$\lambda_{Q\#}$

$Q\#$ as a Quantum **ALGO**rithmic **L**anguage

The Essence of $Q\#$

Kartik Singhal

ks.cs.uchicago.edu

PhD Candidate



Wed/Thu Sept 28/29, 2022

A long-term vision for computing

While a civil engineer as a professional provides specific guarantees about the efficacy and failure modes of a bridge, a software engineer for a software product does not. Computing as a scientific discipline has come a long way, but as an engineering discipline, it has miles to go.

— Vineeth Paleri

A long-term vision for computing

While a civil engineer as a professional provides specific guarantees about the efficacy and failure modes of a bridge, a software engineer for a software product does not. Computing as a scientific discipline has come a long way, but as an engineering discipline, it has miles to go.

— Vineeth Paleri

We are yet to achieve this goal with classical software over the last 70 years; quantum computing, as a fundamentally different way of computation, provides a unique opportunity to do things right in a principled manner.

Approaches to Software Reliability

- Social
  - Code reviews
  - Extreme/Pair programming
- Methodological
  - Design patterns
  - Test-driven development
  - Version control
  - Bug tracking
- Technological
  - "lint" tools, static analysis
  - Fuzzers, random testing
- Mathematical
  - Sound type systems
  - Formal verification



Less "formal": Lightweight, inexpensive techniques (that may miss problems)

This isn't a tradeoff… all of these methods should be used.

Even the most "formal" argument can still have holes:

- Did you prove the right thing?
- Do your assumptions match reality?
- Knuth: *"Beware of bugs in the above code; I have only proved it correct, not tried it."*

More "formal": eliminate *with certainty* as many problems as possible.

Slide Credit: Benjamin Pierce (CIS500: Software Foundations, Lecture 1)

The need for reliable quantum software

Reasoning techniques from classical computing do not transfer:

The need for reliable quantum software

Reasoning techniques from classical computing do not transfer:

State collapse on observation of state.

The need for reliable quantum software

Reasoning techniques from classical computing do not transfer:

State collapse on observation of state.

Simulation beyond tens of qubits is expensive or impossible.

The need for reliable quantum software

Reasoning techniques from classical computing do not transfer:

State collapse on observation of state.

Simulation beyond tens of qubits is expensive or impossible.

The cost of running a program on a physical quantum computer is high; it gets worse if we run incorrect programs.

The need for reliable quantum software

Reasoning techniques from classical computing do not transfer:

State collapse on observation of state.

Simulation beyond tens of qubits is expensive or impossible.

The cost of running a program on a physical quantum computer is high; it gets worse if we run incorrect programs.

Hence, it is important to ensure correctness statically.

The need for reliable quantum software

Reasoning techniques from classical computing do not transfer:

State collapse on observation of state.

Simulation beyond tens of qubits is expensive or impossible.

The cost of running a program on a physical quantum computer is high; it gets worse if we run incorrect programs.

Hence, it is important to ensure correctness statically.

Our approach:

Adapt the best of classical static typing techniques to the quantum realm.[1]

[1] **Singhal**, Hietala, Marshall, Rand. *Q# as a Quantum Algorithmic Language*. QPL 2022

The need for reliable quantum software

Reasoning techniques from classical computing do not transfer:

State collapse on observation of state.

Simulation beyond tens of qubits is expensive or impossible.

The cost of running a program on a physical quantum computer is high; it gets worse if we run incorrect programs.

Hence, it is important to ensure correctness statically.

Our approach:

Adapt the best of classical static typing techniques to the quantum realm.[1]

Bring reasoning techniques as close to programming as possible.[2]

---

[2] **Singhal** & Reppy. *Quantum Hoare Type Theory*. QPL 2020 and Master's Thesis

Outline

The $Q\#$ programming language

$\lambda_{Q\#}$: $Q\#$ as a Quantum Algol

Extended $\lambda_{Q\#}$: Safe $Q\#$ Arrays and More

Additional Directions

Summary

The $Q\#$ programming language (Microsoft, 2017–)

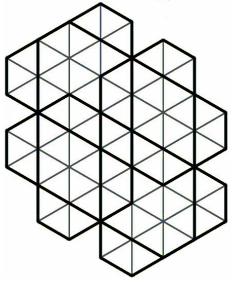
F#-like DSL in the skin of C#-like syntax.[a]

Quantum computer as a co-processor to a classical machine (QRAM).

Clean separation between classical (`function`) and quantum (`operation`) callables. ALGOL-like.

Quantum operations are a monadic sequence of instructions; computation by side effects.

Immutable by default, metaprogramming support (`adjoint` & `controlled` operations).



[a] F# & $Q\#$– A tale of two languages [Azariah 2018]

Teleportation in $Q\#$

```
open Microsoft.Quantum.Intrinsic; // for H, X, Z, CNOT, and M

operation Entangle (qAlice : Qubit, qBob : Qubit) : Unit is Adj {
    H(qAlice);
    CNOT(qAlice, qBob);
}

operation SendMsg (qAlice : Qubit, qMsg : Qubit) : (Bool, Bool) {
    CNOT(qMsg, qAlice);
    H(qMsg);
    return (M(qMsg) == One, M(qAlice) == One);
}

operation DecodeMsg (qBob : Qubit, (b1 : Bool, b2 : Bool)) : Unit {
    if b1 { Z(qBob); }
    if b2 { X(qBob); }
}

operation Teleport (qAlice : Qubit, qBob : Qubit, qMsg : Qubit) : Unit {
    Entangle(qAlice, qBob);
    let classicalBits = SendMsg(qAlice, qMsg);
    DecodeMsg(qBob, classicalBits);
}
```

The need to specify $Q\#$ formally

Sound language design principles lead to programming languages in which programs are easier to write, compose, and maintain.

Previous examples:

- Standard ML [Harper and Stone 2000]
- Featherweight Java [Igarashi, Pierce, and Wadler 2001]; Featherweight Go [Griesemer et al. 2020]
- λ_{JS} [Guha, Saftoiu, and Krishnamurthi 2009]; λ_{Rust} [Jung et al. 2017]

> *$Q\#$ is a living body of work that will grow and evolve over time.*
>
> — *Design Principle 5* [*Heim 2020, Ch. 8*]

Having a well-founded meta-theory of a programming language helps with its evolution.

A recipe for formal language specification[3]

Inside every large language is a small language struggling to get out…
— Tony Hoare

1. Define a well-behaved internal language (a small core).
2. Define an elaboration relation from the external language to the internal language.
3. Specify static and dynamic semantics using the internal language.
   Statics (type system) rule out meaningless programs.
   Dynamics specify behavior of programs at a high abstraction level.
4. Prove meta-theorems such as type preservation and safety.

Study consequences of extensions and variations.

[3] A Type-Theoretic Interpretation of Standard ML [Harper and Stone 2000]

Unsafe programs in $Q\#$

```
operation Cloning () : Unit {
  use q1 = Qubit();
  let q2 = q1;
  CNOT(q1, q2);
}
```

Using same qubit as both control and target.

```
operation Dangling ()
: Qubit {
  use q = Qubit();
  return q;
}
```

Returning a qubit after its lifetime has ended.

$\lambda_{Q\#}$: a core calculus for $Q\#$ (Types)

Qubit Symbols	q			
Typ	τ	$::=$	$\texttt{qref}\langle q \rangle$	qubit ref
			$\texttt{fun}(\tau_1 \,;\, \tau_2)$	function
			$\texttt{cmd}(\tau)$	command
			$\texttt{prod}(i \hookrightarrow \tau_i \mid i \in n)$	n-ary product
			`bool`	boolean
			`unit`	unit

Key idea: **`Qubit`** type in $Q\# \equiv \texttt{qref}\langle q \rangle$ type of indexed qubit references in $\lambda_{Q\#}$.[4]

[4] Alias Types [Smith et al. 2000]

$\lambda_{Q\#}$: Expressions

Exp	e	$::=$	x	variable
			$\texttt{let}[\tau_1\,;\tau_2](e_1\,;x\,.\,e_2)$	let binding
			$\lambda[\tau_1\,;\tau_2](x\,.\,e)$	function
			$\texttt{ap}[\tau_1\,;\tau_2](e_1\,;e_2)$	application
			$\texttt{tuple}[\tau_n](i \hookrightarrow e_i \mid i \in n)$	tuple
			$\texttt{proj}\langle i\rangle[\tau_n](e)$	projection
			$\texttt{true}$	true
			$\texttt{false}$	false
			$\texttt{if}[\tau](e\,;e_1\,;e_2)$	if expression
			$\texttt{unit}$	unit
			$\texttt{cmd}[\tau](m)$	encapsulated command

Essentially, **PFPL**'s language **MA** (Modernized Algol) [Harper 2016, Ch. 34]

$\lambda_{Q\#}$: Commands

Unitaries	U, V			
Cmd	m	$::=$	$\texttt{ret}[\tau](e)$	return
			$\texttt{bnd}[\tau_1 ; \tau_2](e ; x \,.\, m)$	bind
			$\texttt{newqref}[\tau](x \,.\, m)$	new qbit ref
			$\texttt{gateap}\langle U_{2^n}\rangle(e)$	gate application
			$\texttt{diagap}\langle U_{2^n}, V_{2^n}\rangle(e_1 ; e_2)$	controlled gate app.
			$\texttt{meas}(e)$	measure

Key idea: The allocation command, $\texttt{newqref}(x \,.\, m)$, comes with its own binding form so that qubits can never escape lexical scope resulting in an ALGOL-like stack discipline.

Typing of quantum commands

$\boxed{\Gamma \vdash_\Sigma m \div \tau}$ *(m is a well-formed command relative to Σ, returning a value of type τ)*

CMD-NEWQREF

$$\frac{\Gamma, x : \mathbf{qref}\,\langle q \rangle \vdash_{\Sigma,q} m \;\div\; \tau}{\Gamma \vdash_\Sigma \mathbf{newqref}\,(x.m) \;\div\; \tau}$$

CMD-GATEAPREF

$$\frac{\Gamma \vdash_\Sigma e : \mathbf{prod}\,(\overline{i \hookrightarrow \mathbf{qref}\,\langle q_i \rangle}^{\,i \in 1..n})}{\Gamma \vdash_\Sigma \mathbf{gateap}\,\langle U_{2^n} \rangle (e) \;\div\; \mathbf{unit}}$$

CMD-DIAGAPREF

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathbf{qref}\,\langle q \rangle \qquad \Gamma \vdash_\Sigma e_2 : \mathbf{prod}\,(\overline{i \hookrightarrow \mathbf{qref}\,\langle r_i \rangle}^{\,i \in 1..n})}{\Gamma \vdash_\Sigma \mathbf{diagap}\,\langle U_{2^n}, V_{2^n} \rangle (e_1; e_2) \;\div\; \mathbf{unit}}$$

CMD-MEASREF

$$\frac{\Gamma \vdash_\Sigma e : \mathbf{qref}\,\langle q \rangle}{\Gamma \vdash_\Sigma \mathbf{meas}\,(e) \;\div\; \mathbf{bool}}$$

Signature, Σ, keeps track of qubit symbols in scope, each qubit symbol is distinct.

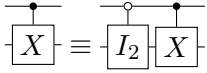
Statically preventing cloning

```
operation Cloning () : Unit {
  use q1 = Qubit();
  let q2 = q1;
  CNOT(q1, q2);
}
```

In $\lambda_{Q\#}$ abstract syntax:

$$\texttt{newqref}(q_1\,.\,\texttt{ret}(\texttt{let}(q_1\,;q_2\,.\,\texttt{cmd}(\texttt{apply}_{D(I_2,X)}(q_1\,;q_2)))))$$

$$CNOT \triangleq \equiv$$



Statically preventing cloning

```
operation Cloning () : Unit {
  use q1 = Qubit();
  let q2 = q1;
  CNOT(q1, q2);
}
```

CMD-DIAGAPREF

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathbf{qref}\,\langle q \rangle \qquad \Gamma \vdash_\Sigma e_2 : \mathbf{prod}\,(\overline{i \hookrightarrow \mathbf{qref}\,\langle r_i \rangle}^{\,i \in 1..n})}{\Gamma \vdash_\Sigma \mathbf{diagap}\,\langle U_{2^n}, V_{2^n} \rangle (e_1; e_2) \;\dot{\sim}\; \mathbf{unit}}$$

In $\lambda_{Q\#}$ abstract syntax:

$$\texttt{newqref}(q_1\,.\,\texttt{ret}(\texttt{let}(q_1\,;\,q_2\,.\,\texttt{cmd}(\texttt{apply}_{D(I_2,X)}(q_1\,;\,q_2)))))$$

but $q_1 : \texttt{qref}\langle q_1 \rangle$ and $q_2 : \texttt{qref}\langle q_1 \rangle$ have the same type!

Safe qubit management

```
operation Dangling ()
: Qubit {
  use q = Qubit();
  return q;
}
```

In $\lambda_{Q\#}$ abstract syntax:

$$\lambda(_\,.\,\texttt{cmd}(\texttt{newqref}(x\,.\,\texttt{ret}(x))))$$

Safe qubit management

```
operation Dangling ()
: Qubit {
  use q = Qubit();
  return q;
}
```

CMD-NEWQREF

$$\frac{\Gamma, x : \mathbf{qref}\,\langle q\rangle \vdash_{\Sigma,q} m \;\dot{\div}\; \tau}{\Gamma \vdash_{\Sigma} \mathbf{newqref}\,(x.m) \;\dot{\div}\; \tau}$$

In $\lambda_{Q\#}$ abstract syntax:

$$\lambda(_\,.\,\mathtt{cmd}(\mathtt{newqref}(x\,.\,\mathtt{ret}(x))))$$

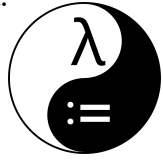
$\Gamma, x : \mathtt{qref}\langle q\rangle \vdash_{\Sigma,q} \mathtt{ret}(x) \dot{\div} \mathtt{qref}\langle q\rangle$

but $\Gamma \nvdash_{\Sigma} \mathtt{newqref}(x\,.\,\mathtt{ret}(x)) \dot{\div} \mathtt{qref}\langle q\rangle$

$\lambda_{Q\#}$ and $Q\#$ are ALGOL-like quantum languages

Safely combine pure (classical) and effectful (quantum) computation.

Obey strict stack discipline for (qubit) memory management.



Functional/Imperative Synthesis

More in the paper:

Equational semantics based on Staton's fully complete equational theory for quantum computation.

Elaboration rules from $Q\#$ to $\lambda_{Q\#}$.

QPL 2022: Kartik Singhal, Kesha Hietala, Sarah Marshall, and Robert Rand. "$Q\#$ as a Quantum Algorithmic Language." arXiv:2206.03532

Extended $\lambda_{Q\#}$: Safe $Q\#$ Arrays and More

Arrays are highly expressive but also challenging to type safely.

```
operation ApplyCNOTChain(qs : Qubit[])
: Unit is Adj + Ctl {
    ApplyToEach(CNOT, Zipped(Most(qs), Rest(qs)));
}
```

Idea:

make implicit specifications in $Q\#$ arrays library explicit

automate reasoning using an industry-scale solver like Z3.

Types and (implicit) specifications

```
operation CNOT (control : Qubit, target : Qubit)
: Unit is Adj + Ctl

operation ApplyToEach<'T> (op : ('T => Unit), reg : 'T[])
: Unit

/// An array containing the elements
/// array[0..Length(array) - 2].
function Most<'T> (array : 'T[]) : 'T[]

/// An array containing the elements
/// array[1..Length(array) - 1].
function Rest<'T> (array : 'T[]) : 'T[]

/// An array containing pairs of the form
///  (left[idx], right[idx]) for each idx.
function Zipped<'T, 'U> (left : 'T[], right : 'U[])
: ('T, 'U)[]
```

Whiley for prototyping

Whiley is a “hybrid imperative and functional programming language” that “provides static checking of pre- and postconditions”.

Boogie is an intermediate verification language and compiler, which acts as a high-level language interface for Z3.

Whiley switched from native solver to Z3 by translating to Boogie.

> *Wy2B/Boogie/Z3 stack offers significant advantages over the native Whiley verifier in terms of the percentage of programs that can be verified automatically. — Pearce, Utting, and Groves [JAR 2022]*

Prototype in Whiley (1)

```
type qbit       is (int n)          where n >= 0
type safePair   is {qbit p, qbit q} where p != q
type distinctOp is method(safePair) -> null

method CNOT(safePair xy) -> null

method ApplyToEach(distinctOp op, safePair[] reg) -> null

function Most(qbit[] qs) -> (qbit[] out)
  requires |qs| > 0
  ensures  |out| == |qs| - 1
  ensures  all { i in 0..|out| | out[i] == qs[i] }

function Rest(qbit[] qs) -> (qbit[] out)
  requires |qs| > 0
  ensures  |out| == |qs| - 1
  ensures  all { i in 0..|out| | out[i] == qs[i + 1] }
```

Prototype in Whiley (2)

```
function Zipped(qbit[] xs, qbit[] ys) -> (safePair[] xys)
  requires |xs| == |ys|
  requires all { i in 0..|xs| | xs[i] != ys[i] }
  ensures  |xys| == |xs|
  ensures  all { i in 0..|xys| | xys[i].p == xs[i]}
  ensures  all { j in 0..|xys| | xys[j].q == ys[j]}:

method ApplyCNOTChain(qbit[] qs) -> null
  requires |qs| >= 2
  requires all { i in 0..|qs| | qs[i] == qs[0] + i }
```

Above specifications and fairly obvious implementations type check automatically. (`Zipped` requires a loop invariant.)

Next: generalize and port to $\lambda_{Q\#}$. w. Jacob Zweifler

Coverage of other $Q\#$ features

Iteration and slices will benefit from the approach just described.

`mutable` bindings are straightforward using ALGOL-style local store.

Parametric polymorphism for $Q\#$ callables is well understood.

Still working on language-level quantum abstractions:

- `within-apply` and `adjoint` functor are similar.
- `repeat-until-fixup` being non-trivial will require care.

Additional Directions

Mechanized metatheory for the core language.

Alternatively, use a tested-semantics approach to validate formalization.

Steps toward a unified framework for specifying, programming, and reasoning about quantum programs.

Semantics preserving compilation to major quantum intermediate languages:

- QIR [Geller 2020] (LLVM-based)
- OpenQASM 3 [Cross et al. 2021]

while integrating with existing tools such as Vellvm (verified LLVM).

Summary

$Q\#$ is one of the only industry-backed standalone quantum programming languages.

In our work, we:

- give $Q\#$ a formal language definition via translation to $\lambda_{Q\#}$.
- extend the $Q\#$ type system for safer quantum programming.

Acknowledgments

Collaborators: Kesha Hietala, Sarah Marshall, Robert Rand, Jacob Zweifler
MS Quantum: Alan Geller, Bettina Heim, Brad Lackey, Aarthi Sundaram
Community: Bob Harper, Ohad Kammar, Jennifer Paykin, Sam Staton