Neko: A quantum map-filter-reduce programming language

ELTON PINTO, Georgia Institute of Technology, USA

1 INTRODUCTION

Programming quantum computers is hard. One has to painstakingly write code that builds a circuit using low-level quantum gates [Svore et al. 2018] [Abraham et al. 2019] [Luo et al. 2020]. In a way, writing a quantum program is analogous to writing assembly: it is tedious, error-prone, and hard to debug. The gate-level abstraction, albeit universal, is non-intuitive and too primitive to be used for rapidly prototyping large-scale quantum applications. There is a need to develop high-level abstractions that enable programmers to productively leverage the idiosyncrasies of quantum computing: quantum parallelism, interference, and entanglement.

In this ongoing work, I present Neko, a high-level quantum programming language that exposes a map-filter-reduce interface for exploiting quantum parallelism through the notion of *first-class superpositions.*

2 BACKGROUND

A quantum computer works by manipulating qubits. Unlike classical bits, qubits can exist in a *superposition*, a linear combination over basis states. Superposition enables the notion of *quantum parallelism* where one can apply a reversible function over the entire superposition in parallel. However, the caveat is that you can measure only one of the outputs. Quantum algorithms employ *interference* to increase the probability of measuring the desired output.

Several quantum algorithms [Grover 1996] [Shor 1999] [Deutsch and Jozsa 1992] follow the pattern of (1) prepare a superposition over the input space, (2) map a function over this superposition, and (3) use interference to filter and reduce over the mapped space. It would then be natural to ask: can we abstract over this pattern?

Suppose we are given an initial quantum state $|\psi\rangle = |0\rangle^{\otimes n} |0\rangle^{\otimes m}$ consisting of an n-qubit index register and an m-qubit data register. We then wish to abstract over the following transformation:

$$|\psi\rangle \xrightarrow{\text{superpose}} \sum_{i=0}^{2^n-1} \alpha_i |i\rangle |0\rangle^{\otimes m} \tag{1}$$

$$\xrightarrow{\text{map}} \sum_{i=0}^{2^n-1} \alpha_i |i\rangle |f(i)\rangle \tag{2}$$

$$\xrightarrow{\text{filter/reduce}} \beta_s |\text{success}\rangle |G(f)\rangle + \beta_f |\text{failure}\rangle |\psi_f\rangle + \beta_e |\text{error}\rangle |\psi_e\rangle \tag{3}$$

where $G(f) = G(f(0), \ldots, f(2^n - 1))$ is a property of the function f based on its individual values, $|\text{success}\rangle$, $|\text{failure}\rangle$ and $|\text{error}\rangle$ are marker indices that allow us to discern the nature of the measured output of the data register, and $|\beta_s|^2 > 0$, i.e., there is a non-zero probability of measuring the correct result. Neko enables you to program in this style using the map-filter-reduce paradigm.

3 NEKO: A QUANTUM MAP-FILTER-REDUCE PROGRAMMING LANGUAGE

Neko is a strictly-typed functional programming language equipped with *first-class superpositions*, allowing users to ergonomically leverage quantum parallelism. It supports a rich set of base types (units, booleans, integers, floats, and lists), first-class functions, let-expressions, conditionals, tensors (which are a generalization over pairs), and reference cells (or refs).

Graduate (MSc.) student advised by Vivek Sarkar, ACM number: 2093702, email: epinto6@gatech.edu.

```
(* Put a list of integers into superposition *)
let xs = superpose [0,1,2,3,4,5,6,7] in

(* Increment each number (in parallel) by 32 *)
let ys = map (λ x. x + 32) xs in

(* Filter out even numbers from the superposition *)
let zs = filter (λ x. x % 2 = 0) in

(* Sample from the superposition *)
sample zs
```

Fig. 1. Example Neko program that samples an even number

A Neko expression can be put into a superposition using the `superpose` primitive. A superposition is made up of individual portions called *chunks* each of which consists of an amplitude and an expression. A superposition can be manipulated using `map`, `filter`, and `reduce`, and can be sampled using `sample`, where the probability of observing one of its chunks is equal to the square of its amplitude. An expression is said to execute in the *context of a superposition* if it is applied to the chunks of a superposition via `map`, `filter`, or `reduce`.

`map` f e applies the function f to each portion of the superposition, leaving the amplitudes unchanged. `filter` f e increases the probability of sampling any chunk that satisfies the predicate f. Finally, `reduce` f e combines chunks using f. The semantics of these operations is formally denoted by Equation (1), Equation (2), and Equation (3).

An expression can be put into a superposition only if its corresponding type implements the *superposition interface*. This interface requires the type to specify the number of chunks and a procedure to produce each chunk given an index. For example, the `list` type could implement the interface by specifying the number of chunks to be equal its length, with each chunk corresponding to an individual element of the list. The superposition interface can also be implemented for scalar types. For example, you can realize a superposition representing a fair coin toss by implementing the interface for the boolean type, with `true` and `false` being the individual chunks.

The semantics of refs is complicated by the presence of superpositions in the language. If an expression executing in the context of a superposition modifies a ref differently for at least two chunks, then the ref becomes *entangled* with the superposition. Put another way, refs exhibit copy-on-write behavior where each chunk modifies its own copy of the ref. An entangled ref cannot be accessed or modified from outside the context of a superposition until it becomes *unentangled*, which can be done by sampling all superpositions with which it has been entangled. The type checker uses a conservative analysis to detect potentially entangled refs that are accessed outside the context of a superposition.

The peculiar characteristics of superpositions coupled with the copy-on-write semantics of reference cells gives rise to the $e \mid \mu \ \mapsto_m \ e' \mid \mu'$ "steps-to" relation, where e, e' are expressions, μ, μ' are stores, and m is the *evaluation mode*. The store μ maps locations l to a tensor of values, and is accessed as $\mu(l)[d][i]$, where d is a *dimension* and i is an index. Each superposition is annotated with a *location-dimension* list, which keeps track of the refs that are potentially entangled with it. The mode m can be either normal (N) or superposed (S_i, where i is an index) and is used to discern expressions that are evaluated in the context of a superposition.

Figure 1 shows a simple Neko program that samples an even number from a list of integers. Appendix A has additional examples along with the full syntax and select operational semantics.

4 COMPILATION TO QUANTUM CIRCUITS

Neko is compiled to a simple quantum circuit language with support for qubit management through qubit registers, single-qubit and multi-qubit gates, qRAM, and measurement. The operational semantics and typing rules are largely similar to that of Hietala et al. [2021] augmented with qubit management and qRAM functionality.

An expression is put into a superposition by entangling an index register put into an equally weighted superposition with data corresponding to its chunks. This encoding results in an exponential reduction in space-complexity, using only a logarithmic amount of space. `map` f e is implemented by compiling f to a reversible gate (using techniques described by Amy et al. [2017] and Li et al. [2022]) and applying it to the superposition. `filter` f e is implemented using Grover search over the predicate f, giving us a quadratic improvement in time-complexity.

Realizing `reduce` f e is not straightforward because the chunks of a quantum superposition cannot share memory (which is precisely why refs are modelled as copy-on-write). Lu et al. [2018] present a deterministic quantum adder for reducing superposition states that purportedly runs exponentially faster than existing classical routines. If their claims are true, I could generalize their work to handle any associative operator. Unfortunately, this is not the case. I show that their algorithm ignores the effects of entanglement which in turn falsifies both, their correctness and time-complexity claims.

One way of realizing `reduce` f e is to replicate the n-element superposition $|\psi\rangle$ and output a circuit that produces $|\psi'\rangle = |\psi\rangle \otimes \cdots \otimes |\psi\rangle$. This state contains all possible permutations (with repetition) of the n-element superposition. Reducing over f can then be implemented by first mapping f and then filtering out the desired permutation. This solution, of course, undoes the space-savings afforded by a superposition.

Are there cases where we can do better? Deutsch [1985] provides an answer. There exists a class of functions f that are computable by quantum parallelism (QPC), i.e., the transformation from Equation (2) to Equation (3) can be realized with $\beta_e = 0$ and little-to-no replication. The class of QPC functions was characterized by Jozsa [1991] using a linear relations formalism. I hope to leverage this in the Neko compiler.

5 RELATED WORK

There have been attempts to develop high-level abstractions for programming quantum computers. Tower [Yuan and Carbin 2022] approaches the problem from the perspective of designing quantum data structures as pointer-based, linked data. Aleph [Paz 2022] develops the notion of *quantum universes*, a high-level abstraction for manipulating quantum states. Neko differs from Aleph in several regards: (1) superpositions in Neko are fine-grained and can be manipulated in more sophisticated ways à la map-filter-reduce, (2) Neko attempts to tackle the problem of reducing a superposition, (3) Neko presents a semantics for copy-on-write refs.

6 FUTURE WORK

The next steps for this work is to implement the Neko compiler, iron out any kinks, and prove relevant properties (like type safety). I hope to continue exploring the implementation of the `reduce` operator, leveraging the QPC characterization developed by Jozsa [1991]. An interesting future direction would be to investigate richer primitives for manipulating the amplitudes of a chunk à la quantum signal processing (QSP) [Low and Chuang 2017] and quantum singular value transforms (QSVT) [Gilyén et al. 2019].

REFERENCES

Héctor Abraham, Ismail Yunus Akhalwaya, Gadi Aleksandrowicz, Thomas Alexander, G Alexandrowics, E Arbel, A Asfaw, C Azaustre, P Barkoutsos, G Barron, et al. 2019. Qiskit: An open-source framework for quantum computing, 2019. *URL https://doi. org/10.5281/zenodo* 2562110 (2019).

Matthew Amy, Martin Roetteler, and Krysta M Svore. 2017. Verified compilation of space-efficient reversible circuits. In *International Conference on Computer Aided Verification*. Springer, 3–21.

David Deutsch. 1985. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400, 1818 (1985), 97–117.

David Deutsch and Richard Jozsa. 1992. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 439, 1907 (1992), 553–558.

András Gilyén, Yuan Su, Guang Hao Low, and Nathan Wiebe. 2019. Quantum singular value transformation and beyond: exponential improvements for quantum matrix arithmetics. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. 193–204.

Lov K Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 212–219.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (jan 2021), 29 pages. https://doi.org/10.1145/3434318

Richard Jozsa. 1991. Characterizing classes of functions computable by quantum parallelism. *Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences* 435, 1895 (1991), 563–574.

Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2022. Verified compilation of Quantum oracles. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 589–615.

Guang Hao Low and Isaac L Chuang. 2017. Optimal Hamiltonian simulation by quantum signal processing. *Physical review letters* 118, 1 (2017), 010501.

Xiaowei Lu, Nan Jiang, Hao Hu, and Zhuoxiao Ji. 2018. Quantum adder for superposition states. *International Journal of Theoretical Physics* 57, 9 (2018), 2575–2584.

Xiu-Zhe Luo, Jin-Guo Liu, Pan Zhang, and Lei Wang. 2020. Yao. jl: Extensible, efficient framework for quantum algorithm design. *Quantum* 4 (2020), 341.

Andres Paz. 2022. aleph. https://github.com/anpaz/aleph

Peter W Shor. 1999. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review* 41, 2 (1999), 303–332.

Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q# enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the real world domain specific languages workshop 2018*. 1–10.

Charles Yuan and Michael Carbin. 2022. Tower: Data Structures in Quantum Superposition. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2, Article 134 (oct 2022), 30 pages. https://doi.org/10.1145/3563297

A MORE ABOUT NEKO

A.1 Example Programs

In each of these examples,

- Neko is extended with support for pattern matching and arrays
- range i j is syntax sugar for $[i, i+1, \ldots, j-1]$.
- for i in j; do e end is syntax sugar for $e[i := j[0]];\ e[i := j[1]];\ \ldots;\ e[i := j[n]]$

Summing up a list of integers. The following program sums all of the odd numbers from 1 to 20.

```
let x = superpose (range 1 20) in

(* Filter out the odd numbers *)
let odd = filter (λ x. x % 2 <> 0) xs in

(* Compute the sum *)
let sum = reduce (λ x y. x + y) odd in

(* Sample the solution *)
sample sum
```

Solving a system of equations. The following program was ported to Neko from Aleph [Paz 2022].

```
let x = superpose (range 0 4) in
let y = superpose (range 0 4) in

(* Create all possible pairs of inputs *)
let input = x ⊗ y in

(* Filter out the input pairs that satisfy both equations *)
let output = filter (λ (x, y).
    let eq1 = 4 * x + 5 * y in
    let eq2 = (-6) * x + 20 * y in
    eq1 = eq2
) inputs
in

(* Sample one of the solution pairs *)
sample output
```

Instruction ordering. The following program executes all possible instruction orderings for a simple register-based assembly language containing three instructions: add, load, and store. It uses exponentially less space for storing all possible instruction orderings compared to a classical implementation.

```
(* Instruction = (opcode, dst, src1, src2) *)

(* dst := src1 + src2 *)
```

```
let add dst src1 src2 = 0 ⊗ dst ⊗ src1 ⊗ src2
in

(* dst := mem[src1 + src2] *)
let load dst src1 src2 = 1 ⊗ dst ⊗ src1 ⊗ src2
in

(* mem[dst + src1] := src2 *)
let store dst src1 src2 = 2 ⊗ dst ⊗ src1 ⊗ src2
in

let r0 = 0 in
let r1 = 1 in
let r2 = 2 in

(* Initialize register file as a 3-element zero-initialized array *)
let regs = [| 0; 3 |] in

(* Initialize memory as a 32-element zero-initialized array *)
let mem = [| 0; 32 |] in

(* Execute an instruction *)
let execute (op, dst, src1, src2) =
    if op = 0 then
        regs[dst] := regs[src1] + regs[src2]
    else if op = 1 then
        regs[dst] := mem[(regs[src1] + regs[src2]) % 32]
    else if op = 2 then
        mem[(regs[dst] + regs[src1]) % 32] := regs[src2]
    else
        ()
in

let prog = superpose [
    add r0 r1 r2,
    load r1 r2 r0,
    store r0 r1 r1
]
in

let prog_orderings = prog ⊗ prog ⊗ prog in

(* Since `execute` is executed in the context of a superposition,
   `mem` and `regs` exhibit copy-on-write behavior and become
   entangled with `prog_orderings` *)
let executions = map (λ instrs.
    for instr in instrs; do
        execute instr
    end
) prog_orderings
in

(* Sample one of the executions *)
sample executions
```

A.2 Syntax

$$
\begin{array}{rll}
\text{Bool } b & ::= \mathsf{true} \mid \mathsf{false} & \\
\text{Integer } i & \in \mathbb{Z} & \\
\text{Real } r, \alpha & \in \mathbb{R} & \\
\text{Binary ops } bop & ::= + \mid * \mid \mathsf{and} \mid \mathsf{or} \mid = \mid \mathsf{<>} \mid \% & \\
\text{Unary ops } uop & ::= - \mid \mathsf{not} & \\
\text{Dimension } d & \in \mathbb{N} & \\
\text{Location } l & & \\
\text{Variable } x & & \\
\text{Type } \tau & ::= \mathsf{unit} \mid \mathsf{bool} \mid \mathbb{Z} \mid \mathbb{R} & \textit{Base types} \\
& \mid \mathsf{ref}\ \tau & \textit{Ref type} \\
& \mid \mathsf{list}\ \tau & \textit{List type} \\
& \mid \tau_1 \otimes \tau_2 & \textit{Tensor type} \\
& \mid \langle(\tau_{\mathrm{t}}, \tau_{\mathsf{chunk}})\rangle & \textit{Superposition type} \\
\text{Expression } e, f & ::= () \mid b \mid i \mid r & \textit{Literals} \\
& \mid [e_1, \ldots, e_n] & \textit{Lists} \\
& \mid \langle(\alpha_1, e_1), \ldots, (\alpha_n, e_n)\rangle^{[(l,d)]} \mid \mathsf{superpose}\ e \mid \mathsf{sample}\ e & \textit{Superpositions} \\
& \mid x \mid \lambda x.e \mid e_1\ e_2 & \textit{Lambda forms} \\
& \mid e_1 \otimes e_2 \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e & \textit{Tensors} \\
& \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 & \textit{Binding} \\
& \mid e_1\ bop\ e_2 & \textit{Binary ops} \\
& \mid uop\ e & \textit{Unary ops} \\
& \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \textit{Conditionals} \\
& \mid e_1;\ e_2 & \textit{Sequencing} \\
& \mid \mathsf{ref}\ e \mid e_1 := e_2 \mid !e & \textit{References} \\
& \mid \mathsf{map}\ f\ e \mid \mathsf{filter}\ f\ e \mid \mathsf{reduce}\ f\ e & \textit{Map, filter, reduce}
\end{array}
$$

$$\frac{\text{mode } m \qquad f = \lambda x.e \qquad xs = \langle(\alpha_i, v_i)\rangle_n^A \\ f\ v_j \mid \mu_{L'} \rightsquigarrow_{S_j,f} v'_j \mid \mu'_{L'}}{\texttt{map}\ f\ xs \mid \mu \mapsto_m \langle(\alpha_i, v'_i)\rangle_n^{L'} \mid \mu'} \text{ E-MapSup}$$

$$\frac{\text{mode } m \qquad f = \lambda x.e \qquad xs = \langle(\alpha_i, v_i)\rangle_n^A \\ k \in P \subseteq \{1, \dots, n\} \qquad k' \in \{1, \dots, n\} \setminus P \\ f\ v_k \mid \mu_{L'} \rightsquigarrow_{S_k,f} \texttt{true} \mid \mu'_{L'} \qquad f\ v_{k'} \mid \mu_{L'} \rightsquigarrow_{S_{k'},f} \texttt{false} \mid \mu'_{L'}}{\texttt{filter}\ f\ xs \mid \mu \mapsto_m \langle(p(i), v_i)\rangle_n^{L'} \mid \mu' \text{ where} \\ p(i) = \begin{cases} \beta_s & i \in P \\ \beta_f & \text{otherwise} \end{cases}, \quad |1 - |\beta_s|^2| < \epsilon} \text{ E-FilterSup}$$

$$\frac{\text{mode } m \qquad f = \lambda x.\lambda y.e \qquad xs = \langle(\alpha_i, v_i)\rangle_n^A \\ f \text{ is associative} \qquad \text{modified?}(f) = \emptyset}{\texttt{reduce}\ f\ xs \mid \mu \mapsto_m \langle(\beta_s, G(f)), (\beta_{f_i}, v_{f_i})^k, (\beta_{e_i}, v_{e_i})^l\rangle_n^A \mid \mu \text{ where} \\ G(f) \text{ reduces the values } v_0, \dots, v_n \text{ using } f \text{ in an arbitrary order} \\ |\beta_s|^2 > 0} \text{ E-ReduceSup}$$

Fig. 2. Operational semantics for map-filter-reduce on superpositions

A.3 Operational Semantics

Figure 2 shows the operational semantics for the map-filter-reduce primitives on a superposition. There is evidently a lot going on. Nonetheless, at a high level, these inference rules encode Equation (1), Equation (2), and Equation (3) with additional judgements for handling references.

Some definitions and shorthand notation:

- The mode m judgement asserts that the evaluation mode is m.
- The modified?(f) judgement returns a list of references captured and potentially modified by f.
- μ_L is shorthand for $\mu(l)[d]$, where $L = [(l, d)]$ is a location-dimension list.
- $e \mid \mu_{L'} \rightsquigarrow_{m,f} e' \mid \mu'_{L'}$ is shorthand for the sequence of judgements
  - modified?(f) = R
  - L' = updims(R, μ)
  - $e \mid \mu_{L'} \mapsto_m e' \mid \mu'_{L'}$

  where updims(R, μ) returns a location-dimension list where the dimension of each location $l \in R$ is incremented by one and the store μ is updated correspondingly.
- $\langle(\alpha_i, v_i)\rangle^A$ is shorthand for $\langle(\alpha_1, v_1), \dots, (\alpha_n, v_n)\rangle^A$.