量子プログラミング言語

蓮尾 一郎（東京大学大学院情報理工学系研究科）
星野 直彦（京都大学数理解析研究所）

量子プログラミング言語とは——研究の動機

量子計算の分野においてアルゴリズムを記述するためには, 図 1 のような量子回路が用いるのが一般的だろう．一方で，（量子でない）古典アルゴリズムを回路で表現することはあまりなく**擬似コード**とよばれるプログラムもどきが用いられる．この一点においてすでに「量子計算のためのプログラミング言語とはどのようなものか？」という疑問が自然に浮かびあがるのであり，筆者らはこの疑問に答えるべく（おもに数学的なアプローチから）研究を行っている．しかし以下ではさらにもう少し，「量子プログラミング言語の実現によって何が可能となるのか？」ということについて論じたい．

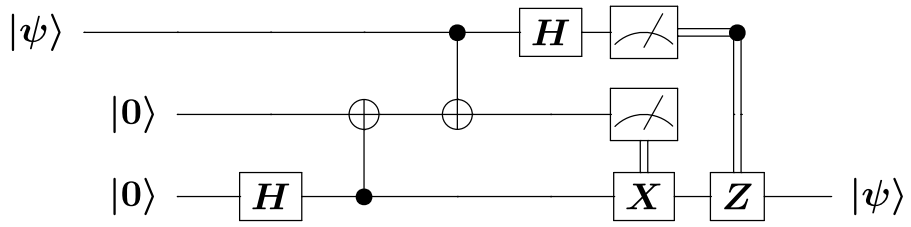

図 1： 量子回路の例（量子テレポーテーション）

論理的記述と物理的実装の分離

まず一つの利点として,「高レベルの」アルゴリズムの記述と「低レベルの」物理的実装との分離があげられる．（古典計算のための）高級プログラミング言語においては，高レベルのプログラムがコンパイラによって低レベルの機械語コードに変換されたのち実行されるため，プログラマーは実行環境のアーキテクチャを意識する必要はない．さらにコンパイラによって数々の最適化が自動的に施される. このシナリオを量子計算に移して考えると，（高級）量子プログラミング言語を用いることにより次のような利点が期待される．

- 量子回路・測定ベース量子計算・位相量子計算などの，量子計算の実現方法に依存しないかたちでアルゴリズムを記述できる．

- **論理量子ビット**と**物理量子ビット**を分離することにより，量子デコヒーレンスに対処するための膨大な数の誤り訂正をアルゴリズムの記述から隠ぺいできる．
- 量子ビットの再利用などの最適化をコンパイラが自動的に行える．

意味論と仕様検証

次の利点として，量子アルゴリズムに対する**仕様検証**を挙げる．ここで仕様とは「アルゴリズムがみたすべき性質」のことを言い，たとえば「量子テレポーテーションによって Bob が受け取る量子ビットは Alice がはじめに持っていたそれと同じである」というような性質である．アルゴリズムと仕様の 2 つを入力として，「みたす（ことが証明できた）」「みたさない（反例がみつかった）」を出力とする問題を仕様検証とよぶのであるが，古典計算に対してはこの仕様検証を行うための手法が盛んに研究され，理論計算機科学において**検証 verification** とよばれる一大分野をなしている．

仕様検証においてはアルゴリズムが仕様をみたすことを**数学的に証明**することを目標とするから，その第一歩としてアルゴリズムのふるまいを数学的に定義してやる必要がある．この後者の営みを**意味論**とよぶ（「このアルゴリズムの『意味』とは何か？」を論じる．筆者らはもともとこの分野の研究者である）．意味論における数学的議論を簡潔にするには，アルゴリズムの構造に関する情報を最大限活用してやることがカギであり，高級プログラミング言語による構造化プログラミングがもたらす恩恵は非常に大きい．

ここまでで「そもそも量子計算において仕様検証は必要なのか？」とお思いの読者もいらっしゃるかもしれない．実は，以上に述べたことは量子計算を**量子通信**に取り替えてもそのまま真であるが，量子鍵配信プロトコルのような量子通信の手順が鍵の秘匿性などの仕様をみたすかどうかは全く自明でない問題である．たとえばすでに古典通信プロトコルにおいても，1978 年に提案された Needham–Schroeder 公開鍵プロトコルの誤りが 1995 年に Lowe によってはじめて発見された．この「事件」によって理論計算機科学の業界では（古典）暗号プロトコルの検証ブームが巻き起こったが，量子通信プロトコルにおいてもこのような誤りが見過ごされている可能性がある．（実際，近年の多くの量子通信プロトコルは正しさの数学的証明付きで発表される）

また，「正しさを数学的に証明する必要があるのか？ テストでもいいのでは？」とお思いの方もいらっしゃるかもしれない．事実，仕様検証には大きなコストがかかるゆえ―

—手動検証には専門家が数週から数ヶ月従事する必要があり，ソフトウェアによる自動検証もスケーラビリティが限られている——現在のソフトウェア品質向上手段としては，いくつかの入力に対して出力をチェックする**テスト**が普通である．しかしテストは明らかに，テスト入力以外の入力に対しての正しさについては状況証拠しか与えてくれない．さらに量子計算においてはアルゴリズムのふるまいが確率的であるため，テストを膨大な回数くり返して統計的に成否を判断する必要がある．今後期待される初期の量子コンピュータの運用コストを考えると，このようなテストは非現実的であろう．

さらに量子プログラミング言語のもう一つの利点として，高級プログラミング言語によるプログラムは人間がアルゴリズムについて考える際の思考過程に近く，新たな量子アルゴリズムの発見につながるかもしれない，ということを挙げておこう．（ただし「慣れてしまえば同じ」ということもあるだろうから，この利点については議論の余地がある）

量子プログラミング言語の 3 つの潮流

量子プログラミング言語の研究は2000年前後に始まった比較的新しいトピックである．量子コンピュータの実用化がまだである以上，古典計算に対するプログラミング言語の研究ほどのにぎわいは期待できない一方，多くの意味論の研究者の興味をひきつけ，主に数学的なアプローチから盛んに研究が行われている．ここ 2〜3 年の動向においては，提案されているいくつかの量子プログラミング言語は次の 3 つの流れに収れんしつつあるようだ．

- Ömer による QCL を代表とする**命令型量子プログラミング言語**
- Altenkirch らによる量子 I/O モナドや Green らによる Quipper などの**量子操作を副作用とする古典関数型言語**
- van Tonder や Selinger, Valiron らによって研究される**量子ラムダ計算**

アプローチとしては，古典計算に関するプログラミング言語の数学的抽象化・一般化を行い，そこから量子プログラミング言語を「数学的にカノニカルな形で」導こうというのが多くの研究者の基本戦略である．関数型プログラミングのスタイルが多いのも（3 つのうち後者 2 つがそうである），その数学的クリーンさによるところが大きい．

上の 3 つの研究の流れをそれぞれ紹介する前に，想定される計算モデルについて述べて

おく.

Knill の QRAM モデル——または「量子データ・古典制御」

これまでに提案された量子プログラミング言語のほとんどが, Knill による **QRAM モデル**における実行を想定している．これは大ざっぱに言って「量子レジスタを持つ古典コンピュータ」であり（図 2），古典コンピュータは古典的計算を行いながら量子レジスタに対して操作を行う．量子レジスタに対する操作は**ユニタリ変換**，**量子測定**と**量子状態の準備**の 3 種類であり，量子測定の結果は古典計算の中で用いるし，新しく準備した量子状態を ancilla として用いることもできる．（ただし，特に量子状態の準備についてはハードウェア的に制約される場合もあろう）

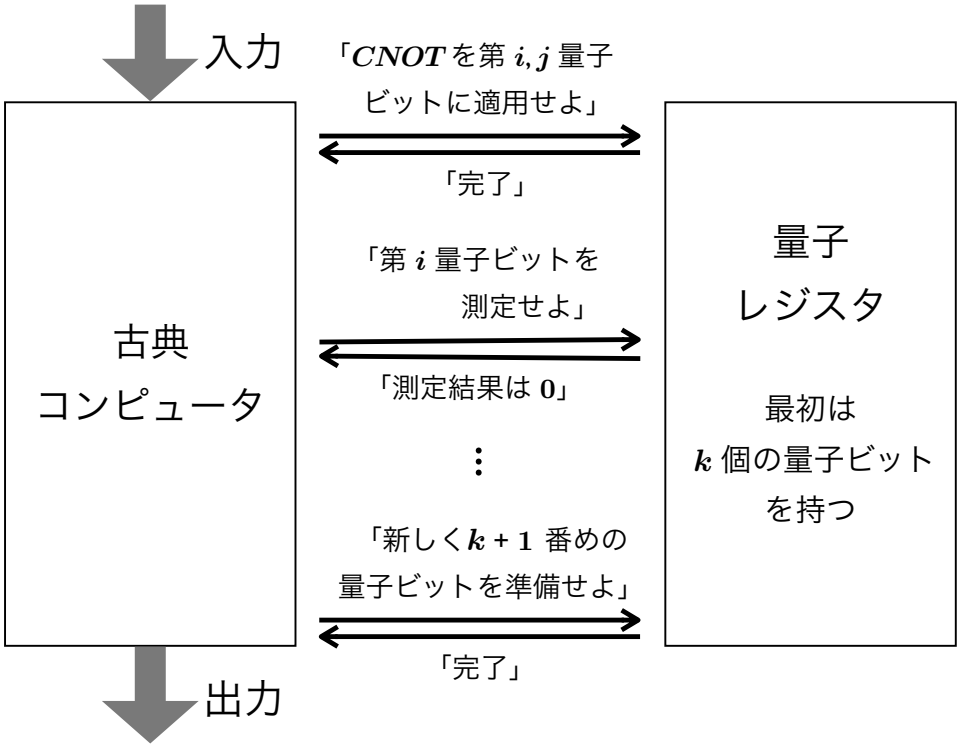

図 2： Knill の QRAM モデル

ここで注意しておきたいのは，QRAM モデルにおいてはプログラムの制御構造はあくまで古典的であることである. より直感的に説明すると, プログラムの実行において「いまプログラムのどこにいるか」を考えるだろう（「4 行目の変数宣言の前」や「6 行目の if 分岐の then のあと」など）. これを**プログラム・カウンタ**とよぶが，量子計算だからといって異なるプログラム・カウンタが量子的に重ね合わされることは (QRAM

モデルにおいては）ない．このような量子計算のパラダイムを**量子データ・古典制御 quantum data, classical control** とよぶ．

多くの量子アルゴリズムにおいては「この量子ビットを測定し，その結果が 0 だったら〇〇，1 だったらx×××...」といった分岐が存在するゆえ，古典制御を何らかの形で行うことの必要性は了解されるだろう．（注： [1, Section 4.4] にあるように，測定結果によるユニタリ変換の分岐は controlled operator による「量子的分岐」に置き換えることができる．しかし，ユニタリ変換以外の，たとえば測定などを伴う分岐について同じことが可能かどうかは明らかではない） また，制御構造を古典に限ること――すなわちプログラム・カウンタの量子的重ね合わせを許さないこと――の正当化に対しては，「多くの量子アルゴリズムが古典制御で書ける」「量子制御のプログラムは難読になる」「古典制御にすることで既存の（古典的）仕様検証手法が適用できる」などの主張が可能である一方で，量子コンピュータを**量子シミュレーション**（量子的物理系のシミュレーションで，量子コンピュータの大きな応用可能分野の一つとされる）のために用いる立場からは反論も聞かれる．たとえば [2] を見よ．

以下，本節冒頭で述べた量子プログラミング言語の 3 つの潮流のそれぞれについて，解説を試みる．

命令型量子プログラミング言語

上にあげた 3 つの潮流のうちのひとつで，C や Pascal のような命令型プログラミング言語に量子的操作のためのプリミティブを付け加えた言語である．Ömer による QCL （最初のバージョンは 1998 年に発表された）が代表的である．

まず例を示す．図 3 は QCL による Deutsch のアルゴリズム（1985 年の改良前のもの）の表現である．ここで qureg は量子レジスタをあらわす型であり，U は適当なユニタリ変換である (U の内容は略す)．命令 measure y,m によって量子レジスタ y が測定され，その結果が古典レジスタ m に格納され，m の値によってループの実行の可否が決定されることが見てとれるだろう．

QCL は最初期の量子プログラミング言語の一つであり，公開されているコンパイラによってプログラムを量子回路に変換したのちにシミュレーションを行うことができる（古典コンピュータによるシミュレーションなので，一般的に指数関数的に遅くなる

が)．しかし，数学的な出自による特筆すべき利点（型による安全性の保証など．下に述べる）においては，他の２つの潮流に一歩譲ると言わざるをえない．

```
procedure deutsch() {
  qureg x[1];                          // レジスタの2つの量子ビットを
  qureg y[1];                          // それぞれ変数 x, y に割り当て
  int m;
  {                                    // ループ
    reset;                             //   量子レジスタをすべて|0〉に初期化
    U(x,y);                            //   ユニタリ変換
    measure y,m;                       //   y を測定
  } until m==1;                        // 測定結果 1 を得るまでくり返し
  measure x,m;                         // x を測定．これは
  print "g(0) xor g(1) =",m;           //   g(0) xor g(1) に一致するはず
  reset;                               // 量子レジスタ
}
```

図 3： QCL による [Deutsch, 1985] のアルゴリズム （[Ömer, 2009] による）

量子操作を副作用とする古典関数型言語

２つめの量子プログラミング言語の潮流は，量子計算を**量子操作を副作用とする古典計算**とみなし，（古典計算のための）関数型プログラミング言語の上で量子的副作用をあらわすモナドを導入することで量子プログラミング言語を得る，というものである．

関数型プログラミングにおける副作用

ここで，**関数型プログラミング**と，そこでの**副作用 side effect** について少し説明しておこう．関数型プログラミングにおいて型 A -> B のプログラムは，型 A のあらわす集合から，型 B のあらわすそれへの関数と理解される．実際，多くの関数型プログラミング言語は**型システム**を持ち，各プログラムの型を明示的に与える（あるいはコンパイラが型推論する）ことにより，プログラムのさまざまな**安全性**を保証することができる．また他の利点として，プログラムの意味が文脈（たとえば命令型プログラムにおける変数の値）によって変化しないという**参照透過性**があり，理論的立場（特に意味論的立場）からよく研究されるのみならず，特に金融分野などにおいて産業上の応用も近年では増えてきた．関数型プログラミング言語としてよく知られたものには，OCaml，Haskell，Scala，F#，Scheme などがある．

このように「プログラム＝関数」というのが関数型プログラミングの基本的アイデアである一方，実際の計算の中には「（純粋な）関数」（入力のそれぞれに対して出力を一つ

対応させる）とは理解しがたいものも存在する．たとえば

- コイントスを行うことで出力が確率的に変化する int -> int 型のプログラム
- 計算過程で標準出力に文字列を出力する int -> int 型のプログラム
- 命令型プログラミングのように変数と値の対応関係を格納する**メモリ状態**を持ち，適宜変数の値を参照したり更新したりしながら計算を行う int -> int 型のプログラム

などは，厳密な意味での関数とはならない．これらの計算の「不純さ」が**副作用**とよばれるものである（**計算効果**ともよばれる）．たとえば上記の例はそれぞれ，**確率的分岐**，**出力**，**状態**の副作用とよばれる．

以上のようなさまざまな計算の副作用を統一的に扱うインターフェイスがあれば，これは関数型プログラミングの可能性を大きく広げるものになろう（純粋な「関数」から，副作用のある「計算」へ）．実際，Haskell に代表されるいくつかの関数型プログラミング言語は**モナド**による副作用の統一的インターフェイスを提供している．モナドはもともと**圏論 category theory** から発生した概念で，その関数型プログラミングにおける使われ方の概略は以下のとおりである．

- 1 つの副作用（たとえば確率的分布）に対して，1 つの型構成子 T を対応させる．（型 A に対して T(A) も型）この T をモナドとよぶ．
- 当該副作用をもつ型 A から型 B の計算は，A -> T(B) 型の関数となる．
- T の定義においては，
  - 純粋な関数を副作用のある計算としてどのように埋め込むか
    (A -> B) -> (A -> T(B)) ，また，
  - 副作用のある計算をどのように合成するか
    (A -> T(B)) -> ((B -> T(C)) -> (A -> T(C)))

  なども定義しておく必要がある．

副作用としての量子操作

話をもとに戻そう．量子プログラミング言語の 3 つの潮流のうち 2 つめは Altenkirch らによる量子 I/O モナドがさきがけであるが，これはは「量子操作を副作用として考え，これに対応するモナド T を定義することで，Haskell を量子プログラミング言語にす

る」というアイデアに基づいた研究である．「量子操作を副作用とする」という言葉の意味は，上の例の状態の副作用を考えるとわかりやすい．状態の副作用においては計算はメモリ状態を隠し持っていて，これを参照したり更新したりしながら進んでいくのであるが，これは図 2 にあげた Knill の QRAM モデルにおいて右側の量子レジスタを古典的メモリ状態に置き換えたものと理解することができる．すなわち，逆に考えると，状態の副作用におけるメモリ状態を量子レジスタに置き換えたのが量子操作の副作用であり，前者の計算においては変数 x の値を参照する lookup(x) や変数の値を更新する update(x,a) などの命令を用いるのに対し，後者の計算では量子状態を測定する measQ(x) やユニタリ変換 u を適用する applyU(u,x) などの命令を用いるのである．

この方向性をさらに推進して，特にスケーラビリティを重視して設計された量子プログラミング言語が 2013 年に Green らによって提案された Quipper である．Quipper は Haskell に埋め込まれる形で定義され，そのコンパイラはプログラムを量子回路に変換する．Quipper はさまざまな量子アルゴリズムを簡潔に表現する目的で注意深く設計され，非常に大きな（たとえば数十兆ゲート）量子回路を生成するプログラムもすでに公開されている．例として Quipper による量子テレポーテーションの表現を図 4 に示す．ここでの Circ が上に述べた量子操作の副作用をあらわすモナドであり（量子回路 quantum circuit から来た名前），関数 alice, bob, teleport のそれぞれが A -> Circ B の型をしていることに注意されたい．

```
alice :: Qubit -> Qubit -> Circ (Bit,Bit)
alice q a = do
    a <- qnot a `controlled` q    // CNOT ゲートを適用
    q <- hadamard q
    (x,y) <- measure (q,a)
    return (x,y)

bob :: Qubit -> (Bit,Bit) -> Circ Qubit
bob b (x,y) = ...                 // 略

teleport :: Qubit -> Circ Qubit
teleport q = do
    (a,b) <- bell00               // Bell 状態
    (x,y) <- alice q a
    b <- bob b (x,y)
    return b
```

図 4： Quipper による量子テレポーテーション （[Green et al., 2013] による）

量子ラムダ計算

3 つの潮流の最後，量子ラムダ計算について解説する．量子ラムダ計算は 2 つめと同じく関数型プログラミングのスタイルをとるが，量子レジスタに対する操作を副作用としてくくり出すことはせず，計算における量子的部分と古典的部分を統一的に扱う．このアプローチは van Tonder の量子ラムダ計算と Selinger の量子フローチャート言語（1 階関数型言語とみなせる）を源流とし，その後も Selinger, Valiron や本稿の筆者らによって盛んに研究されている．この研究の発展は古典プログラミング言語に対する意味論的諸手法（特に**線形論理**と**圏論的意味論**）に牽引される感が強く，仕様検証のための数学的な枠組みが強調される．

まず例を見てみよう．図 5 に [Selinger & Valiron, 2006] で提案された量子ラムダ計算による量子テレポーテーションの表現を示す．ここで x，y，q1，q2 は量子ビット qubit 型の変数であり，b1，b2 は古典ビット bit 型の変数である．s は unit 型の変数であり，unit 型の値 * を渡すことで関数 EPR が呼び出される．このプログラムは量子テレポーテーションにあらわれる操作を素朴に関数として表現しており，関数型プログラミングに慣れた読者にとっては（特に図 4 の Quipper の例と比較しても）自然に見えるだろう．最後に得られる関数 telep は，型 (qubit ⊸ bit ⊗ bit) ⊗ ((bit ⊗ bit) ⊸ qubit) を持つ．

$$
\begin{aligned}
\mathbf{EPR} &= \lambda s.\, CNOT\big\langle\, H(new\,0),\, new\,0\,\big\rangle \\
\mathbf{BellMeas} &= \lambda q_2.\, \lambda q_1.\, let\,\langle x, y\rangle = CNOT\langle q_1, q_2\rangle\, in\,\big\langle\, meas(Hx),\, meas\,y\,\big\rangle \\
\mathbf{U} &= \lambda q.\, \lambda\langle b_1, b_2\rangle.\, if\ b_1\ \ then\,(if\ b_2\ then\,U_{11}q\ else\,U_{10}q) \\
&\qquad\qquad\qquad\qquad\quad else\,(if\ b_2\ then\,U_{01}q\ else\,U_{00}q)
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{telep} = \quad let\,\langle x, y\rangle &= \mathbf{EPR} * in \\
let\,f &= \mathbf{BellMeas}\,x\,in \\
let\,g &= \mathbf{U}\,y \\
in\quad &\langle f, g\rangle
\end{aligned}
$$

図 5： ：[Selinger & Valiron, 2006] の量子ラムダ計算による量子テレポーテーション

線形型システム

さてここで ⊸ や ⊗ など見慣れない記号があらわれた．これらは**線形型システム**の型構成子であり，一般の型システム（-> や × を用いる）とは異なるこの型システムの利用こそが第 3 の量子プログラミング言語の潮流——量子ラムダ計算——の最大の特徴であり利点である．「線形型システム」という名前は線型空間とは（少なくとも表層的なレベルでは）関係がなく，Curry-Howard 対応によって Girard の**線形論理 linear logic** と対応するがゆえの名前である．

少し詳細を述べると，線形関数型 A ⊸ B は「型 A の入力をちょうど一回だけ用いて型 B の出力を返す関数の型」をあらわし，線形積 A ⊗ B は「型 A のデータと型 B のデータをそれぞれ 1 個，あわせて 2 個」をあらわす．この型システムは量子計算によって重要な帰結をもつ——すなわち，「量子状態は複製できない」という**複製不可能性 no-cloning property** を型システムによって強制できるのである！ たとえばラムダ項 λ x^{qubit}. <x,x> は一見 qubit ⊸ (qubit ⊗ qubit) の型を持つように思えるが，線形型システムでは型がつかない．このことはコンパイル時に型エラーとしてプログラマーに通知されるゆえ，実行時エラーを避けることができる．これは QCL や Quipper などにない大きな利点である．

おわりに

本稿では量子プログラミング言語の研究の動機を述べた上で，その大きな 3 つの潮流について手短に解説した．量子プログラミング言語の研究は発展途上の分野であることは

間違いない. 特に量子コンピュータの実用化が未だならないことは大きなハンディキャップであるが，一方でプログラミング言語の研究者（特に意味論の研究者）の入れ込みようは特筆すべきものがある．その理由の一つには，これまで蓄積してきた仕様検証等の抽象的理論がキャッチーな応用先を見つけた，ということがあろう．もう一つの理由としてよく（冗談半分に）言われるのが，「**物理的実装が未だないゆえに**，あまりクリーンとは言えない既存のプログラミング言語に振り回されることなく，クリーンな言語をゼロから理論的に構成するチャンスだ」というものがある．読者がこの主張に同意される自信はないが，ともかく，本稿が量子計算に対するフレッシュな見方を提供できれば，筆者のこの上ないよろこびである．

謝辞

本稿の執筆にあたっては [3] が大きく参考となった．また，[3] の著者である Benoît Valiron 氏や Peter Selinger 氏, Prakash Panangaden 氏，Mingsheng Ying 氏とは，さまざまな機会に有益な議論を行うことができた．ここに謝意を表する．

参考文献

[1] Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information. Cambridge Univ. Press (2000)

[2] Ying, M., Yu, N., Feng, Y.: Alternation in Quantum Programming: From Superposition of Data to Superposition of Programs. CoRR abs/1402.5172 (2014)

[3] Valiron, B.: Quantum Computation: From a Programmer's Perspective. New Generation Comput. 31(1): 1-26 (2013)