Lenguajes de Programación Cuántica

Definición, utilidades y tipos de programación cuántica. Implementación en Q#.

Escuela Técnico Superior de Ingenieros Informáticos
Laura Alejandra Encinar González
15 de Noviembre de 2021

Correo electrónico: alejandra.encinar.gonzalez@alumnos.upm.es

Resumen

En el siguiente documento se presenta la definición de lenguaje de programación cuántica, así como cuáles son sus actuales usos, los tipos que existen y los lenguajes más utilizados. Además, se describe como instalar y utilizar el simulador cuántico de Microsoft mediante su kit de desarrollo cuántico (QDK). También se presentan los primeros pasos para programar en Q#, algunos ejemplos de inicialización de qubits, estado de superposición y entrelazamiento cuántico, así como la implementación del algoritmo de Teleportación. Seguidamente se comparan pequeños fragmentos de código en Q# con Silq demostrando las diferencias más significativas.

Palabras clave: lenguajes de programación cuántica, paradigmas, simuladores cuánticos, QDK, programación en Q#, algoritmo de teleportación, Silq.

Contenido

1. Introducción: ¿Qué es la programación cuántica y para qué sirve?

La programación cuántica es el proceso de ensamblar secuencias de instrucciones, llamadas programas cuánticos, que pueden ejecutarse en una computadora cuántica. Los lenguajes de programación cuántica permiten expresar algoritmos cuánticos utilizando construcciones de alto nivel. Incluso pueden facilitar el descubrimiento y desarrollo de dichos algoritmos antes de que exista hardware capaz de ejecutarlos.

- Según un artículo de Nature Reviews Physics [1], los lenguajes de programación cuántica se están utilizando para:
- Controlar dispositivos físicos ya existentes
- Estimar los costos de ejecución de los algoritmos cuánticos en futuros dispositivos.
- Enseñar conceptos de computación cuántica (qubits, superposición, entrelazamiento).
- Probar y verificar algoritmos cuánticos y sus implementaciones.

Los compiladores y lenguajes de programación cuántica actuales se centran en la optimización de circuitos de bajo nivel que consisten en puertas cuánticas. Las puertas cuánticas son los componentes básicos de los circuitos cuánticos. Son similares a las puertas lógicas reversibles como la puerta Fredkin, la puerta Toffoli, la puerta de interacción y la puerta del interruptor. Sin embargo, la puerta reversible clásica más pequeña tiene que usar tres bits, mientras que la puerta cuántica más pequeña necesita usar solo dos bits.

2. Tipos de lenguajes de programación cuántica

Podemos distinguir entre varios tipos de lenguajes de programación en función de para qué paradigma haya sido creado. Un paradigma de programación es un modelo básico de diseño y desarrollo de programas, que permite producir programas con un conjunto de normas específicas, tales como: estructura modular, fuerte cohesión, alta rentabilidad, etc.

Los paradigmas pueden ser considerados como patrones de pensamiento para la resolución de problemas. Desde luego siempre teniendo en cuenta los lenguajes de programación, según nuestro interés de estudio.

En la programación clásica podemos encontrar numerosos tipos de paradigmas: imperativo, funcional, heurístico, lógico, orientado a objetos… En cambio, en la programación cuántica podemos agrupar todos los lenguajes en tres paradigmas: imperativo, funcional y multiparadigma.

2.1 Lenguaje de programación cuántico imperativo

Los lenguajes de programación imperativos consisten en instrucciones paso a paso que se deben realizar para lograr el resultado deseado. Son aquellos que facilitan los cálculos por medio de cambios de estado, entendiendo como estado la condición de una memoria de almacenamiento. En las computadoras clásicas, los lenguajes imperativos incluyen C, JavaScript, Pascal, Python, etc. Por otro lado, en la computación cuántica también existe una gran variedad de lenguajes como:

- **Pseudocódigo cuántico**: es el primer lenguaje formalizado para la descripción de algoritmos cuánticos que se introdujo y, además, estaba estrechamente relacionado con el modelo de máquina cuántica llamado Quantum Random Access Machine (QRAM) [2].

- **Q Language**: segundo lenguaje de programación cuántica imperativo implementado. Se implementó como una extensión del lenguaje de programación C ++. Proporciona operaciones cuánticas básicas como QHadamard, QFourier, QNot, QSwap y QSwap, que se derivan de la clase base Qop, y permite definir nuevos operadores.

- **QCL**: Quantum Computing Language, uno de los primeros lenguajes de programación cuántica implementados. Es un lenguaje de programación de alto nivel, independiente de la arquitectura, con una sintaxis derivada de lenguajes clásicos como C o Pascal [3]. Esto permite la implementación y simulación completas de algoritmos cuánticos en un formalismo consistente.

- **QMASM**: Quantum Macro Assembler, publicado en 2016. Es un lenguaje de bajo nivel específico para el reconocido cuántico. La importancia de QMASM es que libera al programador de tener que conocer los detalles de hardware específicos del sistema y, al mismo tiempo, permite que los programas se expresen con un bajo nivel de abstracción.

- **Silq**: se publicó originalmente en 2020. Es un lenguaje de programación de alto nivel escrito en lenguaje D que tiene 482 estrellas y 10 colaboradores en github y se actualiza regularmente.

Otros lenguajes imperativos menos populares son Q | SI>, qGCL y Scaffold.

2.2 Lenguaje de programación cuántico funcional

Los lenguajes funcionales no se basan en instrucciones paso a paso, sino que dependen de funciones matemáticas, lo que significa que las entradas se convierten en resultados mediante transformaciones matemáticas. Los lenguajes funcionales son menos populares que los imperativos ya que no admiten controles de flujo (declaraciones de bucles) o condicionales (declaraciones if / else). Sin embargo, debido a estas características, tienen algunos beneficios:

- Menor cantidad de errores y mayor facilidad para la detección y corrección de estos.
- Funciones anidadas.
- Evaluación perezosa: retrasa la evaluación de una expresión hasta que se necesita su valor y de esta manera evita evaluaciones repetidas

Haskell o F# son algunos ejemplos de lenguajes de programación clásica funcional. Mientras tanto, los principales lenguajes funcionales para computadoras cuánticas son:

- **QML**: publicado en 2007, un lenguaje de programación cuántica similar a Haskell basado en una lógica lineal estricta. Puede integrar cálculos cuánticos reversibles e irreversibles.

- **Cálculo lambda cuántico**: se basa en el cálculo lambda clásico introducido en 1930 y se definió por primera vez para cálculos cuánticos en 1996. Utiliza funciones de orden superior. Por lo tanto, es más fuerte que los modelos computacionales cuánticos estándar, como como la máquina cuántica de Turing o el modelo de circuito cuántico.

- **QFC** y **QPL**: Semánticamente QFC y QPL son equivalentes. Sin embargo, en QFC, los programas cuánticos se representan utilizando la sintaxis de diagrama de flujo, pero en QPL la estructura sintáctica de los programas cuánticos se utilizan representaciones textuales.

- **LIQUi|>**: es una extensión de simulación cuántica en el lenguaje de programación F# desarrollada por Quantum Architectures and Computation Group (QuArC). Incluye un lenguaje de programación, algoritmos de optimización y programación y simuladores cuánticos. Se puede utilizar para traducir un algoritmo cuántico escrito como un programa de alto nivel en instrucciones máquina de bajo nivel para un dispositivo cuántico.

También encontramos Quipper y FunQ.

2.3 Lenguaje de programación cuántico multiparadigma

También hay lenguajes de múltiples paradigmas que son específicos de dominio, ya que permiten la implementación y combinación de varias de estas estructuras en el desarrollo de programas. Algunos ejemplos de ello son:

- **Strawberry Fields**: se implementa mediante una biblioteca de Python de código abierto desarrollada por Xanadu Quantum Technologies. Actualmente se utiliza para diseñar, simular y optimizar circuitos ópticos cuánticos de variable continua.

- **Q#**: desarrollado por Microsoft e incluido en su kit de desarrollo de Quantum. Es un lenguaje que ofrece un alto nivel de abstracción. El programador no necesita tener nociones de un estado cuántico o un circuito; en su lugar, Q# implementa programas en términos de instrucciones y expresiones, de forma muy similar a los lenguajes de programación clásicos (C# o F#). Además, incluye distintas funcionalidades cuánticas (como la compatibilidad con funtores y construcciones de flujo de control), que facilitan la expresión de la estimación de fase y de algoritmos de química cuántica.

3. Compilar y ejecutar un programa cuántico en un ordenador clásico

En la actualidad, muchas empresas de tecnología ofrecen servicios cuánticos. Estos se combinan con computadoras y sistemas de escritorio para crear un entorno en el que el procesamiento cuántico resuelve problemas complejos. Algunos ejemplos de empresas que ofrecen estos servicios son:

- **IBM** con el entorno IBM Q: ofrece acceso a varias computadoras cuánticas reales y simulaciones que puede utilizar a través de la nube.

- **Alibaba Cloud** con una plataforma en la nube, dónde se puede ejecutar y probar códigos cuánticos personalizados.

- **Microsoft** con un kit de desarrollo cuántico que incluye el lenguaje de programación Q #, simuladores cuánticos y bibliotecas de desarrollo de código listo para usar.

- **Rigetti** con una plataforma (en versión beta) que está preconfigurada con su Forest SDK.

De esta manera, podemos hacer uso de estos servicios para desarrollar y ejecutar algoritmos cuánticos.

3.1 QDK: kit de desarrollo de Microsoft Quantum

En este caso, usaremos las herramientas de desarrollo cuántico que nos ofrece Microsoft. En concreto, el kit de desarrollo de Microsoft Quantum (QDK) que incluye bibliotecas de Q#, simuladores cuánticos, extensiones para otros entornos de programación y documentación de API [4].

3.1.1 Simuladores Cuánticos

Los simuladores que nos proporciona Microsoft son programas de software que se ejecutan en equipos clásicos y actúan como máquina de destino para los programas de Q#. De esta manera, permiten ejecutar y probar programas cuánticos en un entorno que predice cómo reaccionarán los qubits a distintas operaciones.

Además, un simulador debe proporcionar las implementaciones de operaciones primitivas como H, CNOT y Measure, tanto como el seguimiento y la administración de qubits. El kit de desarrollo de Quantum incluye distintas clases de simuladores cuánticos, que representan diferentes formas de simular un mismo algoritmo cuántico:

- Simulador de estado completo
- Simulador de seguimiento de equipos cuánticos
- Simulador de Toffoli
- Simulador de ruido

Por ejemplo, el simulador de estado completo simula completamente el vector de estado, mientras que el simulador de seguimiento de equipos cuánticos no tiene en cuenta el estado cuántico real, sino que hace un seguimiento del uso de puertas, qubits y otros recursos.

El objetivo de estos simuladores es permitir que los algoritmos permanezcan constantes independientemente de que varíe la implementación de la máquina.

3.1.2 Programación en Q#

Los programas escritos en Q# no modelan el estado cuántico directamente, en su lugar describen como interactúa un equipo clásico con los qubits. Como consecuencia, no tenemos capacidad de consultar o modificar directamente el estado del qubit, en su lugar se deben implementar métodos set para cambiar un estado y utilizar operaciones como Measure (incluida en las bibliotecas) para medir el qubit.

Una vez asignamos un qubit podemos ejercer operaciones y funciones sobre él. Estas operaciones pueden ser implementadas por el usuario o pueden ser invocadas desde las librerías que incluye el kit. En el segundo caso, acciones directas sobre el estado del qubit como X (invierte el estado) y H (puerta de Hadamard) no se definen en Q# sino que son definidas por la máquina de destino. La ventaja de esto es que a la hora de ejecutar

nuestros programas en un hardware cuántico especifico no será necesario cambiar el código.

3.2 Instalación e iniciación del kit de desarrollo de Microsoft Quantum (QDK)

En este caso he optado por la utilización del kit con Visual Studio Codium [5], pero también existe la posibilidad de desarrollar código en Q# mediante un paquete de Python o utilizando otros IDE como Juptyer Notebook o Visual Studio.

Para la instalación necesitaremos tener previamente instalado Net una plataforma de desarrollo [6], de código abierto y gratuita para crear diferentes tipos de aplicaciones.

La instalación del QDK consiste en añadir la extensión Microsoft Quantum Development Kit for Visual Studio Code [7] a nuestro entorno de desarrollo.

Para empezar a programar en Q# ya solo necesitamos crear un nuevo proyecto. Para ello nos ayudaremos del atajo “ctrl + shift + P” que abre la paleta de comandos y seguidamente escribimos y seleccionamos “Q#: create new project”. Esto nos generará automáticamente una carpeta con las bibliotecas y archivos necesarios para compilar nuestros programas. También nos proporciona un código de ejemplo en el archivo program.qs.

A continuación, para compilar nuestro programa, abriremos nuestra terminal en la carpeta del proyecto y ejecutaremos el comando “dotnet run”. Es importante que la primera vez que compilemos un proyecto introduzcamos el comando “dotnet build” antes de ejecutar “dotnet run”.

3.3 Ejemplos:

3.3.1 Inicialización, estado de superposición y medida de qubits

Primero asignamos un qubit (q), seguidamente aplicamos una puerta de Hadamar (con la función `H(q)`) para poner al qubit q en estado de superposición y después realizamos la medida en Z mediante el método `MresetZ()`.

```
@EntryPoint()
operation MeasureSuperposition() : Result {
    use q = Qubit();      // allocates qubit fo
    H(q);                 // puts qubit in supe
    return MResetZ(q);    // measures qubit, re
}
```

3.3.2 Utilización de parámetros

En este caso utilizaremos el parámetro (`n : Int`) para designar el tamaño del array de qubits sobre el que vamos a aplicar una serie de operaciones mediante el método `ApplyToEach()`.

```
//@EntryPoint()
operation MeasureSuperpositionArray(n : Int) : Result[] {
    use qubits = Qubit[n];               // allocate a regi
    ApplyToEach(H, qubits);              // apply H to each
    return ForEach(MResetZ, qubits);     // perform MResetZ
}
```

Para introducir los parámetros compilamos nuestro programa desde la terminal de esta manera:
`dotnet run -<nombre del parámetro> <valor del parámetro>`

Por ejemplo, para este caso:

```
dotnet run -n 5
```

3.3.3 Generador de números aleatorios:

Para este programa utilizaremos un array de bits (tipo entero 0 o 1) en el que cada uno se genera aleatoriamente mediante el método `GenerateRandomBit()`. Más tarde transformaremos este array de bits en un entero con el método `ResultArrayAsInt()`.

```
operation SampleRandomNumberInRange(max : Int) : Int {
    mutable output = 0;
    repeat {
        mutable bits = new Result[0];
        for idxBit in 1..BitSizeI(max) {
            set bits += [GenerateRandomBit()];
        }
        set output = ResultArrayAsInt(bits);
    } until (output <= max);
    return output;
}
```

Este método se implementa utilizando la puerta de Hadamard y la medida de un qubit como lo visto en el ejemplo 3.3.1 Iniciación, estado de superposición y medida de qubits.

```
operation GenerateRandomBit() : Result {
    // Allocate a qubit.
    use q = Qubit();
    // Put the qubit to superposition.
    H(q);
    // It now has a 50% chance of being measured 0 or 1.
    // Measure the qubit value.
    return M(q);
}
```

Finalmente, llamamos a nuestro método `SampleRandomNumberInRange()` con el parámetro `max = 50` para obtener un número aleatorio del 0 al 50 .

```
@EntryPoint()
operation SampleRandomNumber() : Int {
    let max = 50;
    Message($"Sampling a random number between 0 and {max}: ");
    return SampleRandomNumberInRange(max);
}
```

3.3.4 Entrelazamiento

Este ejemplo nos servirá para demostrar y entender el entrelazamiento cuántico. Para ello introduciremos como parámetros `count =1000` (número de veces que probaremos el entrelazamiento) e `intial = one` (estado inicial del primer qubit).

```
dotnet run --count 1000 --initial One
```

El programa nos devolverá el número de veces que el estado de segundo qubit tras la medida era 0, el número de veces que fue 1 y el número de veces que coincidieron los estados de ambos qubits. Los primeros dos resultados deberían ser alrededor del 50% del total y el último resultado debería ser el total, es decir, el parámetro introducido como count.

```
// Return times we saw |0>, times we saw |1>, and times measurements agr
Message("Test results (# of 0s, # of 1s, # of agreements)");
return (count-numOnes, numOnes, agree);
```

Observamos que los resultados se corresponden con lo esperado:

```
Test results (# of 0s, # of 1s, # of agreements)
(519, 481, 1000)
```

Para la implementación de este programa aplicamos una puerta de Hadamard al primer qubit (`H(q0)`) y seguidamente una puerta Control Not a ambos qubits (`CNOT(q0, q1)`)

Después realizamos las medidas (`M(q0)` `M(q1)`) y en función del resultado aumentamos los contadores de número de unos (`numOnes`) y de coincidencias entre los dos estados (`agree`).

```
H(q0);
CNOT(q0, q1);
let res = M(q0);

if M(q1) == res {
    set agree += 1;
}

// Count the number of or
if res == One {
    set numOnes += 1;
}
```

3.3.5 Algortimo de Teleportación

Este ejemplo nos servirá para demostrar y entender el algoritmo de teleportación. Para probarlo enviaremos un mensaje en forma de booleano (true/false generado aleatoriamente) que será mostrado en la variable `{sent}`, mientras que en la variable `{received}` encontraremos el mensaje recibido. En caso de que ambos mensajes coincidan obtendremos la respuesta de “`Teleportation Successful`”.

```
//@EntryPoint()
operation RunProgram () : Unit {
    for idxRun in 1 .. 8 {
        let sent = DrawRandomBool(0.5);
        let received = TeleportClassicalMessage(sent);
        Message($"Round {idxRun}: Sent {sent}, got {received}.");
        Message(sent == received ? "Teleportation successful!" | "");
    }
}
```

Primero reservamos dos qubits uno como origen (`msg`) y otro como destino (`target`).

```
use (msg, target) = (Qubit(), Qubit());
```

También debemos codificar los valores true y false como un estado de un qubit, intuitivamente true será el estado 1 y false el estado 0.

```
// Encode the message we want to send.
if (message) {
    X(msg);
}
```

A continuación, hacemos una llamada al algoritmo de teleportación con ambos qubits como parámetros:

```
Teleport(msg, target);
```

El algoritmo lo hemos implementado como otra operación siguiendo los siguientes pasos:

- Reservamos un qubit como registro (`register`).
- Aplicamos la puerta de Hadamard al registro.
- Aplicamos la puerta CNOT a los qubits de registro y destino.
- Aplicamos la puerta CNOT a los qubits de origen y registro.
- Aplicamos la puerta de Hadamard al origen.

```
operation Teleport (msg : Qubit, target : Qubit) : Unit {
    use register = Qubit();
    // Create some entanglement that we can use to send our
    H(register);
    CNOT(register, target);

    // Encode the message into the entangled pair.
    CNOT(msg, register);
    H(msg);
```

Realizamos las medidas del qubit de origen y de registro y en función del resultado aplicamos la puerta Z o X al destino.

```
// Measure the qubits to extract the classical data we need to
// decode the message by applying the corrections on
// the target qubit accordingly.
// We use MResetZ from the Microsoft.Quantum.Measurement namespac
// to reset our qubits as we go.
if (MResetZ(msg) == One) { Z(target); }
// We can also use library functions such as IsResultOne to write
// out correction steps. This is especially helpful when composin
// conditionals with other functions and operations, or with part
// application.
if (IsResultOne(MResetZ(register))) { X(target); }
```

Realizamos la medida del qubit destino y lo descodificamos a un booleano en función de si es igual a 1.

```
return MResetZ(target) == One;
```

3.4 Comparación con Silq

A pesar de ser un lenguaje de alto nivel bastante intuitivo, Silq presenta algunos avances respecto a la sintaxis de de Q#.

Por ejemplo, Q# no ofrece una forma de asignación de qubits implícita, en su lugar el programador debe utilizar un método explícito de asignación (`set`). Si queremos inicializar un array de n qubits a 0 debemos llamar n veces al método set.

```
operation Set(des:Result,q:Qubit):(){
  ... // omitted
}
operation Solve(qs:Qubit[]):(){ body{
  for (i in 0..Length(qs)-1){
    Set(Zero,qs[i]);
  }
```

En cambio, con Silq solo necesitamos utilizar el operador `:=`

```
def solve(k:!ℕ){
  qs:=0:int[k];
  ... // omitted
}
```

Otro caso sería realizar la operación X con la condición de que los qubits `qs[0]` y `qs[1]` sean 0. Con Q# tendríamos que invertir primero los dos qubits y después realizar la operación Controlled X, y una vez finalizada invertir de nuevo los qubits.

```
X(qs[0]); X(qs[1]);
(Controlled X)(qs, a[0]);
X(qs[0]); X(qs[1]);
```

En contraste con Silq, donde podemos tratar los qubits como booleanos negándolos mediante la exclamación.

```
if !qs[0] && !qs[1] {
  a[0] := X(a[0]);
}
```

4. Conclusiones

En conclusión, los lenguajes de programación cuántica están en constante desarrollo dado que muchas empresas han apostado por el desarrollo de este tipo de tecnología. Han dado acceso a simuladores y máquinas cuánticas a cualquier usuario que desee investigar en estos campos. Además, muchas de estas empresas apuestan por desarrollar software de código abierto. De esta manera, cualquier interesado puede aprender y ayudar al desarrollo de estos lenguajes de programación, lo que hace que evolucionen muy rápido y que cada vez más programadores se animen a aportar código.

La parte negativa de las herramientas que nos brindan estas empresas es que tienen un número limitado de qubits bastante bajo, en mi caso el simulador utilizado era de 30 qubits. Microsoft también ofrece un simulador de mas de 40 qubits pero para usarlo es necesario una subscripción. Si queremos ejecutar nuestros programas cuánticos en una máquina cuántica la única empresa que nos presta esa oportunidad es IBM, con un número de qubits muy reducido (<30). Por lo que, aunque la programación cuántica nos proporcione mucha más potencia de procesamiento, actualmente no podemos explotarlo al máximo.

Por otro lado, el desarrollo de estos lenguajes está muy ramificado, es decir, existen numerosos lenguajes de programación distintos, algunos válidos para varias máquinas y otros sólo válidos para la máquina que desarrolla y ofrece la empresa impulsora de dicho lenguaje. El objetivo sería lograr un lenguaje universal con el que poder ejecutar programas en cualquier máquina. Además, se pretende que el lenguaje sea del más alto nivel posible para así liberar al programador de conocer todos los detalles y especificaciones del hardware, haciéndolo más accesible para usuarios del mundo clásico.

5. Referencias bibliográficas

1. Heim, B., Soeken, M., Marshall, S. *et al.* Quantum programming languages. *Nat Rev Phys* **2,** 709–722 (2020). https://doi.org/10.1038/s42254-020-00245-7
2. E. Knill, Conventions for quantum pseudocode. Los Alamos National Laboratory technical report. (1996).
3. QCL - A Programming Language for Quantum Computers. (2014). http://tph.tuwien.ac.at/~oemer/qcl.html. https://tph.tuwien.ac.at/~oemer/qcl.html
4. Servicio Azure Quantum y kit de desarrollo de Quantum (QDK). (s. f.). https://docs.microsoft.com/. https://docs.microsoft.com/es-es/azure/quantum/install-overview-qdk
5. *VSCodium*. (s. f.). https://vscodium.com/]. https://vscodium.com/
6. DotNET. (s. f.). https://dotnet.microsoft.com. https://dotnet.microsoft.com/download
7. devkit-vscode. (s. f.). devkit-vscode. https://marketplace.visualstudio.com/items?itemName=quantum.quantum-devkit-vscode