Neko: A quantum map-filter-reduce programming language

Elton Pinto [1] Austin Adams [1]

[1]Georgia Institute of Technology

Introduction

Programming quantum computers is hard. One has to painstakingly write code that builds a circuit using low-level quantum gates. The gate-level abstraction, albeit universal, is non-intuitive and too primitive to be used for rapidly prototyping large-scale quantum applications (See Fig. 1).

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.

```
def what_am_i(qc: QuantumCircuit):
    for qubit in range(nqubits): qc.h(qubit)
    for qubit in range(nqubits): qc.x(qubit)
    qc.h(nqubits-1)
    qc.mct(list(range(nqubits-1)), nqubits-1)
    qc.h(nqubits-1)
    for qubit in range(nqubits): qc.x(qubit)
    for qubit in range(nqubits): qc.h(qubit)
```

Figure 1. Can you guess what this program does?

Observation

Several quantum algorithms 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?

$$|\psi\rangle \xrightarrow{\text{superpose}} \sum_{i=0}^{2^n-1} \alpha_i |i\rangle |0\rangle^{\otimes m}$$
$$\xrightarrow{\text{map}} \sum_{i=0}^{2^n-1} \alpha_i |i\rangle |f(i)\rangle$$
$$\xrightarrow{\text{filter/reduce}} \beta_s |\text{success}\rangle |G(f)\rangle + \beta_f |\text{failure}\rangle |\psi_f\rangle + \beta_e |\text{error}\rangle |\psi_e\rangle$$

Solution

Neko is a high-level quantum programming language that exposes a map-filter-reduce interface for exploiting quantum parallelism through the notion of *first-class superpositions*. 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).

```
let x1 = superpose [0,1,2,3,4,5,6,7] in
let x2 = map (λ x. x + 32) x1 in
let x3 = filter (λ x. x % 2 = 0) x2 in
let xs = x1 ⊗ x3 in
let result = reduce (λ (x1,_) (x2,_). x1 + x2) xs in
sample result
```

Figure 2. Example program showcasing Neko's map-filter-reduce interface

First-class Superpositions

A Neko expression can be put into a superposition using the `superpose` primitive if its corresponding type implements the superposition interface (Fig. 3). A superposition can be manipulated using `map`, `filter`, and `reduce`, and can be sampled using `sample`.

```
interface superposition {
  type t
  type chunk

  val num_chunks : t -> int
  val chunk : t -> i -> chunk
}
```

Figure 3. The superposition interface

Semantics of references

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. We can control what forms of entanglement materialize by defining where references can be captured or acquired.

Chunk Local Storage (CLS)
In this system, an expression executing in the context of a superposition is not allowed capture or acquire a reference from the outside.

$$\frac{k \in P \subseteq \{1,\ldots,n\} \qquad k' \in \{1,\ldots,n\} \setminus P \qquad f\ v_k \mid \phi \mapsto \texttt{true} \mid \mu' \qquad f\ v_{k'} \mid \phi \mapsto \texttt{false} \mid \mu''}{\texttt{filter}\ f\ \langle(\alpha_i, v_i)\rangle_n \mid \mu \mapsto \langle(\beta_i, v_i)\rangle_n \mid \mu \text{ such that } \exists\ \beta_j \text{ with } j \in P \text{ and } |1 - |\beta_j|^2| < \epsilon}\ \text{E-FilterSup}$$

$$\frac{G(f, (v_0, \ldots, v_{n-1})) = v \quad \text{where } G \text{ reduces } v_0, \ldots, v_{n-1} \text{ using } f \text{ in an arbitrary order}}{\texttt{reduce}\ f\ \langle(\alpha_i, v_i)\rangle_n \mid \mu \mapsto \langle(\beta_s, v), (\beta_{f_i}, v_{f_i})^k, (\beta_{e_i}, v_{e_i})^l\rangle \mid \mu \quad \text{where } |\beta_s|^2 > 0, k + l + 1 = n, \text{ and } v_{f_i}, v_{e_i} \text{are arbitrary}}\ \text{E-ReduceSup}$$

Figure 4. Operational semantics for `filter` and `reduce` on superpositions under CLS

$$\frac{\Gamma \mid \Sigma \vdash f : \tau_c \to \tau_c' \text{ with } \phi \qquad \Gamma \mid \Sigma \vdash xs : \langle(\tau_t, \tau_c)\rangle \text{ with } \phi}{\Gamma \mid \Sigma \vdash \texttt{map}\ f\ xs : \langle(\tau_t, \tau_c')\rangle \text{ with } \phi}\ \text{T-MapSup}$$

Figure 5. Typing rule for `map` on superpositions under CLS

CLS with immutable access
In this system, an expression executing in the context of a superposition can only capture an immutable handle to a reference.

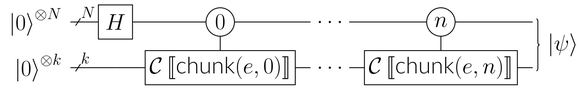
CLS with no restrictions
In this system, no restrictions are placed on where refs can be captured or acquired. Entangled superpositions can be materialized, as shown in Fig. 6.

```
let entangle a b =
  let s = ref 0 in
  let f = λ i x. s := (!s * 10) + i ; x in
  (mapi f a) ⊗ (mapi f b)
```

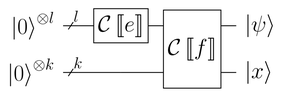
Figure 6. Entangling two superpositions under the CLS with no restrictions semantics

Compilation to Quantum Circuits

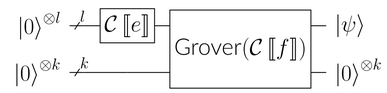
Neko is compiled to a simple quantum circuit language with support for qubit management through qubit registers, single-qubit and multi-qubit gates, qRAM, and measurement. The operational semantics and typing rules are largely similar to that of Hietala et al. (2021) augmented with qubit management and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



$\mathcal{C}[\![\texttt{superpose}\ e]\!]$, $N = \log_2(\text{num\_chunks}(e))$



$\mathcal{C}[\![\texttt{map}\ f\ e]\!]$ when $e : \langle(\tau_t, \tau_c)\rangle$



$\mathcal{C}[\![\texttt{filter}\ f\ e]\!]$ when $e : \langle(\tau_t, \tau_c)\rangle$

$\mathcal{C}[\![\texttt{reduce}\ f\ e]\!] \cong \mathcal{C}[\![\texttt{filteri perm}_{1\ldots n}\ (\texttt{map}\ (\lambda\vec{x}.\texttt{reduce}\ f\ \vec{x})\ e^{\otimes n})]\!]$ when $e : \langle(\tau_t, \tau_c)\rangle$

Figure 7. Select compilation rules

Realizing `reduce` is not straightforward because the chunks of a quantum superposition cannot share memory. One approach is shown in Fig. 7. While it works in the general case, it undoes the space-savings afforded by using superpositions.

Are there cases where we can do better? Jozsa (1991) addresses the issue by characterizing functions that are *computable by quantum parallelism (QPC)* using a linear relations formalism. What's left to unpack is if this formalism can efficiently be leveraged to materialize a better compilation scheme for `reduce`.

Limitations and Future Work

Current Status: A compiler is being worked on; not fully functional yet. Reference semantics in flux. Properties like type safety need to be proved.

Neko does not expose the full power of quantum computation. An interesting future direction would be to investigate richer primitives for manipulating the amplitudes of a chunk à la quantum signal processing (QSP) and quantum singular value transforms (QSVT). Future work could also look into better compilation schemes for `reduce`.

References

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.