A tutorial introduction to quantum circuit programming in dependently typed Proto-Quipper

Peng Fu[1], Kohei Kishida[2], Neil J. Ross[1], and Peter Selinger[1]

[1] Dalhousie University, Halifax, NS, Canada
{frank-fu,neil.jr.ross,peter.selinger}@dal.ca
[2] University of Illinois, Urbana-Champaign, IL, U.S.A.
kkishida@illinois.edu

Abstract. We introduce dependently typed Proto-Quipper, or Proto-Quipper-D for short, an experimental quantum circuit programming language with linear dependent types. We give several examples to illustrate how linear dependent types can help in the construction of correct quantum circuits. Specifically, we show how dependent types enable programming families of circuits, and how dependent types solve the problem of type-safe uncomputation of garbage qubits. We also discuss other language features along the way.

Keywords: Quantum programming languages · Linear dependent types · Proto-Quipper-D

1 Introduction

Quantum computers can in principle outperform conventional computers at certain crucial tasks that underlie modern computing infrastructures. Experimental quantum computing is in its early stages and existing devices are not yet suitable for practical computing. However, several groups of researchers, in both academia and industry, are now building quantum computers (see, e.g., [2,11,16]). Quantum computing also raises many challenging questions for the programming language community [17]: How should we design programming languages for quantum computation? How should we compile and optimize quantum programs? How should we test and verify quantum programs? How should we understand the semantics of quantum programming languages?

In this paper, we focus on quantum circuit programming using the linear dependently typed functional language Proto-Quipper-D.

The no-cloning property of quantum mechanics states that one cannot in general copy the state of a qubit. Many existing quantum programming languages, such as Quipper[9,10], QISKit [21], Q# [26], Cirq [5], or ProjectQ [25], do not enforce this property. As a result, programmers have to ensure that references to qubits within a program are not duplicated or discarded. Linear types have been used for resource aware programming [7,27] and it is now well-known

that they can be used to enforce no-cloning [24]. A variety of programming languages use linear types for quantum circuit programming, e.g., Proto-Quipper-S [23], Proto-Quipper-M [22], and QWire [19]. All well-typed programs in these languages satisfy the no-cloning property.

Dependent types [14] have been one of the main focuses in programming language and type system research in the past decades. Dependent types make it possible to express program invariants and constraints using types [1,3,6]. In the context of quantum circuit programming, dependent types are useful for expressing parameterized families of circuits. For example, one can define a function that inputs a size and outputs a circuit of the corresponding size. Because the type of the output circuit is indexed by the size argument, errors due to an attempt to compose mismatched circuits are detected at compile time. Another important application of dependent types is the type-safe management of garbage qubits, which we discuss in Section 4.

We introduce an experimental quantum circuit programming language called dependently typed Proto-Quipper, or Proto-Quipper-D for short. Following Quipper, Proto-Quipper-D is a functional language with quantum data types and aims to provide high-level abstractions for constructing quantum circuits. Like its predecessors Proto-Quipper-S and Proto-Quipper-M, the Proto-Quipper-D language relies on linear types to enforce no-cloning. Proto-Quipper-D additionally features the use of linear dependent types to facilitate the type-safe construction of circuit families. This paper provides a practical introduction to programming in Proto-Quipper-D.

The paper is structured around several programming examples that showcase the use of linear dependent types in Proto-Quipper-D.

- We give an introduction to dependent types by showing how to use them to prove basic properties of addition in Section 2.
- We show how to program with families of quantum circuits in Section 3.
- We give a new application of existential dependent types and show how it simplifies the construction of certain reversible quantum circuits in Section 4.

An implementation of Proto-Quipper-D is available at: `https://gitlab.com/frank-peng-fu/dpq-remake`.

2 An introduction to dependent types

Proto-Quipper-D supports programming by recursion and pattern matching. For example, the following is a program that defines the addition of Peano numbers.

```
data Nat = Z | S Nat

add : !(Nat -> Nat -> Nat)
add n m =
  case n of
    Z -> m
    S n' -> S (add n' m)
```

In the above program, we use the keyword `data` to define an algebraic data type in the style of Haskell 98 [20]. The type checker will analyze the data type declaration and determine that `Nat` is a *parameter type* (or *non-linear type*). In Proto-Quipper-D, parameter types are types that can be freely duplicated and discarded. The addition function has type `!(Nat -> Nat -> Nat)`. The exclamation mark (pronounced "bang") in front of a function type makes that type a parameter type. This means that addition is a reusable function, i.e., it can be used multiple times. The type of a non-reusable function would be of the form `a -> b` and in particular would not be prefixed by a `!`. In contrast to a reusable function, a non-reusable function must be used exactly once. This guarantees that any quantum data embedded in the function does not get inadvertently duplicated or discarded. Proto-Quipper-D requires all top-level declarations to have parameter types, making them reusable.

With dependent types, we can even encode properties of programs in types. In Proto-Quipper-D, dependent function types are of the form `(x : A) -> B`, where the type `B` may optionally mention the variable `x`. We can think of this dependent function type as the universal quantification $\forall x : A \,.\, B$ of predicate logic. Dependent types therefore allow us to represent properties of programs as types. For example, the following programs correspond to proofs of basic properties of addition.

```
addS : ! (p : Nat -> Type) -> (n m : Nat) ->
             p (add n (S m)) -> p (add (S n) m)
addS p n m h =
  case n of
    Z -> h
    S n' -> addS (λ y -> p (S y)) n' m h

addZ : ! (p : Nat -> Type) -> (n : Nat) -> p (add n Z) -> p n
addZ p n h = case n of
               Z -> h
               S n' -> addZ (λ y -> p (S y)) n' h
```

The type of `addS` expresses the theorem that for all natural numbers n and m, we have $n + Sm = Sn + m$. However, rather than using an equality symbol, we use the so-called *Leibniz equality*. Leibniz defined two things to be equal if they have exactly the same properties. Therefore, the type of `addS` states that for any property `p : Nat -> Type` of natural numbers, and for all natural numbers `n`, `m`, if `add n (S m)` has the property `p`, then `add (S n) m` has the property `p`. Similarly, the type of `addZ` expresses the fact that $n + Z = n$.

Note how the types of dependent type theory play a dual role: on the one hand, they can be read as types specifying the inputs and outputs of functional programs; on the other hand, they can be read as logical statements. This is the so-called *propositions-as-types* paradigm [8]. For example, the last arrow "`->`" in the type of `addS` can be interpreted both as a function type and as the logical implication symbol. This works because a proof of an implication is actually

a function that transforms evidence for the hypothesis into evidence for the conclusion.

Indeed, not only does the type of the function `addS` corresponds to a theorem, but the actual code of `addS` corresponds to its proof. For example, in the branch when `n` is `Z`, the variable `h` has type `p (add Z (S m))`, which equals `p (S m)` by the definition of `add`. This branch is expecting an expression of type `p (add (S Z) m)`, which equals `p (S m)` by definition of `add`, so the type-checking of `h` succeeds.

In practice, we can sometimes use the above equality proofs to convert one type to another. We will give examples of this in Section 3.2. However, we emphasize that Proto-Quipper-D is designed for quantum circuit programming, not general theorem proving like languages such as Coq and Agda. The only kind of primitive propositions we can have are equalities, and the support of dependent data types is limited to *simple types*, as discussed in Section 3.1.

3 Programming quantum circuits

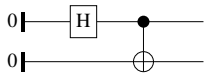
We use the keyword `object` to introduce simple linear objects such as bits and qubits, representing primitive wires in circuits. We use the keyword `gate` to introduce a primitive gate. As far as Proto-Quipper-D is concerned, gates are uninterpreted; they simply represent basic boxes that can be combined into circuits. Each primitive gate has a type specifying its inputs and outputs.

```
object Qubit
object Bit

gate H : Qubit -> Qubit
gate CNot : Qubit -> Qubit -> Qubit * Qubit
gate Meas : Qubit -> Bit
gate Discard : Bit -> Unit
gate Init0 : Unit -> Qubit
gate C_X : Qubit -> Bit -> Qubit * Bit
gate C_Z : Qubit -> Bit -> Qubit * Bit
```

The above code declares primitive types `Qubit` and `Bit` and a number of gates. For example, the gate `H` is a reusable linear function of type `!(Qubit -> Qubit)`, which, by convention, represents the Hadamard gate. Note that the type checker automatically adds the `!` to gate declarations, so it is not necessary to do so manually. The type expression `Qubit * Qubit` denotes the tensor product of two qubits, and thus, the controlled-not gate `CNot` has two inputs and two outputs (where, by convention, the first input is the target and the second is the control). By linearity, the arguments of the `CNot` can only be used once. Thus, an expression such as `CNot x x` will be rejected by the type checker because the argument `x` is used twice. The gate `Meas` corresponds to a measurement, turning a qubit into a classical bit. The type `Unit` represents the unit of the tensor product, i.e., a bundle of zero wires. Thus, the gate `Discard` can be used

to discard a classical bit, and the gate `Init0` can be used to initialize a qubit (by convention, in state $|0\rangle$). We also introduce two classically-controlled gates `C_X` and `C_Z`.

The following program produces a circuit that generates a Bell state:



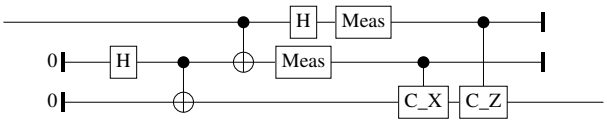

```
bell00 : !(Unit -> Qubit * Qubit)
bell00 u =
  let x = Init0 ()
      y = Init0 ()
      x' = H x
      (y, x') = CNot y x'
  in (y, x')
```

The initialization gate `Init0` inputs a unit, denoted by `()`, and outputs a qubit. If we want to display the circuit generated by the function `bell00`, we can use Proto-Quipper's `box` function:

```
bell00Box : Circ(Unit, Qubit * Qubit)
bell00Box = box Unit bell00
```

The `box` function inputs a circuit-generating function such as `bell00` and produces a completed circuit of type `Circ(Unit, Qubit * Qubit)`. In the Proto-Quipper-D interactive shell, we can then type `:d bell00Box` to display the circuit.

The following program implements quantum teleportation.




```
bellMeas : !(Qubit -> Qubit -> Bit * Bit)
bellMeas x y =
  let (x', y') = CNot x y
      y'' = H y'
  in (Meas x', Meas y'')

tele : !(Qubit -> Qubit)
tele phi =
  let (bob, alice) = bell00 ()
      (a', phi') = bellMeas alice phi
```

```
      (bob', a'') = C_X bob a'
      (r, phi'') = C_Z bob' phi'
      u = Discard phi''
      u = Discard a''
  in r
```

3.1 Simple types

Following Quipper, Proto-Quipper-D makes a distinction between *parameters* and *states*. Parameters are values that are known at circuit generation time, while states are only known at circuit execution time. For example, the type `Nat` represents a parameter, while the type `Qubit` represents a state.

In Proto-Quipper-D, we use the concept of *simple types* to describe states. As discussed earlier, simple types can be introduced using the keyword `object`. In practice, it is more common to create simple types by composing existing ones. For example, `Qubit * Qubit` is also a simple type. For this reason, we call the tensor product a *simple type constructor*. In Proto-Quipper-D, the programmer can also define families of new simple types using the `simple` keyword. For example, the following defines a type family `Vec`, and `Vec Qubit n` is a simple type.

```
simple Vec a : Nat -> Type where
    Vec a Z = VNil
    Vec a (S n) = VCons a (Vec a n)
```

The expression `Nat -> Type` is a *kind expression*. It means that `Vec a n` is a type whenever `n` is a natural number. The two clauses after the `simple` keyword are the definition of the type `Vec a n`. The first clause says that an element of the type `Vec a Z` can be constructed by the constructor `VNil`. The second clause says that an element of the type `Vec a (S n)` can be constructed by applying the constructor `VCons` to a term of type `a` and a term of type `Vec a n`. Therefore, `Vec a n` represents a vector of `n` elements of type `a`.

The type `Vec a n` is an example of *dependent data type*, where the data type `Vec a n` depends on some term `n` of type `Nat`. In the interpreter, we can query the types of `VNil` and `VCons` (by typing `:t VNil`). They have the following types.

```
VNil : forall (a : Type) -> Vec a Z
VCons : forall (a : Type) -> forall (n : Nat) ->
              a -> Vec a n -> Vec a (S n)
```

In Proto-Quipper-D, all data constructors are reusable, so there is no need for them to have an explicit bang-type. The leading `forall` keyword means that programmers do not need to supply that argument when calling the function. We call such quantification *irrelevant quantification*. For example, when using `VCons`, we only need to give it two arguments, one of type `a` and one of type `Vec a n`.

The simple data type declaration is currently the only way to introduce dependent data types in Proto-Quipper-D. Semantically, simple types corresponds to states. Syntactically, a simple type can uniquely determine the size and the constructors of its data. The type checker will check whether a simple data type declaration is well-defined. Note that not all dependent data types are simple types. For example, the following declaration will not pass the type checker.

```
simple ColorVec a : Nat -> Type where
  ColorVec a Z = CNil
  ColorVec a (S n) = VConsBlue a (ColorVec a n)
  ColorVec a (S n) = VConsRed a (ColorVec a n)
```

The `ColorVec` data type is ambiguous when the parameter is `S n`, as the constructor in this case can be either `VConsBlue` or `VConsRed`.

In general, checking whether a simple type is well-defined is equivalent to deciding whether a general recursive function is well-defined and terminating, which is undecidable. Currently, Proto-Quipper-D checks whether a simple data type declaration is well-defined using the same criterion as checking primitive recursion [13].

3.2 Using Leibniz equality

Suppose we want to define a function that reverses the order of the components in a vector. One way to do this is to use an accumulator: we traverse the vector while prepending each element to the accumulator. This can be expressed by the `reverse_aux` function defined below.

```
reverse_aux : ! (a : Type) -> (n m : Nat) ->
                      Vec a n -> Vec a m -> Vec a (add n m)
reverse_aux a n m v1 v2 =
  case n of
    Z -> let VNil = v1 in v2
    S n' ->
      let VCons q qs = v1 in
      let ih = reverse_aux a n' (S m) qs (VCons q v2) in
      addS (Vec a) n' m ih
```

Note that the type of `reverse_aux` indicates that the length of the output vector is the sum of the lengths of the input vectors. In the definition for `reverse_aux`, we use `v1` and `v2` exactly once in each branch, which respects linearity. In the second branch of `reverse_aux`, the type checker expects an expression of type `Vec a (add (S n') m)`, but the expression `ih`, obtained from the recursive call, has type `Vec a (add n' (S m))`. We therefore use the theorem `addS` from Section 2 to convert the type to `Vec a (add (S n') m)`. We can then use `reverse_aux` to define the `reverse_vec` function, which requires a similar type conversion.

```
reverse_vec : ! (a : Type) -> (n : Nat) -> Vec a n -> Vec a n
reverse_vec a n v = addZ (Vec a) n (reverse_aux a n Z v VNil)
```

3.3 Families of quantum circuits

We can use simple data types such as vectors to define functions that correspond to families of circuits. As an example, we consider the well-known quantum Fourier transform [18]. The quantum Fourier transform is the map defined by

$$|a_1, \ldots, a_n\rangle \mapsto \frac{(|0\rangle + e^{2\pi i 0.a_1 a_2 \ldots a_n}|1\rangle) \ldots (|0\rangle + e^{2\pi i 0.a_{n-1} a_n}|1\rangle)(|0\rangle + e^{2\pi i 0.a_n}|1\rangle)}{2^{n/2}}.$$

where $0.a_1 ... a_n$ is the binary fraction $a_1/2 + a_2/4 + ... + a_n/2^n$. Circuits for the quantum Fourier transform can be constructed using the Hadamard gate H and the controlled rotation gates $R(k)$ defined by

$$R(k) = \begin{pmatrix} 1\,0\,0 & 0 \\ 0\,1\,0 & 0 \\ 0\,0\,1 & 0 \\ 0\,0\,0 & e^{2\pi i/2^k} \end{pmatrix}.$$

The family of gates $R(k)$ can be declared in Proto-Quipper-D as follows:

```
gate R Nat : Qubit -> Qubit -> Qubit * Qubit
```

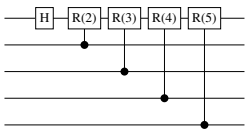
Applying the Hadamard gate to the first qubit produces the following state

$$H_1|a_1, \ldots, a_n\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i 0.a_1}|1\rangle) \otimes |a_2, \ldots, a_n\rangle,$$

where the subscript on the gate indicates the qubit on which the gate acts. We then apply a sequence of controlled rotations using the the first qubit as the target. This yields

$$R(n)_{1,n} \ldots R(2)_{1,2} H_1|a_1, \ldots, a_n\rangle = \frac{1}{2^{1/2}}(|0\rangle + e^{2\pi i 0.a_1 a_2 \ldots a_n}|1\rangle) \otimes |a_2, \ldots, a_n\rangle,$$

where the subscripts i and j in $R(k)_{i,j}$ indicate the target and control qubit, respectively. When $n = 5$, the above sequence of gates corresponds to the following circuit.



To construct such a circuit in Proto-Quipper-D, we first define the `rotate` function, which will produce a cascade of rotations with a single target. The rotations in the above circuit are then generated by `oneRotation 4`.

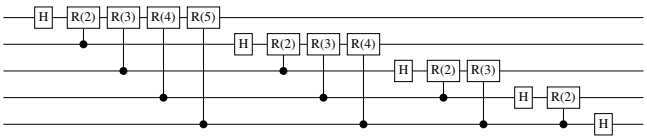
```
rotate : ! forall (y : Nat) -> Nat ->
                Qubit -> Vec Qubit y -> Qubit * Vec Qubit y
rotate k q v =
```

```
    case v of
      VNil -> (q, VNil)
      VCons x xs ->
        let (q', x') = R k q x
            (q'', xs') = rotate (S k) q' xs
        in (q'', VCons x' xs')

oneRotation : ! (n : Nat) ->
                  Circ(Qubit * Vec Qubit n, Qubit * Vec Qubit n)
oneRotation n =
  box (Qubit * Vec Qubit n)
    (λ x -> let (q, v) = x in rotate 2 (H q) v)
```

The `rotate` function uses the input vector `v` for controls and recursively applies the rotation gate `R` to the target qubit `q`, updating the rotation angle at each step. To program the full quantum Fourier transform, we apply the Hadamard and controlled rotations recursively to the rest of input qubits.

```
qft : ! forall (n : Nat) -> Vec Qubit n -> Vec Qubit n
qft v =
  case v of
    VNil -> VNil
    VCons q qs ->
      let q' = H q
          (q'', qs') = rotate 2 q' qs
          qs'' = qft qs'
      in VCons q'' qs''

qftBox : ! (n : Nat) -> Circ(Vec Qubit n, Vec Qubit n)
qftBox n = box (Vec Qubit n) qft
```

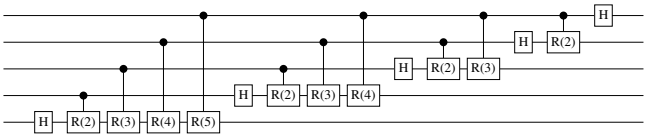
For example, `qftBox 5` generates the following circuit.



The input qubits of the circuit above use a big-endian ordering. We can convert to little-endian ordering by reversing the input vector.

```
qftBoxLittle : ! (n : Nat) -> Circ(Vec Qubit n, Vec Qubit n)
qftBoxLittle n = box (Vec Qubit n) (λ v -> qft (reverse_vec Qubit n v))
```

Then `qftBoxLittle 5` generates the following circuit.



3.4 Type classes for simple types and parameter types

Proto-Quipper-D is equipped with a type class mechanism that allows the user to define type classes and instances [28]. In addition, Proto-Quipper-D has two built-in type classes called `Simple` and `Parameter`, which are useful for programming with simple types and parameter types, respectively. The user cannot directly define instances for these two classes. Instead, instances for `Simple` and `Parameter` are automatically generated from data type declarations.

When a simple data type is defined, the type checker automatically makes the type an instance of the `Simple` class and, if appropriate, of the `Parameter` class. Similarly, when algebraic data types such as `List` and `Nat` are defined, the type checker makes instances of the `Parameter` class when possible. For example, consider the following programs.

```
data List a = Nil | Cons a (List a)

kill : ! forall a -> (Parameter a) => a -> Unit
kill x = ()

test1 : !(List Nat -> Unit)
test1 x = kill x

test2 : !(List Qubit -> Unit)
test2 x = kill x
```

The argument of the function `kill` must be a parameter. The expression `test1` is well-typed, because `List Nat` is a member of the `Parameter` class. But `test2` fails to type-check because `List Qubit` is not a member of the `Parameter` class.

Simple types are useful for describing the types of certain operations that require a circuit, rather than a family of circuits. Examples are boxing, unboxing, and reversing a circuit:

```
box : (a : Type) -> forall (b : Type) ->
          (Simple a, Simple b) => !(a -> b) -> Circ(a, b)

unbox : forall (a b : Type) ->
           (Simple a, Simple b) => Circ(a, b) -> !(a -> b)

reverse : forall (a b : Type) ->
            (Simple a, Simple b) => Circ(a, b) -> Circ(b, a)
```

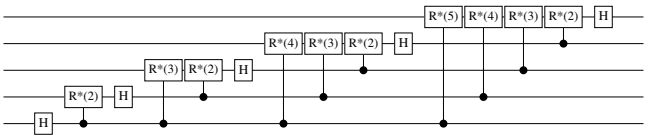
The type of `box` implies that only functions of simple type can be turned into boxed circuits. The following program will not type-check because `List Qubit` is not a simple type.

```
boxId : Circ(List Qubit, List Qubit)
boxId = box (List Qubit) (λ x -> x)
```

With the built-in function `reverse`, we can now compute the inverse of `qftBox`.

```
boxQftRev : ! (n : Nat) -> Circ(Vec Qubit n, Vec Qubit n)
boxQftRev n = reverse (qftBox n)
```

By definition, the family of circuits represented by `boxQftRev` is obtained by taking the inverse of every member of the family of circuits represented `qftBox`. For example, `boxQftRev 5` generates the following circuit.



4 Type-safe management of garbage qubits

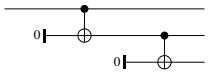
In quantum computing, it is often necessary to provide classical oracles to a quantum algorithm. These oracles are reversible implementations of classical boolean functions. Consider the example of the single bit full adder. If the inputs are `a`, `b` and `carryIn`, then the boolean expression `xor (xor a b) carryIn` calculates the sum of `a, b` and `carryIn` while the boolean expression `(a && b) || (a && carryIn) || (b && carryIn)` calculates the output carry.

We can implement the single bit adder as a reversible quantum circuit. Suppose that the boolean operations `xor`, `||`, and `&&` are given as reversible circuits of type `!(Qubit -> Qubit -> Qubit * Qubit)`. Here, the first qubit in the output of each function is the result of the operation, whereas the second qubit is a "garbage" qubit that cannot be discarded since this would violate linearity. As a result, the following naive implementation of the adder generates 7 garbage qubits and has a 9-tuple of qubits as its return type.

```
adder : ! (Qubit -> Qubit -> Qubit ->
              Qubit * Qubit * Qubit * Qubit * Qubit *
                  Qubit * Qubit * Qubit * Qubit)
adder a b carryIn =
  let (a1, a2, a3) = copy3 a
      (b1, b2, b3) = copy3 b
      (carryIn1, carryIn2, carryIn3) = copy3 carryIn
      (g1, r) = xor a1 b1
```

```
      (g2, s) = xor carryIn1 r
      (g3, c1) = a2 && b2
      (g4, c2) = a3 && carryIn2
      (g5, c3) = b3 && carryIn3
      (g6, c4) = c1 || c2
      (g7, carryOut) = c4 || c3
  in (s, carryOut, g1, g2, g3, g4, g5, g6, g7)
```

Due to linearity, the copying of a classical qubit must be explicit. In the code above, `copy3` is a function that produces three copies of a qubit that is in a classical state, i.e., `copy3` corresponds to the following circuit.



The above implementation of the adder is hard to read and awkward to compose with other circuits, because its type keeps track of all the garbage qubits produced throughout the computation. In Proto-Quipper-D, we solve this problem using monads [12], existential dependent types, and existential circuit boxing.

Instead of using the type `!(Qubit -> Qubit -> Qubit * Qubit)`, we give `xor`, `||`, and `&&` the type `!(Qubit -> Qubit -> WithGarbage Qubit)`, where `WithGarbage` is a monad that will take care of the garbage qubits. The idiomatic implementation of the full adder in Proto-Quipper-D is the following.

```
adder : !(Qubit -> Qubit -> Qubit -> WithGarbage (Qubit * Qubit))
adder a b carryIn = do
  let (a1, a2, a3) = copy3 a
      (b1, b2, b3) = copy3 b
      (carryIn1, carryIn2, carryIn3) = copy3 carryIn
  s <- [| xor (xor a1 b1) (pure carryIn1)|]
  carryOut <- [|[|(a2 && b2) || (a3 && carryIn2)|] || (b3 && carryIn3)|]
  return (s, carryOut)
```

Proto-Quipper-D implements idiom brackets [15] of the form `[| f a b c |]`. This expression will be translated to `join (ap (ap (ap (pure f) a) b) c)`, where `ap`, `pure` and `join` have the following types.

```
ap : ! forall (a b : Type) -> forall (m : Type -> Type) ->
           (Monad m) => m (a -> b) -> m a -> m b

pure : ! forall (m : Type -> Type) ->
           (Monad m) => forall (a : Type) -> a -> m a

join : ! forall (a : Type) -> forall (m : Type -> Type) ->
             (Monad m) => m (m a) -> m a
```

We now briefly discuss the definition of the `WithGarbage` monad.

```
data WithGarbage a = WG ((n : Nat) * Vec Qubit n) a

instance Monad WithGarbage where
  return x = WG (Z, VNil) x
  bind wg f = let WG ng r = wg
                  (n, g) = ng
                  WG mg' r' = f r
                  (m, g') = mg'
              in WG (add n m, append g g') r'
```

The type `(x : A) * B` is an *existential dependent type*, corresponding to the existential quantification $\exists x : A \,.\, B$ of predicate logic. Just as for dependent function types, the type B may optionally mention the variable x. The elements of the type `(n : Nat) * Vec Qubit n` are pairs `(n, v)`, where `n : Nat` and `v : Vec Qubit n`. Thus, `WithGarbage a` contains a vector of qubits of a unknown length and a value of type `a`. In the definition of the `WithGarbage` monad, the `return` function does not generate any garbage qubits. The `bind` function combines the garbage qubits from the two computations `wg` and `f`. Note that it uses the `append` function to concatenate two vectors.

The standard way to dispose of a qubit (and turn it into garbage) is via the following `dispose` method.

```
class Disposable a where
  dispose : a -> WithGarbage Unit

instance Disposable Qubit where
  dispose q = WG (1, VCons q VNil) ()
```

So for example, we can implement `xor` as follows. Note that the implemented circuit is not optimal, but it serves to illustrate the point.

```
xor : !(Qubit -> Qubit -> WithGarbage Qubit)
xor x y =
  do let z = Init0 ()
         (z', x') = CNot z x
         (z'', y') = CNot z' y
     dispose x'
     dispose y'
     return z''
```

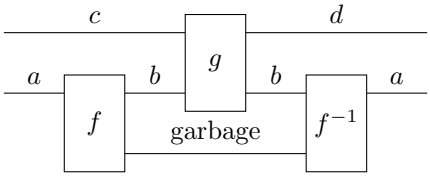
Using the `WithGarbage` monad, we can essentially program as if the extra garbage qubits do not exist. Next, we need a type-safe way to uncompute the garbage qubits. We achieve this with the function `with_computed` below, which takes a garbage-producing function and turns it into a function that produces no garbage. The implementation of `with_computed` relies on the following built-in function:

```
existsBox : (a : Type) -> forall (b : Type) ->
               (Simple a, Parameter b) => (p : b -> Type) ->
               !(a -> (n : b) * p n) ->
               (n : b) * ((Simple (p n)) => Circ(a, p n))
```

Intuitively, the `existsBox` construct is used to box an existential function. It takes a circuit generating function of type `!(a -> (n : b) * p n)` as input and turns it into an *existential circuit* of the type `(n : b) * Circ(a, p n)`. Using `existsBox`, we can define `with_computed`:

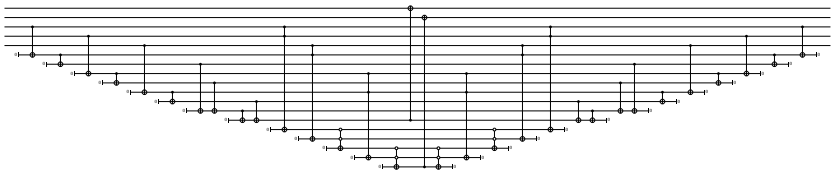
```
with_computed : ! forall d ->  (a b c : Type) ->
                    (Simple a, Simple b) =>
                    !(a -> WithGarbage b) ->
                    !(c * b -> d * b) -> (c * a -> d * a)
with_computed a b c f g input =
  let (y, x) = input
      (_, circ) = existsBox a (λx -> Vec Qubit x * b) (λz -> unGarbage (f z))
      h' = unbox circ
      (v, r) = h' x
      circ_rev = unbox (reverse circ)
      (d, r') = g (y, r)
      res = circ_rev (v, r')
  in (d, res)
```

The `with_computed` function inputs a function `f : a -> WithGarbage b` and a second function `g : c * b -> d * b`, and produces a garbage-free circuit `c * a -> d * a` corresponding to the following diagram. Of course each wire may correspond to multiple qubits, as specified in its type.



Note that this construction is type-safe, because it guarantees that there will be no uncollected garbage, regardless of how much garbage the function f actually produces. However, Proto-Quipper-D does not guarantee the *semantic* correctness of the resulting circuit; it could happen that a qubit that is supposed to be returned in state $|0\rangle$ is returned in some other state. Since semantic correctness is in general undecidable, Proto-Quipper-D makes no attempt to prove it. Consequently, a failure of semantic correctness is considered to be a programming error, rather than a type error. However, the *syntactic* correctness of the generated circuits is guaranteed by the type system.

Using the `with_computed` function and a few helper functions, we can obtain the following reversible version of `adder`.



5 Case studies

Beyond the simple examples that were considered in this tutorial, we have conducted two nontrivial programming case studies using Proto-Quipper-D. The first one is an implementation of the binary welded tree algorithm [4], which features the use of the dependent vector data type. The second is a boolean oracle for determining the winner of a completed game of Hex, which features the use the of `WithGarbage` and `State` monads. Both implementations are distributed with Proto-Quipper-D, in `test/BWT.dpq` and `test/Hex3.dpq`, respectively. The largest oracle contains 457,383 gates. For this oracle, type checking is nearly instantaneous (it takes less than 1 second), and circuit generation takes about 2.5 minutes on a 3.5 GHz CPU (4 cores), 16 GB memory desktop machine.

6 Conclusion

In this tutorial, we introduced the quantum programming language Proto-Quipper-D through a series of examples. Proto-Quipper-D is an experimental language and is currently under active development. Due to space constraints, we did not discuss all of the features of Proto-Quipper-D. Our goal was to highlight the use of linear and dependent types in quantum circuit programming. All the programs in the tutorial are available in `test/Tutorial.dpq` of the Proto-Quipper-D distribution.

Acknowledgements

This work was supported by the Air Force Office of Scientific Research under award number FA9550-15-1-0331. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the U.S. Department of Defense.

References

1. Agda documentation. `https://agda.readthedocs.io/en/v2.6.0.1/`, accessed: 2020-02-01
2. Arute, F. et al. (84 authors): Quantum supremacy using a programmable superconducting processor. Nature **574**, 505–510 (2019)

3. Bove, A., Dybjer, P.: Dependent types at work. In: International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development. pp. 57–99. Springer (2008)
4. Childs, A.M., Cleve, R., Deotto, E., Farhi, E., Gutmann, S., Spielman, D.A.: Exponential algorithmic speedup by a quantum walk. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing. pp. 59–68 (2003)
5. Circ. `https://cirq.readthedocs.io/en/stable/`, accessed: 2020-02-01
6. Coq documentation. `https://coq.inria.fr/documentation`, accessed: 2020-02-01
7. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987)
8. Girard, J.Y., Lafont, Y., Taylor, P.: Proofs and Types. Cambridge University Press (1989)
9. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: An introduction to quantum programming in Quipper. In: Proceedings of the 5th International Conference on Reversible Computation. pp. 110–124. Springer (2013)
10. Green, A.S., Lumsdaine, P.L., Ross, N.J., Selinger, P., Valiron, B.: Quipper: A scalable quantum programming language. In: Proceedings of the 34th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation. vol. 48, pp. 333–342. ACM (2013)
11. IBM Quantum Experience. `https://quantum-computing.ibm.com`, accessed: 2020-02-01
12. Jones, M.P.: Functional programming with overloading and higher-order polymorphism. In: International School on Advanced Functional Programming. pp. 97–136. Springer (1995)
13. Kleene, S.C.: Introduction to Metamathematics. Van Nostrand (1968)
14. Martin-Löf, P., Sambin, G.: Intuitionistic Type Theory. Bibliopolis Naples (1984)
15. McBride, C., Paterson, R.: Applicative programming with effects. Journal of Functional Programming **18**(1), 1–13 (2008)
16. Monroe, C., Campbell, W.C., Duan, L.M., Gong, Z.X., Gorshkov, A.V., Hess, P., Islam, R., Kim, K., Pagano, G., Richerme, P., Senko, C., Yao, N.Y.: Programmable quantum simulations of spin systems with trapped ions (2019), `arXiv:1912.07845`
17. Mosca, M., Roetteler, M., Selinger, P.: Quantum Programming Languages (Dagstuhl Seminar 18381). Tech. rep., Schloss Dagstuhl, Leibniz-Zentrum für Informatik (2019)
18. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge University Press (2002)
19. Paykin, J., Rand, R., Zdancewic, S.: Qwire: A core language for quantum circuits. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. ACM SIGPLAN Notices, vol. 52, pp. 846–858. ACM (2017)
20. Peyton Jones, S.: Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
21. Qiskit. `https://qiskit.org/`, accessed: 2020-02-01
22. Rios, F., Selinger, P.: A categorical model for a quantum circuit description language. Extended abstract. In: Proceedings of the 14th International Workshop on Quantum Physics and Logic, QPL 2017, Nijmegen. Electronic Proceedings in Theoretical Computer Science, vol. 266, pp. 164–178 (2018)
23. Ross, N.J.: Algebraic and Logical Methods in Quantum Computation. Ph.D. thesis, Dalhousie University, Department of Mathematics and Statistics (2015), available from `arXiv:1510.02198`
24. Selinger, P., Valiron, B.: A lambda calculus for quantum computation with classical control. Mathematical Structures in Computer Science **16**(3), 527–552 (2006)

25. Steiger, D.S., Häner, T., Troyer, M.: Projectq: An open source software framework for quantum computing (2016), arXiv:1612.08091
26. Svore, K., Geller, A., Troyer, M., Azariah, J., Granade, C., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., Roetteler, M.: Q#: Enabling scalable quantum computing and development with a high-level DSL. In: Proceedings of the Real World Domain Specific Languages Workshop, RWDSL 2018. Association for Computing Machinery (2018)
27. Wadler, P.: Linear types can change the world! In: Broy, M., Jones, C. (eds.) TC 2 Working Conference on Programming Concepts and Methods. pp. 546–566 (1990)
28. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 60–76. ACM (1989)