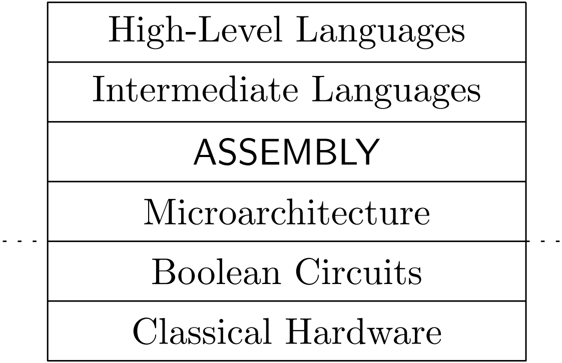
On Dynamic Lifting and Effect Typing in Circuit Description Languages

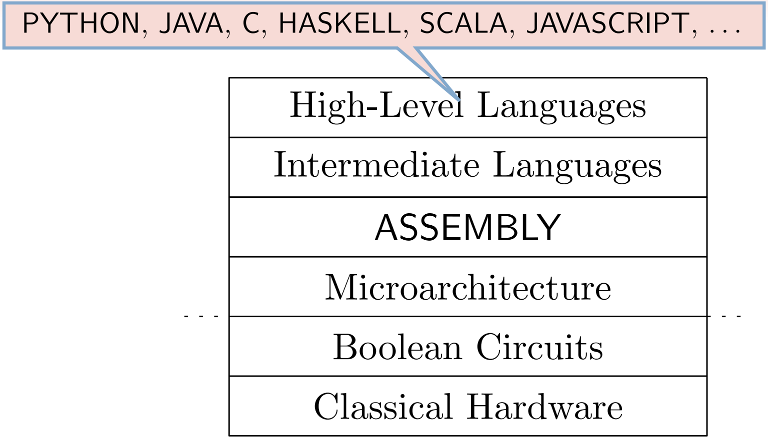
Andrea Colledan **Ugo Dal Lago**

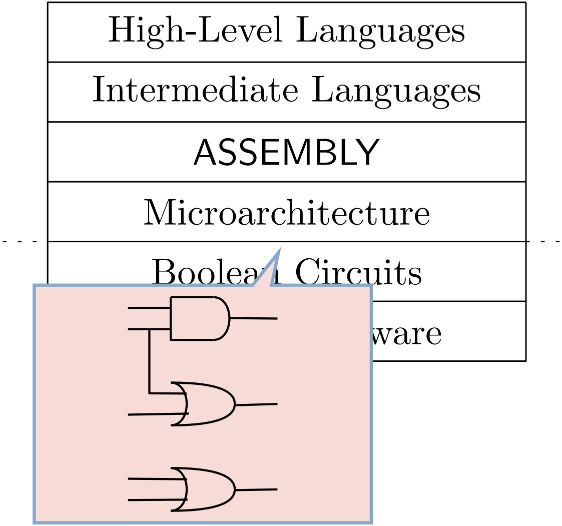
TYPES Workshop, Nantes, June 21st 2022

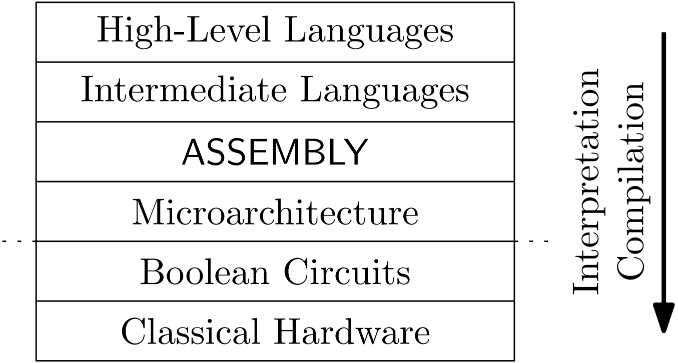
Part I

Context and Outline

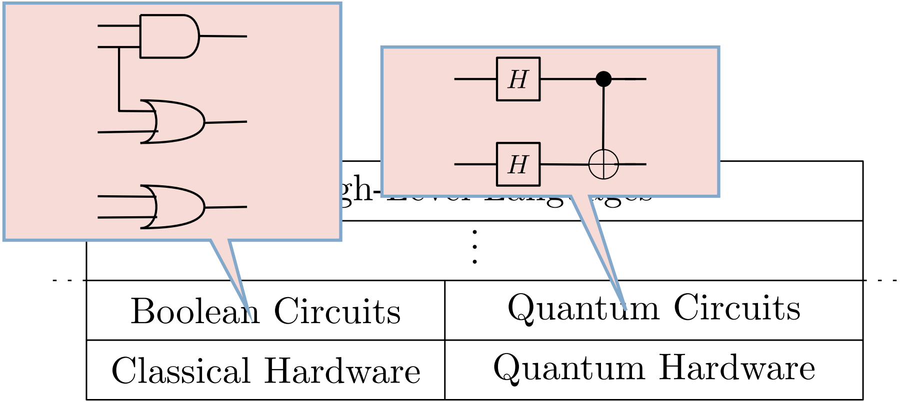

High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits
Classical Hardware


PYTHON, JAVA, C, HASKELL, SCALA, JAVASCRIPT, . . .
High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits
Classical Hardware


High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolea Circuits
ware


High-Level Languages
Intermediate Languages
ASSEMBLY
Microarchitecture
Boolean Circuits
Classical Hardware
Interpretation
Compilation

High-Level Languages	
⋮	
Boolean Circuits	Quantum Circuits
Classical Hardware	Quantum Hardware


H
H
Boolean Circuits
Quantum Circuits
Classical Hardware
Quantum Hardware

High-Level Languages

...

ircuits

rdware

- How could we *construct* high-level quantum programs?
- How could we compile a high-level program down to a *mixed* architecture?
- How to take advantage of the presence of quantum circuits, and of the computation power they provide?

THE QRAM MODEL

Conventions for Quantum Pseudocode

LANL report LAUR-96-2724

E. Knill

knill@lanl.gov, Mail Stop B265
Los Alamos National Laboratory
Los Alamos, NM 87545

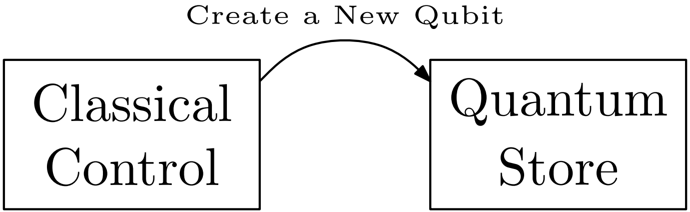
June 1996

Abstract

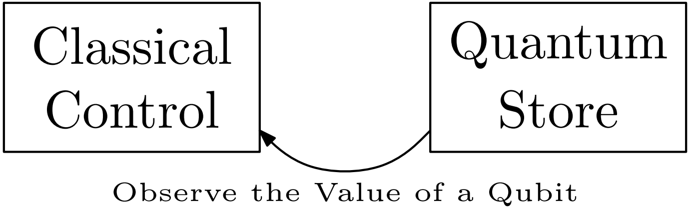
A few conventions for thinking about and writing quantum pseudocode are proposed. The conventions can be used for presenting any quantum algorithm down to the lowest level and are consistent with a quantum random access machine (QRAM) model for quantum computing. In principle a formal version of quantum pseudocode could be used in a future extension of a conventional language.
Note: This report is preliminary. Please let me know of any suggestions, omissions or errors so that I can correct them before distributing this work more widely.

Quantum Data and Classical Control

Quantum Data and Classical Control



Quantum Data and Classical Control



Quantum Data and Classical Control

A Brief Survey of Quantum Programming Languages

Peter Selinger

Department of Mathematics, University of Ottawa
Ottawa, Ontario, Canada K1N 6N5
selinger@mathstat.uottawa.ca

Abstract. This article is a brief and subjective survey of quantum programming language research.

1 Quantum Computation

Quantum computing is a relatively young subject. It has its beginnings in 1982, when Paul Benioff and Richard Feynman independently pointed out that a quantum mechanical system can be used to perform computations [11, p.12]. Feynman's interest in quantum computation was motivated by the fact that it is computationally very expensive to simulate quantum physical systems on classical computers. This is due to the fact that such simulation involves the manipulation is extremely large matrices (whose dimension is exponential in the size of the quantum system being simulated). Feynman conceived of quantum computers as a means of simulating nature much more efficiently.

The evidence to this day is that quantum computers can indeed perform

A SURVEY

ANOTHER SURVEY

Preprint, *Proceedings of the Second International Conference on Quantum, Nano, and Micro Technologies (ICQNM 2008)*, IEEE Computer Society, pp. 66-71, 2008.

A Survey of Quantum Programming Languages: History, Methods, and Tools

Donald A. Sofge, *Member, IEEE*

***Abstract*— Quantum computer programming is emerging as a new subject domain from multidisciplinary research in quantum computing, computer science, mathematics (especially quantum logic, lambda calculi, and linear logic), and engineering attempts to build the first non-trivial quantum computer. This paper briefly surveys the history, methods, and proposed tools for programming quantum computers circa late 2007. It is intended to provide an extensive but non-exhaustive look at work leading up to the current state-of-the-art in quantum computer programming. Further, it is an attempt to analyze the needed programming tools for quantum programmers, to use this analysis to predict the direction in which the field is moving, and to make recommendations for further development of quantum programming language tools.**

***Index Terms*— quantum computing, functional programming, imperative programming, linear logic, lambda calculus**

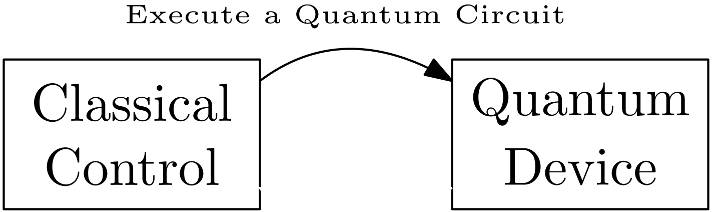
I. Introduction

THE importance of quantum computing has increased significantly in recent years due to the realization that we are rapidly approaching fundamental limits in shrinking the size of silicon-based integrated circuits (a trend over the past

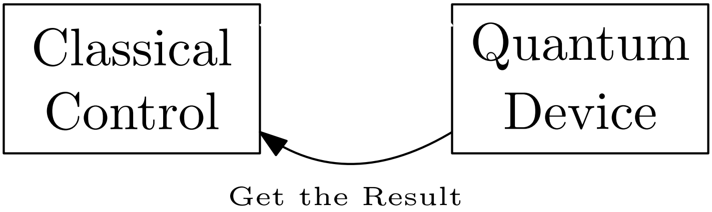
However, existing classical (non-quantum) programming languages lack both the data structures and the operators necessary to easily represent and manipulate quantum data. Quantum computing possesses certain characteristics that distinguish it from classical computing such as the superposition of quantum bits, entanglement, destructive measurement, and the no-cloning theorem. These differences must be thoroughly understood and even exploited in the context of quantum programming if we are to truly realize the potential of quantum computing. We need native quantum computer programming languages that embrace the fundamental aspects of quantum computing, rather than forcing us to adapt and use classical programming languages and techniques as ill-fitting stand-ins to develop quantum computer algorithms and simulations. Ultimately, a successful quantum programming language will facilitate easier coding of new quantum algorithms to perform useful tasks, allow or provide a capability for simulation of quantum algorithms, and facilitate the execution of quantum program code on quantum computer hardware.

II. Origins And History Of Quantum Computing

Quantum Data and Classical Control, Revisited



Quantum Data and Classical Control, Revisited



Quipper: A Scalable Quantum Programming Language

Alexander S. Green
Dalhousie University
agreen@mathstat.dal.ca

Peter LeFanu Lumsdaine
Institute of Advanced Studies
p.l.lumsdaine@gmail.com

Neil J. Ross
Dalhousie University
Neil.JR.Ross@Dal.Ca

Peter Selinger
Dalhousie University
selinger@mathstat.dal.ca

Benoît Valiron
University of Pennsylvania
benoit.valiron@monoidal.net

Abstract

The field of quantum algorithms is vibrant. Still, there is currently a lack of programming languages for describing quantum computation on a practical scale, i.e., not just at the level of toy problems. We address this issue by introducing Quipper, a scalable, expressive, functional, higher-order quantum programming language. Quipper has been used to program a diverse set of non-trivial quantum algorithms, and can generate quantum gate representations using trillions of gates. It is geared towards a model of computation that uses a classical computer to control a quantum device, but is not dependent on any particular model of quantum hardware. Quipper has proven effective and easy to use, and opens the door towards using formal methods to analyze quantum algorithms.

Keywords Quipper; Quantum Programming Languages

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory

1. Introduction

The earliest computers, such as the ENIAC and EDVAC, were both rare and difficult to program. The difficulty stemmed in part

This paper is a stepping stone towards meeting this challenge. We approach quantum computation from a programmer's perspective: how should one design a programming language that can implement real-world quantum algorithms in an efficient, legible and maintainable way? We introduce Quipper, a declarative language with a monadic operational semantics that is succinct, expressive, and scalable, with a sound theoretical foundation.

When we speak of Quipper being "scalable", we mean that it goes well beyond toy algorithms and mere proofs of concept. Many actual quantum algorithms in the literature are orders of magnitude more complex than what could be realistically implemented in previously existing quantum programming languages. We put Quipper to the test by implementing seven non-trivial quantum algorithms from the literature:

- Binary Welded Tree (BWT). To find a labeled node in a graph [4].
- Boolean Formula (BF). To evaluate a NAND formula [2]. The version of this algorithm implemented in Quipper computes a winning strategy for the game of Hex.
- Class Number (CL). To approximate the class group of a real quadratic number field [8].

QUIPPER

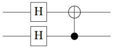
QUIPPER

- Quantum circuits can be constructed and manipulated within a fully-fledged functional programming language, namely HASKELL.

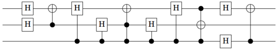
QUIPPER

- Quantum circuits can be constructed and manipulated within a fully-fledged functional programming language, namely HASKELL.
- **Quantum Circuit Construction**

```
mycirc :: Qubit -> Qubit -> Circ (Qubit, Qubit)
mycirc a b = do
  a <- hadamard a
  b <- hadamard b
  (a,b) <- controlled_not a b
  return (a,b)
```



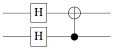

```
mycirc2 :: Qubit -> Qubit -> Qubit
  -> Circ (Qubit, Qubit, Qubit)
mycirc2 a b c = do
  mycirc a b
  with_controls c $ do
    mycirc a b
    mycirc b a
  mycirc a c
  return (a,b,c)
```



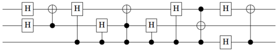
QUIPPER

- Quantum circuits can be constructed and manipulated within a fully-fledged functional programming language, namely HASKELL.
- **Quantum Circuit Construction**

```
mycirc :: Qubit -> Qubit -> Circ (Qubit, Qubit)
mycirc a b = do
  a <- hadamard a
  b <- hadamard b
  (a,b) <- controlled_not a b
  return (a,b)
```

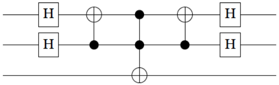



```
mycirc2 :: Qubit -> Qubit -> Qubit
  -> Circ (Qubit, Qubit, Qubit)
mycirc2 a b c = do
  mycirc a b
  with_controls c $ do
    mycirc a b
    mycirc b a
  mycirc a c
  return (a,b,c)
```



- **Quantum Circuit Transformation**

```
timestep :: Qubit -> Qubit -> Qubit
  -> Circ (Qubit, Qubit, Qubit)
timestep a b c = do
    mycirc a b
    qnot c `controlled` (a,b)
    reverse_simple mycirc (a,b)
    return (a,b,c)
```



PROTO-QUIPPER-M

A Categorical Model for a Quantum Circuit Description Language (Extended Abstract)

Francisco Rios and Peter Selinger
Dalhousie University
Halifax, Canada

Quipper is a practical programming language for describing families of quantum circuits. In this paper, we formalize a small, but useful fragment of Quipper called *Proto-Quipper-M*. Unlike its parent Quipper, this language is type-safe and has a formal denotational and operational semantics. Proto-Quipper-M is also more general than Quipper, in that it can describe families of morphisms in any symmetric monoidal category, of which quantum circuits are but one example. We design Proto-Quipper-M from the ground up, by first giving a general categorical model of *parameters* and *state*. The distinction between parameters and state is also known from hardware description languages. A parameter is a value that is known at circuit generation time, whereas a state is a value that is known

PROTO-QUIPPER-M

- Formalization of **a fragment** of QUIPPER.

PROTO-QUIPPER-M

- Formalization of **a fragment** of QUIPPER.
- Linear lambda calculus with constructs to manipulate circuits:

$$M, N ::= \cdots \mid \ell \mid (\vec{\ell}, C, \vec{\ell'}) \mid \mathsf{apply}(M, N) \mid \mathsf{box}_T\, M.$$

PROTO-QUIPPER-M

- Formalization of **a fragment** of `QUIPPER`.
- Linear lambda calculus with constructs to manipulate circuits:

$$M, N ::= \cdots \mid \ell \mid (\vec{\ell}, C, \vec{\ell'}) \mid \mathsf{apply}(M, N) \mid \mathsf{box}_T\, M.$$

The usual constructions from linear λ-calculi:

- Abstractions and applications;
- Linear products;
- Linear coproducts.

PROTO-QUIPPER-M

Labels

- Formalization of **a fragment** of `QUIPPER`.
- Linear lambda calculus with constructs to manipulate circuits:

$$M, N ::= \cdots \mid \ell \mid (\vec{\ell}, C, \vec{\ell'}) \mid \mathsf{apply}(M, N) \mid \mathsf{box}_T\, M.$$

The usual constructions from linear λ-calculi:

- Abstractions and applications;
- Linear products;
- Linear coproducts.

PROTO-QUIPPER-M

Boxed Circuits

Labels

- Formalization of **a fragment** of QUIPPER.
- Linear lambda calculus with constructs to manipulate circuits:

$$M, N ::= \cdots \mid \ell \mid (\vec{\ell}, C, \vec{\ell'}) \mid \mathsf{apply}(M, N) \mid \mathsf{box}_T\, M.$$

The usual constructions from linear λ-calculi:

- Abstractions and applications;
- Linear products;
- Linear coproducts.

PROTO-QUIPPER-M

Boxed Circuits

Modifies the underlying circuit

Labels

- Formalization of **a fragment** of QUIPPER.
- Linear lambda calculus with constructs to manipulate circuits:

$$M, N ::= \cdots \mid \ell \mid (\vec{\ell}, C, \vec{\ell'}) \mid \mathsf{apply}(M, N) \mid \mathsf{box}_T\, M.$$

The usual constructions from linear λ-calculi:

- Abstractions and applications;
- Linear products;
- Linear coproducts.

PROTO-QUIPPER-M

- Formalization of **a fragment** of QUIPPER.
- Linear lambda calculus with constructs to manipulate circuits.

$$M, N ::= \cdots \mid \ell \mid (\vec{\ell}, C, \vec{\ell'}) \mid \mathsf{apply}(M, N) \mid \mathsf{box}_T\, M.$$

Labels

Boxed Circuits

Modifies the underlying circuit

Turns a function into a circuit

The usual constructions from linear λ-calculi:

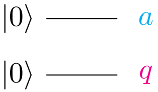
- Abstractions and applications;
- Linear products;
- Linear coproducts.

PROTO-QUIPPER-M, by Example

$$\mathsf{let}\ \langle q, a\rangle = \mathsf{apply}(\mathsf{qinit}_2, *)\ \mathsf{in}$$
$$\mathsf{let}\ \langle q', a'\rangle = \mathsf{apply}(\mathsf{CNOT}, \langle q, a\rangle)\ \mathsf{in}$$
$$\mathsf{let}\ q'' = \mathsf{apply}(\mathsf{H}, q')\ \mathsf{in}\ \langle q'', a'\rangle$$

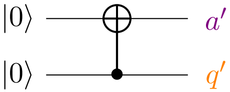
PROTO-QUIPPER-M, by Example

$\rightarrow$ let $\langle q, a\rangle = \text{apply}(\text{qinit}_2, *)$ in
let $\langle q', a'\rangle = \text{apply}(\text{CNOT}, \langle q, a\rangle)$ in
let $q'' = \text{apply}(\text{H}, q')$ in $\langle q'', a'\rangle$



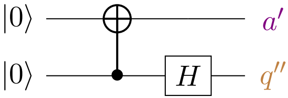
PROTO-QUIPPER-M, by Example

$$\begin{aligned}
&\mathsf{let}\ \langle q, a\rangle = \mathsf{apply}(\mathsf{qinit}_2, *)\ \mathsf{in}\\
\rightarrow\ &\mathsf{let}\ \langle q', a'\rangle = \mathsf{apply}(\mathsf{CNOT}, \langle q, a\rangle)\ \mathsf{in}\\
&\mathsf{let}\ q'' = \mathsf{apply}(\mathsf{H}, q')\ \mathsf{in}\ \langle q'', a'\rangle
\end{aligned}$$



PROTO-QUIPPER-M, by Example

$$\text{let } \langle q, a\rangle = \text{apply}(\text{qinit}_2, *) \text{ in}$$
$$\text{let } \langle q', a'\rangle = \text{apply}(\text{CNOT}, \langle q, a\rangle) \text{ in}$$
$$\rightarrow \text{let } q'' = \text{apply}(\text{H}, q') \text{ in } \langle q'', a'\rangle$$



PROTO-QUIPPER-M: Operational Semantics

$$M$$

PROTO-QUIPPER-M: Operational Semantics

Term

$$M$$

PROTO-QUIPPER-M: Operational Semantics

$$M \Downarrow (C, V)$$

Term

Circuit

PROTO-QUIPPER-M: Operational Semantics

Term

$$M \Downarrow (C, V)$$

Circuit

Value

PROTO-QUIPPER-M: Operational Semantics

- Big-step
- Call-by-value

Term

$$M \Downarrow (C, V)$$

Circuit

Value

PROTO-QUIPPER-M: Type System

$$
\begin{aligned}
T, U &::= \quad \mathsf{Qubit} \mid \mathsf{Bit} \mid \langle T, U \rangle. \\
A, B &::= \quad \cdots \mid \mathsf{Qubit} \mid \mathsf{Bit} \mid \langle A, B \rangle \mid \mathsf{Circ}(T, U).
\end{aligned}
$$

PROTO-QUIPPER-M: Type System

$$
\begin{aligned}
T, U &::= \quad \mathsf{Qubit} \mid \mathsf{Bit} \mid \langle T, U \rangle. \\
A, B &::= \quad \cdots \mid \mathsf{Qubit} \mid \mathsf{Bit} \mid \langle A, B \rangle \mid \mathsf{Circ}(T, U).
\end{aligned}
$$

$$\Gamma; Q \vdash M : A$$

PROTO-QUIPPER-M: Type System

$$T, U ::= \quad \mathsf{Qubit} \mid \mathsf{Bit} \mid \langle T, U \rangle.$$
$$A, B ::= \quad \cdots \mid \mathsf{Qubit} \mid \mathsf{Bit} \mid \langle A, B \rangle \mid \mathsf{Circ}(T, U).$$

$$\Gamma; Q \vdash M : A$$

Types of Term Variables (Γ)

Types of Labels (Q)

PROTO-QUIPPER-M: Type System

$$T, U ::= \quad \mathsf{Qubit} \mid \mathsf{Bit} \mid \langle T, U \rangle.$$
$$A, B ::= \quad \cdots \mid \mathsf{Qubit} \mid \mathsf{Bit} \mid \langle A, B \rangle \mid \mathsf{Circ}(T, U).$$

$$\Gamma; Q \vdash M : A$$

$$\mathit{apply}\,\frac{\Phi, \Gamma_1; Q_1 \vdash M : \mathsf{Circ}(T, U) \quad \Phi, \Gamma_2; Q_2 \vdash N : T}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash \mathsf{apply}(M, N) : U} \qquad \mathit{box}\,\frac{\Gamma; Q \vdash M : !(T \multimap U)}{\Gamma; Q \vdash \mathsf{box}_T\, M : \mathsf{Circ}(T, U)}$$

Part II

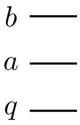
Dynamic Lifting

Teleportation

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
 a <- qnot a `controlled` q
 q <- hadamard q
 (x,y) <- measure (q,a)
 b <- gate_X b `controlled` y
 b <- gate_Z b `controlled` x
 return b
```

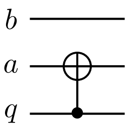
Teleportation

```
   teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
→  teleport b a q = do
     a <- qnot a `controlled` q
     q <- hadamard q
     (x,y) <- measure (q,a)
     b <- gate_X b `controlled` y
     b <- gate_Z b `controlled` x
     return b
```



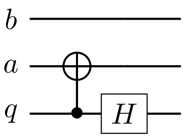
Teleportation

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
→ a <- qnot a `controlled` q
  q <- hadamard q
  (x,y) <- measure (q,a)
  b <- gate_X b `controlled` y
  b <- gate_Z b `controlled` x
  return b
```



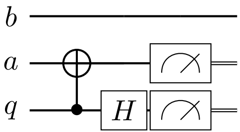
Teleportation

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
  a <- qnot a `controlled` q
→ q <- hadamard q
  (x,y) <- measure (q,a)
  b <- gate_X b `controlled` y
  b <- gate_Z b `controlled` x
  return b
```



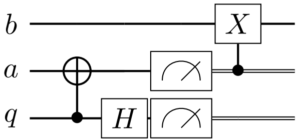
Teleportation

```
  teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
  teleport b a q = do
   a <- qnot a `controlled` q
   q <- hadamard q
→  (x,y) <- measure (q,a)
   b <- gate_X b `controlled` y
   b <- gate_Z b `controlled` x
   return b
```



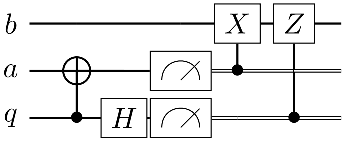
Teleportation

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
 a <- qnot a `controlled` q
 q <- hadamard q
 (x,y) <- measure (q,a)
→ b <- gate_X b `controlled` y
 b <- gate_Z b `controlled` x
 return b
```



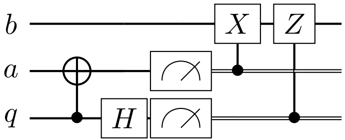
Teleportation

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
 a <- qnot a ‘controlled‘ q
 q <- hadamard q
 (x,y) <- measure (q,a)
 b <- gate_X b ‘controlled‘ y
→ b <- gate_Z b ‘controlled‘ x
 return b
```



Teleportation

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
 a <- qnot a `controlled` q
 q <- hadamard q
 (x,y) <- measure (q,a)
 b <- gate_X b `controlled` y
 b <- gate_Z b `controlled` x
→ return b
```

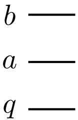


Teleportation with Dynamic Lifting

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
 a <- qnot a `controlled` q
 q <- hadamard q
 (x,y) <- measure (q,a)
 (u,s) <- dynamic_lift(x,y)
 b <- if s then gate_X b else return b
 b <- if u then gate_Z b else return b
 return b
```

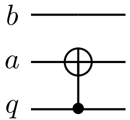
Teleportation with Dynamic Lifting

```
   teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
→  teleport b a q = do
     a <- qnot a `controlled` q
     q <- hadamard q
     (x,y) <- measure (q,a)
     (u,s) <- dynamic_lift(x,y)
     b <- if s then gate_X b else return b
     b <- if u then gate_Z b else return b
     return b
```



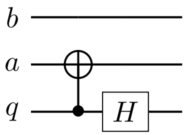
Teleportation with Dynamic Lifting

```
   teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
   teleport b a q = do
→   a <- qnot a `controlled` q
    q <- hadamard q
    (x,y) <- measure (q,a)
    (u,s) <- dynamic_lift(x,y)
    b <- if s then gate_X b else return b
    b <- if u then gate_Z b else return b
    return b
```



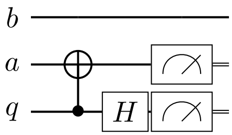
Teleportation with Dynamic Lifting

```
  teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
  teleport b a q = do
   a <- qnot a `controlled` q
→  q <- hadamard q
   (x,y) <- measure (q,a)
   (u,s) <- dynamic_lift(x,y)
   b <- if s then gate_X b else return b
   b <- if u then gate_Z b else return b
   return b
```



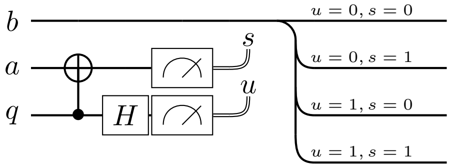
Teleportation with Dynamic Lifting

```
   teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
   teleport b a q = do
    a <- qnot a `controlled` q
    q <- hadamard q
→  (x,y) <- measure (q,a)
   (u,s) <- dynamic_lift(x,y)
   b <- if s then gate_X b else return b
   b <- if u then gate_Z b else return b
   return b
```



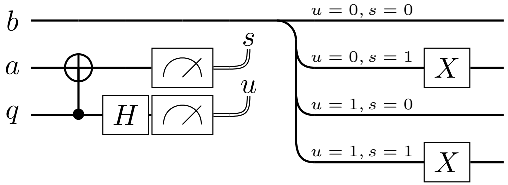
Teleportation with Dynamic Lifting

```
    teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
    teleport b a q = do
     a <- qnot a `controlled` q
     q <- hadamard q
     (x,y) <- measure (q,a)
→    (u,s) <- dynamic_lift(x,y)
     b <- if s then gate_X b else return b
     b <- if u then gate_Z b else return b
     return b
```



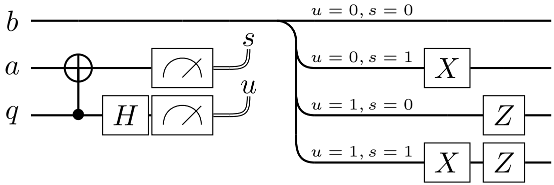
Teleportation with Dynamic Lifting

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
  a <- qnot a `controlled` q
  q <- hadamard q
  (x,y) <- measure (q,a)
  (u,s) <- dynamic_lift(x,y)
→ b <- if s then gate_X b else return b
  b <- if u then gate_Z b else return b
  return b
```



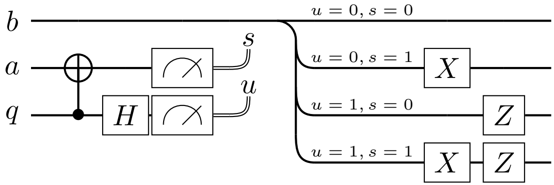
Teleportation with Dynamic Lifting

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
  a <- qnot a `controlled` q
  q <- hadamard q
  (x,y) <- measure (q,a)
  (u,s) <- dynamic_lift(x,y)
  b <- if s then gate_X b else return b
→ b <- if u then gate_Z b else return b
  return b
```



Teleportation with Dynamic Lifting

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b a q = do
 a <- qnot a `controlled` q
 q <- hadamard q
 (x,y) <- measure (q,a)
 (u,s) <- dynamic_lift(x,y)
 b <- if s then gate_X b else return b
 b <- if u then gate_Z b else return b
→ return b
```

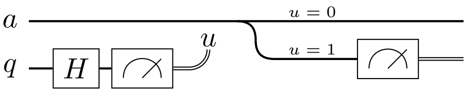


Beyond Uniform Dynamic Lifting

- In the previous example, the various branches induced by dynamic lifting are **uniformly typed**, both in the term and in the circuit.

Beyond Uniform Dynamic Lifting

- In the previous example, the various branches induced by dynamic lifting are **uniformly typed**, both in the term and in the circuit.
- There are cases in which uniformity does not hold, at least if we want to be modular.
  - Measurement-based quantum computing:



- It would be nice to allow for the wildest forms of dynamic lifting, without imposing any restriction.

Our Contribution

A Conservative Extension of `PROTO-QUIPPER-M`...

Our Contribution

A Conservative Extension of `PROTO-QUIPPER-M`...

... Capturing a Very General form of Dynamic Lifting...

Our Contribution

A Conservative Extension of `PROTO-QUIPPER-M`...

... Capturing a Very General form of Dynamic Lifting...

... And Enjoying Type Soundness

Concrete Categorical Model of a Quantum Circuit Description Language with Measurement

Dongho Lee
Université Paris-Saclay, CentraleSupélec, LMF, France & CEA, List, France

Valentin Perrelle
Université Paris-Saclay, CEA, List, France

Benoît Valiron
Université Paris-Saclay, CentraleSupélec, LMF, France

Zhaowei Xu
Université Paris-Saclay, LMF, France

Abstract

In this paper, we introduce dynamic lifting to a quantum circuit-description language, following the Proto-Quipper language approach. Dynamic lifting allows programs to transfer the result of measuring quantum data – qubits – into classical data – booleans – . We propose a type system and an operational semantics for the language and we state safety properties. Next, we introduce a concrete categorical semantics for the proposed language, basing our approach on a recent model from Rios&Selinger for Proto-Quipper-M. Our approach is to construct on top of a concrete category of circuits with measurements a Kleisli category, capturing as a side effect the action of retrieving

PROTO-QUIPPER-L

Proto-Quipper with dynamic lifting

Peng Fu, Kohei Kishida, Neil J. Ross, Peter Selinger

April 28, 2022

Abstract

Quipper is a functional programming language for quantum computing. Proto-Quipper is a family of languages aiming to provide a formal foundation for Quipper. In this paper, we extend Proto-Quipper-M with a construct called *dynamic lifting*, which is present in Quipper. By virtue of being a circuit description language, Proto-Quipper has two separate runtimes: circuit generation time and circuit execution time. Values that are known at circuit generation time are called *parameters*, and values that are known at circuit execution time are called *states*. Dynamic lifting is an operation that enables a state, such as the result of a measurement, to be lifted to a parameter, where it can influence the generation of the next portion of the circuit. As a result, dynamic lifting enables Proto-Quipper programs to interleave

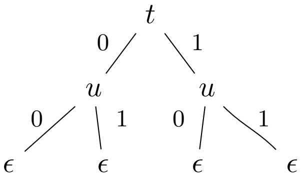
PROTO-QUIPPER-Dyn

Part III

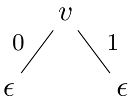
PROTO-QUIPPER-K

Lifting Trees

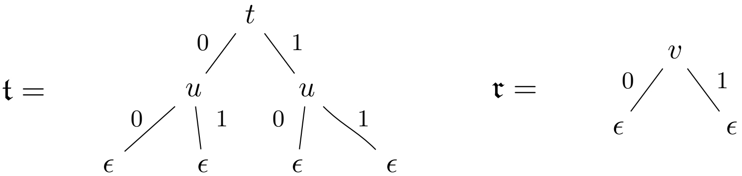
$\mathfrak{t} =$



$\mathfrak{r} =$

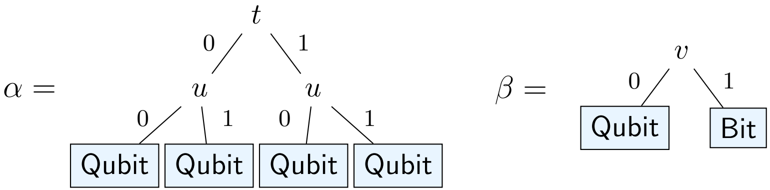


Lifting Trees

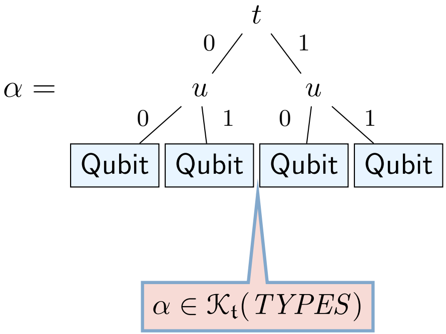


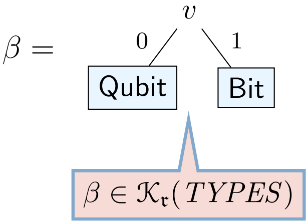
- One can **associate objects** to the leaves of any lifting tree.
- This way, lifting trees become mathematical representation of an object whose identity **depends** on the value of one or more lifted variables
- The objects one attaches to the lifting tree's leaves may be anything: terms, values, types, etc.

Lifted Types

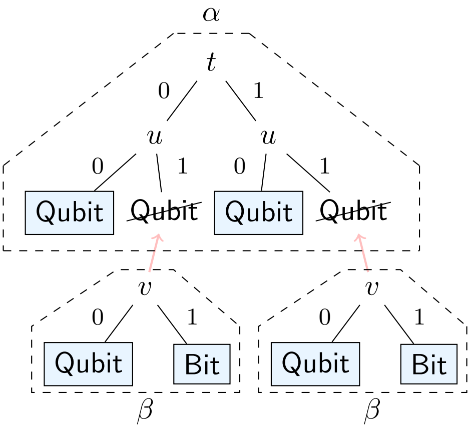


Lifted Types

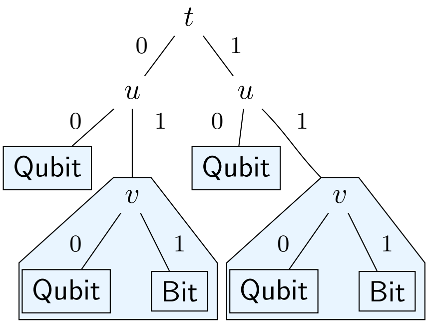




Manipulating Lifted Objects

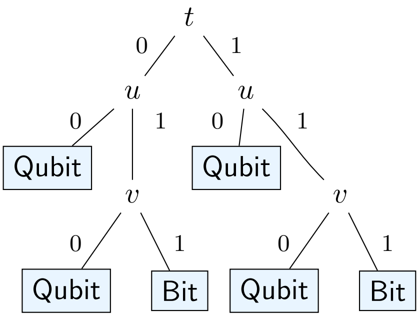


Manipulating Lifted Objects



$$= \alpha[\beta]^{\{u=1\}}$$

Manipulating Lifted Objects



$$= \lfloor \alpha[\beta]^{\{u=1\}} \rfloor$$

PROTO-QUIPPER-K

$$M, N ::= \cdots \mid \mathsf{let}\ x = M\ \mathsf{in}\ \mu.$$

PROTO-QUIPPER-K

Minor variation on PROTO-QUIPPER-M

$\mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERMS})$

$$M, N ::= \cdots \mid \mathsf{let}\ x = M\ \mathsf{in}\ \mu.$$

PROTO-QUIPPER-K

Minor variation on PROTO-QUIPPER-M

$\mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERMS})$

$$M, N ::= \cdots \mid \mathsf{let}\ x = M\ \mathsf{in}\ \mu.$$

$$A, B ::= \cdots \mid A \multimap_{\mathfrak{t}} \beta \mid\ !\alpha \mid \mathsf{Circ}_{\mathfrak{t}}(T, \gamma).$$

PROTO-QUIPPER-K

Minor variation on PROTO-QUIPPER-M

$\mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERMS})$

$$M, N ::= \cdots \mid \mathsf{let}\ x = M\ \mathsf{in}\ \mu.$$

$$A, B ::= \cdots \mid A \multimap_{\mathfrak{t}} \beta \mid !\alpha \mid \mathsf{Circ}_{\mathfrak{t}}(T, \gamma).$$

$$\Gamma; Q \vdash_c^{\mathfrak{t}} M : \alpha$$

PROTO-QUIPPER-K

Minor variation on PROTO-QUIPPER-M

$\mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERMS})$

$$M, N ::= \cdots \mid \mathsf{let}\ x = M\ \mathsf{in}\ \mu.$$

$$A, B ::= \cdots \mid A \multimap_{\mathfrak{t}} \beta \mid !\alpha \mid \mathsf{Circ}_{\mathfrak{t}}(T, \gamma).$$

$$\Gamma; Q \vdash_c^{\mathfrak{t}} M : \alpha$$

$\alpha, \beta, \gamma \in \mathcal{K}_{\mathfrak{t}}(\mathit{TYPES})$

PROTO-QUIPPER-K

Minor variation on PROTO-QUIPPER-M

$\mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERMS})$

$$M, N ::= \cdots \mid \mathsf{let}\ x = M\ \mathsf{in}\ \mu.$$

$$A, B ::= \cdots \mid A \multimap_{\mathfrak{t}} \beta \mid !\alpha \mid \mathsf{Circ}_{\mathfrak{t}}(T, \gamma).$$

$$\Gamma; Q \vdash_c^{\mathfrak{t}} M : \alpha$$

$\alpha, \beta, \gamma \in \mathcal{K}_{\mathfrak{t}}(\mathit{TYPES})$

$$\mathrm{let}\ \frac{\Phi, \Gamma_1; Q_1 \vdash_c^{\mathfrak{t}} M : \alpha \quad \mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERM}) \quad \Phi, \Gamma_2, x : \alpha; Q_2 \Vdash_c^{\mathfrak{t}[\mathfrak{r}_a]_a} \mu : \theta}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\lfloor \mathfrak{t}[\mathfrak{r}_a]_a \rfloor} \mathsf{let}\ x = M\ \mathsf{in}\ \mu : \lfloor \theta \rfloor}$$

PROTO-QUIPPER-K: Operational Semantics

$\phi \in \mathcal{K}_{\mathfrak{t}}(\mathit{VALUES})$

$$M \Downarrow (C, \phi)$$

Type Soundness

Subject Reduction

If $\vdash^{\mathsf{t}} M : \alpha$ and $\exists C, \phi. M \Downarrow (C, \phi)$, then $\vdash^{\mathsf{t}} (C, \phi) : \alpha$.

Proof. By induction and case analysis on the last rule used to derive $M \Downarrow (C, \phi)$.

Progress

If $\vdash^{\mathsf{t}} M : \alpha$, then either $\exists C, \phi. M \Downarrow (C, \phi)$ or $M \Uparrow$.

Proof. We prove that if $\vdash^{\mathsf{t}} M : \alpha$ and $\nexists C, \phi. M \Downarrow (C, \phi)$, then $M \Uparrow$. We proceed by coinduction and case analysis on M.

Future Work

- Understanding the **monadic status** of our branching effect.
- Studying the **relationship** between branching and regular circuits.
- Giving a **denotational** account of `PROTO-QUIPPER-K`.
- ...

Thank You!

Questions?