Remarks on type syntax in the proposed OpenQASM 3

Niel de Beaudrap

Dept. Informatics,
University of Sussex
niel.debeaudrap@sussex.ac.uk

18 December 2020

Abstract

Some remarks pertaining to type annotations and the type system of the proposed OpenQASM 3 syntax, as it stands at 12 December 2020. To provide better consistency with OpenQASM 2, and in particular to achieve more concise and more uniform syntax, I suggest a weak type system involving a hierarchy of classical types, and also recommend that `gate` declarations and `def` subroutines in OpenQASM 3 *not* have explicit type declarations. I outline the syntactical motivations for this, and some semantic ramifications of the suggested conventions.

1 Reflections on OpenQASM 2

OpenQASM 2 is presented as an intermediate representation for quantum computations, and in particular, one which invites the user to consider it as an abstract assembly language. It differs from a typical assembly language in the lack of instruction addressing — justified by the lack of any control flow that would require it, given the presence of `if` statements,

and deliberate omission of loops or recursive subroutines. The `gate` definitions provide a way of describing more complex circuits, and invite an interpretation in terms of control flow and scope, but in practise act more as macro definitions. Finally, the lack of a wide variety of types made it unnecessary to do substantial type-checking of programs: it suffices to ensure that classical arguments are not provided where quantum arguments are required, and vice-versa.

While differing somewhat in presentation from a typical assembly language, the above features allow it to achieve something of the terseness of a typical assembly language. I would regard this as a positive aspect of OpenQASM 2, as it reduces the amount of visual clutter in the language — something which should not be neglected for a format whose aim is to be human-readable.

2 Reconsidering the type syntax of OpenQASM 3

OpenQASM 3 aims to develop upon the functionality of OpenQASM 2, and in particular to offer a wide variety of classical types *and* the capacity to perform computations on such classical types. This is well-motivated by the aim of supporting more versatile control of quantum devices.

I think that the wide variety of classical types is an excellent addition in OpenQASM 3, and that it is reasonable (if not necessarily essential) also to include purely classical computations in the language. It is also reasonable to consider ways to prevent this wide range of classical types from causing confusion while writing quantum programs.

In these respects, the design priorties of OpenQASM 3 extend beyond those of OpenQASM 2. Nevertheless, I argue that it is both desirable and possible to retain the concise, low-level syntax of OpenQASM 2 while supporting these features. This has the significant effect of retaining *nearly* complete backwards compatibility with OpenQASM 2, while presenting a simple and consistent syntax for the extended functionality of OpenQASM 3. In the following, I set out arguments motivating a change in the syntax of type annotations in OpenQASM 3 programs, particularly in `gate` definitions and in `def` subroutines, from that which is currently documented. To this end, I also suggest an approach to types and automatic typecasting.

3 Backwards compatibility with OpenQASM 2?

The willingness and intent to break with complete backwards compatibility with OpenQASM 2 is made clear in a response [1] to a comment I made on the GitHub, at least in the case of `opaque` declarations from OpenQASM 2. There are other signs as well of anticipated departures from OpenQASM 2 syntax. The current OpenQASM 3 documentation provides an example,

```
gate cphase(angle[32]: θ) a, b
{
  U(0, 0, θ/2) a;
  CX a, b;
  U(0, 0, -θ/2) b;
  CX a, b;
  U(0, 0, θ/2) b;
}
cphase(π/2) q[0], q[1];
```

in which the argument θ is given an explicit type. If all gate declarations were to be given type declarations, then OpenQASM 3 would break backwards compatibility with OpenQASM 2 in a more significant way.

In the design of interfaces for quantum computing — particularly in such early stages as we are currently in — backwards compatibility is not to be prized above all else. Usefulness, both in the sense of versatility and accessibility, must take priority; and while backwards-compatibility is itself a form of accessibility, it must take a back-seat to language design choices which make new features easy to understand and access.

As we develop our ideas about how to make use of quantum technology, it is almost inevitable that backwards compatibility will have to be abandoned at some stage in the interests of usefulness. I would argue however that IBM and OpenQASM is not yet at that stage.

In my opinion, IBM has an early-mover advantage in cultivating a user-base who are familiar with their systems. It is likely to be too early to risk alienating existing users by having incompatible versions of a programming interface (identically named apart from revision number). Furthermore, while OpenQASM 3 does introduce a variety of new ideas, it is not especially likely that they require a break from backwards compatibility, except in the most low-level cases (as with `opaque` declarations).

While it is possible to disagree on these points, and possibly to present other arguments to abandon backwards compatibility, I would strongly suggest that OpenQASM 3 be developed *as much as practically possible* — not as a design requirement, but as a high priority — to be backward compatibile with OpenQASM 2. If it will be important to make a more substantial break with backwards compatibility in the near future, this should perhaps be managed by introducing some separate, differently-branded IR or low-level programming language "X" parallel to the OpenQASM languages, and to present some version of OpenQASM (such as OpenQASM 3, or a later version) as a bridge from the existing OpenQASM line to the entirely new paradigm offered by "X". In either case, it would seem advantageous to maintain backwards compatibility of OpenQASM 3 with OpenQASM 2.

Below, I will write as though one of the central design priorities for OpenQASM 3 is to retain backwards compatibility with OpenQASM 2, for anything not requiring pulse-level control. Not much of what I have to say will be relevant if this is not a priority — except perhaps as a way to realise a more concise aesthetic for OpenQASM 3 programs, consistent with the simplicity of OpenQASM 2 syntax.

4 Syntax recommendations for OpenQASM 3

4.1 Recommendation: a hierarchy of 'weak' types

Given as a premise that OpenQASM 3 should aim as much as possible to be backwards compatible with OpenQASM 2, it follows that the following snippet should be valid in OpenQASM 3:

```
qreg q[4];
creg c[4];
// ...
if (c == 4) u1 (pi/2) q[3];
```

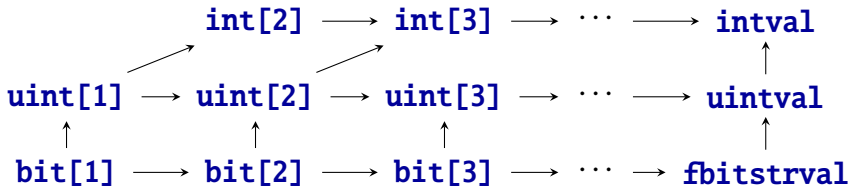
If OpenQASM 3 is to be backwards compatible with OpenQASM 2, it must either not maintain strong type distinctions between bit-strings and integers, or it must allow for automatic typecasting in one direction or the other (*e.g.*, from bit-strings to integers). Furthermore, the current documentation about classical instructions contains a curious example of typecasting:

```
angle[20] a = pi / 2;
angle[20] b = pi;
a + b; // 3/2 * pi
angle[10] c;
c = angle(a + b); // cast to angle[10]
```

There is some type conversion to be done between the result `a + b` and the lvalue `c`, but the cast is not of a non-angular value to an angular one, but rather from higher precision to lower — a detail not described in the casting operation. The conversion to be performed (*i.e.*, one which loses precision in the result) is in principle unambiguous, and so I would suggest that the type convention of OpenQASM 3 require no type-casting here at all, to ease readbility (and possibly reduce confusion as to what the type-cast achieves).

Based on the preceding examples — and given that the types of `length` and `stretch` are already an example of a system of types and subtypes — I would suggest an approach of defining a hierarchy of types, in which we think of the types which may be used in variable declarations as limited-precision and implementation-specific realisations of idealised types.

- Bitstring types `bit[m]` and `creg[m]` are taken as synonyms of one another, and are taken to be subtypes of unsigned integer types `uint[m]` of the same length. At the same time, those same unsigned integer types `uint[m]` are regarded as a subtype of signed integer types `int[m+1]` of greater length. We also order each of these types according to the number of bits in their width, corresponding to padding with zeroes in higher indexed / more significant bits. This gives rise to the following hierarchy of types (where `t` → `u` indicates that `t` is a subtype of `u`):

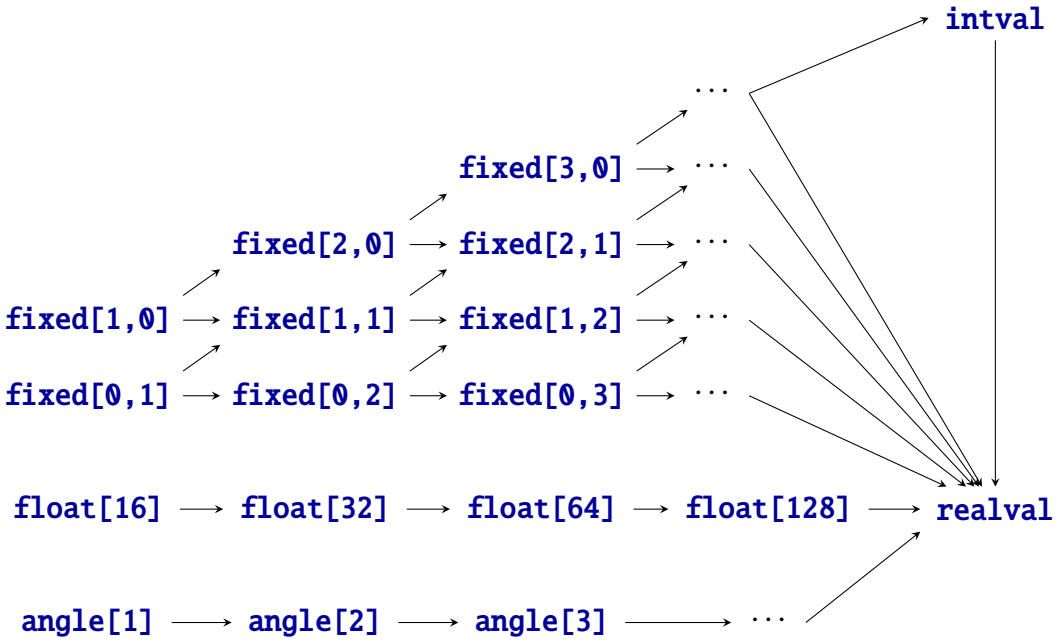


  On the right, we have three limiting 'abstract' types, which cannot be directly instantiated, and are only approximated by their various subtypes:

- **fbitstrval**, of infinitely long bitstrings which only have finitely many **1** symbols (representable using bitstrings of finite length by truncating any infinite tail of **0** symbols);
- **uintval**, of arbitrary unsigned integers (and which are considered to be the proper type of array lengths and bit-widths);
- **intval**, of arbitrary signed integers (and which are considered to be the proper type of individual array indexes and **for** loop bounds/increments/iterators).

We order these as well, so that **fbitstrval** is a subtype of **uintval**, which is a subtype of **intval**.

- Real-valued types **angle[m]**, **fixed[m,f]**, and **float[m]** are all ordered according increasing values of their parameters, and have a common abstract limiting type of **realval**, which is taken to be the proper type of arguments of the **U** operation. We also take **intval** to be the limiting type of **float[m,0]** for increasing **m**, and take **intval** to be a subtype of **realval**.



- For any binary operation or relation whose arguments are not of the same type as one another: if they are both subtypes of a common type,

we interpret them as representing values of their common super-type, and evaluate the result accordingly.

- For any expression in which an expression of some type is assigned to an lvalue of a subtype, we automatically cast the result as follows.
  - We cast a **realval** as a **float[m]** by setting the result to **0** if the mantissa is lower than **float[m]** values can represent, setting the result to **NAN** if the value has a larger mantissa than **float[m]** values can represent, and truncating the precision in the appropriate way otherwise. (We perform a similar operation to cast **float[m]** to **float[n]**, for **m** > **n**.)
  - We cast a **realval** as a **fixed[m,f]** (or as an **int[m+1]**, implemented as **fixed[m,f]** for **f** = 0) by truncating the precision to **f** fractional bits and discarding all but the **m** least significant bits of the integer part. (We perform a similar truncation to cast **fixed[m,f]** as a subtype with fewer integer or fractional bits.)
  - We cast a **realval** as an **angle[m]** by dividing the value by 2π, discarding the integer part, and truncating to **m** bits of precision to represent a fractional multiple of 2π. (We perform a similar truncation of the precision to cast **angle[m]** as **angle[n]** for **m** > **n**.)
  - We cast an **intval** as a **uintval** by substituting any negative value with zero. (We perform a similar substitution to cast **int[m+1]** as **uint[m]**.)
  - We cast a **uintval** as a **uint[m]** by discarding all but the **m** least significant bits. (We perform a similar truncation to cast **uint[m]** as **uint[n]** for **m** > **n**.)
  - We cast a **uintval** as a **fbitstrval** by evaluating its little-endian binary expansion (and similarly for casting **uint[m]** as **bits[m]** for any **m**).
  - We cast a **fbitstrval** as a **bits[m]** by discarding all but the first **m** bits. We perform a similar truncation to cast **bits[m]** as **bits[n]** for **m** > **n**.

Notice that an unwary programmer may set themselves up for some surprising typecasts if they are sloppy in the expressions or comparisons that they write. (For instance: a `float[32]` value with sufficiently large mantissa, so that it does not have any significant figures below the units place when divided by 2π, will evaluate to 0 when typecast as an `angle[m]` value for any `m`.) However, most expressions in practise will be evaluated and typecast with some change of precision in an easily predicted way; and the ability to flexibly re-cast expressions is in my opinion also appropriate to a low-level language such as OpenQASM 3 represents.

4.2 Recommendation: type inference in parameters instead of type annotations

Given as a premise that OpenQASM 3 should aim as much as possible to be backwards compatible with OpenQASM 2, it follows that the following snippet should be valid in OpenQASM 3:

```
gate rotZ (theta) q { U(0, 0, theta) q; }
```

In particular, gate definitions should not require type information to be provided. This entails that, to the extent that OpenQASM 3 makes any commitments at all to the types of parameters in `gate` declarations, a compiler must be capable of inferring these types from context.

One might consider the possibility of having an *optional* type specification, so that the following snippet would also be valid:

```
gate rotZ (angle[32]: theta) q { U(0, 0, theta) q; }
```

However, unless very carefully circumscribed by a specific use-case, I think that this might lead to confusion among programmers both about syntax — are type annotations required, or not? For something as important as `gate` declarations (and declarations of other subroutines), it is most likely a good idea to avoid having multiple allowed forms of syntax except in well-understood special cases, to reduce the already steep learning curve associated with learning to program quantum computers.

There is a second reason to prefer not to have any explicit type declarations for gate arguments. This is because a declaration such as that above,

with a specific type for `theta`, is needlessly specific — no matter which type is given — unless the purpose is to describe a type which is tailored to a specific hardware platform. That is: defining a procedure which takes `angle[32]` parameters makes most sense when the procedure is intended to be run only on a hardware platform which supports operations specifically on `angle[32]` values (or where `angle[32]` operations compile to operations which the platform does support).

As `U` is capable of accepting arguments which are in principle floating-point or fixed-point real numbers, committing to `theta` being an angle specifically seems somewhat arbitrary (even granting that the type system described above would allow such a real value to be correctly interpreted). Unless to target a specific hardware platform, it is unclear what benefit is obtained from declaring `theta` in the snipped above to have a specific precision as a specification of an angle. This gate would be reasonably well-defined whichever angle parameter is passed as an argument, and that the question of the precision available would be one which is more important to the hardware — the number of bits which *it* uses to represent angles, and/or the precision to which angles can be specified — than to the program logic.

The foregoing would suggest that an OpenQASM 3 compiler should (in the vast majority of cases) be left to infer any type information that is required. If we apply this together with the type hierarchy described above, this means that the signatures which the compiler would infer, would involve only the abstract types — `fbitstrval`, `uintval`, `intval`, and `realval` — or those OpenQASM 3 types (such as `bool`, `length`, *etc.*) which lie outside of that type hierarchy. For instance, from

```
gate rotZ ( θ ) q { U(0, 0, θ) q; }
```

and the fact that `U` only takes parameters of `realval`, we infer that the parameter of `rotZ` must also be of type `realval`.

In those cases where the declaration *is* intended to target hardware platforms which support a specific datatype, I also suggest that the syntax more closely match the syntax for variable declaration, *e.g.*:

```
gate rotZ32 ( angle[32] θ ) q { U(0, 0, θ) q; }
```

which omits the colon from the existing syntax.

4.3 Recommendation: revise or eliminate the type annotations from the specifications of `def` subroutines

A final recommendation which I would make regarding type declarations, concerns new functionality in OpenQASM 3, which (by that fact) cannot be motivated by backwards compatibility with the *functionality* of OpenQASM 2. However, having accepted a specific syntax for `gate` declarations (specifically, one which *excludes* type annotations), I would argue that for the sake of stylistic consistency, the type annotations in `def` subroutines should also be changed from that which is currently documented.

The following example syntax is currently presented in the section on subroutines:

```
def xcheck qubit[4]:d, qubit:a -> bit {
  reset a;
  for i in [0: 3] cx d[i], a;
  return measure a;
}
```

In such a declaration, we have annotations for the types of the operands (and a classical byproduct), using a syntax that does not occur anywhere in OpenQASM 2, nor in any other context in OpenQASM 3, excepting perhaps in the syntax for classical parameters. In particular, the only instances where type annotations for quantum data occur is in identifier declarations:

```
qubit d[4];
qubit a;
bit c;
```

where, notably, the width of the register occurs *after* the identifier name.

Note that qubits are the only type (apart from bit registers storing classical return values/byproducts) which may occur as operands in `def` subroutines. This means that describing each such operand as being some number *specifically of qubits* is redundant.[1] I would suggest that

[1] In fairness, it would not be astonishing if in the next few years, it became practical to have a QPU in which qudits of dimension > 2 were exposed to the user interface; even then this might be expected to be the exception rather than the rule.

the operands instead be declared essentially as they would if being declared in the global scope, only omitting the redundant **qubit** or **qreg** specification: for instance,

```
def xcheck d[4], a -> bit {
  reset a;
  for i in [0:3] CX d[i], a;
  return measure a;
}
```

This syntax still has one remaining piece of type annotation, which is the return type **bit**.

This small non-uniformity in syntax — of type annotations specifically for the classical return types of **def** subroutines — is relatively innocent. However, we may remove this non-uniformity as well by considering the prospects to make the syntax more uniform with the OpenQASM 2 syntax for measurements.

In the above examples, the syntax **measure a** lends itself to the interpretation of a procedure with a side-effect on the qubit **a**, with return type **bit**. This syntax would have been well-motivated for OpenQASM 2, and it is also reasonable as a supplementary, alternative syntax to the more assembly-style syntax **measure a -> c** provided in OpenQASM 2. However, in the interests of backwards compatibility, I would suggest that the OpenQASM 2 syntax also be retained.[2] In this syntax, the classical output is stored in a register provided as an operand. Considering this classical value as an operand would suggest that in the example of the **def** subroutine above, we should amend the syntax to

```
def xcheck d[4], a -> c {
  reset a;
  for i in [0:3] CX d[i], a;
  measure a -> c;
}
```

[2] An argument could be made that there should be at most one syntax for storing measurement outcomes in OpenQASM 3, in order to reduce confusion and unimportant distractions over style. In this case, I think that providing two different forms of syntax is worthwhile, to provide an assembly-style syntax (and retain backwards compatibility with OpenQASM 2) on the one hand, and to provide a more conventional procedural-style syntax on the other.

We might then invoke this subroutine with the syntax

```
xckeck q[4], x -> r;
```

for a quantum register `qubit q[4]` and a single qubit `qubit x`, and a classical bit `bit r`. This achieves uniformity in the syntax for operations with classical byproducts, the syntax for the operands of `gate` declarations, and the syntax for declarations of global quantum/classical registers.

As both the 'operand' and 'return value' idioms would be available to the programmer, another realisation of this `def` subroutine might be as follows:

```
def xcheck d[4], a -> c {
  reset a;
  for i in [0:3] CX d[i], a;
  c = measure a;
}
```

Regardless of which way the body of `xcheck` is defined, we could invoke it with either the operand syntax above, or with the invocation

```
r = xckeck q[4], x;
```

If it is not considered too confusing to provide two different forms of syntax for measurement outcomes, this seems harmless enough, and a question perhaps of personal style.

We note that `def` subroutines also give us cause to refine our earlier prescription that subroutine arguments should contain no type annotation information whatsoever. In the current OpenQASM 3 documentation, the following purely classical subroutine is presented:

```
const n = /* some size, known at compile time */;
def parity(bit[n]:cin) -> bit {
  bit c;
  for i in [0: n - 1] {
    c ^= cin[i];
  }
  return c;
}
```

It is clear that the type (specifically, the width) of the argument `cin` could not be inferred from the subroutine body. However, given that no array of any other kind occur in OpenQASM 3 other than those of qubits and classical bits, and given that only classical arguments may occur as parameters, we may adopt the syntax

```
const n = /* some size, known at compile time */;
def parity (cin[n]) -> r {
  r = cin[0];
  for i in [1: n-1]
    r ^= cin[i];
}
```

and thereby indicate both that `cin` is a classical bit register, and specifically that it has width `n`, without an explicit type annotation.

It should be acknowledged that this notation is a departure from the style of a high-level programming language, despite the references to 'type inference' and 'abstract types'. To some extents, it prioritises backwards compatibility with OpenQASM 2, and syntactic uniformity with the concision of OpenQASM 2 in particular, over clarity. I would suppose that the sacrifice to clarity is in this case minor. If one wished to improve clarity, at the cost of uniformity of syntax, it would be reasonable to adopt a syntax such as the following for `def` subroutines:

```
def xcheck qubit d[4], qubit a -> bit {
  bit c;
  reset a;
  for i in [0:3] CX d[i], a;
  measure a -> c;
  return c;
}

const n = /* some size, known at compile time */;
def parity (bit cin[n]) -> bit {
  bit c;
  for i in [0: n-1]
    c ^= cin[i];
  return c;
}
```

```
qubit q[4];
qubit a;
bit c = xcheck d, a;
bit buffer[n];
c = parity(buffer);
```

which includes explicit type annotations (in a style consistent with variable declarations), declarations of classical bits with limited scope, and explicit return statements. This would be a departure from the syntax of `gate` declarations, without sacrificing backwards compatibility with OpenQASM 2 *for* those gate declarations. This may be an alternative declaration convention, which could be offered to allow the programmer to be explicit about their type declarations, and also provide the opportunity to extend the syntax (either sooner or later) to accomodate array arguments or higher-dimensional qudits:

```
def qutritStabiliserMeas (uint[2] zMask[10], uint[2] xMask[10])
  qutrit q[10], qutrit a -> uint[2] r {
    reset a;
    qft3 a;
    for i in [0:9] {
      pow[zMask[i]] @ qutritCZ a, q[i];
      pow[xMask[i]] @ qutritCX a, q[i];
    }
    inv @ qft3 a;
    measure a -> r;
  }
```

5 Closing remarks

In these early days of quantum computation, I have regarded OpenQASM 2 as a promising language to build upon, to describe and perhaps even teach quantum computation. It is concise, and involves very little overhead in the form of structure or bookkeeping, which is appropriate to a platform to be programmed on a very low level. I also recognise and appreciate the relatively responsible role that IBM has played in promoting quantum technologies, and feel that the design of OpenQASM 2 has played a role.

For this reason, I am interested in helping the development of the OpenQASM family of languages to be successful, in growing to play the role of a text-based *langua franca* of the field. In the service of this particular aim, my hopes are for simplicity of syntax (without sacrificing functionality), and consistency of the 'OpenQASM ' brand. This has played a role in my choices of recommendations above.

I offer the above recommendations and ideas in the hope of improving the prospects of OpenQASM 3 for widespread adoption, bearing in mind that uniformity, concision, and clarity of syntax will play a role in how well it is received. As backwards compatibility with OpenQASM 2 will also play such a role, and allow existing users to build on their understanding of OpenQASM 2 to grasp OpenQASM 3, I consider this a very significant priority — even if this backwards compatibility is not absolute, and even if it is soon to be abandoned under the brand of a different platform to be offered in parallel by IBM.

References

[1] T. Alexander. `https://github.com/Qiskit/openqasm/pull/81#issuecomment-744694738`