Trabajo Práctico Especial: Flamingo

Autómatas, teoría de lenguajes y compiladores

Pedro Momesso
Marina Fuster
Enrique Tawara
Juan Bensadon

Índice

Objetivo del lenguaje

La computación cuántica es un fenómeno que permaneció en el mundo de lo estrictamente teórico desde su concepción a principios de los 1980s hasta hace unos pocos años, en la década del 2000, cuando se desarrollaron los primeros prototipos funcionales. En 2011, la empresa D-Wave anunció la llamada “primera computadora cuántica comercial”, y desde entonces se han sumado grandes empresas como IBM y Google a la carrera por la supremacía cuántica. Según una estimación reciente por la revista *ScientificAmerica*, se podría llegar a ella tan pronto como antes del fin de 2019.

Lo que nos planteamos como objetivo del proyecto fue crear un lenguaje de programación que no sólo permitiera trabajar con tipos de datos clásicos, sino incluir también registros cuánticos, junto con una serie de operadores que se les pudieran aplicar, simulando el comportamiento de una computadora cuántica y permitiendo así programar y testear circuitos o algoritmos cuánticos (sin obtener obviamente los beneficios de eficiencia de emplear una computadora cuántica real).

Tomamos como objetivo final hacer posible y demostrar la implementación en Flamingo (nuestro lenguaje) de dos circuitos cuánticos clave: el circuito que emplea el *Algoritmo de Deutsch*, y el circuito de *Teletransportación Cuántica*. Estos algoritmos, además de haber jugado un rol importante a la hora de demostrar el poder de la computación cuántica frente a la clásica, integran conceptos como interferencia y enlazamiento de Qbits, que son centrales a la computación cuántica.

Consideraciones realizadas

Para implementar la simulación en sí, decidimos utilizar Java en lugar de C, ya que es un lenguaje que nos facilita el trabajo al tratar con números complejos y operaciones matriciales, sin encontrarnos con problemas de manejo de memoria. A la par de esta decisión, resolvimos que nuestro compilador generaría o bien bytecode o código java sin compilar. Este código generado utilizará nuestra librería de simulación para las operaciones con Qbits y registros cuánticos.

Una vez decidido esto, comenzamos a trabajar en el Lexer y Parser, y a medida que avanzamos nos dimos cuenta de que, dadas las restricciones de tiempo con las que nos encontramos, sería mejor idea generar código java y que el compilador recompile y corra el código generado utilizando *javac* y *java.*

Al pensar en la gramática del lenguaje, si bien el abanico de posibilidades era amplio, optamos por atenernos a una gramática más bien familiar, con la adición de elementos cuánticos de la manera más intuitiva posible. (La definición de registros cuánticos empleando notación *Ket*, por ejemplo, “Reg myReg = |010>”, o la aplicación de operadores a un Qbit n de un registro, por ejemplo, “H(myReg n)”).

Desarrollo del TP

Empleando conocimientos previos, junto con muchos adquiridos durante el proyecto, logramos implementar antes que nada una librería de simulación para circuitos cuánticos que funcionara satisfactoriamente. Todos los circuitos que probamos funcionaban y la matemática detrás era consistente.

Comenzamos entonces a trabajar en el Lexer y el Parser. Nos decidimos en una gramática y construimos las producciones correspondientes a ella. Al mismo tiempo, creamos scripts de ejemplo para los circuitos que queríamos correr, utilizando la gramática del nuevo lenguaje. Junto con la creación de estos, comenzamos a investigar la generación de bytecode, convirtiéndolos manualmente a java y analizando las clases generadas con el comando *javap -c*.

Si bien la tarea de generación de bytecode era factible, optamos por generar código java sin compilar y compilarlo tras ser generado, ya que nos pareció más adecuado dado el tiempo del que disponíamos antes de la entrega del proyecto. Si bien esta alternativa fue lo más eficiente en tiempo, se entiende que el compilador pierde portabilidad al requerir el uso de Java Development Kit para poder llevar a cabo la compilación, programa que no se encuentra instalado en una gran parte de las computadoras. De haber optado por generación directa de bytecode se habría logrado un mayor alcance de sistemas donde operar, al ser necesario únicamente Java Virtual Machine.

Gramática de Flamingo

Las reglas de producción escritas en forma BNF son las siguientes, con “Program” el símbolo distinguido, se encuentra en el archivo de “grammar.bnf” disponible en el repositorio entregado junto con el informe.

En cuanto a la semántica de la misma, existen cuatro tipos de datos asociados a los distintos símbolos: number, string, gate y sym. El primero es utilizado, como su nombre bien lo indica, para los símbolos asociados a expresiones numéricas. El segundo es utilizado para generar código Java a medida que se parsea el archivo de entrada. El tercero es utilizado para generar código relacionado a compuertas cuánticas. El cuarto es utilizado para los símbolos que se encuentren presentes en el archivo de entrada.

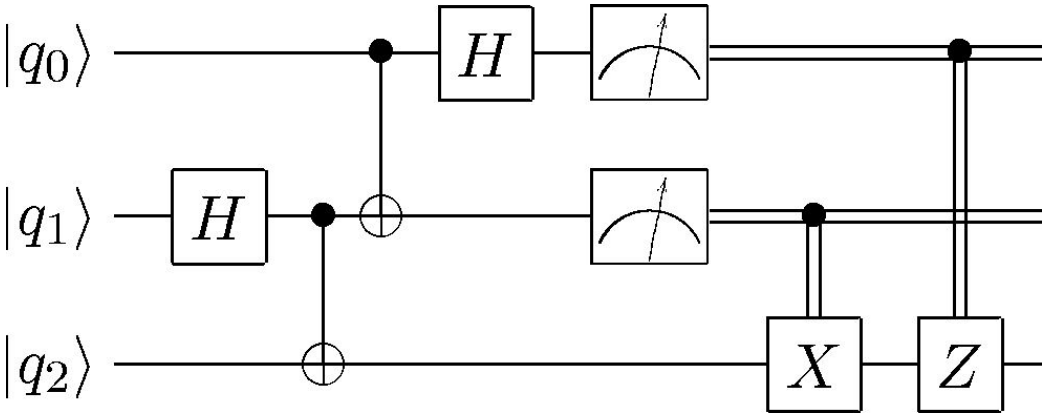
Las expresiones numéricas en la medida que no contengan símbolos presentes en la tabla se consideró oportuno evaluarlas y así el código generado para las mismas consiste en su valor pasado a texto. Para aplicar expresiones con una precedencia deseada se deben utilizar expresiones parentizadas.

En cuanto a definiciones, se optó por una gramática de definciones en donde se debe especificar el tipo de la variable a definir. Sin embargo, en la medida que se desarrollaba esta sección, se llegó a la realización que era posible factorizar una cierta parte común a estas producciones y que la misma se correspondía con una gramática de redefiniciones. A su vez, servía para definiciones en donde se infiere el tipo de la variable a definir a partir del valor a la derecha del símbolo de asignación. A este tipo de definiciones se las llamó “implícitas”. Cabe notar que por una cuestión de tiempo no se llegaron a

desarrollar definiciones implícitas para todos los tipos, sino únicamente para tipos de datos numéricos. Cabe notar que como limitación adicional de Flamingo se encuentra la imposibilidad de asignar un tipo numérico punto flotante a tipo numérico entero o tipo numérico entero a tipo numérico punto flotante. El siquiera intentar hacerlo resulta en un error semántico. La limitación se debe a falta de implementación y no por imposibilidad de realizarlo.

En cuanto a las expresiones relacionales, consideramos una expresión booleana llamada BoolExp que contempla no sólo expresiones con los operadores !, && y ||, sino también se permiten las combinaciones de las mismas con expresiones relacionales de tipos de datos Int y Float. Nótese que estas operaciones relacionales retornan un valor booleano ficticio, es decir, es un string que será insertado en el código java generado (en el cual sí corresponderá a una expresión booleana cuyo resultado será booleano).

A continuación mostraremos el circuito de teletransportación cuántica junto con su código equivalente en Flamingo:




```
Reg reg = |000>;
FLAMINGO "We want to teleport the value of the first qbit to that of the third one.";
FLAMINGO reg;

//We first generate a bell state between Qbits 1 and 2
H(reg 1);
CNOT(reg 1);

//Apply corresponding gates to Qbits 0 and 1
CNOT(reg 0);
H(reg 0);

//We apply measurements and store them in variables
Int firstMeasure = M(reg 0);
Int secondMeasure = M(reg 1);

//Based on the values of the measurements,
//we apply corrections to get the desired result
if (secondMeasure == 1){
    X(reg 2);
}

if (firstMeasure == 1){
    Z(reg 2);
}

FLAMINGO reg;
```

Decisiones tomadas en el trabajo

Respecto a los operadores de comparación, decidimos que fuera posible comparar únicamente tipos de datos Float e Integer. Dejamos afuera Strings y Registros por una cuestión de tiempo, ya que estos casos requerían un trato distintivo a la hora de generar el código Java.

Otra decisión refiere a la asignación a variables de tipo Float los tipos de datos que acepta. Decidimos que no haya casteo automático de entero a punto flotante en caso de querer asignar un Int en una variable Float.

Decidimos eliminar el tipo de dato *boolean,* introducido al comienzo de la implementación del lenguaje, dado que se logró resolver el parseo de los distintos tokens recurriendo al tipo de dato string y se consideró innecesario para la funcionalidad con la que se contaba agregar otro tipo de dato. Hay ciertas optimizaciones, sin embargo, que se podrían haber realizado de contar con un tipo boolean, ya que su uso habría permitido reemplazar directamente en el código el valor booleano resultante de la expresión.

Por último, se implementó una gramática que respetase la precedencia en cuanto a expresiones aritméticas pero no logramos completar el mismo cometido para ciertas expresiones booleanas (que incluyen *AND* y *OR*). Esto podría haberse logrado en forma similar a la precedencia lograda para las operaciones aritméticas.

Dificultades encontradas

Un desafío que se enfrentó fue el de mantener un registro, durante el parseo de los programas, de las variables ya declaradas junto con su tipo. De lo contrario, el compilador podría generar código java donde se accede a una variable que no ha sido declarada, o donde se declara una variable múltiples veces. Ésto se resolvió con la introducción de una “tabla de símbolos”, estructura cuya implementación es un hashmap, donde se lleva registro de todos los símbolos (nombres de variables) en el programa a medida que se parsea el mismo. Esta tabla es global para el programa, es decir, por diseño todas las variables tienen alcance global. Si bien esto no permite definición de alcances distintos, era lo más eficiente y simple en tiempo para lograr. La tabla se inicializa una vez inicializado el parseo del programa.

Por otra parte, la aparición de conflictos desplazamiento/reducción y reducción/reducción durante el proceso de generación del parser también presentó un desafío. Estos conflictos no fueron resueltos, ya que al hacerlo, se minimizaba la capacidad de combinar distintas expresiones booleanas y se consideró una mejor decisión dejar la funcionalidad extendida. Sin embargo y a pesar de la aparición de estos conflictos, no surgieron errores a la hora de parsear los archivos de prueba incluídos en el repositorio.

Algo que no se llegó a corregir la cantidad de *Magic Numbers* que aparecen. No es elegante ni facilita la comprensión, por lo que debe ser resuelto para mayor legibilidad y prolijidad del código ya que no será sencillo para otros programadores que quieren acceder al mismo.

Futuras extensiones

En el futuro, es perfectamente posible agregar más operadores cuánticos a la librería y al lenguaje sin mayor esfuerzo. Posiblemente de mayor importancia, sin embargo, sería otorgar al programador la posibilidad de crear sus propios operadores definiéndolos como matrices. Esto traería varias complicaciones, como incluir en el lenguaje los números complejos como tipo de dato.

También facilitaría la modularización del código permitir al programador definir y llamar funciones propias. En particular, dentro de la programación cuántica, esto representaría la posibilidad de crear Oráculos reusables en diversos circuitos.

Otra clara mejora posible para el compilador sería modificar la generación de código de manera que se genere bytecode en lugar de código java. De esta manera, el compilador no dependerá de que el equipo donde se ejecute tenga instalado el JDK, ya que las clases de la librería cuánticas se pueden precompilar.

Referencias

NIELSEN, Michael A.; CHUANG, Isaac L. (2000) *Quantum Computation and Quantum Information.* UK: Cambridge University Press.

https://en.wikipedia.org/wiki/Quantum_computing
https://www.scientificamerican.com/article/a-new-law-suggests-quantum-supremacy-could-happen-this-year/