EP 演習報告書

量子コンピュータ用プログラミング言語: Ryoshi-lang

本間大一優

近年量子コンピュータは、ハードウェア技術の発展により性能が高まったことに加え、機械学習への利用など適応領域が広がってきており、社会的に注目を集めている。しかし、量子コンピュータはとても難しく、到底使えるものではない、研究用のものだと考えている人が多い。私はこの状況を生み出している要因は量子コンピュータプログラミングの記法にあると考えた。現在、量子コンピュータプログラミングは量子ビット(以下Qubit)に対して量子ゲートを適応していき、量子回路を書くという流れで行われる。ひとつひとつの Qubit に処理を施し複雑な量子状態を作れるという点で大変この記法は優れている。しかし、量子に対して深い知識がない人から見るとこの記法は難易度が高い。なぜならば、量子ゲートは行列で表されるため、それが適応された後 Qubit がどう変化するか想像しにくく、それに加え量子回路は古典コンピュータに対するプログラムとは形が全く異なるからである。そのため、量子ゲートに関する知識がなくても直感的に量子プログラムを記述できる言語 Ryoshi-lang を開発し、量子コンピュータの一般化を目指した。

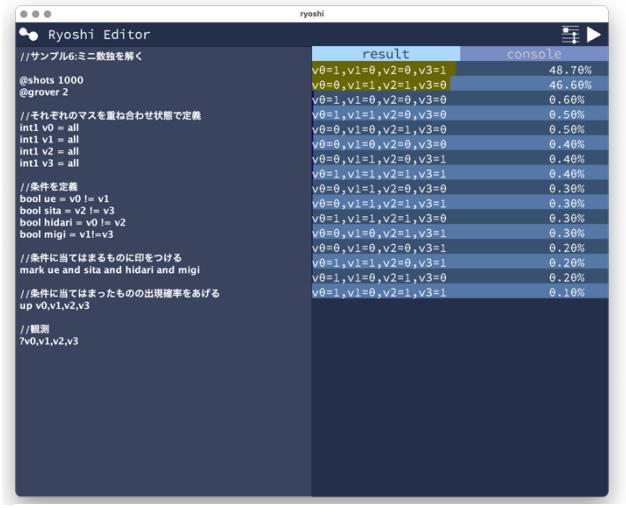

図 1

まず、Ryoshi-Editor(図 1)上でのプログラムの編集からコンパイル、実行、結果表示までの一連の流れを説明する(図 2)。

実行ボタン(図 1 右上)が押された後、Ryoshi-Editor 上で編集されているプログラムは文字列として Ryoshi-Editor が内包する Ryoshi-Compiler に渡される。Ryoshi-Compiler では ryoshi プログラムの字句解析、構文解析、意味解析を段階的に行い、Python コードを出力する。出力された Python コードは内部で自作 Python ライブラリ(ryoshi-Librar)を使っている。このライブラリは IBM が提供する量子コンピュータプログラム用 Python ライブラリ・Qiskit を Qubit 同士の比較や、任意の整数の重ね合わせ状態の生成を簡単に行うためにラップしたものである。そして、Ryoshi-Editor が、

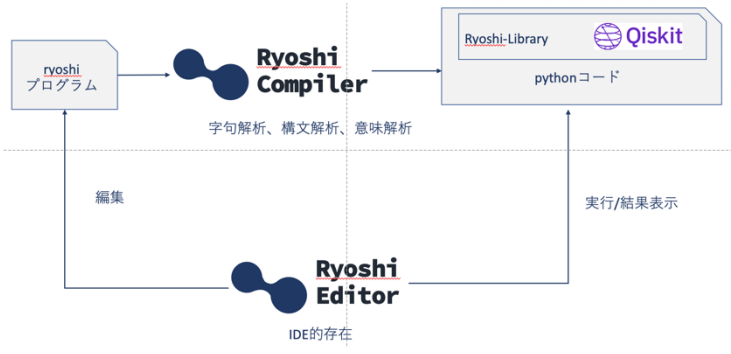

図 2

Ryoshi-Compiler から出力された Python コードを実行し、結果を取得し画面上に表示する。これがプログラムの編集から結果表示までの一連の流れである。

Ryoshi-lang でのプログラミングの流れと禁止事項ついて紹介する。従来の量子コンピュータプログラミングでは量子レジスタ(Qubit の集まり)の確保, 量子レジスタの初期化,量子レジスタに対する操作,量子レジスタの観測、という流れで行われる。この一連の流れを Ryoshi-lang では,変数の宣言&初期値設定,変数に対する処理,変数の観測という形で行う。また、量子コンピュータでは観測以外の不可逆な演算や Qubit のコピーは不可能である。この性質のために、Ryoshi-lang では変数への値の再代入の禁止がされている。(a = b これは b の値をコピーしており、かつ a の情報が失われる不可逆な演算である)

まず、変数の宣言と初期値設定について説明する。Ryoshi-lang は静的型付けトラスコンパイル言語である。そのため、型を指定して変数を定義する必要がある。変数定義の文法は非常に従来の言語に似ているが、ユーザー側が int 型の型の大きさを決める必要がある点において従来と異なる。量子コンピュータで扱える Qubit 数は現在最大でも 32bit ほどで、C 言語と同じ型を導入してしまうと int 型の変数を 1 つ定義しただけで 32bit 使ってしまうためである。例えば符号なし 16bit 整数は int16 のように表現する。また、変数の初期値には true,false や整数の他に、2 つの整数の重ね合わせ状態,すべての組み合わせの重ね合わせ状態を設定することができる。前者は 2|3,後者は all というふうに表現する。また、初期値を書かないと自動的に 0 または false に設定される。以下にサンプルコードとコンパイル後の量子的な処理を示す。

//4bit 符号なし整数の変数,a を宣言し 2 で初期化する。

int4 a = 2

4bit の Qubit レジスタ(name=u_a)を確保し,2bit 目に対して X ゲート(Not ゲート)を適応する。(0000->0010) レジスタ名の先頭に付与された u_はユーザーが定義したという意味。コンパイラが自動的にレジスタを作成することがありそれと区別するためにつけている。コンパイラが作成した場合は c_が付与される。

//真偽値型の変数,a を宣言し false で初期化する

bool b

1bit の Qubit レジスタ(name=u_b)を確保する。

//真偽値型の変数,a を宣言し true と false の重ね合わせ状態を生成する

bool a = all

1bit の Qubit レジスタ(name=u_a)を確保し、H ゲート(重ね合わせ状態にするゲート)を適応する。

//4bit 符号なし整数の変数,a を宣言し 2 と 3 の重ね合わせ状態を生成する

int4 a = 2 | 3

4bit の Qubit レジスタ(name=u_a)を確保し、量子もつれを用いた重ね合わせ状態を作る独自アルゴリズムを行う

//4bit 符号なし整数の変数,a を宣言しすべての組み合わせ(0~15)の重ね合わせ状態を生成する。

int4 a = all

4bit の Qubit レジスタ(name=u_a)を確保し、H ゲート(重ね合わせ状態にするゲート)を全てのビットに適応する。

次に、変数に対する処理を行う式文について説明する。以下にサンプルコードを示す。

//a に b を加える

a+=b

レジスタ(name=u_a,u_b)に量子加算回路を適応する

//a から b を引く

a-=b

レジスタ(name=u_a,u_b)に量子減算回路(量子加算回路の逆演算)を適応する

//真偽値 b を反転する

!b

レジスタ(name=u_b)に X ゲートを適応する。

//a と b が等しいかどうかの結果を c の初期値とする

bool c = a == b

計算用レジスタ(name=c_1)とレジスタ(name=u_c)を確保する。レジスタ(name=u_a,u_b)のそれぞれの Qubit を制御ビットとする CNOT ゲートを対応する計算用レジスタの Qubit にかける。(XOR 演算)　計算用レジスタの全ての Qubit に X ゲートをかける。計算用レジスタ全てを制御ビットとする Multi Control X ゲートをレジスタ(name=u_c)にかける。

//a と b が異なるかどうかの結果を c の初期値とする

bool c = a != b

==の演算を施した後、最後にレジスタ(name=u_c)に X ゲートをかける

//a かつ b かどうかを c の初期値とする

bool c = a and b

レジスタ(name=u_c)を確保し、レジスタ(name=u_a)とレジスタ(name=u_b)を制御ビットとする ccx ゲートをかける。

//a または b かどうかを c の初期値とする

bool c = a or b

レジスタ(name=u_c)を確保する。レジスタ(name=u_a,u_b)に X ゲートをかける。レジスタ(name=u_c)にレジスタ(name=u_a, u_b)を制御ビットとする ccx ゲートをかける。レジスタ(name=u_a,u_b)に X ゲートをかける。レジスタ(name=u_c)に X ゲートをかける。

前述したように量子コンピュータでは不可逆な演算が許されていないため、一つの演算子を実行するたびにその結果を格納するレジスタを新しく作らなければいけない。しかし、演算を行うたびに bool a = b == c のように新しい変数を定義しては不便である。そこで、コンパイラが必要と判断した場合自動的にレジスタを作成するコードを挿入することで、a == 3 and c or !d や a and !(a == b)といった少し複雑な式もコンパイル可能となっている。

次に、変数に処理を行う命令文について説明する。これらの命令文によって簡単にグローバーのアルゴリズムを実装できる。グローバーのアルゴリズムは量子コンピュータにおいて重ね合わせ状態から、条件に当てはまる値のみの観測確率を上げるためによく利用される。具体的なアルゴリズムとしては以下の通りである。まず、オラクル関数を用いて条件に当てはまる値の位相を反転させる。次に、Diffuser と呼ばれる位相が反転した組み合わせのみ観測確率を上げる機構をかける。以上 2 つを適切な回数繰り返す。最後に観測する。以上が大まかなグローバーのアルゴリズムの流れである。ちなみに、繰り返す回数によって観測確率は波状に変化するため、適切な繰り返し回数を設定することが重要である。以下に命令文のサンプルコードを示す

//a(0~3)、b(0~3)のうち、a==b の組み合わせ(0,0)(1,1)(2,2)(3,3)の位相を反転させる

int2 a = all

int2 b = all

bool c = a == b

mark c

レジスタ(name = u_c)に Z ゲートを適応し位相を反転させる。

//複数条件も可能。a,b,c のうち a==b かつ b==c の位相を反転させる

int2 a = all

int2 b = all

int2 c = all

bool c = a == b

bool d = b == c

mark c,d

レジスタ(name = u_c, u_d)に multi control Z ゲートを適応し位相を反転させる。

*//前述したコンパイラの自動レジスタ作成機能により上記コードは以下のようにも記述*可能

int2 a = all

int2 b = all

int2 c = all

mark a==b and b==c

//変数 a,b の全組み合わせのなかで位相が反転されたものの出現確率を上げる

up a,b

アンコンピュートを行い、レジスタ(name = u_a,u_b)に Diffuser をかける

up を行う際、Diffuser を掛ける前に必ずアンコンピュートという処理が必要となる。これは量子もつれを解消させる操作であり、今まで量子回路にかけてきた量子ゲートを逆にかけることで実現できる。そのためにコンパイラ内でどのレジスタ同士がもつれ合っているのか保存しており、その保存された情報をもとにアンコンピュートを行っている。

次に観測について説明する。観測とは Qubit を物理的に観測し、0 か 1 どちらかの結果を得ることを意味する。また、観測は重ね合わせ状態やもつれなどの情報を全て失うため破壊的であり不可逆な操作である。Ryoshi-lang では観測もかんたんに行える。いかにサンプルコードを示す。

//a,b を観測する

?a,b

レジスタ(name=u_a,u_b)に対して観測を実行し、古典レジスタ(name=u_a_c,u_b_c)に結果を保存する

最後に実行設定の記述について説明する。Ryoshi-lang では実行回数や実行方法、グローバーのアルゴリズムの繰り返し回数などをソースコード内で設定可能である。以下にサンプルコードを示す。

//試行回数(デフォルト:1 回)

@shots 1000

//実行場所(デフォルト:simulator)

@device simulator //シミュレータ

@device actual //実機

//グローバーのアルゴリズムの繰り返し回数(デフォルト 1)

@grover 2

また、予期しないプログラム(真偽値に整数値を足す,int4 に 128 を設定しようとしているなど)がある場合はエラーが出るようになっている。

今後の展望

現在、符号なし整数しか対応していないため、符号付き整数についても対応したい。また、不等号や掛け算といった処理も実現可能であれば実装したい。現在、量子性を活かしたプログラムはグローバーのアルゴリズムを使ったものに限られているが、他にもアルゴリズムはたくさんあるので(QFT,位相キックバック等)それについても実装を検討したいと思う。

参考にした書籍

Go 言語で作るインタプリタ(Thorsten Ball、 設樂 洋爾) ：この中の Pratt 構文解析のコードを参考にしました