Introducing BRAT

ROSS DUNCAN, Quantinuum
MARK KOCH, Quantinuum
ALAN LAWRENCE, Quantinuum
CONOR MCBRIDE, University of Strathclyde
CRAIG ROY, Quantinuum

BRAT is a functional programming language for writing programs which interleave quantum and classical execution, with a focus on composition and guarantees of safety. It employs a unique syntax to enable juxtaposition of different parts of circuits, and type level parameters for reasoning about programs during type checking. We illustrate the features of BRAT by showing a number of examples, including some illegal programs, and discuss future directions.

1 INTRODUCTION

Realistic applications of quantum computers involve a mixture of classical and quantum operations, at quite different time-scales. In the large scale we might interleave complete subroutines, such as estimating the value of a quantum observable and calculating its classical parameter updates, as seen in the VQE algorithm [11]. At the small scale, possibly even at real-time, measurement outcomes may determine the control flow that governs later quantum operations, as seen in the recently-demonstrated repeat-until-success pattern [2, 10]. BRAT is a functional programming language designed to allow interleaving these different runtimes in a controlled manner while eliminating sources of costly runtime errors.

BRAT offers a number of distinctive features. Firstly, a unique compositional syntax which is designed to ease the writing of quantum circuits, by analogy to quantum circuit diagrams.

Secondly, BRAT features a strong type system that can catch bugs at compile time. Like Qimaera [3] and Proto-Quipper-D [4], it uses linearity and dependent types to improve the safety of quantum programs and distinguish between quantum and classical data at the type level. BRAT additionally uses its linear type system to track the allocation and deallocation of qubits within a circuit without exceeding a fixed budget.

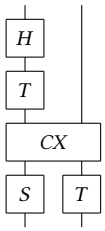
Thirdly, BRAT's strong typing allows programmers to use typed holes when writing their programs and its implementation includes a language server which can connect to their editor make writing programs easier.

2 COMPOSITIONAL SYNTAX

Composition of systems via the tensor product is one of the fundamental principles of quantum mechanics. It forms the foundation of the categorical approach to quantum theory [5] and gives circuits an inherent monoidal structure. In order to capture this structure, BRAT features the notion of monoidal composition as a core part of the language.

Given two values `a :: A` and `b :: B`, we construct a *row* by writing `a,b` and give it the *row type* `A,B`. In the string diagrammatic picture of [5], this correspond to parallel composition of data. Rows are associative, i.e. they satisfy `a,(b,c) == (a,b),c == a,b,c`. Importantly, a row `f,g` of two functions `f :: A -> X` and `g :: B -> Y` actually behaves like a new function `A,B -> X,Y` acting on the row type `A,B`. For example, it may be called by writing `(f,g)(a,b)` or using the syntactic sugar `a,b |> f,g`. This naturally extends parallel composition of data to parallel composition of

Authors' addresses: Ross Duncan, Quantinuum; Mark Koch, Quantinuum; Alan Lawrence, Quantinuum; Conor McBride, University of Strathclyde; Craig Roy, craig.roy@quantinuum.com, Quantinuum.



(a)

```
circ(Qubit, Qubit) -o Qubit, Qubit
circ = {
  H, id;
  T, id;
   CX;
  S, T
}
```

(b)

```
circ(Qubit, Qubit) -o Qubit, Qubit
circ(q1, q2) =
  let q1 = T(H(q1)) in
  let q1, q2 = CX(q1, q2) in
  S(q1), T(q2)
```

(c)

Fig. 1. Circuit (a) and corresponding BRAT implementation in compositional style (b) and point style (c)

computation. BRAT also enables sequential composition using the `;` infix composition operator. For example, given a third function `h :: X,Y -> C,D`, users may write `f,g;h` to obtain a new function `A,B -> C,D`.

This syntax allows us to write quantum circuits purely compositionally. With some suggestive formatting, this results in code that comes very close to the natural string diagrammatic representation of circuits as illustrated by Figure 1b. In comparison, Figure 1c shows an implementation in a more traditional point style which is also supported by BRAT .

Finally, we want to remark that BRAT's type system is designed in the style of Call by Push Value [7]. As such, we differentiate between function types which describe *computations*, and all other types which describe *values*. Computations can be embedded into values by surrounding them with braces, constructing a *thunk*. For example, the code in Figure 1b uses braces to embed the contained computation into a value that is then bound to the name `circ`. Thus, we end up with a *thunk type* `circ :: { Qubit,Qubit -o Qubit,Qubit }`. In Figure 1c the thunking happens implicitly, so no braces are needed.

3 QUANTUM PROGRAMMING

3.1 Quantum Kernels

As seen in Figure 1, BRAT represents quantum operations as functions acting on qubits. We call these functions *quantum kernels* and write their signature with a lollipop (`-o`) instead of the regular arrow used for classical functions (`->`). Kernels are executed on the quantum device whereas classical functions run offline on a classical computer. This way, a glance at a function signature suffices to determine where a given piece of code is executed.

To achieve this, BRAT's type system enforces a strict distinction between the kernel fragment and classical fragment of the language, drawing from work by McBride [8] and Atkey [1]. Kernel types (for example qubits, classical bits and vectors thereof) may never be used in the signature of a classical function and vice versa. In particular, parametrised kernel operations like the R_Z gate have the signature `Rz(Float) -> { Qubit -o Qubit }`, i.e a *classical* function that takes a `Float` and *returns* a quantum kernel `{ Qubit -o Qubit }`. In other words, the computation of the rotation angle occurs offline in the classical fragment and yields a thunked kernel as a result. By composing these operations, users can write classical programs that construct kernels which can then be executed on the quantum computer.

3.2 Linearity

Qubits in BRAT obey *linear typing*, i.e. they cannot be copied or implicitly discarded. This allows us to catch programming errors at compile time, rather than allowing them to propagate to costly

runtime errors. For example, Figure 2 shows programs where a qubit is implicitly dropped or used twice, violating the no-discarding and no-cloning theorems, respectively.

```
fst(Qubit, Qubit) -o Qubit
fst(a, b) = a
```

(a) BRAT will complain that b hasn't been used

```
copy(Qubit, Qubit) -o Qubit, Qubit
copy(qa, qb) = CX(qa, qa)
```

(b) BRAT will complain that qa is used twice

Fig. 2. Type errors caused by violating linearity

3.3 Measurement and Qubit Budget

Another special BRAT feature is the handling of qubits after they are measured. For this purpose, we introduce the `Money` type, drawing from work by Hofmann [6]. Money is returned whenever a qubit is measured and acts as a token that may be used to allocate a new qubit. Concretely, we offer two functions `measure(Qubit) -o Bit,Money` and `fresh(Money) -o Qubit` that may be used to exchange money and qubits. Similar to qubits, `Money` is typed linearly. As a result, the sum of available qubits and money remains constant throughout the program, thus statically tracking the qubit budget available to the user.

3.4 Real-Time Branching

With mid-circuit measurement now widely available, we want to use measurement outcomes to influence the execution of other parts of a circuit. For example, the quantum teleportation protocol requires corrections conditioned on measurement results. For this purpose, we allow users to match on the values of bits returned by `measure`. This not only allows us to conditionally apply gates, but also to completely alter the control-flow of later quantum operations. For example, Figure 3 implements a repeat-until-success protocol from Paetznick and Svore [10] where the same circuit is applied again and again until the measurement outcome is `false`.

```
rus(Money, Qubit) -o Money, Qubit
rus = {
  fresh, id;
  (H;T), id;
      CX;
    H,   id;
      CX;
  (T;H), id;
  measure, id;
  rus'
}

rus'(Bit, Money, Qubit) -o Money, Qubit
rus'(false, m, q) = m, q      -- success
rus'(true, m, q) = rus(m, q)  -- try again
```

Fig. 3. Repeat-until-success protocol from [10].

Note that the matching on `Bit` occurs in the kernel and not the classical fragment. This is in line with the runtime distinction drawn in Section 3.1: Since the branching occurs in real-time on the device, the corresponding code is placed in a kernel. Thus, the user has full control over whether classical logic runs offline or in real-time on the quantum device.

3.5 Dependent Qubit Registers

All examples shown so far implement circuits on a static number of qubits. However, many quantum algorithms actually operate on variable sized qubit registers. In BRAT, these registers are represented by dependent vectors of type `Vec(Qubit, n)` for some length `n`. Using them, we can define variable-sized circuits with the signature `circ(n :: #) -> { Vec(Qubit, n) -o Vec(Qubit, n)}`. Here, `#` is the kind of type-level numbers. Following the discussion in Section 3.1, we can see that `circ` is a classical function that takes a number `n` as input and returns a kernel whose dimension depends on `n`.

```
qft(n :: #) ->
   { Vec(Qubit, n) -o
     Vec(Qubit, n) }
qft(zero) = { [] => [] }
qft(succ(n)) = {
    uncons(n);
    H, vid(n);
    entangle(n, 2);
    id, qft(n);
    recons(n)
}
```

```
entangle(n :: #, Nat)
      -> { Qubit, Vec(Qubit, n) -o
           Qubit, Vec(Qubit, n) }
entangle(zero, _) = { q, qs => q, qs }
entangle(succ(n), x) = {
    id, uncons(n);
    CRx(2.0 * pi / 2 ^ x), vid(n);
    swap, vid(n);
    id, entangle(n, succ(x));
    swap, vid(n);
    id, recons(n)
}
```

Fig. 4. Quantum Fourier Transform in BRAT

For example, Fig. 4 shows a program which recursively generates a Quantum Fourier Transform circuit. Concretely, an $n+1$ qubit QFT consists of a Hadamard gate on the first qubit which is then entangled with all other qubits, followed by an n qubit QFT on the other qubits. Here, `vid(n)` is the identity function on qubit vectors of length `n`. The entanglement step is carried out by the `entangle` function that applies controlled rotations with increasingly smaller angles between the first qubit and all other qubits. As an aside, the `swap` function here isn't strictly necessary, since BRAT offers a feature called *port pulling* which may be used to reorder rows. However, we have omitted this here for brevity. Also note that we are able to use an *irrefutable* pattern match on the empty vector in the `zero` case of `qft`, since we've ensured the length of the vector argument by pattern matching on `n`.

Overall, this example illustrates how dependent types help the programmer to ensure correctness when traversing qubit registers. If the user made a mistake, for example passing the wrong list length into `entangle`, it will be caught by the type checker. Furthermore, termination is ensured since the recursion is bounded by a decreasing `#` parameter.

Finally, we remark that dependent vectors are also available in the classical fragment of BRAT. Additionally, we can use the same dependent function notation to describe classical polymorphic functions. For example, the length function on lists has the polymorphic signature:
`len(X :: *, List(X)) -> Nat`.

4 COMPARISON TO OTHER LANGUAGES

4.0.1 Q#. While similarly functional, Q# doesn't offer any guarantees about linearity of qubits, though it does maintain a separation between classical functions and quantum kernels and otherwise treats them as the same. In a Q# kernel, one can always initialise a new qubit, so it doesn't enforce that the number of available qubits is fixed.

4.0.2 Proto-Quipper-D. This circuit generation language is also based on Quantitative Type Theory [1], which it uses to distinguish between classical functions (to be included in the middle of a program) and kernels. It's notion of "parameter types" are very similar to BRAT's type level values.

In Proto-Quipper-D, qubits are tracked via a `WithGarbage` monad which indicates that new qubits can be instantiated, whereas BRAT's qubit tracking is more fine grained to allow judgement of whether some computations can be composed in our world of limited resources.

4.0.3 Qimaera. BRAT treats the absence of a qubit (i.e. the ability to reset) as a value, which can be arbitrarily structured, whereas Qimaera manages the qubit resources in a circuit by tracking the number of qubits as a type parameter and uses a state monad to deal with measurement and resetting. We believe that with better algebraic manipulation of tensors (beyond treating circuits as

vectors, as Qimaera and current BRAT both do), this will lead to a more convenient programming paradigm.

Qimaera represents quantum gates as data and provides functions for performing juxtaposition and composition of unitaries. In BRAT, quantum gates are treated similarly to classical functions and, thus, the juxtaposition and composition operators are more general. This results in a typechecker which doesn't require the same proof obligations as Qimaera's does for it's unitaries, while still tracking qubit count and facilitating composition.

Dynamic lifting In Qimaera is implicit. The results of a measurement are the standard Idris boolean type, but in BRAT the distinction between the classical and quantum worlds is stronger, making it clear which functions inspecting booleans should run as part of a kernel vs between shots as a post-processing step.

5 FUTURE WORK

5.1 hugr + tket2 Integration

The backend for BRAT is intended to target Quantinuum's nascent quantum IR for quantum programming: hugr [12]. This will allow easy integration with the tket [13] compiler to facilitate targeting a number of different quantum hardware providers.

5.2 Programming Features

Less verbose syntax. Type arguments to polymorphic functions could be omitted from the surface language and implicitly inferred, as in Agda and Haskell. Also, monomorphic identity functions like `vid` and `id` could be omitted in favour of a general, variadic identity operation passing through "the rest of the row type".

More dependent types. So far we have support for types dependent on type-level natural numbers: `#`. However, we intend to extend this with a much richer range of type level values. In particular, generalizing from natural numbers (which can be seen as lists of the uninformative unit type '[]') to arbitrary lists. Thinnings [9] will be able to be used as proofs of $n \leq m$ for `n :: #, m :: #`. Following this, we intend to target ergonomic types for dealing with tensors of data.

REFERENCES

[1] Robert Atkey. 2018. Syntax and Semantics of Quantitative Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science* (Oxford, United Kingdom) *(LICS '18)*. Association for Computing Machinery, New York, NY, USA, 56–65. https://doi.org/10.1145/3209108.3209189

[2] Natalie C. Brown, John Peter Campora, Cassandra Granade, Bettina Heim, Stefan Wernli, Ciaran Ryan-Anderson, Dominic Lucchetti, Adam Paetznick, Martin Roetteler, Krysta Svore, and Alex Chernoguzov. 2023. Advances in compilation for quantum hardware – A demonstration of magic state distillation and repeat-until-success protocols. arXiv:2310.12106 [quant-ph]

[3] Liliane-Joy Dandy, Emmanuel Jeandel, and Vladimir Zamdzhiev. 2021. Qimaera: Type-safe (Variational) Quantum Programming in Idris. arXiv:2111.10867 [cs.PL]

[4] Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. 2023. Proto-Quipper with Dynamic Lifting. *Proc. ACM Program. Lang.* 7, POPL, Article 11 (jan 2023), 26 pages. https://doi.org/10.1145/3571204

[5] Chris Heunen and Jamie Vicary. 2019. *Categories for Quantum Theory: An Introduction.* Oxford University Press. https://doi.org/10.1093/oso/9780198739623.001.0001 arXiv:https://academic.oup.com/book/43710/book-pdf/50991591/9780191060069_web.pdf

[6] Martin Hofmann. 2003. Linear types and non-size-increasing polynomial time computation. *Information and Computation* 183, 1 (2003), 57–85. https://doi.org/10.1016/S0890-5401(03)00009-9 International Workshop on Implicit Computational Complexity (ICC'99).

[7] Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA '99)*. Springer-Verlag, Berlin, Heidelberg, 228–242.

[8] Conor McBride. 2016. *I Got Plenty o' Nuttin'*. Springer International Publishing, Cham, 207–233. https://doi.org/10.1007/978-3-319-30936-1_12

[9] Conor McBride. 2018. Everybody's Got To Be Somewhere. In Proceedings of the 7th Workshop on *Mathematically Structured Functional Programming*, Oxford, UK, 8th July 2018 *(Electronic Proceedings in Theoretical Computer Science, Vol. 275)*, Robert Atkey and Sam Lindley (Eds.). Open Publishing Association, 53–69. https://doi.org/10.4204/EPTCS.275.6

[10] Adam Paetznick and Krysta M. Svore. 2014. Repeat-until-Success: Non-Deterministic Decomposition of Single-Qubit Unitaries. *Quantum Info. Comput.* 14, 15–16 (nov 2014), 1277–1301.

[11] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. 2014. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications* 5, 4213 (07 2014). https://doi.org/10.1038/ncomms5213 arXiv:arXiv:1304.3061

[12] Quantinuum. 2023. *HUGR: Hierarchical Unified Graph Representation.* Quantinuum. https://github.com/CQCL/hugr

[13] 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. https://doi.org/10.1088/2058-9565/ab8e92