古典・量子ハイブリッドな高機能プログラミング言語の設計及び処理系の開発
― Qitten ―

1. 背景

コンピュータを動かすためにはそのプログラムが必要であり、プログラムを作成するためにはプログラミング言語が必要である。これは量子コンピュータでも同様であり、量子コンピュータを使う以上、そのためのプログラミング言語（以下量子言語と呼称）が必要となる。

実際 IBM Quantum などのプラットフォーム上で動作するプログラムを作成するためには、量子アセンブリと呼ばれる OpenQASM などで記述されたプログラムが必要となる。現在のところ量子アセンブリは主に次のような方法で生成されている。

- Q＃などの言語からコンパイルして生成する
- Qiskit などの SDK から生成する
- 直接手で書く

しかしながら、これらの方法では、複雑なプログラムを記述する際には、量子ゲートレベルの記述と古典的な制御構文しか用いることができないことなどが障壁となっている。この障壁を解消するため、古典同様、量子に関しても、より高級な言語、すなわち記述性や安全性など様々な機能でプログラマを支える言語の開発が志向されている。

より高級な形でプログラムを記述するための量子言語として開発されてきた言語としては、Silq や Quipper などの量子言語が挙げられる。これらの言語には、線形型システムや量子非計算（uncomputation）のシステム、量子・古典のプログラムを混合しつつ記述することが可能なハイブリッド性などが採用され、記述性などを高める工夫がなされてきた。その一方で、これらの高級な量子言語を以ってしても、言語機能が未だ不足しているという課題や、プログラムをコンパイルして実際の量子コンピュータ上で実行できないという課題が認められ、理想とはほど遠い。そのような現状もあってか、高級言語と言えるほど高い記述性を持つ言語で普及しているものは未だないと言える。

2. 目的

本プロジェクトでは、実用的かつ先駆的な言語 Qitten を設計・開発することで、Qitten を通して、量子計算になじみのない多くのプログラマを量子プログラミングにいざない、量子ソフトウェアの進歩に貢献することを目標として実施した。

3. ソフトウェア開発内容

本プロジェクトでは以下の設計・開発を行った。

- (開発内容 1) 高級な古典・量子ハイブリッドな新しい言語 Qitten の設計

- （開発内容 2) Qitten のコンパイラの開発
- （開発内容 3) コンパイル時の最適化アルゴリズムの設計
- （開発内容 4) Qitten のシミュレータの開発

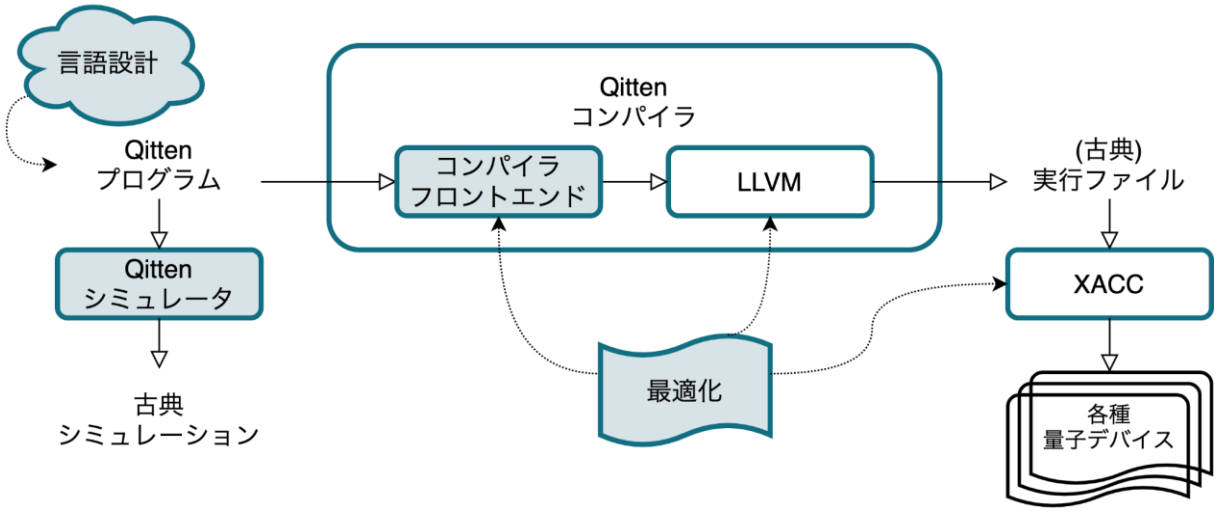

図 1. 開発の全体構成図

（開発内容 1）では古典プログラムと量子プログラムを混ぜて記述可能なハイブリッドな言語 Qitten を設計した。Qitten では、線形型システムや、量子的な高級文法である量子 if 式などを導入することで、言語の実用性・可読性・安全性の向上を試みた。とりわけこの仕様の検討においては、古典プログラミング言語の中で最近シェアを強めている言語「Rust」と構文を可能な限り共通させることで、現代の古典プログラミング言語との親和性にも配慮した。図 2 にプログラムコードの例を示す。

```
fn main() {
    …
    let mut bit: qbool = qfalse;
    let bit = h(bit);
    …
}
fn h(mut bit: qbool) -> qbool {
    inline_qasm!(“Hadmard”, bit);
    bit
}
```

```
fn main() -> bool {
    …
    let mut bit: qbool = qfalse;
    let bit: qbool  = h(bit);
    …
    measure(bit)
}
```

図 2. Qitten プログラムコードの例

（開発内容 2）では現存する量子デバイスを対象とした実行ファイルを生成する Qitten コンパイラを開発した。このコンパイラは古典プログラミング言語のコンパ

イラ基盤としてもよく用いられるソフトウェア「LLVM」や、量子的な処理を複数の異なる量子計算機で実行できるような統合インターフェイスを提供するソフトウェア「XACC」等の既存のソフトウェア資産を活用することで、古典計算に関する最適化の実装や個別の量子コンピュータにあわせた実装を最小化することに成功した。また、通常コンパイラフロントエンドと呼ばれる、コンパイラの処理過程の上流に位置する部分の実装に関しては、プログラミング言語 Rust のコンパイラである rustc と近い構造での実装を採用している。これにより、rustc に関連した既存の開発知見を活かしつつ、将来的な rustc とのマージなどの方向性を余地として残すこともできている。

（開発内容 3） 量子的な高級文法である量子 if 式（以下 qif）に関しては、図 3 のように、qif のネストや else を重ねる記述が許されている。このような量子的な if 文をコンパイルした前例は我々の知る限りない。

qif のコンパイル時の扱いとしては、qif のもつ then と else の二つのブロックは、分岐としてではなく直列なコードブロックとして扱われることが大きな特徴の一つである。そして、それぞれのブロックに相当する量子回路の合成は、制御条件に当たる量子ビットで制御された制御付き量子ゲートに置き換えることで行われる。この方法は qif のネストが深くない場合、回路サイズが問題になる事はないが、qif のネストが深くなる場合、回路サイズが肥大化する問題がある。我々はこの問題を、それぞれのブロックについて補助量子ビットに制御条件を集約することで解決した。すなわち、補助量子ビットに一般化 Toffoli ゲートを用いて制御条件の連言を取り、導入した 1 補助量子ビットのみに制御された制御付き量子ゲートを続けて合成することで問題を解決した。

```
fn main() {
   …
   qif flag1 {
      gates::X(targ);
      qif flag2 {
         gates::Z(targ)
      }
   } else {
      gates::Y(targ)
   }
   ….
}
```

```
fn main() {
   …
   qif flag1 {
      gates::Hadmard(targ)
   } else qif flag2 {
      gates::Y(targ)
   }
   ….
}
```

図 3. 量子 if 式を用いたプログラム例

（開発内容 4）では言語仕様を生かした古典シミュレータを、既存の古典シミュレータでの知見やソフトウェア資産を活かしながら作成した。

4. 新規性・優位性

今回我々は、初めて高級な文法を含むハイブリッド言語をコンパイルすることに成功した。これによりユーザーは、高級な文法で記述したプログラムを実際の量子コンピュータ（IBM Quantum, IonQ 等）上で実行することができるようになった。表 1 は既存の高級量子言語との比較表である。

表 1. 既存言語との比較

	Silq	Quipper	**Qitten**
型システム	線形	線形でない	線形
非計算	自動* + 隠蔽	自動 (明示的)	なし
手続き or 関数	手続き	関数型	手続き
ライブラリ	なし	Haskell のものあり	今後整える
ハイブリッド	yes	yes	yes
量子 if がある	yes	no	yes
コンパイラ	ない	△	ある

5. 期待されるユーザー価値と社会へのインパクト

これまでの言語と比較し、Qitten ではユーザーはより簡単かつ直感的にプログラムを記述することができるようになり、量子プログラマの開発環境の改善の第一歩となった。また、制御付き量子操作という概念を古典の if に関係させることで、古典プログラマにとっての記述難易度を下げ、量子ソフトウェア開発への参入障壁の低減に向けた第一歩ともなった。

6. 氏名（所属）

平田 賢吾 (京都大学 理学研究科)
寺本 幸生 (東京大学情報理工学系研究科コンピュータ科学専攻)
米内 貴志 (株式会社 Flatt Security)

（参考）関連 URL

・成果報告会のスライド　https://speakerdeck.com/terapoon/gu-dian-liang-zi-haiburitudonagao-ji-neng-puroguraminguyan-yu-falseshe-ji-ji-bichu-li-xi-falsekai-fa

・Qitten のデモ動画 https://www.youtube.com/watch?v=Dlt1ufYQZws