**Какво е квантовата теория?**

Квантовата теория е дял от физиката, която се занимава със света на атомите и по малките (суб-атомни) частици вътре в тях. Законите на физиката на това ниско ниво, а именно атом, са различни от класическите физични правила, който наблюдаваме в нашия всекидневен живот.

**Лимитаций на класическия компютър**

Класическият компютър е помогнал много в трансформацията и дигитализацията на света, всестранно подобрявайки всичко, което човек до момента е измислил, от хранителната индустрия до електричеството и отвъд. Класическият компютър обаче има свойте лимитаций.

Класическият компютър ползва транзистори за да формира логически превключватели, които или позволяват на напрежението да премине през тях или го блокират (включено и изключено). На тази така проста логика се държи цялата тази комплескна и сложна архитектура на съвременния "класически" компютър. С времето и прогресите в минатюризацията на електро-елементи, големината на тези транзистори започнала да намалява, съвременните компютри станали все по малки и по малки, до появата на смартфоните. Обаче тази тенденция няма да продължава безкрайно, защото може да се каже че почти сме достигнали границата на физическите лимити при минатюризацията на транзистора и не само. Днес, големината на транзисторите достига 1 нанометър. На атомично ниво при големина по малка от 1 нанометър, електроните имат непредвидено поведение и не следват нормалните принципи на физиката. На атомично ниво, електроните демонстрират, т.нар. ефект на "квантовия тунел", дефакто понякога поведението им е като вълна и ще премине дори когато превключвателят блокира напрежение, игнорирайки логическите превключватели като цяло. Тъй като не може да имаме перфектни транзистори с по-малък размер от 1 нанометър, може да изведем извод, че официално сме достигнали лимитите на минатюризацията при транзистори. Това от своя страна ни пречи да продължаваме да увеличаваме изчислителната сила на нашите компютри, докато същевременно намаляваме техния размер.

Множество проблеми нуждаещи се от изчислителната мощ на компютъра са започнали експоненциално да се зараждат, а за разрешаването на някой от гях ще отнеме няколко години, а за други, няколко десетки години, със нашите съвременни "класически компютъри", които притежаваме. Затова трябва да потърсим други по ефикасни начини при разрешаватето на тези проблеми, а именно Квантови Компютри.

**Какво е квантов компютър?**

Квантовите компютър са машини построени на принципите на квантовата механика, която предприема нов подход при обработването на информация, ползвайки кюбити, което многостранно увеличава тяхната изчислителна мощ.

**Какво е кюбит?**

Фундаменталната изчислителна единица в класическия компютър е бит, което може да държи бинарни стойности, т.е или 0 или 1. Аналога на бита и кюбит (квантов бит) при квантовите компютри. Кюбита може да има специални свойства, който да помогнат за разрешаването на сложни проблеми по бързо от класическия бит. Едно от тези свойства е суперпозицията, което може да държи 0 и 1 едновременно. Когато много кюбита взаимодействат съгласувано един с друг, те могат да разучават много голямо множество от избори и да обработват информацията за фракция от времето, което ще отнеме на дори най-силния съвременен суперкомпютър.

**Какви са моментните предизвикателства на квантовия компютър?**

В момента Микрософт, Гугъл и ИБМ разполагат с квантови компютри, които имат силата да манипулират по малко или равно на 20 кюбита, като също така ИБМ стоят прототип който ще може да работи с 50 кюбита. Уви тези машини не са изцяло стабилни, а изграждането на стабилни такива може да отнеме над 10 години, тъй като квантовата наука включва в себе си и други науки, като свръхпроводници и нанотехнологий, който още се развиват и сами по себе си са достатъчно предизвикателни и сложни.

**Q-Sharp**

Натурален модел за квантови изчисления е да третираме квантовия компютър като копроцесор, подобен на тези ползвани при графичните процесори (GPU). Първичната логика за изпълнение се изпълнява чрез класически код, на класически компютър, когото можем наречем хост. Когато е подходящо и се наложи, хост програмата може да повика субрутина, която да изпълни въпросната част от логиката, на допълнителен процесор. Когато субрутината приключи, хост програмата взема нейния резултат.

Това прави и специализирания от Микрософт език Q#, който е специфичен език създаден за изразяване на квантови алгоритми. Той може да се използва за писането именно на тези субрутини, които да могат да се изпълнят на допълнителния квантов процесор, под контрола на класически компютър в класическа програма, която да бъде хост. Докато квантовите процесори не станат широко разпространени, въпросните субрутини ще се изпълняват на симулатор.

**1. Въведение в Microsoft Quantum Development Kit**

Фундаментала на квантовите изчисления се гради върху това да се запазва информация в квантовите състояния на материята и да се ползва квантови операций, за да се пресметне информация, ползвайки силата на квантовата интерференция. Такъв пример за квантова интерференция и програмиране, било да се разреши проблем, смятан за много труден от един конвенционален компютър. Проблемът бил известен като факториране, а експериментът проведен от д-р. Питър Шор през 1994-а се счита за реален старт на проучванията по проблема.

Алгоритъмът на Шор показва, че квантовият компютинг може бързо да декриптира повечето традиционни форми на асиметрична криптография. Повече от две десетилетия по-късно обещанията (и заплахите) от квантовия компютинг са тук. Вече не под формата на теоретични модели, а като реално работещи квантови компютри със софтуер, хардуер, в мрежи и с други комуникационни устройства.

Едно от големите предизвикателства е кубитовете да са достатъчно стабилни, за дълго време и без грешки, които да ги направят използваеми в сериозния компютинг (Граймс използва нетехническия термин "перфектни").

Според д-р Марк Джаксън, теоретичен физик и научен ръководител по бизнес развитие на Cambridge Quantum Computing (CQC), квантов компютър с едва 49 перфектни кубита надминава по производителност традиционните бинарни компютри. За да се разбият повечето настоящи документи, криптирани с публичен ключ, са необходими поне 4000 перфектни кубита или много повече, ако кубитовете не са перфектни.

Поне 44 компании и организации по света в момента разработват свои версии на квантови компютри. Сред тях са Google, IBM, Intel и Microsoft. Бъдейки все още поле на големи проучвания, екипът от Microsoft, представят своя Quantym Development Kit, за да подсилят и подкрепят проучванията по въпроса за квантовите изчисления, предоставяйки инструменти, който програмистите могат да ползват за да експериментират квантови алгоритми и да разрешават нетрадиционни проблеми. Екипът също така предоставя и езикът Q# , език от високо ниво, който е специално създаден за да адресира тези изпитания, пред които квантовата информация ни поставя.

**2. Модел на езика Q-Sharp и типовете данни.**

Q# е силно типизиран език и внимателното ползване на типове може да помогне на компилатора да даде силна надеждност на програмите повреме на компилация. За целта, конвертирането между типове в езика трябва да е експлицитно, ползвайки повиквания към функций, който изразяват тази конверсия. Набор от такива функций са ни подадени в Microsoft.Quantum.Convert namespace, като кастването към съвместими типове става имплицитно.

Q# ни дава малак набор от примитивни типове данни и два типа (масиви и тъпъли) за създаването на структурирани типове. Езикът поддържа базовия процедурен модел за писане на програми с условни конструкций и цикли, като също така подъжра и типове дефинирани от потребителя, като функций. По долу можем да разгледаве всичките типове и цялостния модел на езика Q#

**a). Примитивни типове:**

* Типа **Int** репрезентира 64-битов signed integer, т.е **2, 107, -5**
* Типа **BigInt** репрезентира signed intiger от своеволна големина, т.е. **2L, 107L, -5L.**
* Типа **Double** репрезентира числа с плаваща запетая, т.е. **0.0, -1.3, 4e-7**
* Типа **Bool** репрезентира булеви израз, който може да е **true** или **false**.
* Типа **Qubit** репрезентира квантов бит или кюбит, чиято единствена операция, освен подаването им на друга операция е да изпробват и проверяват за идентичност.
* Типа **Pauli** репрезентира едно от четирите Pauli операций върху един кюбит. Този тип се ползва за обозначаването на базовите операций за въртене и специфициране и измерване на наблюдаваното. Този тип е енумерация и има четири въможни стойности: **PauliI**, **PauliX**, **PauliY**, **PauliZ**, който са константни типове на **Pauli**.
* Типа **Result** репрезентира резултат от измерване. Този тип е енумерация и има две възможни стойности, **One** и **Zero**, които са константи. **Zero** покачва че се е измерило +1 собствена стойност, т.е **eugenevalue**, а **One** показва че е отчетено -1 собствена стойност.
* Типа Range репрезентира поредица от целочислени типове, нотирани по следния начин **start...step..stop**, където денотирането на step не е задължително. т.е **start..1..stop**, т.е **1..2..7**, което репрезентира поредицата **1,3,5,7.**
* Типа **String** е поредица от символи. Типа по скоро се ползва за да се генерират съобщения до класическия компютир или при изкарването на съобщения за грешка.
* Типа **Unit** е тип, който позволява само една стойност.
* Константите **true, false, PauliI, PauliX, PauliY, PauliZ, One** и **Zero** са резервирани символи в езика **Q#**

**b). Масиви**

Типа масив в **Q#** се репрезентира като **'T[]** , т.е **Qubit[]** или **Int[][].** Например колекция от целочислени типове се анотира като Int[], докато масив от масиви от **(Bool, Pauli)** стойности, се нотира като **(Bool, Pauli)[][].** Тук можем да забележим как репрезентацията е по-скоро представена като назъбен масив а не квадратна матрица. Това е така защото езикът **Q#** не подържа квадратни матрици или многомерни масиви.

Масивите в езика може да бъде изписан чрез квадратни скоби около елементите в масив, като при **[PauliI, PauliX, PauliY, PauliZ].** Типа на литерала се определя на базата на това какви типове се съхраняват в масива. Също така алтернативно може да ползваме ключовата дума new за да декларираме и инициализираме масив, през конструктур. Например:

|  |
| --- |
| let zeros = new Int[13];  // new also allows for creating empty arrays:  let emptyRegister = new Qubit[0]; |

При всички случай, веднъж щом масив е бил конструиран, главната фукция Length може да бъде извикана за да се достъпи броят на елементите в колекцията.

**c). Тъпъли**

При дадени нула или повече различни типове T0, T1, …, Tn, можем да обозначим нов тип **tuple,** като (T0, T1, … Tn). Типовете който се ползват за да се конструира нов тип **tuple**, могат самите те ба бъдат tuple, например **(Int, (Qubit, Qubit)).** Измеренията разбира се винаги са крайни, понеже **tuple** не могат по никакъв начин да съдържат самите себе си във себе си.

Стойностите на нов **tuple** са **tuples,** формирани от поредица от стойности от всеки **tuple** във **tuple.** Например, **(3, false)** е **tuple,** чийто тип е от тип **tuple** **(int, bool)**. Възможно е да се създават мартици от тъпъли, тъпъли от съб-тъпъли и прочие.

Тъпълите са силна концепция използвана в езика да събират стойности заедно в една стойност, правейки ги по лесни за предаване нататък. По този начин можем да изразим че всяка операця има само един вход и само един изход.

**d). Дефинирани от потребителя типове**

Можем да дефинираме нови типове, съдържащи единични стойности от всеки легален тип. Например за всеки tuple от тип T, можем да декларираме нов тип, който да е подтип от T със ключовата дума newtype.

|  |
| --- |
| newtype Complex = (Double, Double); |

Тази декларация ще създаде нов тип с два анонимни обекта от тип Double.

|  |
| --- |
| newtype Complex = (Re : Double, Im : Double); |

Тази декларация ще създаде нов тип с два найменовани обекта, съдържащи стойности от тип Double. Предимството на найменованите типове е че може да бъдат достъпени чрез операторът **::** , например ако имаме функция, можем да направим следното.

|  |
| --- |
| function Addition (c1 : Complex, c2 : Complex) : Complex {  return Complex(c1::Re + c2::Re, c1::Im + c2::Im);  } |

За да достъпим анонимни айтъми в колекцията, трябва първо опакованата стойност да бъде извлечена с постфикс операторът **!** .Като "извличането" позволява на стойноста, съдържана в дефинирания от потребителя тип, стойност. Например:

|  |
| --- |
| function PrintMsg (value : Nested) : Unit {  let (d, (\_, str)) = value!;  Message ($"{str}, value: {d}");  } |

**e) Операций и функций.**

Операция в Q# наричаме квантова субрутина. Това рутина която се повиква, чието съдържание е квантова операция.

Функция в Q# е класическа субрутина, приложена в квантов алгоритъм. Тя може да съдържа и класически код без квантови операций. Функцийте не може да разпределят кюбити или да викат операций. Възможно е обаче да предават операций и кюбити за изчиследие. Функцийте са изцяло детерминистични, с идеята че ако ги викаме с едни и същи агрументи, те видаги ще произвеждат едни и същи резултати.

Взети заедно, операцийте и фукнцийте наричаме **callables**. Всичките **Q# callables** взимат една стойност и връщат една стойност. Входа и изхода могат да бъдат **tuples,** а **callables** които нямат резултат връщат **Unit**. **Callables** който нямат вход, могат да приемат празен **tuple** за вход.

Един пример за проста квантова програма, която комбинира операций и функций е следната програма, която ще генерира произволно което може да бъде или 0 или 1, тъй като кюбита е единица, чиято информация може да бъде във суперпозиция. Което от своя страна означава че повреме на измерване, един кюбит може да бъде или 0 или 1, обаче повреме на изпълнение състоянието на кюбита репрезентира вероятност да прочете или 0 или 1. Тази вероятност е т.нар. суперпозиция на кюбитите, която можем да ползваме за генериране както на произволни числа, така и на по-сложни комбинаторни алгоритми.

|  |
| --- |
| namespace Quantum {  open Microsoft.Quantum.Intrinsic;  operation QuantumRandomNumberGenerator() : Result {  using(q = Qubit()) { // Allocate a qubit.  H(q); // Put the qubit to superposition. It now has a 50% chance of being 0 or 1.  let r = M(q); // Measure the qubit value.  Reset(q);  return r;  }  }  } |

**3. Начин на работа**

Един от фундаменталните градивни единици на **Q#** е **Qubit** типа, който не може да бъде копиран или директно адресиран, точно както един истински qubit. Вместо това, той може да бъде измерен, а резултата му запазен в променлива от тип **Result**, която в **Q#** може да приема само две стойности, **Zero** или **One**, които са константи. **Zero** покачва че се е измерило +1 собствена стойност, т.е еугенова стойност (**eugenevalue)**, а **One** показва че е отчетено -1 собствена стойност. Конструкций като тази винаги ще гарантират, че алгоритмите ще спазват законите на квантовата физика и ще работят правилно върху квантови компютри или симулатори.

**Q#** също съдържа и класическите логически оператори, като условни конструкций и цикли, със съответно модификаций, нужни за да сме уверени, че правилата на квантовата физика се спазват. Затова **Q#** програмите често се комбинират с хост програми написани на **C#** или **Python**, което ни дава добра организация между класически и квантов код.

Съответно можем да заключим че езика, създаден от Microsoft, успешно ни позволява да извършваме следното:

**a)** Изучаване на квантови изчисления – до сега за да научим как работят квантовите изчисления е било нужно да разбираме квантова физика и самият модел на електрическата верига. Сега можем да поемем по друг път, а именно да се учим и да експериментираме с квантови изчисления като директно пишем и създаваме квантови програми.

**b)** Създаваме квантови алгоритми – програмният език, чрез множеството от нарастващи библиотеки и дефинирани типове от потребителя ни позволява да имплементираме инструменти и да създаваме и експериментираме квантови алгоритми. Например за да свържем два кюбита q1 и q2, вместо индивидуално да заплитаме два кюбита чрез порти, можем да ползваме абстракция, която прави това вместо нас, а именно **PrepareEntangledState([q1], [q2]).**

**c)** Можем да симулираме изпълнението на наша програма, ползвайки квантовия симулатор, даден ни от Microsoft, т.е. QDK, който от своя страна дава ресурси за анализирането на програмата, която ще се изпълни. Това е особенно полезно за научни работници и създателите на алгоритми, който могат да анализират и симулират количеството от ресурси, което тяхната прогама би следвало да изразходи, по време на изпълнение. По този начин те могат да моделират и корегират своя алгоритъм, правейки го усъществим дори и на машини с по малък брой от кюбити.