Linear Quipper: 埋め込み線形型付き量子プログラミング言語

菅野 翔太　松田 一孝

本研究では，埋め込み線形型付き量子プログラミング言語 Linear Quipper を提案する．Quipper は Haskell に埋め込まれた量子プログラミング言語であり，高階関数等の Haskell の機能を利用し量子回路を簡潔に記述することができる．しかし，Quipper は量子ビットと量子回路を素朴に Haskell のデータ及び関数として表現しているため，量子ビットの複製や破棄を許してしまうという問題がある．本研究では，Polakow による線形 λ 計算の埋め込み技法を応用し，Quipper に線形型を導入した Linear Quipper を実現する．具体的には，Linear Quipper は Quipper の構文を表現する型クラスを提供する．この型クラスの各メソッドは Quipper の各言語要素に対応し，メソッドの型はその言語要素が持つべき線形型を表現している．そのため，この型クラスに適切なインスタンスを与えることにより Linear Quipper のプログラムを Quipper のプログラムに変換することができ，Quipper の実装を再利用可能という利点がある．また，GHC 拡張を活用することにより，少なくとも単純な回路については Linear Quipper では Quipper とほぼ同等の記述により量子回路を記述可能である．

This paper proposes Linear Quipper, an embedded linear quantum programming language, extending Quipper with Linear Types. Quipper is an embedded quantum programming language in Haskell that makes use of Haskell's functionalities including its powerful type system and higher-order functions to describe quantum circuits. However, since Quipper naively represents quantum bits and circuits respectively by Haskell data and functions, it allows users to define circuits that clone or discard quantum bits, which are physically impossible. In this paper, we propose Linear Quipper, which addresses this limitation by adopting the Polakow's embedding method of the linear lambda calculus. Linear Quipper provides a type class that represents the syntax of Quipper. Each method of the type class represents a languages construct in Quipper and has a type corresponding to the linear type the language construct should have. We can translate Linear Quipper programs to Quipper programs via an appropriate instance of the type class, which enables us to reuse the Quipper's implementation. Furthermore, thanks to GHC extensions, Linear Quipper has similar programmability to Quipper at least for simple circuits, allowing users to reuse Quipper programs only with small changes.

1 序論

ゲート型量子コンピュータの実用化が期待される中，量子回路を記述するための様々なプログラミング言語（量子プログラミング言語）が提案されている．例えば，命令型量子プログラミング言語には，QCL [6] や Microsoft により開発された Q# [10] 等があり，関数型量子プログラミング言語には，QPL [8] や QML [1], Quipper [5] といったものがある．

Quipper [5] も量子プログラミング言語の 1 つであり，Haskell に埋め込まれた形で実装されている．Quipper では量子回路は Haskell の関数として表現されているため，ユーザーは高階関数や強力な型システムなど Haskell の様々な機能を利用して量子回路の記述が可能である．また，Quipper は関数型の量子プログラミングも，命令型の量子プログラミングもサポートしている．さらに，Quipper では量子回路のシミュレーションに加え，量子回路の描画出力など

Linear Quipper: An Embedded Linear Quantum Programming Language

(This is an unrefereed paper. Copyrights belong to the Authors.)

Shota Sugano　Kazutaka Matsuda, 東北大学大学院情報科学研究科, Graduate School of Information Sciences, Tohoku University.

様々な便利な機能も提供されている．

しかし Quippe では量子ビットや量子回路を Haskell の通常のデータや関数として素朴に表現しているため，量子ビットの複製や破棄といった記述を許してしまうという問題がある．例えば，Quipper では $qcirc\ q_1\ q_2 = return\ (q_1, q_1)$ といった関数を定義することができる．この例では量子ビット q_1 は複製され，q_2 が破棄されているため量子回路として意味をなさない．そこで Quipper の関数に適切な線形型を導入し，このような操作を禁止したい．

本研究では，Polakow による線型 λ 計算の埋め込み技法 [7] を応用して，Quipper の関数型サブセットに線形型を導入した Linear Quipper を提案する．Polakow の埋め込みの基本的なアイデアは，線形 λ 計算における term-in-context を Haskell による型推論が可能な形で型クラスとして表現することである．我々はその型クラスを拡張し，Quipper の各関数を「その関数が持つべき線形型」を表現する型を持つメソッドとして追加する．拡張された型クラスの適切なインスタンスにおいてそれらの追加されたメソッドを元の Quipper の関数を用いて実装することにより，Quipper の実装を再利用することが可能である．

この拡張は単純ではないためさらなる工夫が必要である．問題の 1 つは Quipper には素朴には線形型を与えるのが難しい関数が存在することである．例えば Quipper の関数 *controlled* は第 2 引数を制御ビットとしたような第 1 引数の回路を表現するが，第 2 引数に出現する量子ビットを消費しない．我々は Polakow の埋め込み技法において変数の線形性と型が別々に管理されているという特徴を利用し，*controlled* にも適切な表現を与える．また，それだけだと Quipper のプログラムと Linear Quipper のプログラムが大きく異なってしまうようという問題がある．例えば，Quipper では **do** 記法を用いて量子回路をプログラムするが，素朴に Polakow の埋め込みを応用するのでは **do** 記法が利用できなくなる．我々はこの問題点を GHC の RebindableSyntax 拡張を利用することにより解消し，Linear Quipper においても **do** 記法を用いた回路記述を可能にする．

まとめると，本研究の貢献は以下である．

- Polakow による線形 λ 計算の埋め込み技法を応用し，Quipper に線形型を導入した Linear Quipper を提案する．
- Quipper には単純に線形型を与えるのが難しい関数 *controlled* が存在するが，Polakow による埋め込み技法の特徴を活用し，適切にその表現方法を与える．
- GHC 拡張を活用し，Linear Quipper においても Quipper と同様に **do** 記法を用いた回路記述を可能にする．

本論文は次のように構成される．2 節では Quipper 及び Polakow の埋め込み技法について簡潔に紹介する．3 節では Linear Quipper の基本設計及び単純な実現方法の問題点とその解決方法について述べる．4 節では量子テレポーテーションの例に Quipper の記述性と Linear Quipper の記述性を比較する．5 節では関連研究について議論し，6 節では本論文をまとめる．

2 準備

本節では，まず埋め込み量子プログラミング言語 Quipper [4,5] について簡潔に述べ（2.1 節），その後 Polakow の線形 λ 計算の Haskell への埋め込み手法 [7] について述べる（2.2 節）．

2.1 Quipper

Quipper は Haskell に埋め込まれた量子プログラミング言語である [4,5]．Quipper では，関数的な記述や命令的な記述の両方が可能であるが，本稿では関数的な記述方法に絞って述べる．

関数的な Quipper においては，量子ビットは *Qubit* という型で表現されていて，n 入力 n 出力の量子回路は

$$\underbrace{Qubit \to \cdots \to Qubit}_{n} \to Circ\ \underbrace{(Qubit, \ldots, Qubit)}_{n}$$

という型の関数として表現される．ここで *Circ* は量子回路を表すモナドである．Quipper では，$qnot :: Qubit \to Circ\ Qubit$ や $hadamard :: Qubit \to Circ\ Qubit$ などの多数の基本的な量子ゲートが定義されていて，それをモナドの演算（$\gg\!=$，*return*）で

組み合わせることにより，量子回路の記述が可能である．たとえば，$circA, circB :: Qubit \to Qubit \to (Qubit, Qubit)$ という量子回路が与えられたときに，$circA$ の二つ目の出力を，$circB$ の一つ目の入力に繋げることで得られる 3 入力 3 出力回路を以下のように記述できる．

$$
\begin{array}{l}
circAB :: Qubit \to Qubit \to Qubit \\
\quad \to Circ\ (Qubit, Qubit, Qubit) \\
circAB\ i_1\ i_2\ i_3 = \mathbf{do} \\
\quad (s_1, s_2) \leftarrow circA\ i_1\ i_2 \\
\quad (t_1, t_2) \leftarrow circB\ s_2\ i_3 \\
\quad return\ (s_1, t_1, t_2)
\end{array}
$$

こうして定義された量子回路に対し，Quipper ではシミュレーションしたり，また図として出力したりすることが可能である [4,5].

Quipper は量子回路の記述を助けるための様々な関数を提供している．そうした関数の中で重要なものの一つである，本論文で主に議論の対象とする *controlled* 関数について述べる．

2.1.1 *controlled* 関数

制御付きのゲートとは，制御ビットが 1 であるときのみ特定のゲートを適用するゲートである．たとえば，制御 NOT ゲートは，二つの入力を取り，一つのビットが 1 であるときだけ，もう一方のビットを反転させる．制御 NOT ゲートは可逆回路を記述するための基本回路の一つであり，量子回路においても重要である．Quipper では，こうした制御付きのゲートを作るための関数，*controlled* を提供している．たとえば，*controlled* を使うことで以下のように制御 NOT ゲートを定義することができる．

$$
\begin{array}{l}
cnot :: Qubit \to Qubit \to Circ\ (Qubit, Qubit) \\
cnot\ a\ b = \mathbf{do} \\
\quad a' \leftarrow qnot\ a\ \text{`}controlled\text{`}\ b \\
\quad return\ (a', b)
\end{array}
$$

ここで，$qnot :: Qubit \to Circ\ Qubit$ は（量子）NOT ゲートである．

具体的には，*controlled* は $ControlSource\ c \Rightarrow Circ\ a \to c \to Circ\ a$ という型を持つ．直観的には，$controlled\ (g\ a)\ c$ は制御ビット（の組）c が全て 1 であるときにゲート g を a に適用するようなゲートを表す．ここで，制約 $ControlSource\ c$ は c が制御ビットとして有効な型，具体的には古典ビットまとは量子ビットからなる組であることを表している．これにより，以下のように複数の制御ビットを持つ関数を *controlled* 関数一つで記述が可能となっている．

$$
\begin{array}{l}
ccnot :: Qubit \to Qubit \to Qubit \\
\qquad\qquad \to Circ\ (Qubit, Qubit, Qubit) \\
ccnot\ a\ b\ c = \mathbf{do} \\
\quad a' \leftarrow qnot\ a\ \text{`}controlled\text{`}\ (b, c) \\
\quad return\ (a', b, c)
\end{array}
$$

2.1.2 問題点

Quipper は量子ビットを Haskell の通常のデータとして表現しているため，次のような記述が可能である．

例 1 (*badcirc*)．この例では，量子ビット a の複製，量子ビット b の破棄が起きている．これは量子回路として実現不可能な操作である．

$$
\begin{array}{l}
badcirc :: Qubit \to Qubit \to Circ\ Qubit \\
badcirc\ a\ b = \mathbf{do} \\
\quad a \leftarrow hadamard\ a \\
\quad b \leftarrow hadamard\ b \\
\quad a \leftarrow qnot\ a\ \text{`}controlled\text{`}\ a \\
\quad return\ a
\end{array}
$$

2.2 線形 λ 計算の Polakow による Haskell への埋め込み

Polakow は線形 λ 計算を Haskell に埋め込む手法を提案した [7].

素朴な線形 λ 計算の埋め込み手法の一つは，$\Gamma \vdash e : \tau$ という項を $LLam\ \Gamma\ \tau$ のような de Bruijn 表現で表すことである．しかしながら，この手法は埋め込み実装において問題がある．例として，以下の線形 λ 計算における関数適用の型付け規則を考える．[†1]

$$
\frac{\Gamma = \Gamma_1 \uplus \Gamma_2 \quad \Gamma_1 \vdash e_1 \vdash \tau \multimap \tau' \quad \Gamma_2 \vdash e_2 \vdash \tau}{\Gamma \vdash e_1\ e_2 : \tau'}
$$

ここで，$\Gamma \vdash e_1\ e_2 : \tau'$ であることを確認する

†1 本稿では説明を単純にするため，特に必要な場合を除いて非線形型環境を無視する．

には，$\Gamma = \Gamma_1 \uplus \Gamma_2$ という型環境 Γ_1 と Γ_2 で，$\Gamma_1 \vdash e_1 : \tau \multimap \tau'$ で $\Gamma_2 \vdash e_2 : \tau$ であるものを発見する必要がある．しかし，このような型環境の発見を Haskell の型推論器に行わせるのは用意ではない．

この問題点を解決するため，Polakow は IO-$\top$ システムという線形 λ 計算の変種を提案している [7]．主なアイデアは元の線形 λ 計算における型環境 Γ を，その差分 $\Gamma_1 \setminus \Gamma_2$ が Γ を表すような**入力型環境** Γ_1 と**出力型環境** Γ_2 の組により表現することである．たとえば，線形 λ 計算の $\multimap$ に関するフラグメントに対する型付け規則は以下となる．

$$\frac{}{(\Gamma_1, x : \tau, \Gamma_1') \setminus (\Gamma_2, \Box, \Gamma_2') \vdash x : \tau}$$

$$\frac{\Gamma_1 \setminus \Gamma_2 \vdash e_1 : \tau \multimap \tau' \quad \Gamma_2 \setminus \Gamma_3 \vdash e_2 : \tau}{\Gamma_1 \setminus \Gamma_3 \vdash e_1\ e_2 : \tau'}$$

$$\frac{(\Gamma_1, x : \tau) \setminus (\Gamma_2, \Box) \vdash e : \tau'}{\Gamma_1 \setminus \Gamma_2 \vdash \lambda x.e : \tau \multimap \tau'}$$

入力の型環境が与えられたときに出力の型環境が高々一つに定まることに注意する．しかし上記の型システムは，線形 λ 計算の加法的直積型（&）のフラグメントを考えた場合には十分でなく，出力の型環境が入力の型環境から定まらなくなる．それを避けるために IO-$\top$ システムでは，型判断に $\top$（& の単位元）の導入則に対応する型付け規則がその導出木において使用されたかを表すフラグ b を含め，$\Gamma_1 \setminus \Gamma_2 \vdash_b e : \tau$ という形の型判断を扱う．しかし，本稿では & および $\top$ は扱わないため，この詳細は割愛する．

Polakow の埋め込みでは，$\Gamma_1 \setminus \Gamma_2 \vdash_b e : \tau$ なるような項を型クラス *LLC* を用いて表す．直観的には，*LLC repr* は *repr vid b i o t* が上記の項を表す抽象型であることを表している．ここで，*repr* は

$$\begin{array}{l} \mathit{repr} :: \mathit{Nat} \to \mathit{Bool} \to \\ \quad [\mathit{Maybe\ Nat}] \to [\mathit{Maybe\ Nat}] \to \\ \qquad \mathit{Type} \to \mathit{Type} \end{array}$$

という kind を持ち，*vid*，*b*，*i*，*o*，*t* はそれぞれ以下を表している．

- *vid* は現在の de Brujin レベル（すなわち，束縛の深さ）を表す．型レベルの変数名の表現として利用される．
- b は $\Gamma_1 \setminus \Gamma_2 \vdash_b e : \tau$ の b を表す．
- i は入力型環境 Γ_1 を表す．しかし型情報自体は持たない．
- o は出力型環境 Γ_2 を表す．しかし型情報自体は持たない．
- t は項の線形型 τ を表す．

ここで，*Nat* は

$$\mathbf{data}\ \mathit{Nat} = Z \mid S\ \mathit{Nat}$$

で定義される kind（DataKinds 拡張により昇華）である．

LLC は線形 λ 計算の各構文に対応するメソッドを持つ．以下，本稿で使用するもののみに絞り説明する．

2.2.1 *llam* および (^) メソッド

メソッド *llam* は，線形関数抽象 $\lambda x.e$ に対応する．

$$\begin{array}{l} \mathit{llam} :: \\ \quad (\mathit{VarOk}\ b\ \mathit{var}) \Rightarrow \\ \quad (\mathit{LVar}\ \mathit{repr}\ \mathit{vid}\ s \to \\ \qquad \mathit{repr}\ (S\ \mathit{vid})\ b\ (\mathit{Just}\ \mathit{vid} : i)\ (\mathit{var} : o)\ t) \to \\ \quad \mathit{repr}\ \mathit{vid}\ b\ i\ o\ (s \multimap t) \end{array}$$

LVar repr vid a は線形 λ 計算における変数式の型である．

$$\begin{array}{l} \mathbf{type}\ \mathit{LVar}\ \mathit{repr}\ (\mathit{vid} :: \mathit{Nat})\ a = \\ \quad \mathit{forall}\ (v :: \mathit{Nat}) \\ \qquad (i :: [\mathit{Maybe\ Nat}]) \\ \qquad (o :: [\mathit{Maybe\ Nat}]). \\ \quad \mathit{Consume}\ \mathit{vid}\ i\ o \Rightarrow \mathit{repr}\ v\ \mathit{False}\ i\ o\ a \end{array}$$

直観的には，*LVar* 型は以下の規則に対応している．

$$\frac{}{(\Gamma_1, x : \tau, \Gamma_1') \setminus (\Gamma_2, \Box, \Gamma_2') \vdash_{\mathit{False}} x : \tau}$$

ここで，*Consume vid i o* という制約により，i が型レベル変数名 *vid* を含む型環境であり，o はそれを消費して得られるものであることが保証される．$\top$ の扱い立ち入る必要があるため *VarOk* の定義は割愛するが，制約 *VarOk b var* は b が *False* ならば（つまり，$\top$ の導入が行われていない），*var* が *Nothing* とならなければならないことを述べておく．メソッド *llam* は，Haskell の関数により，線形 λ 計算の束縛を表現している．そのため，$\mathit{llam}\ \$\ \lambda x \to x$ のように，Haskell の関数抽象を用いて線形関数を定義可能である．

メソッド (^) は線形関数の適用を表す．

$(\hat{}) :: (Or\ b_1\ b_2\ b) \Rightarrow$
$\quad repr\ vid\ b_1\ i\ h\ (s \multimap t) \rightarrow$
$\quad repr\ vid\ b_2\ h\ o\ s \rightarrow$
$\quad repr\ vid\ b\ i\ o\ t$

ここで，$Or\ b_1\ b_2\ b$ は b が $b_1 \vee b_2$ であることを表す型クラスである．

2.2.2　*ulam* および $$ メソッド

Polakow による埋め込み手法は，非線形な関数とその適用についても表現可能である．

メソッド *ulam* は，非線形関数抽象を表す．

$ulam :: (UVar\ repr\ s \rightarrow repr\ vid\ b\ i\ o\ t) \rightarrow$
$\quad repr\ vid\ b\ i\ o\ (s \rightarrow t)$

ここで，$UVar\ repr\ a$ は以下のように定義される型である．

$\mathbf{type}\ UVar\ repr\ a =$
$\quad forall\ (vid :: Nat)\ (i :: [Maybe\ Nat]).$
$\quad repr\ vid\ False\ i\ i\ a$

$UVar\ repr\ vid\ a$ は線形 λ 計算における非線形な変数式を表す．つまり，以下の規則に対応する．

$$\frac{}{\Delta; \Gamma \setminus \Gamma \vdash_{False} x : \Delta(x)}$$

ここで，Δ は非線形型環境である．

対応して，($$) は非線形関数の適用を表す．[†2]

$(\$\$) :: repr\ vid\ b\ i\ o\ (a \rightarrow b) \rightarrow$
$\quad repr\ vid\ b'\ []\ []\ a \rightarrow$
$\quad repr\ vid\ b\ i\ o\ b$

ここで，$e_1\ \$\$\ e_2$ の e_2 が閉じた式であることが要求されていることに注意する．

2.2.3　*bang* および *letBang* メソッド

線形 λ 計算において，$!s$ は複数回の使用が可能な型を表す．Polakow の埋め込みにおいては，$!s$ は $Bang\ s$ のように表現される．

メソッド *bang* は，*Bang* を導入する．

$bang :: repr\ vid\ b\ []\ []\ s \rightarrow$
$\quad repr\ vid\ False\ i\ i\ (Bang\ s)$

ここで，$repr\ vid\ b\ []\ []\ s$ は直観的には（線形な変数に関して）閉じた式を表している．閉じた式は，外部で導入された変数を含まないため複数回の使用が可能となる．

それに対し，*letBang* は，線形 λ 計算における式 $\mathbf{let}\ !x = e_1\ \mathbf{in}\ e_2$ に対応するメソッドであり，*Bang* を除去する．

$letBang ::$
$\quad (Or\ b_1\ b_2\ b) \Rightarrow$
$\quad repr\ vid\ b_1\ i\ h\ (Bang\ s) \rightarrow$
$\quad (UVar\ repr\ a \rightarrow repr\ vid\ b_2\ h\ o\ t) \rightarrow$
$\quad repr\ vid\ b\ i\ o\ t$

ここでも *llam* と同様，線形 λ 計算の束縛が Haskell の関数により表現されているが，*llam* と異なりここでは *UVar* が使われている．そのため，$letBang\ e\ k$ の k は入力を複数回使用可能である．

2.2.4　*one*，*letOne*，(⊗) および *letStar* メソッド

Polakow の埋め込み手法は，テンソル積型（乗法的直積型）およびその単位元についても表現可能である．

メソッド (⊗) はテンソル積を導入する．

$(\otimes) :: (Or\ b_1\ b_2\ b) \Rightarrow$
$\quad repr\ vid\ b_1\ i\ h\ a \rightarrow$
$\quad repr\ vid\ b_2\ h\ o\ b \rightarrow$
$\quad repr\ vid\ b\ i\ o\ (a \otimes b)$

第一引数の入力型環境が i，出力型環境境が h，第二引数の入力型環境が h，出力型環境が o となっていることに注意する．このことは，第 1 引数と第 2 引数の式の間で変数の重複がないことを表している．

メソッド *letStar* は，線形 λ 計算における式 $\mathbf{let}\ x \otimes y = e_1\ \mathbf{in}\ e_2$ に対応する．ここでも，*llam* や *letBang* の場合と同様に線形 λ 計算における束縛が Haskell における関数として表現されていることに注意する．

†2　Haskell では $repr\ vid\ b'\ []\ []\ a$ のように書くと [] が（型レベルの）空リストでなくリストの型構成子として解釈されてしまうため本来は $repr\ vid\ b'\ '[]\ '[]\ a$ のように記述する必要があるが，本稿では記述の簡便のためこれを無視する．

$$
\begin{aligned}
\mathit{letStar} :: (&\mathit{VarOk}\ b_2\ \mathit{vid}_1,\\
&\mathit{VarOk}\ b_2\ \mathit{vid}_2,\\
&\mathit{Or}\ b_1\ b_2\ b) \Rightarrow\\
&\mathit{repr}\ \mathit{vid}\ b_1\ i\ h\ (a \otimes b) \to\\
&\ (\mathit{LVar}\ \mathit{repr}\ \mathit{vid}\ a \to\\
&\ \ \mathit{LVar}\ \mathit{repr}\ (S\ \mathit{vid})\ b \to\\
&\ \ \mathit{repr}\ (S\ (S\ \mathit{vid}))\ b_2\\
&\quad\quad (\mathit{Just}\ \mathit{vid} : \mathit{Just}\ (S\ \mathit{vid}) : h)\\
&\quad\quad (\mathit{vid}_1 : \mathit{vid}_2 : o)\ c) \to\\
&\mathit{repr}\ \mathit{vid}\ b\ i\ o\ c
\end{aligned}
$$

メソッド one はテンソル積の単位元を表す.

$$\mathit{one} :: \mathit{repr}\ \mathit{vid}\ \mathit{False}\ i\ i\ \mathit{One}$$

メソッド letOne は，線形 λ 計算における式 $\mathbf{let}\ \langle\rangle = e_1\ \mathbf{in}\ e_2$ に対応するメソッドであり，One 型の値を除去する.

$$
\begin{aligned}
\mathit{letOne} :: (&\mathit{Or}\ b_1\ b_2\ b) \Rightarrow\\
&\mathit{repr}\ \mathit{vid}\ b_1\ i\ h\ (\mathit{One}) \to\\
&\mathit{repr}\ \mathit{vid}\ b_2\ h\ o\ a \to\\
&\mathit{repr}\ \mathit{vid}\ b\ i\ o\ a
\end{aligned}
$$

たとえば，a と $a \otimes \mathit{One}$ の間の相互変換は以下のように記述できる.

$$
\begin{aligned}
&\mathit{introOneL} :: \mathit{LLC}\ \mathit{repr} \Rightarrow\\
&\quad\quad \mathit{UVar}\ \mathit{repr}\ (a \multimap a \otimes \mathit{One})\\
&\mathit{introOneL} = \mathit{llam}\ \$\ \lambda a \to a \otimes \mathit{one}\\
&\mathit{elimOneL} :: \mathit{LLC}\ \mathit{repr} \Rightarrow\\
&\quad\quad \mathit{UVar}\ \mathit{repr}\ (a \otimes \mathit{One} \to a)\\
&\mathit{elimOneL} = \mathit{llam}\ \$\ \lambda ao \to \mathit{letStar}\ ao\ \$\ \lambda a\ o \to\\
&\quad\quad\quad\quad \mathit{letOne}\ o\ \$\ a
\end{aligned}
$$

3 Linear Quipper

本節では，Quipper に Polakow の埋め込み技法を適用することにより，埋め込み線形量子プログラミング言語 Linear Quipper を実現する.

3.1 基本的なアイデアとその問題点

基本的なアイデアは，Quipper における関数を線形関数におきかえること，つまり n 入力 n 出力の量子回路を以下のような線形型を用いて表現することである.

$$\underbrace{\mathit{Qubit} \multimap \cdots \multimap \mathit{Qubit}}_{n} \multimap \mathit{Circ}\,\underbrace{(\mathit{Qubit} \otimes \cdots \otimes \mathit{Qubit})}_{n}$$

それに対応し，Quipper の提供する各関数について，元々 $\to$ を用いていた部分を $\multimap$ で置き換え，以下のように型クラス LLC をそれらの関数で拡張した型クラス $\mathit{LinearQuipper}$ を提供する.

$$
\begin{aligned}
&\mathbf{class}\ \mathit{LLC}\ \mathit{repr} \Rightarrow \mathit{LinearQuipper}\ \mathit{repr}\ \mathbf{where}\\
&\quad \ldots
\end{aligned}
$$

たとえば，Quipper の関数 $\mathit{hadamard} :: \mathit{Qubit} \to \mathit{Circ}\ \mathit{Qubit}$ は，以下の $\mathit{LinearQupper}$ のメソッドとして表現される.

$$\mathit{hadamard} :: \mathit{UVar}\ \mathit{repr}\ (\mathit{Qubit} \multimap \mathit{Circ}\ \mathit{Qubit})$$

ここで，UVar は $\mathit{hadamard}$ 自体は何回も使用できることを表している．また，古典ビットを入力とする回路は非線形関数に対応させ，古典ビットを出力とする回路は Bang 型を返す関数に対応させる．たとえば，Quipper の関数 $\mathit{qinit_qubit} :: \mathit{Bool} \to \mathit{Circ}\ \mathit{Qubit}$ と $\mathit{measure_qubit} :: \mathit{Qubit} \to \mathit{Circ}\ \mathit{Bit}$ は以下の $\mathit{LinearQuipper}$ のメソッドとして表現される.

$$
\begin{aligned}
&\mathit{qinit_qubit} :: \mathit{Uvar}\ \mathit{repr}\ (\mathit{Bool} \to \mathit{Circ}\ \mathit{Qubit})\\
&\mathit{measure_qubit} ::\\
&\quad \mathit{UVar}\ \mathit{repr}\ (\mathit{Qubit} \multimap \mathit{Circ}\ (\mathit{Bang}\ \mathit{Bit}))
\end{aligned}
$$

しかし，このアイデアを単純に適用するだけでは以下の理由により不十分である.

- Quipper では，**do** 記法を用いて回路を記述するが，単純に $(\gg\!=) :: m\ a \to (a \to m\ b) \to m\ b$ を，$(\gg\!=) :: \mathit{UVar}\ \mathit{expr}\ (m\ a \multimap (a \multimap m\ b) \multimap m\ b)$ に置き換えてしまうと **do** 記法が使用できなくなる.
- Quipper の一部の関数は単純に線形型を与えることが難しい．たとえば，前節の cnot の例が示すように，$\mathit{controlled}$ 関数は第 2 引数の量子ビットを消費しない.
- $(\otimes)$ や Bang 型は回路の記述で頻繁に使用されるが，LLC のメソッドをそのまま利用したのでは，$\mathit{letBang}$ や $\mathit{letStar}$ を用いて陽に Bang や $(\otimes)$ を一つ一つ分解しなければならない．これにより，ネストされたパターンを使用している Quipper のプログラムに対し，対応する Linear Quipper

のプログラムは大きく形が異なったものになってしまう．

以下では，これらのそれぞれの問題点についての解決法を議論する．

3.2 モナドのメソッドの表現

前述の通り，我々は Linear Quipper においても **do** 記法を利用したい．一つの方法は GHC 拡張の RebindableSyntax 使うことであるが，そのためには *return* および $(\gg=)$ は

$(\gg=) :: ??? \to (??? \to ???) \to ???$

$\mathit{return} :: ??? \to ???$

という形の型を持たなければならない．このことは，たとえば $\mathbf{do}\ \{p \leftarrow e_1; \mathit{return}\ \ e_2\}$ という **do** 式が $e_1 \gg= \lambda p \to \mathit{return}\ \ e_2$ といった形の式に翻訳されるためである．しかし，$(\gg=)$ を素朴に $(\gg=) :: \mathit{UVar}\ \ \mathit{expr}\ \ (m\ \ a \multimap (a \multimap m\ \ b) \multimap m\ \ b)$ と表現してしまうと，型が上記の形でないため RebindableSyntax を活用できない．

3.2.1 $(\gg=)$ メソッド

基本的なアイデアは，$(\gg=)$ を以下の型付け規則を持つ言語要素として扱うことである．

$$\frac{\Gamma_1 \vdash e_1 : \mathit{Circ}\ \tau \quad \Gamma_2, x : \tau \vdash e_2 : \mathit{Circ}\ \tau'}{\Gamma_1 \uplus \Gamma_2 \vdash e_1 \gg= (\lambda x.e_2) : \mathit{Circ}\ \tau'}$$

この構文要素は *llam* や (^) の類推から以下の型を与えることができる．

$$\begin{array}{l}(\gg=) :: \\ \quad (\mathit{Or}\ b_1\ b_2\ b, \mathit{VarOk}\ b_2\ \mathit{var}) \Rightarrow \\ \quad \mathit{repr}\ \mathit{vid}\ b_1\ i\ h\ (\mathit{Circ}\ s) \to \\ \quad (\mathit{LVar}\ \mathit{repr}\ \mathit{vid}\ a \to \\ \quad\quad \mathit{repr}\ (S\ \mathit{vid})\ b_2\ (\mathit{Just}\ \mathit{vid} : h) \\ \quad\quad\quad\quad (\mathit{var} : o)\ (\mathit{Circ}\ s)) \to \\ \quad \mathit{repr}\ \mathit{vid}\ b\ i\ o\ (\mathit{Circ}\ t)\end{array}$$

上記において，束縛の扱いは *llam* と同様であり，環境や ⊤ フラグの扱いは (^) と同様である．

3.2.2 *return* メソッド

同様のアイデアにより，*return* もより簡潔に表現することができる．具体的には，*return* を以下の型付け規則を持つ言語要素として考える．

$$\frac{\Gamma_1 \vdash e : \tau}{\Gamma \vdash \mathit{return}\ e : \mathit{Circ}\ \tau}$$

対応して，我々は *return* に以下の型を与えることができる．

$$\begin{array}{l}\mathit{return} :: \mathit{repr}\ \mathit{vid}\ b\ i\ o\ a \to \\ \quad\quad \mathit{repr}\ \mathit{vid}\ b\ i\ o\ (\mathit{Circ}\ a)\end{array}$$

これらのメソッドより，Linear Quipper においても **do** 記法を利用して量子回路をプログラムすることが可能である．たとえば，入力に *qnot* を二度適用する回路は以下のように記述可能である．

$$\begin{array}{l}\mathit{qnotnot} :: \\ \quad \mathit{LinearQuipper}\ \mathit{repr} \Rightarrow \\ \quad \mathit{UVar}\ \mathit{repr}\ (\mathit{Qubit} \multimap \mathit{Circ}\ \mathit{Qubit}) \\ \mathit{qnotnot} = \mathit{llam}\ \$\ \lambda a \to \mathbf{do} \\ \quad a' \leftarrow \mathit{qnot}\ a \\ \quad a'' \leftarrow \mathit{qnot}\ a' \\ \quad \mathit{return}\ a''\end{array}$$

ここで，上記で定義した $(\gg=)$ および *return* がスコープ内に存在していることを仮定している．

3.2.3 $(\gg)$ メソッド

Quipper では，$\mathit{qterm_qubit} :: \mathit{Qubit} \to \mathit{Bool} \to \mathit{Circ}\ ()$ を用いて $\mathit{qterm_qubit}\ q\ b$ と書くことで，量子ビット q の値が b となっていることをアサートし，量子ビットを破棄することが可能である．これは，q が出力 ancilla ビットになっていることを意味する．しかし，上記の $\gg=$ を使ったのでは $\mathbf{do}\ \{\mathit{qterm_qubit}\ e_1\ b; e_2\}$ といったコードを排除してしまう．これは上記のコードが $\mathit{qterm_qubit}\ e_1\ b \gg= \lambda_ \to e_2$ へと変換されるためである．ここで，$(\gg=)$ の第二引数は，線形関数でなければならないが $\lambda_ \to e_2$ は入力を破棄しているため，型チェックが通らない．

RebindableSyntax 拡張は，スコープ内に $(\gg)$ が定義されていれば，$\mathbf{do}\ \{e_1; e_2\}$ を $e_1 \gg e_2$ のように $(\gg)$ を用いて翻訳する．そのため，我々は以下のように定義される $(\gg)$ を提供している．

$$\begin{array}{l}(\gg) :: \\ \quad (\mathit{Or}\ b_1\ b_2\ b, \mathit{VarOk}\ b_2\ \mathit{var}) \Rightarrow \\ \quad \mathit{repr}\ \mathit{vid}\ b_1\ i\ h\ (\mathit{Circ}\ \mathit{One}) \to \\ \quad \mathit{repr}\ (S\ \mathit{vid})\ b_2\end{array}$$

$$(\mathit{Nothing} : h)\ (\mathit{var} : o)\ (\mathit{Circ}\ e_2) \rightarrow$$
$$\mathit{repr}\ \mathit{vid}\ b\ i\ o\ (\mathit{Circ}\ e_2)$$
$$(\gg)\ e_1\ e_2 = e_1 \ggg \lambda x \rightarrow \mathit{letOne}\ x\ e_2$$

3.3 controlled 関数

Quipper は $\mathit{controlled} :: \mathit{ControllSource}\ \ c\ \Rightarrow \mathit{Circ}\ a \rightarrow c \rightarrow \mathit{Circ}\ a$ 関数を提供している．ここで c は量子ビットか古典ビットかブール値からなる組であり，第 2.1 節の *cnot* や *ccnot* の例が示すように，量子ビットを使用した場合でもそれを消費しない．

素朴には，*controlled* には $\mathit{ControlSource}\ \ c\ \Rightarrow \mathit{Circ}\ a \multimap c \multimap \mathit{Circ}\ (a \otimes c)$ と a と c の組を返す関数として実現可能である．しかしながら，制御つきゲートは可逆回路において最も基本的な回路の一つであり，量子回路においても頻繁に利用されるため，この変換はプログラミングを煩雑にしてしまう．

関数 *controlled* に適切な線形型を与えることは容易ではない．たとえば，以下の型付け規則は，e_2 がリソースを消費しないことを表しているはいるものの，e_2 の型検査に現在の型環境とはまったく関係のない型環境を使用しているため正しくない．

$$\frac{\Gamma_1 \vdash e_1 : \mathit{Circ}\ \tau \quad \Gamma_2 \vdash e_2 : \text{control bit の型}}{\Gamma_1 \vdash e_1\ `\mathit{controlled}`\ e_2 : \mathit{Circ}\ \tau}$$

幸運にも *LLC* では変数の使用情報と型情報が別々に管理されている（前者は入出力型環境，後者は Haskell が管理している）ため，e_2 がリソースを消費しないことと，現在の型環境と同じ型として使用されることの両方を同時に表現することが可能である．具体的には，*controlled* メソッドは以下の型を持つ．

$$\begin{aligned}\mathit{controlled} &:: \mathit{ControlSource}\ c \Rightarrow \\ &\mathit{repr}\ \mathit{vid}\ b\ i\ h\ (\mathit{Circ}\ a) \\ &\rightarrow \mathit{repr}\ \mathit{vid}\ b\ h\ o\ c \\ &\rightarrow \mathit{repr}\ \mathit{vid}\ b\ i\ h\ (\mathit{Circ}\ a)\end{aligned}$$

制約 $\mathit{ControlSource}\ c$ は c が，古典/量子ビットあるいはブール値のテンソル積からなることを表している．ここで，*controlled* の返り値の入力型環境と出力型環境と，その第一引数の型環境が一致していることに注意する．また，$\otimes$ の型付け規則のように，第一引数の出力型環境は第二引数の入力型環境となっている．そのため，上の型により，*controlled* の第二引数に使われた量子ビットを消費しないことと，第一引数と第二引数に同一の量子ビットを使用しないことの二点が保証される．

なお，*ControlSource* は Quipper の提供する同名の型クラスと同一である．この型クラスが Linear Quipper においても再利用可能なのは，$a \otimes b$ が (a, b) の型シノニムとして定義されていることによる．この $a \otimes b$ と (a, b) の同一視は安全なである．なぜなら，型 *repr* が抽象化されている限りにおいて $a \otimes b$ 型の値は Linear Quipper のメソッドを通してしか操作できないためである．

例 2 (*cnot*, *ccnot*)**.** 第 2.1 節の *cnot* と *ccnot* は Linear Quipper でもほぼ同等の形で記述することができる．

$$\begin{aligned}&\mathit{cnot} :: \mathit{LinearQuipper}\ \mathit{repr} \Rightarrow \\ &\quad \mathit{UVar}\ \mathit{repr}\ (\mathit{Qubit} \multimap \mathit{Qubit} \multimap \\ &\qquad\qquad \mathit{Circ}\ (\mathit{Qubit} \otimes \mathit{Qubit})) \\ &\mathit{cnot} = \mathit{llam}\ \$\ \lambda a \rightarrow \mathit{llam}\ \$\ \lambda b \rightarrow \mathbf{do} \\ &\quad a' \leftarrow \mathit{qnot}\ a\ `\mathit{controlled}`\ b \\ &\quad \mathit{return}\ (a' \otimes b)\end{aligned}$$

$$\begin{aligned}&\mathit{ccnot} :: \mathit{LinearQuipper}\ \mathit{repr} \Rightarrow \\ &\quad \mathit{UVar}\ \mathit{repr}\ (\mathit{Qubit} \multimap \mathit{Qubit} \multimap \mathit{Qubit} \multimap \\ &\qquad\qquad \mathit{Circ}\ (\mathit{Qubit} \otimes \mathit{Qubit} \otimes \mathit{Qubit})) \\ &\mathit{ccnot} = \mathit{llam}\ \$\ \lambda a \rightarrow \\ &\qquad\qquad \mathit{llam}\ \$\ \lambda b \rightarrow \mathit{llam}\ \$\ \lambda c \rightarrow \mathbf{do} \\ &\quad a' \leftarrow \mathit{qnot}\ a\ `\mathit{controlled}`\ (b \otimes c) \\ &\quad \mathit{return}\ (a' \otimes b \otimes c)\end{aligned}$$

3.4 パターンマッチングの表現

Linear Quipper のコアなメソッドのみでは，ネストしたテンソル積を扱うのは煩雑である．たとえば，以下は Quipper で NAND 回路を記述したプログラムである．可逆性のため，返り値の組の第 2 要素と第 3 要素はゴミ出力となっている．

$$
\begin{aligned}
&nand_{\mathrm{orig}}\ a\ b = \mathbf{do}\\
&\quad c \leftarrow qinit\ False \quad \text{-- ancilla bit}\\
&\quad (c,a,b) \leftarrow ccnot\ c\ a\ b\\
&\quad c \leftarrow qnot\ c\\
&\quad return\ (c,a,b)
\end{aligned}
$$

同等のプログラムを Linear Quipper で記述すると以下となる．

$$
\begin{aligned}
&nand_{\mathrm{lin}} = llam\ \$\ \lambda a \to llam\ \$\ \lambda b \to \mathbf{do}\\
&\quad c \leftarrow qinit_qubit\ \$\$\ false\\
&\quad cab \leftarrow ccnot\ \hat{}\ c\ \hat{}\ a\ \hat{}\ b\\
&\quad letStar\ cab\ \$\ \lambda c\ ab \to\\
&\quad\quad letStar\ ab\ \$\ \lambda a\ b \to \mathbf{do}\\
&\quad\quad\quad c \leftarrow qnot\ \hat{}\ c\\
&\quad\quad\quad return\ (c \otimes a \otimes b)
\end{aligned}
$$

ここで，Linear Quipper では $ccnot$ の出力は $(Qubit \otimes Qubit) \otimes Qubit$ であるため，それを分解するのに $letStar$ のネストが必要となっている．一般に，複数個の Qubit を返す関数を利用する際に，その個数だけ $letStar$ のネストが必要になってしまう．

Haskell に詳しい読者は PatternSynonym を用いれば問題がないように思うかもしれないが，これも難しい．なぜならば，LLC では $repr \ldots (a \otimes b) \multimap (repr \ldots a, repr \ldots b)$ という関数を定義することが難しいためである．テンソル積と違い組についてはその要素の使用に制限がないことに注意する．

そこで，我々は un-unparsing コンビネータ [2] に倣い，パターンを継続渡し形式で表現することにより，合成可能なパターンを実現する．より具体的には，パターンは，そのマッチングによって束縛される自由変数の値を引き数にとる継続を入力にとる関数として表現される．

もっとも基本形なパターンは変数パターンである．これは以下の関数として表される．

$$
\begin{aligned}
&pvar ::\\
&\quad (Or\ b_1\ b_2\ b, VarOk\ b_1\ v) \Rightarrow\\
&\quad repr\ vid\ b_2\ h\ o\ a \to\\
&\quad (LVar\ repr\ vid\ a \to\\
&\quad\quad repr\ (S\ vid)\ b_1\ (Just\ vid : i)\ (var : h)\ b) \to\\
&\quad repr\ vid\ b\ i\ o\ b\\
&pvar\ a\ k = llam\ k\ \hat{}\ a
\end{aligned}
$$

たとえば，$pvar\ e_1\ \$\ \lambda x \to e_2$ と書くことにより，$\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ に対応するプログラムを表現することができる．ここで，$pvar$ の型は，$(\gg\!=)$ の型から $Circ$ を除いたものであることに注意する．

テンソル積パターン $p_1\ \texttt{@*}\ p_2$ は，$letStar$ を用いて入力を分解し，それを部分パターン p_1 と p_2 に渡し，その結果を組として継続に渡す．

$$
\begin{aligned}
&(\texttt{@*}) :: (VarOk\ b_2\ var_1, VarOk\ b_2\ var_2,\\
&\quad Or\ b_1\ b_2\ b,\\
&\quad Consume\ vid\ i_1\ i_2,\\
&\quad Consume\ (S\ vid)\ i_2\ o_2) \Rightarrow\\
&\quad (repr\ v_1\ False\ i_1\ o_1\ a_1 \to\\
&\quad\quad (a_2 \to t_1) \to\\
&\quad\quad repr\ (S\ (S\ vid))\ b_2\\
&\quad\quad (Just\ vid : Just\ (S\ vid) : h)\\
&\quad\quad (var_1 : var_2 : o_3)\ c) \to\\
&\quad (repr\ v_2\ False\ i_2\ o_2\ b_1 \to\\
&\quad\quad (b_2 \to t_2) \to t_1) \to\\
&\quad repr\ vid\ b_1\ i_3\ h\ (a_1 \otimes b_1) \to\\
&\quad ((a_2, b_2) \to t_2) \to\\
&\quad repr\ vid\ b\ i_3\ o_3\ c\\
&p_1\ \texttt{@*}\ p_2 = \lambda z\ k \to letStar\ z\ \$\ \lambda a\ b \to\\
&\quad p_1\ a\ (\lambda x \to p_2\ b\ (\lambda y \to k\ (x, y)))
\end{aligned}
$$

たとえば，$(pvar\ \texttt{@*}\ pvar)\ e_1\ \$\ \lambda(x,y) \to e_2$ と書くことにより，$\mathbf{let}\ (x \otimes y) = e_1\ \mathbf{in}\ e_2$ に対応するプログラムを表現することができる．

また，パターンと束縛する変数を近い場所に書けるようにするために，我々は以下の関数も提供している．

$$
\begin{aligned}
&letQ :: t_1 \to (t_1 \to t_2 \to t_3) \to t_2 \to t_3\\
&letQ\ e\ p\ k = p\ e\ k
\end{aligned}
$$

たとえば，これらの関数により $nand_{\mathrm{lin}}$ を以下のよ

うに記述することが可能となる．

$$
\begin{aligned}
&\mathit{nand}_{\mathrm{lin}} = \mathit{llam}\ \$\ \lambda a \to \mathit{llam}\ \$\ \lambda b \to \mathbf{do}\\
&\quad c \leftarrow \mathit{qinit}\ \$\$\ \mathit{false}\\
&\quad \mathit{cab} \leftarrow \mathit{ccnot}\ \hat{}\ c\ \hat{}\ a\ \hat{}\ b\\
&\quad \mathit{letQ}\ \mathit{cab}\\
&\qquad\quad (\mathit{pvar}\ \texttt{@*}\ \mathit{pvar}\ \texttt{@*}\ \mathit{pvar})\ \$\\
&\qquad\qquad \lambda((c, a), b) \to \mathbf{do}\\
&\qquad c \leftarrow \mathit{qnot}\ \hat{}\ c\\
&\qquad \mathit{return}\ (c \otimes a \otimes b)
\end{aligned}
$$

ここで，*pvar* や (`@*`) を用いることによって *letStar* のネストの煩雑さを抑られていることに注意する．

こうした継続を用いたパターンの表現は，非線形変数を導入する *letBang* にも適用可能である．しかし，線形な変数を導入するパターンと非線形変数を導入するパターンを組み合わせることは単純ではなく，その検討は今後の課題とする．

3.5 Linear Quipper の項から Quipper の項への変換

Linear Quipper は型クラスで Quipper の構文を表現しているため，適切なインスタンスを与えることによりその項を Quipper の項に変換することが可能である．我々は，そのアイデアに従い以下の変換関数を提供している．

$$
\begin{aligned}
&\mathit{convert} :: \mathit{Corr}\ a\ r \Rightarrow\\
&\quad (\mathit{forall}\ \mathit{repr}.\ \mathit{LinearQuipper}\ \mathit{repr} \Rightarrow\\
&\qquad\qquad \mathit{repr}\ \mathit{vid}\ b\ i\ o\ a) \to r
\end{aligned}
$$

ここで，*Corr* は以下のように定義される型クラスであり，線形型システムにおける型構成子と通常の型構成子との対応を表現している．

$$
\begin{aligned}
&\mathbf{class}\ \mathit{Corr}\ a\ b \mid a \to b\ \mathbf{where}\\
&\quad \mathit{to} :: a \to b\\
&\quad \mathit{from} :: b \to a
\end{aligned}
$$

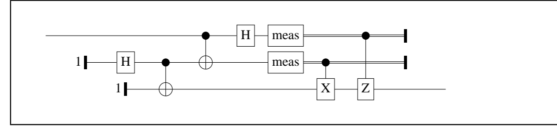

図 1　量子テレポーテーションを行う回路

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Corr}\ \mathit{Qubit}\ \mathit{Qubit}\ \mathbf{where} \ldots\\
&\ldots\\
&\mathbf{instance}\ \mathit{Corr}\ \mathit{One}\ ()\ \mathbf{where} \ldots\\
&\mathbf{instance}\ (\mathit{Corr}\ a\ a', \mathit{Corr}\ b\ b') \Rightarrow\\
&\qquad \mathit{Corr}\ (a \multimap b)\ (a' \to b')\ \mathbf{where} \ldots\\
&\mathbf{instance}\ (\mathit{Corr}\ a\ a', \mathit{Corr}\ b\ b') \Rightarrow\\
&\qquad \mathit{Corr}\ (a \otimes b)\ (a', b')\ \mathbf{where} \ldots\\
&\mathbf{instance}\ (\mathit{Corr}\ a\ a')\\
&\qquad \Rightarrow \mathit{Corr}\ (\mathit{Bang}\ a)\ a'\ \mathbf{where} \ldots.
\end{aligned}
$$

4 評価

本節では，量子テレポーテーションを行う回路（図 1）を Quipper および Linear Quipper の両者で記述することにより，Linear Quipper の記述性を評価する．Quipper における，量子テレポーテーンションのプログラムは[4] のものを用いた．

図 2 および図 4 に Quipper における量子テレポーテーションの実装を，図 3 および図 5 に Linear Quipper における量子テレポーテーションの実装を示す．図 2 と図 3 を比較すると，比較的単純な補助関数については，Quipper と Linear Quipper のプログラムの差は小さい．一方で，図 4 と図 5 を比較すると，テンソル積や ! 型の分解を伴う *teleport* 関数については，Quipper と Linear Quipper の差は大きいことがわかる．これは，3.4 節で述べたように，Haskell レベルのパターンマッチングを用いてテンソル積や ! 型の分解を行えないことが原因である．特に *teleport* 関数においては，*bell00* も *alice* についても組を返すので，その差が顕著となる．

通常の線形でないプログラムに対しては，図 5 のようなプログラムも，継続モナドを用いることで **do** 記法を用いたプログラムとして表現することが可能な場合がある．同様のアプローチが Linear Quipper に対しても可能か否かの検討は今後の課題である．

```
plus_minus :: Bool → Circ Qubit
plus_minus b =
  q ← qinit b
  hadamard q

share :: Qubit → Circ (Qubit, Qubit)
share a = do
  b ← qinit False
  b ← qnot b ‘controlled‘ a
  return (a, b)

bell00 :: Circ (Qubit, Qubit)
bell00 = do
  a ← plus_minus False
  share a

alice :: Qubit → Qubit → Circ (Bit, Bit)
alice q a = do
  a ← qnot a ‘controlled‘ q
  q ← hadamard q
  (x, y) ← measure (q, a)
  return (x, y)

bob :: Qubit → (Bit, Bit) → Circ Qubit
bob b (x, y) = do
  b ← gate_X b ‘controlled‘ y
  b ← gate_Z b ‘controlled‘ x
  cdiscard (x, y)
  return b
```

図 2　Quipper における量子テレポーテーションの実装（補助関数）

```
plus_minus :: (LLCQuipper repr) ⇒
  UVar repr (Bool → Circ Qubit)
plus_minus = ulam $ λb → do
  q ← qinit_qubit $$ b
  hadamard ˆ q

share :: LLCQuipper repr ⇒
  UVar repr (Qubit ⊸ Circ (Qubit ⊗ Qubit))
share = llam $ λa → do
  b ← qinit_qubit $$ false
  b ← qnot ˆ b ‘controlled‘ a
  return (a ⊗ b)

bell00 :: LLCQuipper repr ⇒
  UVar repr (Circ (Qubit ⊗ Qubit))
bell00 = do
  a ← plus_minus $$ false
  share ˆ a

alice :: LLCQuipper repr ⇒
  UVar repr (Qubit ⊸ Qubit ⊸
    Circ (Bang Bit ⊗ Bang Bit))
alice = llam $ λq → llam $ λa → do
  a ← qnot ˆ a ‘controlled‘ q
  q ← hadamard ˆ q
  x ← measure_qubit ˆ q
  y ← measure_qubit ˆ a
  return (x ⊗ y)

bob :: LLCQuipper repr ⇒
  UVar repr (Qubit ⊸ Bit → Bit → Circ Qubit)
bob = llam $ λb →
  ulam $ λx → ulam $ λy → do
  b ← gate_X ˆ b ‘controlled‘ y
  b ← gate_Z ˆ b ‘controlled‘ x
  cdiscard_bit $$ x
  cdiscard_bit $$ y
  return b
```

図 3　Linear Quipper における量子テレポーテーションの実装（補助関数）

5　関連研究

量子プログラミング言語の基礎理論としては量子 λ 計算 [11] がある．これは λ 計算を基に，量子回路を関数として表現する方法を提案している．また，型付き量子 λ 計算 [9] もまた提案されている．これは線形論理を基にし，静的に量子ビットの複製や破棄を防い

```
teleport :: Qubit → Circ Qubit
teleport q = do
  (a, b) ← bell00
  (x, y) ← alice q a
  bob b (x, y)
```

図 4　Quipper における *teleport* 関数

```
teleport :: LLCQuipper repr ⇒
  UVar repr (Qubit ⊸ Circ Qubit)
teleport = llam $ λq →
  letQM bell00 (pvar @* pvar) $ λ(a, b) →
  letQM (alice ^ q ^ a) (pvar @* pvar) $ λ(x, y) →
  letBang x $ λx →
  letBang y $ λy → bob ^ b $$ x $$ y

letQM m p k = m >>= λa → letQ a p k
```

図 5　Linear Quipper における *teleport* 関数

でいる．

実装としては，命令型と関数型それぞれに様々な量子プログラミング言語が提案されている．命令型量子プログラミング言語としては QCL [6] を始め，最近では Q# [10] が提案されている．関数型プログラミング言語では QPL [8] や，QML [1]，Quipper 等がある．QPL はこれらの中で最初に提案され，他の関数型量子プログラミング言語に影響を与えている．QML は Haskell に似た文法で量子回路を記述でき，線形型システムを備えている．Quipper は Haskell の埋め込み言語であり，Haskell の関数として量子回路を記述する．

Quipper の基本概念については，*"Thus, the basic abstraction offered by Quipper is that a quantum operation is a function that inputs some quantum data, performs state changes on it, and then outputs the changed quantum data."* [5] と述べられている．よって本研究ではそのポリシーにしたがって Quipper の各関数に線形型を与えた．しかしながら Quipper では，*Qubit* 型は実際にはワイヤを表しており，たとえば以下のように命令的な記述も許容される．

```
cnot :: Qubit → Qubit → Circ Qubit
cnot a c = do
  qnot a `controlled` c
  return a
```

Linear Quipper の実装方針では，*Qubit* は量子ビットそのものを表現しているため，このような命令的な記述を扱うには大きな変更が必要となる．

Haskell に線形型を取り入れる技法としては，GHC の LinearTypes 拡張 [3] の実装が行われている．LinearTypes 拡張を用いると，Polakow の埋め込み技法 [7] とは異なり，Haskell の通常の構文を用いて，線型型プログラムを記述できるようになる．例えば，(^) や ($$) といったメソッドが不要となり，Linear Quipper では煩雑になってしてしまうテンソル積の分解も，パターンマッチを用いて簡潔に記述できるようになると予想される．また，関数の線形型に関する多相型を利用でき，例えば量子ビットと古典ビットに関する一部の処理を別々に定義する必要がなくなると予想される．しかし LinearTypes 拡張付きの Haskell を含む通常の線型型付きプログラミング言語においては，一部の Quipper の関数を直接表現するのは難しい．例えば，Quipper の関数 *controlled* は，Linear Quipper のように制御ビットを消費しないことを制約する型を与えるのは困難であり，$controlled :: ControlSource\ c \Rightarrow Qubit \multimap c \multimap Circ\ (Qubit \otimes c)$ のような型を持つ関数として実装する必要があるだろう．また，現時点においては LinearTypes 拡張は GHC に取り入れられる予定はない．それに対して Linear Quipper は現在の GHC（バージョン 8.2.2）で利用可能である．

6　結論

本研究の成果は，埋め込み線形型付き量子プログラミング言語 Linear Quipper を提案したことである．Linear Quipper では量子ビットの複製，破棄を許容しないため，より安全な量子回路を設計することができる．またその中で，単純に線形型をつけることができない Quipper のメソッドのための LLC の応用や，記述性の向上のための RebindableSyntax 拡張を用いた **do** 構文の利用手法，継続渡し形式を用いた (⊗)

に対するパターンマッチングの提案を行った．

現時点では，Linear Quipper は Quipper の全ての関数を備えているわけではない．それらの実装が今後の課題である．

参考文献

[1] Altenkirch, T. and Grattage, J.: A Functional Quantum Programming Language, *20th IEEE Symposium on Logic in Computer Science (LICS 2005), 26-29 June 2005, Chicago, IL, USA, Proceedings*, IEEE Computer Society, 2005, pp. 249–258.

[2] Asai, K., Kiselyov, O., and Shan, C.: Functional un|unparsing, *Higher-Order and Symbolic Computation*, Vol. 24, No. 4(2011), pp. 311–340.

[3] Bernardy, J., Boespflug, M., Newton, R. R., Peyton Jones, S., and Spiwack, A.: Linear Haskell: practical linearity in a higher-order polymorphic language, *PACMPL*, Vol. 2, No. POPL(2018), pp. 5:1–5:29.

[4] Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., and Valiron, B.: An Introduction to Quantum Programming in Quipper, *Reversible Computation - 5th International Conference, RC 2013, Victoria, BC, Canada, July 4-5, 2013. Proceedings*, Dueck, G. W. and Miller, D. M.(eds.), Lecture Notes in Computer Science, Vol. 7948, Springer, 2013, pp. 110–124.

[5] Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., and Valiron, B.: Quipper: a scalable quantum programming language, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Boehm, H. and Flanagan, C.(eds.), ACM, 2013, pp. 333–342.

[6] Ömer, B.: A Procedural Formalism for Quantum Computing, Master's thesis, Department of Theoretical Physics Technical University of Vienna, 1998.

[7] Polakow, J.: Embedding a full linear Lambda calculus in Haskell, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Lippmeier, B.(ed.), ACM, 2015, pp. 177–188.

[8] Selinger, P.: Towards a quantum programming language, *Mathematical Structures in Computer Science*, Vol. 14, No. 4(2004), pp. 527–586.

[9] Selinger, P. and Valiron, B.: A lambda calculus for quantum computation with classical control, *Mathematical Structures in Computer Science*, Vol. 16, No. 3(2006), pp. 527–552.

[10] Svore, K. M., Geller, A., Troyer, M., Azariah, J., Granade, C. E., Heim, B., Kliuchnikov, V., Mykhailova, M., Paz, A., and Roetteler, M.: Q#: Enabling scalable quantum computing and development with a high-level domain-specific language, *CoRR*, Vol. abs/1803.00652(2018).

[11] van Tonder, A.: A Lambda Calculus for Quantum Computation, *SIAM J. Comput.*, Vol. 33, No. 5(2004), pp. 1109–1135.