisations are easier within the SSA dialect.

Ensemble IR [47] provides MLIR dialects for representing collections (or ensembles) of related quantum circuits, as in the randomised compilation example of Section 5.1. Similar to the current work, it allows gates to be specified via an extensible set of strings, rather than having fixed operations for each gate, and attempts to perform optimisations and transformations at the hybrid level rather than circuit level. It is more specialised to probabilistic scenarios than a general hybrid setting, containing primitive operations for certain

probabilistic tasks such as sampling from distributions. Future work could study how these dialects could be used in conjunction with our own, given their similar approaches and setting.

Quantinuum recently introduced t|ket⟩ 2, extending t|ket⟩ 1 [43] to hybrid quantum-classical programs. It is built on HUGR [25], a custom-built library for creating quantum-classical representations. A possible avenue for future work is looking at the feasibility of introducing the framework, operations, and accompanying translations to this library.

6.2 Quantum Gadgets for Optimisation

A range of quantum gadgets already exist in the literature for optimising quantum circuits. Phase polynomials [3] are a gadget for representing combinations of CX and T gates, primarily used to reduce the T-count of a circuit.

The ZX calculus [7] is a graph-based method of representing linear maps. It introduces quantum gadgets called red and green spiders, and it has been shown these gadgets can be used to optimise quantum circuits [24]. As future work it would be interesting to explore adding ZX spiders to the dynamic gate model, which is nontrivial because the ZX calculus makes no distinction between the inputs and outputs of spiders.

The t|ket⟩ 1 [43] quantum circuit compiler makes use of two quantum gadgets inspired by the ZX calculus: phase gadgets, and their generalisation Pauli gadgets. These gadgets are used like the XZ gadgets in this paper: subcircuits can be represented by phase gadgets, which can be optimised before translating back to a traditional circuit representation.

Future work could investigate which of these are suitable for implementation as dynamic gates, allowing them to be used in the optimisation of hybrid programs rather than static quantum circuits.

6.3 Other Future Directions

Much of the current work focusses on optimisations concerning Clifford gates, with the XZ gadgets representing Pauli gates and the XZS gadgets adding the Clifford phase gate. The optimisations in this paper have utility outside of Clifford programs, such as the randomised compilation in Section 5.1 on Clifford+T circuits and the universal MBQC programs of Section 5.3. In the future it would be good to explore optimisations which target hybrid non-Clifford operations. Adapting phase polynomials (see the previous section) to the dynamic gate framework could be a way to achieve this.

The current work does not discuss subprocedures or subcircuits. While these can be represented with MLIR's `func` dialect, a more complete solution would be able to interact with conversions between reference and value-based semantics, Pauli propagation, and inverting unitaries. An interesting direction for further research is to explore implementing a `subcircuit` operation producing a gate value from a circuit, which could then be applied dynamically.

A different direction to extend the current work is to add quantum data types other than qubits. The addition of quantum registers has been explored in other quantum IRs, with QIRO and QSSA both introducing custom quantum register types with operations for manipulating them. Note that the MLIR `tensor` type is immutable, making it difficult to use on qubits with value-semantics. Other work [45, 49] has explored adding higher-level data types to quantum languages, which will be necessary to represent in future quantum IRs.

Finally, while quantum noise channels were used to motivate dynamic gates, their use within a noise-aware compilation pipeline remains relatively unexplored. Natively representing noise channels with dynamic gates should in particular work for the classical simulation of noisy circuits, where it should be possible to optimise the noisy circuit before instantiating the noise for a particular sample, much like in our randomised compilation case study.

Acknowledgments

The authors would like to thank Chris Vasiladiotis for feedback on earlier drafts of this paper. This work was funded by EPSRC Grant EP/W032635/1, Robust and Reliable Quantum Computing

References

[1] Paetznick Adam and Krysta M. Svore. 2014. Repeat-Until-Success: Non-deterministic Decomposition of Single-Qubit Unitaries. *Quantum Information and Computation* 14, 15&16 (Nov. 2014), 1277–1301. doi:10.26421/qic14.15-16-2

[2] QIR Alliance. 2025. QIR Runner. https://github.com/qir-alliance/qir-runner.

[3] Matthew Amy, Dmitri Maslov, and Michele Mosca. 2014. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10 (Oct. 2014), 1476–1489. doi:10.1109/TCAD.2014.2341953

[4] Ville Bergholm, Josh Izaac, Maria Schuld, Christian Gogolin, Shahnawaz Ahmed, Vishnu Ajith, M. Sohaib Alam, Guillermo Alonso-Linaje, B. AkashNarayanan, Ali Asadi, Juan Miguel Arrazola, Utkarsh Azad, Sam Banning, Carsten Blank, Thomas R. Bromley, Benjamin A. Cordier, Jack Ceroni, Alain Delgado, Olivia Di Matteo, Amintor Dusko, Tanya Garg, Diego Guala, Anthony Hayes, Ryan Hill, Aroosa Ijaz, Theodor Isacsson, David Ittah, Soran Jahangiri, Prateek Jain, Edward Jiang, Ankit Khandelwal, Korbinian Kottmann, Robert A. Lang, Christina Lee, Thomas Loke, Angus Lowe, Keri McKiernan, Johannes Jakob Meyer, J. A. Montañez-Barrera, Romain Moyard, Zeyue Niu, Lee James O'Riordan, Steven Oud, Ashish Panigrahi, Chae-Yeun Park, Daniel Polatajko, Nicolás Quesada, Chase Roberts, Nahum Sá, Isidor Schoch, Borun Shi, Shuli Shu, Sukin Sim, Arshpreet Singh, Ingrid Strandberg, Jay Soni, Antal Száva, Slimane Thabet, Rodrigo A. Vargas-Hernández, Trevor Vincent, Nicola Vitucci, Maurice Weber, David Wierichs, Roeland Wiersema, Moritz Willmann, Vincent Wong, Shaoming Zhang, and Nathan Killoran. 2022. PennyLane: Automatic Differentiation of Hybrid Quantum-Classical Computations. arXiv:1811.04968 [physics, physics:quant-ph] doi:10.48550/arXiv.1811.04968

[5] Anne Broadbent and Elham Kashefi. 2009. Parallelizing Quantum Circuits. *Theoretical Computer Science* 410, 26 (June 2009), 2489–2510. doi:10.1016/j.tcs.2008.12.046

[6] M. Cerezo, Andrew Arrasmith, Ryan Babbush, Simon C. Benjamin, Suguru Endo, Keisuke Fujii, Jarrod R. McClean, Kosuke Mitarai, Xiao Yuan, Lukasz Cincio, and Patrick J. Coles. 2021. Variational Quantum Algorithms. *Nature Reviews Physics* 3, 9 (Sept. 2021), 625–644. doi:10.1038/s42254-021-00348-9

[7] Bob Coecke and Ross Duncan. 2011. Interacting Quantum Observables: Categorical Algebra and Diagrammatics. *New Journal of Physics* 13, 4 (April 2011), 043016. doi:10.1088/1367-2630/13/4/043016

[8] A. D. Córcoles, Maika Takita, Ken Inoue, Scott Lekuch, Zlatko K. Minev, Jerry M. Chow, and Jay M. Gambetta. 2021. Exploiting Dynamic Quantum Circuits in a Quantum Algorithm with Superconducting Qubits. *Physical Review Letters* 127, 10 (Aug. 2021), 100501. doi:10.1103/PhysRevLett.127.100501

[9] Alexander Cowtan, Silas Dilkes, Ross Duncan, Will Simmons, and Seyon Sivarajah. 2020. Phase Gadget Synthesis for Shallow Circuits. *Electronic Proceedings in Theoretical Computer Science* 318 (May 2020), 213–228. doi:10.4204/EPTCS.318.13

[10] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3 (Sept. 2022), 12:1–12:50. doi:10.1145/3505636

[11] Vincent Danos, Elham Kashefi, and Prakash Panangaden. 2007. The Measurement Calculus. *J. ACM* 54, 2 (April 2007), 8–es. doi:10.1145/1219092.1219096

[12] Dhruv Devulapalli, Eddie Schoute, Aniruddha Bapat, Andrew M. Childs, and Alexey V. Gorshkov. 2024. Quantum Routing with Teleportation. *Physical Review Research* 6, 3 (Sept. 2024), 033313. doi:10.1103/PhysRevResearch.6.033313

[13] David P. Divincenzo. 1997. Mesoscopic Electron Transport. Kluwer, Chapter Topics in quantum computers, 657–677. arXiv:9612126 [cond-mat] doi:10.48550/arXiv.cond-mat/9612126

[14] Lior Ella, Lorenzo Leandro, Oded Wertheim, Yoav Romach, Lukas Schlipf, Ramon Szmuk, Yoel Knol, Nissim Ofek, Itamar Sivan, and Yonatan Cohen. 2023. Quantum-Classical Processing and Benchmarking at the Pulse-Level. arXiv:2303.03816 [quant-ph] doi:10.48550/arXiv.2303.03816

[15] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. 2014. A Quantum Approximate Optimization Algorithm. arXiv:1411.4028 [quant-ph] doi:10.48550/arXiv.1411.4028

[16] Mathieu Fehr, Michel Weber, Christian Ulmann, Alexandre Lopoukhine, Martin Paul Lücke, Théo Degioanni, Christos Vasiladiotis, Michel Steuwer, and Tobias Grosser. 2025. xDSL: Sidekick Compilation for SSA-Based Compilers. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization (CGO '25)*. Association for Computing Machinery, New York, NY, USA, 179–192. doi:10.1145/3696443.3708945

[17] Michael Foss-Feig, Arkin Tikku, Tsung-Cheng Lu, Karl Mayer, Mohsin Iqbal, Thomas M. Gatterman, Justin A. Gerber, Kevin Gilmore, Dan Gresh, Aaron Hankin, Nathan Hewitt, Chandler V. Horst, Mitchell Matheny, Tanner Mengle, Brian Neyenhuis, Henrik Dreyer, David Hayes, Timothy H. Hsieh, and Isaac H. Kim. 2023. Experimental Demonstration of the Advantage of Adaptive Quantum Circuits. arXiv:2302.03029 [quant-ph] doi:10.48550/arXiv.2302.03029

[18] Google Quantum AI and Collaborators. 2025. Quantum Error Correction below the Surface Code Threshold. *Nature* 638, 8052 (Feb. 2025), 920–926. doi:10.1038/s41586-024-08449-y

[19] Cassandra Granade and Nathan Wiebe. 2022. Using Random Walks for Iterative Phase Estimation. arXiv:2208.04526 [quant-ph] doi:10.48550/arXiv.2208.04526

[20] Maurice H. Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., USA.

[21] David Ittah, Ali Asadi, Erick Ochoa Lopez, Sergei Mironov, Samuel Banning, Romain Moyard, Mai Jacob Peng, and Josh Izaac. 2024. Catalyst: A Python JIT Compiler for Auto-Differentiable Hybrid Quantum Programs. *Journal of Open Source Software* 9, 99 (July 2024), 6720. doi:10.21105/joss.06720

[22] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization. *ACM Transactions on Quantum Computing* 3, 3 (Sept. 2022), 1–32. doi:10.1145/3491247

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

[24] Aleks Kissinger and John van de Wetering. 2020. Reducing the Number of Non-Clifford Gates in Quantum Circuits. *Physical Review A* 102, 2 (Aug. 2020), 022406. doi:10.1103/PhysRevA.102.022406

[25] Mark Koch, Agustín Borgna, Seyon Sivarajah, Alan Lawrence, Alec Edgington, Douglas Wilson, Craig Roy, Luca Mondada, Lukas Heidemann, and Ross Duncan. 2025. HUGR: A Quantum-Classical Intermediate Representation. arXiv:2510.11420 [cs.PL] doi:10.48550/arXiv.2510.11420

[26] Raymond Laflamme, Cesar Miquel, Juan Pablo Paz, and Wojciech Hubert Zurek. 1996. Perfect Quantum Error Correcting Code. *Physical Review Letters* 77, 1 (July 1996), 198–201. doi:10.1103/PhysRevLett.77.198

[27] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. doi:10.1109/CGO.2004.1281665

[28] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[29] Rikus R. le Roux, George van Schoor, and Pieter A. van Vuuren. 2014. A Survey on Reducing Reconfiguration Cost: Reconfigurable PID Control as a Special Case. *IFAC Proceedings Volumes* 47, 3 (Jan. 2014), 1320–1330. doi:10.3182/20140824-6-ZA-1003.01544

[30] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308–320. doi:10.1109/TSE.1976.233837

[31] Alexander McCaskey and Thien Nguyen. 2021. A MLIR Dialect for Quantum Assembly Languages. In *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*. 255–264. doi:10.1109/QCE52317.2021.00043

[32] M. S. Moreira, G. G. Guerreschi, W. Vlothuizen, J. F. Marques, J. van Straten, S. P. Premaratne, X. Zou, H. Ali, N. Muthusubramanian, C. Zachariadis, J. van Someren, M. Beekman, N. Haider, A. Bruno, C. G. Almudever, A. Y. Matsuura, and L. DiCarlo. 2023. Realization of a Quantum Neural Network Using Repeat-until-Success Circuits in a Superconducting Quantum Processor. *npj Quantum Information* 9, 1 (Nov. 2023), 118. doi:10.1038/s41534-023-00779-5

[33] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information* (10th anniversary edition ed.). Cambridge university press, Cambridge.

[34] Anurudh Peduri, Siddharth Bhat, and Tobias Grosser. 2022. QSSA: An SSA-based IR for Quantum Computing. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*. ACM, Seoul South Korea, 2–14. doi:10.1145/3497776.3517772

[35] Christophe Piveteau and David Sutter. 2024. Circuit Knitting With Classical Communication. *IEEE Trans. Inf. Theor.* 70, 4 (April 2024), 2734–2745. doi:10.1109/TIT.2023.3310797

[36] John Preskill. 2018. Quantum Computing in the NISQ Era and Beyond. *Quantum* 2 (2018), 79.
[37] QIR Alliance 2021. *QIR Specification.* QIR Alliance.
[38] Robert Raussendorf and Hans J. Briegel. 2001. A One-Way Quantum Computer. *Physical Review Letters* 86, 22 (May 2001), 5188–5191. doi:10.1103/PhysRevLett.86.5188
[39] Alex Rice. 2026. Paper artifact: xdslproject/inconspiquous. doi:10.5281/zenodo.22147114
[40] Joschka Roffe. 2019. Quantum error correction: an introductory guide. *Contemporary Physics* 60, 3 (2019), 226–245. doi:10.1080/00107514.2019.1667078
[41] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principle of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. doi:10.1145/73560.73562
[42] C. Ryan-Anderson, J. G. Bohnet, K. Lee, D. Gresh, A. Hankin, J. P. Gaebler, D. Francois, A. Chernoguzov, D. Lucchetti, N. C. Brown, T. M. Gatterman, S. K. Halit, K. Gilmore, J. A. Gerber, B. Neyenhuis, D. Hayes, and R. P. Stutz. 2021. Realization of Real-Time Fault-Tolerant Quantum Error Correction. *Physical Review X* 11, 4 (Dec. 2021), 041058. doi:10.1103/PhysRevX.11.041058
[43] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology* 6, 1 (Nov. 2020), 014003. doi:10.1088/2058-9565/ab8e92
[44] Shigeyuki Takano and Hideharu Amano. 2022. Reconfiguration Cost for Reconfigurable Computing Architectures. In *2022 23rd ACIS International Summer Virtual Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD-Summer)*. 62–67. doi:10.1109/SNPD-Summer57817.2022.00019
[45] Matan Vax, Peleg Emanuel, Eyal Cornfeld, Israel Reichental, Ori Opher, Ori Roth, Tal Michaeli, Lior Preminger, Lior Gazit, Amir Naveh, and Yehuda Naveh. 2025. Qmod: Expressive High-Level Quantum Modeling. arXiv:2502.19368 [quant-ph] doi:10.48550/arXiv.2502.19368
[46] Joel J. Wallman and Joseph Emerson. 2016. Noise Tailoring for Scalable Quantum Computation via Randomized Compiling. *Physical Review A* 94, 5 (Nov. 2016), 052325. doi:10.1103/PhysRevA.94.052325
[47] Sourish Wawdhane, Sashwat Anagolum, Poulami Das, and Yunong Shi. 2025. Ensemble-IR: Concise Representation for Quantum Ensemble Programs. arXiv:2507.09581 [quant-ph] doi:10.48550/arXiv.2507.09581
[48] Noson S. Yanofsky and Mirco A. Mannucci. 2008. *Quantum Computing for Computer Scientists.* Cambridge University Press. doi:10.1017/cbo9780511813887
[49] Charles Yuan and Michael Carbin. 2022. Tower: Data Structures in Quantum Superposition. *Tower: Data Structures in Quantum Superposition* 6, OOPSLA2 (Oct. 2022), 134:259–134:288. doi:10.1145/3563297

A Dynamic Gate IR Specification

Here we list the attributes, types, and operations of the Dynamic Gate IR, which are to be combined with common MLIR dialects (e.g. arith and builtin). Crucially, the IR is open in nature; a user can define new gate attributes or operations for creating gates, and combine them with the operations listed below.

Types			
`!qu.bit`	Qubit type		
`!gate.type<`n`>`	Type of n-qubit gates		
`!measurement.type<`n`>`	Type of n-qubit measurements		
`!angle.type`	Type of angles		
Attributes			
Angle attribute	θ	:=	`#angle.attr<`x`>` for $0 \leq x < 2\pi$
Gate attributes	g	:=	`#gate.id<`n`>` | `#gate.x` | `#gate.y` | `#gate.z` | `#gate.h` | `#gate.s` | `#gate.s_dagger` | `#gate.t` | `#gate.t_dagger` | `#gate.rx<`θ`>` | `#gate.ry<`θ`>` | `#gate.rz<`θ`>` | `#gate.j<`θ`>` | `#gate.rzz<`θ`>` | `#gate.cx` | `#gate.cz` | `#gate.toffoli`
Measurement attributes	m	:=	`#measurement.comp_basis` | `#measurement.x_basis` | `#measurement.xy<`θ`>`
Allocation attributes	a	:=	`#qu.zero` | `#qu.plus`

Operations	**Type**	
`qu.alloc<`a`>`	() -> `!qu.bit`^{n}	where qubits(a) = n
`qssa.gate<`g`>`	`!qu.bit`^{n} -> `!qu.bit`^{n}	where qubits(g) = n
`qssa.dyn_gate`	(`!gate.type<`n`>`, `!qu.bit`^{n}) -> `!qu.bit`^{n}	
`qssa.measure<`m`>`	`!qu.bit`^{n} -> `i1`^{n}	where qubits(m) = n
`qssa.dyn_measure`	(`!measurement.type<`n`>`, `!qu.bit`^{n}) -> `i1`^{n}	
`qref.gate<`g`>`	`!qu.bit`^{n} -> ()	where qubits(g) = n
`qref.dyn_gate`	(`!gate.type<`n`>`, `!qu.bit`^{n}) -> ()	
`qref.measure<`m`>`	`!qu.bit`^{n} -> `i1`^{n}	where qubits(m) = n
`qref.dyn_measure`	(`!measurement.type<`n`>`, `!qu.bit`^{n}) -> `i1`^{n}	
`gate.constant` g	() -> `!gate.type<`n`>`	where qubits(g) = n
`gate.xz`	(`i1`, `i1`) -> `!gate.type<1>`	
`gate.xzs`	(`i1`, `i1`, `i1`) -> `!gate.type<1>`	
`gate.dyn_rx`	`!angle.type` -> `!gate.type<1>`	
`gate.dyn_ry`	`!angle.type` -> `!gate.type<1>`	
`gate.dyn_rz`	`!angle.type` -> `!gate.type<1>`	
`gate.dyn_j`	`!angle.type` -> `!gate.type<1>`	
`gate.dyn_rzz`	`!angle.type` -> `!gate.type<2>`	
`measurement.constant` m	() -> `!measurement.type<`n`>`	where qubits(m) = n
`measurement.dyn_xy`	`!angle.type` -> `!measurement.type<1>`	
`angle.constant` θ	() -> `!angle.type`	
`angle.add`	(`!angle.type`, `!angle.type`) -> `!angle.type`	
`angle.scale`	(`!angle.type`, `f64`) -> `!angle.type`	
`angle.negate`	`!angle.type` -> `!angle.type`	
`angle.cond_negate`	(`i1`, `!angle.type`) -> `!angle.type`	
`prob.bernoulli` f	() -> `i1`	where $0 \leq f \leq 1$
`prob.uniform` T	() -> T	where T is an integer type