Report

jyd&Nina&chl

2021 年 5 月 8 日

摘要

我们总结了智能合约语言的部分现状，简要介绍了以 ProjectQ 以及 $Q^{\#}$ 为代表的量子程序语言，并描述了它们部分特征。

关键字: 区块链，智能合约语言，量子程序语言

Abstract

We summarize part of the current situation of smart Contract languages, briefly introduce the quantum programming languages represented by projectQ and $Q^{\#}$, and describe some of their characteristics.

Keywords: blockchain, smart contract languages, quantum programming languages

1 引言

区块链技术是近些年研究的热门方向。本质上讲，区块链是一个记录了网络一切变化的分布式数据库。智能合约，在 1996 年被 Nick Szabo 首次提出。“合约”指的是记录执行条件与对应执行的条款，而“智能”指的是自动、可编程。智能合约可以被看成一段可执行代码，它在一定条件下自动执行，且不受人为干预，例如自动售货机、免密支付等。智能合约并非只能依靠区块链实现，区块链与智能合约如此紧密的原因是区块链可以保证智能合约的不可篡改。一般来说，不同的区块链平台提供实现智能合约的语言也各不相同，例如 Solidity 用于以太坊，而 Script 用于比特币。在第 2 节中是文章 Smart Contract Languages: A Multivocal Mapping Study 的一些内容的总结 [3]。

量子计算也是现在的热门研究方向，目前成熟的通用量子计算机尚未面世，但量子编程语言的研究已经开始，以期在硬件成熟后能马上使用。目前的量子编程语言包括 qiskit、projectq、Q#、QPanda 等。本文主要介绍两种语言。第一种是 projecq，这是在 python 的基础上的扩展，部分底层使用了 C++ 实现来提高效率；第二种是 Q#，这是由微软提供的量子编程语言。对于 projectq，还会在附件中提供一个简单的实例代码。

2 智能合约语言

2.1 已存在的智能合约语言

目前存在的智能合约语言总共 101 种，可分为三类语言范式：命令式语言、声明式语言和符号语言。在这些语言中，用于实现的语言包括命令式 28 种，声明式 25 种，符号 3 种，其中有 9 种命令声明混合型的；用于规范的语言包括声明式 33 种和符号 3 种。用于实现的明显比用于规范的更多且种类更丰富。

2.2 这些语言的特征

2.2.1 这些语言基于哪些平台？

最常用的平台是以太坊，共支持 23 种不同的智能合约语言；其次是平台无关的语言，共 22 种。第三是比特币，共 7 种，它和以太坊都是被大众熟知的区块链平台。

除此之外，有 7 种语言尚是理论、未被实现或是无法在区块链平台上部署的规范语言。还有 13 种语言没有区块链平台。

2.2.2 这些语言是单独的还是已有语言的扩展？

实现语言里单独的共 41 种，扩展的共 24 种；规范语言里单独的有 21 种，扩展的有 15 种。学术界里单独语言 33 种，扩展的有 27 种；工业界里单独的有 29 种，扩展的 12 种。

不难看出，单独的智能合约语言在不同情况下的数目都大于扩展的语言，在实现语言、工业界中表现的最为明显。

2.2.3 关于智能合约语言的研究焦点在哪？

最受瞩目的四个方面分别是验证、金融合同、安全性和法律问题。

验证方面的研究集中在如何通过分析或者提供机制，达到验证完整性、正确性等属性的目的。

金融合同指多方现金交易流的合法共识。智能合约在这方面的研究集中在如何提供易于非技术人员理解的，可在分布式账簿上管理的语言。

安全性方面的研究是在语言里增加安全机制来提高安全性，例如防止攻击。

法律问题方面的研究是如何使非计算机专家能够访问的法律协议的模型，并建立开发者与法律专家的交流通道。

2.3 关于 Solidity& 特征

Solidity 是一种静态类型的类似 java 脚本的编程语言，用于开发在以 Ethereum 虚拟机（EVM）上运行的智能合约。(Ethereum 是一个基于区块链创建分散在线服务的平台，在智能合约的基础上工作。）语言于 2014 年首次引入。最新版本 0.8.4 于 2021 年发布。下图是一个简单的代码:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
uint storedData;

function set(uint x) public {
storedData = x;
}

function get() public view returns (uint) {
return storedData;
}
}
```

Solidity 是一种面向对象语言，contract 类似于 class(类)，第一行表示

源代码是在 GPL3.0 版本下获得许可的，第二行表示代码是为从 0.4.16 及更高版本到 0.9.0 的 solidity 版本编写的。Pragma 关键字用于启用某些编译器功能。uint 为无符号整数类型。

下面是 Solidity 的一些特点：

1. 以太坊底层是基于帐户，而非 UTXO 的，所以有一个特殊的 Address 的类型。用于定位用户，定位合约，定位合约的代码（合约本身也是一个帐户）；

2. 由于语言内嵌框架是支持支付的，所以提供了一些关键字，如 payable，可以在语言层面直接支持支付；

3. 存储是使用网络上的区块链，数据的每一个状态都可以永久存储，所以需要确定变量使用内存，还是区块链；

4. 运行环境是在去中心化的网络上，会比较强调合约或函数执行的调用的方式。因为原来一个简单的函数调用变为了一个网络上的节点中的代码执行，给人分布式的感觉；

5. 异常机制：一旦出现异常，所有的执行都将会被回撤，这主要是为了保证合约执行的原子性，以避免中间状态出现的数据不一致。

3 ProjectQ

本节主要介绍量子编程语言 ProjectQ 及其高级语言层面的特性，实例代码会在附件中给出。

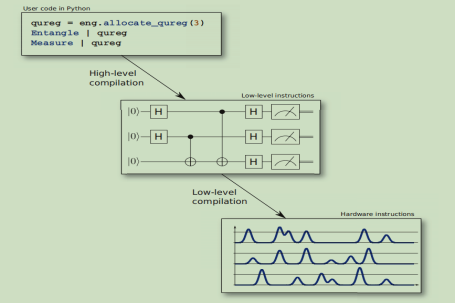
3.1 ProjectQ 概要 & 设计思想

ProjectQ 是 2017 年由苏黎世联邦理工学院发起的一个开源的量子计算软件框架，其中的高级量子语言是嵌入在 Python 中的特定领域语言。是首个有针对各种类型硬件，高性能模拟器都有仿真能力的编译器架构，可以最终在实际硬件上编译、模拟、仿真并最终运行量子算法。

如 [1] 中所述，ProjectQ 的设计思想基于以下四点：1. 自由开放；2. 学习曲线简单；3. 易拓展；4. 代码高质量。并且 ProjectQ 有以下三种主要优势：1. 用模拟器测试前可以先执行高级语言；2. 模块化和可拓展设计；3. 后端到实际量子硬件。

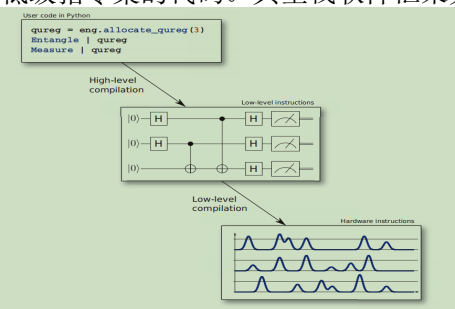
ProjecQ 有直观的语法和模块化编译器设计，量子编译器可以将高级语言转换为硬件指令，优化量子程序的所有不同的中间表示。另外，如下图所

示，高级的抽象有助于更快的开发，到低级指令集的自动编译可以使用户在任意可用后端编译他们的算法。



3.2 ProjectQ 的编译

用户用 Python 写程序，然后程序被发送到模块化编译器的前端，也就是 Mainengine。编译器由单个编译器引擎组成，这些引擎转换各种后端支持的低级指令集的代码。其全栈软件框架如下图所示：



3.3 基本类型与量子门

基本类型是量子寄存器 qureg。使用函数 allocate_qubit() 可以分配长度为一的量子寄存器，即 qubit。函数 allocate_qureg(int n) 分配指定长度的量子寄存器。

在使用辅助量子比特时，电路完成后需要将使用过的量子比特复位，这时就需要知道哪些量子比特被使用。ProjectQ 在初始化 qubit 时提供脏位，例如 qubit = allocate_qubit(dirty=True)。

量子计算的基本操作是量子门。ProjectQ 提供一些基本的量子门以供调用，例如 Pauli 门 X、Y、Z，Hadamad 门 H，和量子傅里叶变换 QFT。ProjectQ 提供函数 get_inverse 实现门的反转。想要把量子门作用在量子态

上只需要使用 | 操作符，如下所示：

```
H | qubit
```

这描述了将 H 门作用在量子比特 qubit 上。也可以把门操作装成函数，通过 python 的函数调用它。例如：

```
def RX_C(qubit, c):  #用函数包装
     Rx(c) | qubit
     H | qubit
```

为了实现将 $HR_x(1)$ 作用在 qubit 上，我们可以使用语句

```
RX_C(qubit,1)
```

3.4 特点

3.4.1 量子门及函数

可以用 Python 函数或者量子门来描述对量子数据类型的操作。如通常的 Python 函数 (关于常数加法的例子)：

```
def add_constant ( quint , c ) :
  QFT | quint
  # addition in the phases :
  phi_add ( quint , c )
  get_inverse ( QFT ) | quint
```

也即：

```
add_constant ( my_quint , 11)
```

但我们也可以用 ProjectQ 门的方式来作上述操作，并且定义一个拆分规则：

```
class AddConstant ( BasicGate ) :
  def __init__ ( self , c ) :
    self . c = c # store constant to add
# provide one possible decomposition :
register_decomposition ( AddConstant ,
add_constant_decomposition1 )
```

这使我们对量子门的操作可以使用通常的语法:

```
AddConstant (11) | my_quint
```

后一种方法允许我们在更高的抽象层次上进行优化。比如在上面这个例子中，我们把常量加法器定义为一个门，这就使我们通过直接加常数把两个 Addconstant 门合并了。另一个好处是对于不同的编译器，我们可以对(例如)Addconstant 操作使用不同的拆分规则，通过评估函数可以得到对后端而言最优的一种拆分规则，从而起到优化作用。

3.4.2 元指令

在高级语言层面，为了使一些常见的复杂量子门能被更简便地使用，引入了元指令。元指令的调用语法如下:

```
with MetaInstructionObject :
  ...
```

其中 Meta Instruction Object 可以是 Control、Dagger、Loop、compute 中的一种。这里主要介绍 Control、Dagger 和 Loop。

Control(eng,control_qubits): 控制门电路，在控制比特为 $|1\rangle$ 时执行代码块。

Dagger(eng): 反转整个幺正的代码块。

Loop(eng, num_iterations): 循环代码块指定次数。

3.5 后端

ProjectQ 支持十分丰富的后端, 包括各个硬件平台, 例如 IBM 等; 还有各种模拟器, 例如 ProjectQ 自己提供的基于 C++ 的模拟器; 以及量子电路图绘制后端 CircuitDrawer。在实例代码中使用的后端就是 CircuitDrawer。

4 $Q^{\#}$

本节基于 [2] 简要介绍关于量子编程语言 $Q^{\#}$

4.1 特点 & 设计思想

在 $Q^{\#}$ 之前设计的量子程序语言，很难模拟有非平凡分支的算法 (比如 repeat-until-success 算法)，但作为算法定义的语言 $Q^{\#}$ 则可以轻松地处理 (它直接此做了声明)。其在处理大量量子与经典需要交互的算法上优势很大。另一方面，$Q^{\#}$ 是独立语言，这和其它量子程序语言有本质区别，比如上文介绍的 ProjectQ 就是嵌入在 Python 中的。虽然主流语言支持的很多函数它无法使用，但它也有许多主流语言所不具备的优势。

这么设计的出发点是不把量子计算机作为一个完备的，成熟的设备，而是作为经典计算机的一个协处理器。即主控制逻辑在经典主机上运行经典代码，在适当的或者是必要的时机，主程序可以去调用运行在量子协处理器上的子程序。

另外，在算法设计上，$Q^{\#}$ 内置的库支持很多量子算法，可以直接调用。在它的标准库中。也支持定义 oracle。

4.2 算子 & 函数

在 $Q^{\#}$ 中算子与函数是 “一等公民”，它们可以作为参数传递给其它算子与函数，也可以作为算子或函数的结果返回。它们还可以组成元组或者可以建立关于算子与函数的数组。

Functors 是定义算子到算子的一个函数，比如 Adjoint,Controll。所以 Functors 可以执行比传统高级函数更为复杂的函数。

4.3 声明

Q# 有三个关于变量的声明：1.let;2.mutable;3.set。分别是创建不可变的量，创建可改变的量以及修改可变变量的值。注意，对数组本身以及每一项，根据定义的不同，要么不可改，要么可改变。显然，算子和函数的参数是不会被改变的。

5 个人理解与认识

5.1 智能合约

阅读的智能合约语言文献主要内容是对这一类语言的调查统计，并没有具体到某一个语言。据我读后的理解，智能合约语言是一类比较特殊的语言，因为它是为了一个十分具体的用途设计，因此有一些独特的特性，例如 Solidity 中的 Address 类型，以及直接实现支付的关键字 payable。智能合约语言的相关研究也有着金融、法律相关的问题，是一个有多个交叉的研究方向。智能合约的相关应用在生活中现在越来越常见，因此对它的安全性比较关注，希望这些语言能够提供更有安全保障的机制。

5.2 ProjectQ

ProjecQ 中我比较喜欢的特性是 MetaInstruction。正如它被设计的目的，它确实在构造一些复杂量子电路时提供了便利。在 ProjectQ 里，已经可以将一些操作打包成一个函数，但调用时也是以函数的形式，例如 3.1 节中的例子。如果能实现像通用门一样调用这个函数，可以变得更加方便，比如

```
RX_C(2) | qubit
```

此外，CircuitDrawer 输出的量子电路 latex 代码，最终生成的电路图格式并不是都很美观，经常有文字越出框外等影响阅读的情况，之后还需要手动调整，期待能有改进。

5.3 Q#

关于 Q#，其优势在于脱离了宿主语言的窠臼，自成一派，比如基本脱离了门电路这种单元，使它具有重构性，不用需要改的时候从头到脚改一遍。坏消息是微软似乎没有开源。此外，Q# 和 java 的结构比较相像，上手起来也比较容易。

我觉得比较有意思的一个是在 Q# 这个文章中提到的它支持 repeat-until-success 这样的循环结构，如下面这段代码所展示的：

```
sing ancilla = Qubit[1] {
  repeat {
    let anc = ancilla[0];
    H(anc);
    T(anc);
    CNOT(target,anc);
    H(anc);
    (Adjoint T)(anc);
    H(anc);
    T(anc);
    H(anc);
    CNOT(target,anc);
    T(anc);
    Z(target);
    H(anc);
    let result = M([anc],[PauliZ]);
  } until result == Zero
fixup {
();//空语句
}
}
```

repeat 语句的执行过程为：首先执行循环体，然后检查条件表达式是否为 true，如果为 true，那么该语句执行完毕；反之，则执行 fixup 语句中的内容，接着执行 repeat 语句中内容，如此循环直至检查的结果为 true。

此外，在 Q# 中，我们不需要重新定义一个量子门的复数共轭形式和受控形式，相反，我们只需要在定义新的量子门操作时进行相应的声明即可。具体代码请看附件。

关于引入新特征，希望在 Q# 中可以通过下标去查询数组元素，这样或许会更方便。但感觉量子计算机应该有自己本身的一套逻辑，虽说 Q# 向前迈了很大一步，但它还是作为经典计算机的辅助，期待未来有以量子计算机定义的逻辑为基础的高级语言，如果是中文的就更好了！

5.4 以太坊

以太坊 (Ethereum) 的目标是提供一个带有图灵完备语言的区块链，用这种语言可以创建合约来编写任意状态转换功能，除此之外，内置的持久化状态存储也是其设计的目标之一。

安全性的保证：Solidity 被设计为静态类型和强类型的语言，保证对于一些常见错误，让开发者能够迅速通过编译捕捉到，增强的语言的安全性。

以太坊在扩展性上的优势：1. 以太坊在比特币区块压缩的基础上，又采用了状态快照的方式来节约硬盘空间。具体来说，就是在区块头的结构中不但记录了当前区块所有交易的根散列，还记录了当前区块及过去所有区块中的状态根散列。这些状态包括所有的 UTXO、账户余额、合约存储等，所以节点只需要保留最新的区块和完整的状态信息即可；2. 提出了“幽灵协议”：它是以太坊对现有 POW 算法的改进，提出的动机是当前快速确认的区块链因为区块的高作废率而受到的低安全性困扰。

而在数字资产上，以太坊没有内置的多种数字资产支持，而是通过智能合约来实现相应的功能。这样的设计虽然很简洁，保证了较高的自由度，但也会带来一系列的负面影响，比如所有的资产创建者不得不自己编写重复的业务逻辑，而用户也没有办法通过统一的方式去操作自己的资产。此外，关于账户系统，基于 UTXO 系统的比特币可以很容易地对交易进行并行验证，因为 UTXO 之间是没有关联的，对任何一个 UTXO 的状态改变都可以独立进行且与顺序无关；而以太坊则是基于余额的账户系统，因为可能会同时发生多笔交易对同一个账户进行资产操作，需要进行一些额外的步骤来处理，所以并不容易实现并行，希望以太坊考虑添加并行计算的特点，这对区块链未来的扩展性会有很大的帮助。

参考文献

[1] Damian S Steiger, Thomas Häner, and Matthias Troyer. Projectq: an open source software framework for quantum computing. *Quantum*, 2:49, 2018.

[2] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q# enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, pages 1–10, 2018.

[3] Ángel Jesús Varela-Vaca and Antonia M. Reina Quintero. Smart contract languages: A multivocal mapping study. *ACM Comput. Surv.*, 54(1), January 2021.