Phantom Types for Quantum Programs

Robert Rand
University of Pennsylvania
rrand@seas.upenn.edu

Jennifer Paykin
University of Pennsylvania
jpaykin@seas.upenn.edu

Steve Zdancewic
University of Pennsylvania
stevez@cis.upenn.edu

Abstract

We explore the design space of using dependent types to type check and verify quantum circuits. We weigh the trade-offs between the expressivity of dependent types and the costs imposed by large proof terms. As a middle ground, we propose lightweight dependent types, or *phantom types*, which provide useful type information for programming while specifying the properties to be externally verified.

Keywords dependent types, verification, quantum computing

1 QWIRE

The QWIRE language [3, 4] is a quantum circuit generation language implemented in the Coq proof assistant.[1] QWIRE uses linear types to guarantee that its circuits are well-formed and therefore respect the no-cloning and no-deleting theorems of quantum mechanics. QWIRE also uses Coq's dependent types to specify families of circuits and verify those circuits' correctness. This is accomplished by translating QWIRE circuits into functions on a family of matrices called *density matrices*, which represent distributions over quantum states.

In principle QWIRE could use dependent types to specify a number of valuable properties: to indicate the dimensions of matrices, to structure the patterns of wires in our circuits and enforce linearity, and to guide the translation of circuits to their denotational semantics. However, dependently-typed programming in Coq is a slow and painful process due to the burden of carrying around proof terms. Coq is arguably better suited to extrinsically verifying properties of (simply-typed) programs than programming directly with dependent types.

As a compromise, we find that using lightweight dependent types, or phantom types [2], provides much of the desired structure of dependent types without the bulk. The lightweight types do not enforce all of our desired specifications, but provide just enough structure to guide the programmer. The types also characterize which additional properties need to be externally verified in Coq.

2 Matrices for Quantum Programs

Consider the type of complex-valued matrices represented as a function from indices to values.

```
Definition fun_matrix := ℕ * ℕ → ℂ.
```

Compared to a list or array-based representation, functions make it easy to represent two operations frequently used in quantum calculations: the conjugate transpose of a matrix ($A^\dagger$), and the Kronecker product of two matrices ($A \otimes B$).

$$A^\dagger(i,j) = (A(j,i))^*$$
$$(A \otimes B)(i,j) = A(i/m, j/n) \times B(i\%m, j\%n)$$

where c^* is the complex conjugate of c, $m = \dim_x(B)$, and $n = \dim_y(B)$. Matrix multiplication is also a common operation, where

$$(A \times B)(i,j) = \sum_{k=0}^{n-1} A(i,k)B(k,j)$$

for $n = \dim_y(A) = \dim_x(B)$.

As the Kronecker product and matrix multiplication depend on the dimensions of a matrix, it might seem reasonable to record a matrix's dimensions in its type, which is the approach taken by SSReflect's Mathematical Components library [1]:

```
Definition dep_matrix (m n : ℕ) := ℕ_m * ℕ_n → ℂ.
```

where `ℕ_n` is the type of natural numbers less than n. Now matrix multiplication and addition can only be applied to matrices of the correct dimension, and we can specify properties, like a matrix being square, at the typing level rather than as external judgments.

Unfortunately, working with bounded natural numbers (or, in the equivalent situation with lists, length-indexed vectors) requires significant overhead. For example, it is not trivial to show that $A \otimes B$ has the dimensions $\dim_x(A) \cdot \dim_x(B) \times \dim_y(A) \cdot \dim_y(B)$, and this proof must be constructed every time we compute the Kronecker product. Even looking up a specific cell in the constructed matrix requires proofs that the indices falls within the specified bounds.

A compromise is to use *phantom types*, where the dimensions of a matrix occur in its type but are not actually enforced:

```
Definition phantom_matrix (m n : ℕ) := ℕ * ℕ → ℂ.
```

Here, the m and n parameters specify the dimensions of the matrix, while the arguments to the function specify an index into the matrix. One way to think of `phantom_matrix` is as a `fun_matrix` with additional fields storing the dimensions of the matrix, where those fields are accessible in the types.

The dimensions in the type make multiplication and the Kronecker product easy to express, and act as a lightweight form of specification for the programmer. However, there is no proof burden internal to the matrices themselves. Instead, it is possible to show a matrix is well-formed within its specified bounds by means of an external predicate:

```
Definition wf_matrix {m n} (A : phantom_matrix m n) :=
  ∀ x y, x ≥ m ∨ y ≥ n → A(x,y) = 0.
```

Phantom types occupy a convenient middle ground in allowing information to be stored in the types, while pushing the majority of the work to external predicates. For example, in quantum verification we define predicates characterizing pure quantum states (square matrices satisfying $\rho\rho = \rho$ and $trace(\rho) = 1$) and mixed states (weighted sums of pure states), as well as unitary transformations (square matrices where $U^\dagger = U^{-1}$).

In the next section we look at the analogous case for QWIRE circuits, where phantom types provide structure while retaining a low intrinsic proof obligation.

[1] https://github.com/jpaykin/QWIRE

CoqPL'18, January 13, 2018, Los Angeles, CA, USA
.

3 Typing QWIRE Programs

A QWIRE circuit is a sequence of gate applications to patterns (nested tuples) of qubit- or bit-valued wires. A well-typed circuit uses these wires linearly, to prevent duplicating qubit-valued wires. Using dependent types we can define patterns paramaterized by a linear typing context and a wire type `WType`.

```
Inductive dep_pat : Ctx → WType → Set :=
| qubit : ∀ x, dep_pat (singleton x Qubit) Qubit
| bit   : ∀ x, dep_pat (singleton x Bit) Bit
| unit  : dep_pat ∅ One
| pair  : ∀ {Γ1 Γ2 Γ W1 W2}, Γ1 ∪ Γ2 = Some Γ →
          dep_pat Γ1 W1 → dep_pat Γ2 W2 → dep_pat Γ (W1⊗W2).
```

Here $\Gamma_1 \cup \Gamma_2$ is the partial disjoint merge of two typing contexts and $W_1 \otimes W_2$ is the linear pair of two wire types.

Using higher-order abstract syntax, we can define a type of well-formed circuits.

```
Inductive dep_circuit (Γ : Ctx) (W : WType) : Set :=
| output : dep_pat Γ W → dep_circuit Γ W
| gate   : ∀ {Γ1 Γ'} {m : Γ1 ∪ Γ' = Some Γ},
           dep_gate W1 W2 → dep_pat Γ1 W1 →
           (∀ {Γ2 Γ2'} {m2 : Γ2 ∪ Γ' = Some Γ2'},
                       dep_pat Γ2 W2 → dep_circuit Γ2' W)→
           dep_circuit Γ W.
```

Higher-order abstract syntax, which uses Coq variable binding for patterns, has the advantage of eliminating proofs of meta-theoretic properties. For example, the composition of two circuits can be defined by case analysis on the structure of the first circuit:

```
Program Fixpoint compose {Γ1 Γ1' Γ'} {W W'}
        {m1 : Γ1 ∪ Γ = Some Γ1'}
        (c : Circuit Γ1 W)
        (f : ∀ {Γ2 Γ2'} {m2 : Γ2 ∪ Γ = Some Γ2'},
             Pat Γ2 W → Circuit Γ2' W')
        : Circuit Γ1' W' :=
  match c with
  | output p   ⇒ f p
  | gate g p h ⇒ gate g p (fun q ⇒ compose (h q) f)
  end.
```

The remaining obligations, omitted as implicit arguments, can be solved using proof tactics. However, in order to use this function, or even gate application, the programmer must provide those implicit proofs each time. As a result, even simple applications of the `compose` function do not reduce efficiently to a normal form, due to the size of the proof terms. The size of such a proof depends not only on the size of the input circuit `c`, but also on the size of the input context Γ_1', which might increase in recursive calls if a gate creates new wires. Thus the complexity of the dependent `compose` is on the order of $O(|c||\Gamma_1'|)$, rather than $O(|c|)$ for a non-dependent version.

A non-dependent circuit omits the typing information:

```
Inductive untyp_circuit : Set :=
| output : untyp_pat → untyp_circuit
| gate   : untyp_gate → untyp_pat →
           (untyp_pat → untyp_circuit)→
           untyp_circuit.
```

However, this approach has other drawbacks. For one, we would like to know the input and output type of a circuit in order to compute its denotation. A group of n bits or qubits corresponds to a 2^n density matrix, so a circuit from m qubits to n qubits corresponds to a superoperator from an $m \times m$ matrix to an $n \times n$ matrix.

In addition, using untyped circuits to pattern match against the output of a gate application is problematic, since it must include a bogus error case. For example, the following circuit applies a `swap` gate and then swaps the output qubits, producing an identity:

```
Definition unswap := gate swap (qubit x, qubit y) (fun z ⇒
  match z with
  | pair (qubit y, qubit x) ⇒ output (qubit x, qubit y)
  | _ ⇒ output unit
  end).
```

With intrinsically typed circuits we can avoid this error case by using the type information of the pattern.

As we did for matrices, we can use lightweight dependent types to store some but not all type information in circuits and patterns. In particular, by retaining only the output type and not the input contexts, we can restore dependent pattern matching, eliminating the error case in `unswap`:

```
Inductive phantom_circuit (W : WType) : Set :=
| output : phantom_pat W → phantom_circuit'' W
| gate   : ∀ {W1 W2}, dep_gate W1 W2 → phantom_pat W1 →
           (phantom_pat W2 → phantom_circuit W) →
           phantom_circuit W.
```

The parameter `W` to `phantom_circuit` is used by the constructors and so is not technically a phantom type, but it serves the same purpose by carrying limited type information without a heavy proof burden.

We can then define an extrinsic judgment `Typed_Circuit Γ W c` which ensures that the circuit `c : phantom_circuit W` uses exactly the wires in Γ, linearly. We further define a *boxed circuit* as a function from a pattern of some type `w` to a circuit, thereby giving the circuit an input type as well.

```
Inductive Box (W1 W2 : WType) : Set :=
| box : (phantom_pat W1 → phantom_circuit W2) → Box W1 W2.
```

This enables us to translate our circuits into functions on appropriately sized (phantom) matrices.

4 Discussion

Using phantom types for both matrices and circuits leads to a happy medium by tracking information in the size of our data and leaving the bulk of the proof obligations to extrinsic verification. Our Coq development provides a denotational semantics using density matrices that has already allowed us to verify a number of small quantum protocols, including Deutsch's algorithm, and phantom types provide the right level of lightweight expressivity to do so. In future work we will formally prove that every well-typed circuit corresponds to a function between valid quantum states.

References

[1] Assia Mahboubi and Enrico Tassi. 2016. *Mathematical Components.* Electronic resource, available from https://math-comp.github.io/mcb/book.pdf.

[2] Daan Leijen and Erik Meijer. 1999. Domain Specific Embedded Compilers. *SIGPLAN Not.* 35, 1 (Dec. 1999), 109–122. https://doi.org/10.1145/331963.331977

[3] Jennifer Paykin, Robert Rand, and Steve Zdancewic. 2017. QWIRE: A Core Language for Quantum Circuits. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* ACM, New York, NY, USA, 846–858. https://doi.org/10.1145/3009837.3009894

[4] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2017. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Proceedings of the 14th International Conference on Quantum Physics and Logic, QPL 2017.*