CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS
DEL INSTITUTO POLITÉCNICO NACIONAL
DEPARTAMENTO DE COMPUTACIÓN

Simulación de Cómputo Cuántico

Tesis que presenta

Mireya Paredes López

Para obtener el grado de

Maestro en Ciencias

En la especialidad de

Computación

Director de la Tesis: **Dr. Guillermo B. Morales Luna**

México, D. F., a 10 de Enero del 2007.

Dedico esta tesis a la memoria de mi hermana "Livis".

Agradecimientos

Este logro ha sido producto del esfuerzo de muchas personas e instituciones, a quienes quiero agradecer el gran apoyo que me brindaron durante los últimos años.

Quiero expresar mi agradecimiento al CINVESTAV y muy particularmente a todos los investigadores del ahora Departamento de Computación. Primero por darme la oportunidad de colaborar con ustedes y segundo por haberme puesto los retos necesarios para demostrar mi capacidad.

Sin el apoyo económico del Consejo Nacional de Ciencia y Tecnología (CONACyT) no me hubiera sido posible dedicarme de tiempo completo a mis estudios de maestría, por lo que le agradezco el gran interés en formar profesionistas que como yo están interesados en hacer de México un país cada vez mejor.

Otro gran apoyo económico que recibí fue por parte del Consejo Mexiquense de Ciencia y Tecnología (COMECyT) mediante la Beca-Tesis promoción 2006. Con esta beca pude completar mis estudios, por lo que agradezco plenamente al Estado de México por apoyar y financiar el avance tecnológico de la entidad.

Sin duda alguna, agradezco la guía de mi asesor Guillermo Morales. Sus consejos, observaciones y pláticas amenas me hicieron conocer la gran persona que es.

Una persona a quien debo agradecerle su colaboración en esta tesis es al Dr. Carlos Rentería, profesor de ESFM, quien me ayudó a encontrar una solución a un problema importante en este trabajo (toma de mediciones). Sus conocimientos y habilidades con las matemáticas fueron la clave principal.

Agradezco todo el apoyo incondicional a Amilcar, quien siempre estuvo al tanto del trabajo. Los días que me tuve que levantar temprano, él lo hizo, los días que me tuve que quedar tarde, él se quedó tarde. Su motivación constante fue lo que me mantuvo firme en todo momento.

Un especial agradecimiento al excelente profesor Rafael Baquero, quien con toda la paciencia del mundo me enseñó una nueva forma de ver la vida. Espero en el futuro fungir como su servidora y con un granito de arena poder ayudar a materializar sus sueños.

Quiero nombrar a mis amigos Noel Ramírez, mejor conocido como el "troll", e Israel Vite, a quienes admiro y respeto por la paciencia que me han tenido y quienes además sé que no son compañeros de viaje sino amigos para toda la vida.

Agradezco a todos mis compañeros de viaje por su amistad y compañerismo, quienes hicieron más placentera mi estancia en el CINVESTAV, especialmente a Edna, a Gildardo, a Jorge, a Francisco, a Leonor y a Héctor. Todavía recuerdo las horas de tensión para pasar cierta materia.

Indiscutiblemente Sofia es la secretaria más simpática y eficiente que he conocido, por ello, quiero agradecerle todo el soporte que me brindó durante mi estancia.

Definitivamente quiero agradecer a mis padres, quienes no siempre estuvieron de acuerdo con mi decisión, pero aún así siempre la respetaron. La presencia de ellos y la de mis hermanos siempre fue suficiente para continuar con el siguiente día.

Finalmente, quiero mencionar a mi gran amigo, compañero y amante Mario Augusto, a quien le debo todo.

Índice general

Índice de figuras

Índice de tablas

Lista de Algoritmos

Resumen

La computación cuántica es un nuevo paradigma de cómputo. Esta nueva área es interesante debido a la propiedad de *paralelismo implícito*, concepto que se desprende de los orígenes de la mecánica cuántica. Esta propiedad permite codificar una cantidad exponencial de información mediante un sistema cuántico de varios qubits. En la actualidad los simuladores cuánticos constituyen una forma de hacer investigación en esta área, y los hay tanto físicos como formales (software). Además, existe gran interés en probar y diseñar nuevos algoritmos cuánticos que reduzcan la complejidad del problema que resuelven en una complejidad polinomial.

Presentamos *GAMA* el cual es un nuevo lenguaje de programación para llevar a cabo la simulación de algoritmos cuánticos. Este lenguaje es de tipo imperativo, estructurado y de propósito específico. *GAMA* cuenta con un módulo de depuración que permite llevar la secuencia de los diferentes cambios de los estados de las variables en las diferentes fases del algoritmo.

Este ambiente de simulación cuántica está enfocado principalmente al aprendizaje y de esta manera contribuye al área de computación cuántica. Como parte de la efectividad del simulador, probamos el algoritmo de Deutsch-Jozsa.

Abstract

Quantum computing is a new paradigm of Computation. This area is interesting due to its inherent property called "quantum paralellism", with origin in Quantum Mechanics. This property allows to codify an exponential quantity of information within a linear number of qubits. At present time quantum simulators provide great support in this area, either in physical (hardware) or formal (software) forms. Also, many people have been interested in probing and designing new quantum algorithms in order to diminish the time complexity of the algorithms aimed to solve intractable problems even to a polynomial order.

We introduce *GAMA* which is a new programming language to simulate quantum algorithms. It is imperative, structured with a specific purpose. *GAMA* has a debugging module allowing to track the evolution of all variables appearing in an algorithm.

This environment for quantum simulation is focused mainly in teaching and learning to illustrate quantum computing. In order to prove the simulator's correcteness, we tested it using the Deutsch-Jozsa algorithm.

Introducción

En 1982, Richard Feynman del Instituto de Tecnología de California sugirió que la construcción de computadoras cuánticas podría ayudar a entender las dificultades implícitas en la simulación de sistemas cuánticos difíciles de realizar en computadoras clásicas. En 1985 Deutsch investiga el posible poder computacional de estas computadoras físicas, definiendo la máquina de Turing cuántica y algunas propiedades de sus circuitos. Benioff, Deutsch y Feynman, basándose en la idea de que la simulación de una computadora cuántica en una clásica requiere un mayor número de operaciones, plantean que la computación cuántica posee un mayor potencial de cálculo, debido al principio de *superposición* de la Mecánica Cuántica, que hace las veces de *paralelismo cuántico* [6]. Esta propiedad permite codificar una cantidad exponencial de información mediante un sistema cuántico de varios *qubits*.

En 1994, Peter Shor presenta un algoritmo cuántico para factorizar números grandes con una complejidad polinomial [44]. Esto provocó que muchas personas se interesaran en el área. Así, mientras unos trataban de crear un dispositivo cuántico, diversos analistas trataban de encontrar nuevos algoritmos cuánticos. Desde entonces, la literatura se ha ido desenvolviendo en algoritmos cuánticos y teoría de complejidad cuántica. En 1995, Lov Grover consideró otro problema importante: el problema de canalizar una búsqueda a través de un espacio de búsqueda sin estructura; aunque éste no tuvo el éxito del de Shor, son muchas las aplicaciones de este algoritmo en técnicas de búsqueda.

En 2001, el algoritmo de la factorización de Shor fue probado experimentalmente con una computadora cuántica de 7-qubits, la cual ha sido la más poderosa que se ha construido hasta la actualidad [30].

Al día de hoy, los simuladores cuánticos constituyen una forma de hacer investigación en esta área, y los hay tanto físicos como formales (software). En particular, la línea de investigación que nos interesa es la de los lenguajes de programación para la simulación de algoritmos cuánticos que ha tenido gran auge desde el 2001.

En realidad, el avance de implementaciones físicas de las computadoras cuánticas ha sido limitado, por lo que, muchas personas pensarían que no tiene sentido proponer un lenguaje de programación sin tener hardware alguno. Daremos algunas razones de la existencia de estos lenguajes de acuerdo a la referencia [18].

- El gran progreso de los sistemas de criptografía cuántica. Algunos componentes de estos sistemas han sido comercializados y tal parece que será una tecnología importante inclusive antes de que las computadoras cuánticas de gran capacidad aparezcan.

- La amplia gama de lenguajes de programación ha causado problemas en la ingeniería de software. Para llevar a la práctica la computación fue necesario el desa-

	QCL	Q language	qGCL	(Block-)QPL
Nuevo lenguaje	√		√	√
Biblioteca estándard		√		
Lenguaje imperativo	√	√	√	
Lenguaje funcional				√
Simulador disponible	√	√		
Enfoque pragmático	√	√		
Enfoque teórico			√	√
Semántica formal			√	√
Lenguaje universal	√	√	√	√
Módulo de depuración				
Definición dinámica de Qgates	√			

Tabla 1: Lenguajes de programación cuántica.

rrollo de un estudio teórico. Desde este punto de vista, el diseño de los lenguajes de programación cuántica antes de que el hardware exista, en algunos aspectos, es una situación ideal.

- La semántica, la lógica y el enfoque teórico en los lenguajes han dado una nueva perspectiva sobre la teoría cuántica. Las personas interesadas en este tema han aunado los fundamentos de la mecánica cuántica, los cuales son invaluables inclusive si no llegaran a existir las computadoras cuánticas.

En la tabla 1 enunciamos los lenguajes de programación cuántica más mencionados en la literatura de acuerdo a las referencias [39, 18]. Además, presentamos sus características principales. Aunque los simuladores conforman otra forma de investigación, los hemos dejado de lado para concentrarnos únicamente en los lenguajes de programación. Sin embargo, una buena referencia que lista tanto simuladores como lenguajes es [19].

- 1996 *Q-gol* por Greg Baker [5]. Éste fue el primer intento de diseño e implementación de lenguage de programación cuántica; sin embargo, no es conciso y no es universal, es decir, no es posible implementar y simular todos los algoritmos cuánticos conocidos. Sin embargo, algunas ideas significativas acerca del control de los estados cuánticos.

- 1998, 2000, 2001, 2002, 2003 QCL por Bernhard Ömer [35]. Éste es el lenguaje más avanzado de programación cuántica que ha sido implementado hasta la actualidad. Su sintaxis es parecida a la de C. Además, cuenta con su propio simulador de algoritmos cuánticos y tiene funciones muy específicas como: Manejo de memoria y derivación automática de operadores condicionales.

- 2002 *Q Language* por Stefano Bettelli [7]. Éste se trata de un lenguaje de alto nivel sobre C++ y en una colección de operadores primitivos. Estos operadores se apegan al modelo QRAM.

- 2001 qGCL por Paolo Zuliani [47]. Este lenguaje es una extensión del lenguaje *guarded command language* de Dijkstra (GCL) y está expresado en términos probabilistas. Es un lenguaje imperativo y tiene un alto nivel de notación matemática; además cuenta con un mecanismo por paso para el refinamiento de la verificación del programa. Tiene una semántica formal e incluye tres tipos de primitivas cuánticas de alto nivel: preparación de estados, evolución y observación. Es un lenguaje universal aunque no se ha implementado.

- 2004 QPL por Peter Selinger [43]. En este artículo se describe la sintaxis y la semántica de un lenguaje de programación cuántico con características de alto nivel como *loops*, procedimientos recursivos y tipos de datos estructurados. El lenguaje es funcional por naturaleza y tiene una semántica denotacional interesante con respecto a la asignación de operadores a cada fragmento del programa. Se puede representar el programa cuántico por medio de diagramas o bien de texto.

- 2005 QML por Thorsten Altenkirch y Jonathan Grattage [20]. Éste se trata de un lenguaje de programación funcional de tipos finitos. Los programas QML son interpretados por morfismos en la categoría **FQC** (Finite Quantum Computations), la cual provee una semántica constructiva de computaciones cuánticas irreversibles como las compuertas cuánticas. QML integra reversibilidad e irreversibilidad en un lenguaje.

Es importante para la comunidad apreciar este nuevo paradigma de computación, ya que existen cambios muy radicales en la forma de pensar acerca de computación, programación y complejidad. A medida que se vayan entendiendo mejor los algoritmos cuánticos es como surgirán nuevas ideas para diseñar nuevos algoritmos que obtengan soluciones con complejidades polinomiales en términos de primitivos cuánticos a problemas con complejidades exponenciales o bien NP difíciles.

La creación de un lenguaje de programación de alto nivel de tipo imperativo y estructurado (GAMA) nos dará otra visión de la computación cuántica, es decir, será más simple trabajar con conceptos abstractos por naturaleza. Este lenguaje será con fines prácticos, por lo que es necesario la adaptación de un módulo de depuración capaz de visualizar los cambios en los estados del algoritmo. Dando origen a un lenguaje de programación cuántica con nuevas propiedades, incorporando y aprovechando las mejores características y exprerienciias de los lenguajes anteriormente expuestos. Además, con ésto obtendremos una herramienta propia, con facilidad de uso y con una manipulación total de la herramienta, lo cual facilitaría posibles trabajos futuros en el tema.

El objetivo general que nos hemos planteado en el desarrollo de esta tesis es realizar un prototipo para la *simulación cuántica*. Se ha de permitir el uso de cualesquiera *compuertas cuánticas* (*quantum gates*), y del operador de *toma de mediciones* (*measurement of qubits*). Esto con el fin de visualizar en una computadora el funcionamiento de un algoritmo cuántico.

Para ello, describiremos el diseño de nuestro lenguaje de programación cuántica, el cual ha de consistir de conectivos básicos de concatenación, condicionales y de iteración de procesos, así como de definición de operadores primitivos y funciones básicas como asignación de variables y pruebas de enunciados booleanos; el desarrollo de un

intérprete para su ejecución y el de un módulo de depuración de las variables involucradas en un algoritmo. Y finalmente, comprobaremos la efectividad de la herramienta mediante la ejecución de un algoritmo cuántico.

Este documento está dividido en seis capítulos. En el capítulo 1 presentamos los antecedentes que dieron origen a la computación cuántica, así como sus fundamentos. En el capítulo 2 detallamos el diseño del lenguaje de programación cuántica: GAMA. En los capítulos 3 y 4 describimos el desarrollo tanto de su intérprete de ejecución como de su entorno de simulación. En el capítulo 5 realizamos las pruebas de efectividad del simulador mediante el algoritmo de Deustch-Jozsa. Finalmente, en el capítulo 6 mostramos un compendio de las conclusiones y observaciones hechas durante el desarrollo de esta tesis.

Capítulo 1

Un vistazo a la computación cuántica

En la computación cuántica existen diversos conceptos fundamentales entre los que se cuenta el *qubit* (unidad básica en el cómputo cuántico). En este capítulo los describimos, con el fin de dar una perspectiva general de éstos. Si se desea una explicación más detallada recomendamos consultar las referencias [32, 37].

En la sección 1.1 narramos los sucesos principales que se han ido desenvolviendo en la historia de la computación cuántica. En la sección 1.2 describimos los conceptos fundamentales tales como los bits cuánticos (sección 1.2.1), los registros cuánticos (sección 1.2.2), las compuertas cuánticas (sección 1.2.4), el mecanismo para la toma de mediciones (sección 1.2.3), los algoritmos cuánticos (sección 1.2.5)y el paralelismo cuántico 1.2.6. Además, presentamos un caso particular de un algoritmo cuántico, el de Deutsch-Jozsa para decidir si una función es balanceada o constante (sección 1.3).

1.1. Antecedentes

A finales del siglo XIX fueron descubiertos nuevos fenómenos que no correspondían al régimen de las teorías físicas de esa época. La existencia de una catástrofe ultravioleta con energías finitas y la de una espiral de electrones dentro del núcleo de un átomo son algunos ejemplos de estos fenómenos. La explicación inicial a estos fenómenos consistió en agregar nuevas hipótesis a la teoría física. Sin embargo, cada vez que se tenía un mejor entendimiento del átomo y de la radiación se tornaban confusas.

Posteriormente, la física teórica obtuvo dos importantes conclusiones: el carácter dual de la radiación electromagnética y la existencia de valores discretos para cantidades físicas. Estas conclusiones permitireron a Planck, Einstein, Bohr y otros desarrollar la teoría cuántica. En 1925 Heisenberg y en 1926 Schrödinger y Dirac formularon la mecánica cuántica, donde están incluidas las teorías cuánticas precedentes. La mecánica cuántica es considerada un conjunto de reglas matemáticas aplicadas a la descripción de nuevas teorías físicas. Entre estas se cuenta la física de materia condensada, electrodinámica cuántica y teoría cuántica de campos, de donde se desprenden diversas e importantes aplicaciones. Aunque estas teorías son simples, algunos físicos las encuentran poco intuitivas. De hecho, Albert Einstein fue uno de los principales criticos de estas teorías. Desde sus inicios, la mecánica cuántica ha sido un área de constante investigación para ayudar a entender y esclarecer sus argumentos. La simulación de

sistemas cuánticos [1] es una de las herramientas más utilizadas en esta investigación. Por ello, desde los 70 se han desarrollado diversas técnicas para obtener un control completo en estos sistemas. Por ejemplo, los métodos para atrapar partículas en una "trampa atómica" ha servido para probar diferentes aspectos en su comportamiento con gran precisión. El interés sobre este tipo de sistemas radica principalmente en la posible generación de ciencia a partir del desarrollo de métodos para probar nuevos regímenes de la naturaleza, como ha ocurrido anteriormente. Un ejemplo de ello es la invención de la Radioastronomía que de 1930 a 1940 dió pauta a una espectacular secuencia de descubrimientos, incluyendo entre ellos, el centro galáctico de la Via Láctea, en la constelación de Sagitario.

En 1982, Feynman sugirió que la construcción de computadoras cuánticas podría ayudar a entender las dificultades implícitas en la simulación de estos sistemas cuánticos. Idea que lleva a Deutsch al planteamiento de un nuevo modelo computacional basado en las teorías de la mecánica cuántica, capaz de procesar información de origen cuántico. Esta información está basada en la naturaleza cuántica de las partículas elementales, donde la unidad básica es el *qubit*. El qubit consiste en un sistema cuántico de dos estados y representa la superposición de estos. Las ideas de Feynman y el trabajo de Deutsch dieron origen a la computación cuántica. Ésta tiene elementos tales como el bit cuántico, los registros cuánticos y las compuertas cuánticas. Desde sus inicios la computación cuántica ha generado gran expectación debido al *paralelismo implícito* detallado en la sección 1.2.6. Esta propiedad implicaría tener una velocidad de procesamiento superior a la de los dispositivos clásicos. Sin embargo, a pesar del gran esfuerzo invertido en la creación de sistemas procesadores de información cuántica, éste ha arrojado resultados simples. Un ejemplo de ello son los prototipos de computadoras cuánticas capaces de realizar docenas de operaciones con unos pocos bits cuánticos y esto es lo que conforma el estado del arte del área [30]. De cualquier manera, ésto muestra uno de los grandes retos del siglo XXI, donde el desarrollo de nuevas técnicas para la manipulación de sistemas cuánticos permita llevar a la práctica los beneficios del procesamiento cuántico de grandes cantidades de información.

Por otro lado, la ciencia de la computación fue uno de los grandes acontecimientos del siglo XX. Sin embargo, la noción de procedimiento se puede remontar a algunas tablas cuneiformes por la época de Hammurabi, rey de Babilonia (1792 - 1750 a.C.), a los elementos de Euclides (siglo IV a. C.), a la aritmética romana (siglo I d. C.), a Cardano (siglo XVI), a Descartes (siglo XVII), a Pascal (siglo XVII) ó Peano (siglo XIX). En 1936, Alan Turing introdujo un modelo computacional en su trabajo *"On computable numbers, with an application to the Entscheidungsproblem"* [31], que en la actualidad se llama máquina de Turing, en el cual se estudió el problema planteado por David Hilbert sobre la decibilidad de las Matemáticas. Con este modelo, Turing comprobó que existían problemas que una máquina no podía resolver y además mostró la existencia de una *Máquina Universal de Turing* que podía simular cualquier otra máquina de Turing, con lo cual formalizó el concepto de *algoritmo*. La tesis de Church-Turing es considerada el primer axioma de la computación, permitiendo estudiar problemas resolubles mediante un algoritmo.

En 1952, el matemático John von Neumann, muy conocido por ser aportador y revo-

[1] Un sistema cuántico es una entidad formada por partículas que interactúan entre sí a escalas microscópicas.

lucionario de la computación, incorporó un nuevo paradigma de computación electrónica basado en las ideas de Turing, diseño conocido como *Arquitectura de von Neumann*, la cual ha sido base de las computadoras físicas desde entonces. Con la invención del *transistor* se logró la miniaturización de las computadoras y la posibilidad de la producción masiva de computadoras personales. De acuerdo a la ley empírica de Gordon Moore (1965), el número de transistores en una pieza de silicio se duplica cada 18 meses aproximadamente. Increíblemente esta aproximación se ha apegado a la realidad desde 1960, y se prevé que para el año 2017 la miniaturización llegará a su límite debido a que los dispositivos lógicos alcanzarán el tamaño atómico o molecular. Para asumir este inconveniente, habrá que estudiar un nuevo paradigma de computación. Y es ahí, donde radica la importancia del estudio de la teoría de la computación cuántica.

Un algoritmo es eficiente si es ejecutable en un tiempo polinomial. Por otro lado, uno ineficiente es aquel que su ejecución ocupa un tiempo superpolinomial (la mayoría de las veces es exponencial). En los 60 se creía que la máquina de Turing era al menos tan poderosa como cualquier otro modelo de computación, es decir, cualquier otro modelo se reduce eficientemente con el modelo de Turing. Una versión de la tesis de Church-Turing enuncia: *"cualquier proceso algorítmico puede ser simulado eficientemente usando una máquina de Turing"*. Si se acepta esta tesis entonces no importa en cual tipo de máquina se ejecute el algoritmo, ese modelo puede ser simulado eficientemente usando una máquina de Turing estándar. Sin embargo, se ha demostrado que las computadoras analógicas pueden resolver problemas eficientemente sin que exista una que pueda ser simulada de manera eficiente por una máquina de Turing.

En 1970, Robert Solovay y Valker Strassen inventaron un método para revisar si un número es primo o compuesto, usando un algoritmo probabilista (no determinista). Este algoritmo determinó si un número era primo o compuesto después de un número determinado de repeticiones, con cierta certeza. Esta prueba de primalidad fue de gran importancia, debido a que era la primera prueba no determinista, sin embargo, los algoritmos probabilistas contradecían la tesis de Church y Turing, ya que sugerían que hay problemas con una solución eficiente que no pueden ser resueltos en una máquina de Turing. Ésta fue una de las razones por las que Deutsch (1985) se interesa en encontrar un modelo de computación que garantice la simulación eficiente de cualquier otro y busca en la teoría de la física, encontrar un fundamento que valide su tesis. Basado en un dispositivo con orígenes en la mecánica cuántica define una *Máquina Universal Cuántica*, aunque en la actualidad se desconoce si este modelo será suficiente para cumplir la tesis de Church y Turing.

Deutsch además se preguntaba si este nuevo modelo era capaz de resolver problemas eficientemente, incluso con una máquina de Turing probabilista. Este hecho atrajo el interés de mucha gente en la década subsecuente y culminó en las soluciones de Peter Shor (1994) a dos problemas muy importantes: el problema de la factorización de números grandes y el problema del logaritmo discreto; los cuales podrían ser resueltos eficientemente en una computadora cuántica.

En 1995, Lov Grover consideró otro problema importante: el problema de canalizar una búsqueda a través de un espacio de búsqueda sin estructura; aunque éste no tuvo el éxito del de Shor, son muchas las aplicaciones de este algoritmo en técnicas de búsqueda.

1.2. Conceptos generales

En el *espacio de estados* de un sistema cuántico existen conceptos tales como: posición, momento, polarizaciones, *spin*, partícula, los cuales son modelados por un espacio de Hilbert de funciones de onda [40]. En la computación cuántica es suficiente trabajar con sistemas cuánticos finitos y tratar con espacios vectoriales complejos de dimensión finita.

1.2.1. Bits cuánticos

En el modelo de computación clásica la unidad básica de información es el *bit*, el cual asume uno de dos posibles valores, ya sea 0 o 1. Análogamente, el modelo de computación cuántica tiene como unidad básica el *qubit*, que puede asumir una superposición de dos valores, o estados, distintos, digamos $|0\rangle$ o $|1\rangle$, los cuales representan los dos valores de un bit clásico.

Antes de entrar en más detalle, describiremos la notación que será utilizada en todo el texto y que además es convencional en mecánica cuántica para la descripción de estados y transformaciones cuánticas. El espacio de estados cuánticos y las transformaciones que lo operan pueden ser descritos en términos de vectores y matrices o en la compacta notación de Dirac [1958]. Los *kets* como $|x\rangle$ denotan vectores columna y son utilizados típicamente para representar estados cuánticos. La pareja *bra*, $\langle x|$, denota el conjugado transpuesto de $|x\rangle$. Combinando $\langle x|$ y $|y\rangle$, lo cual quedaría $\langle x||y\rangle$, también se puede escribir $\langle x|y\rangle$, denota el producto interior de dos vectores. Por el contrario, la notación $|x\rangle\langle y|$ es el producto exterior de $|x\rangle$ y $\langle y|$. Por ejemplo, si por $|0\rangle$ se tiene a un vector unitario, entonces se cumple que $\langle 0|0\rangle = 1$ y además si $|1\rangle$ es otro vector unitario ortogonal a $|0\rangle$, se cumple que $\langle 0|1\rangle = 0$. Particularmente, si los vectores $|0\rangle$ y $\langle 1|$ corresponden a la base canónica en $\mathbb{C}^2$, el producto exterior $|0\rangle\langle 1|$, también puede ser representado en forma matricial:

$$|0\rangle\langle 1| = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}.$$

Esta notación tiene la ventaja de poder especificar las transformaciones cuánticas (ver Sección 1.2.4) sobre estados cuánticos en términos de la base canónica. Por ejemplo, la transformación que cambia los estados $|0\rangle$ y $|1\rangle$ está dada por la matriz

$$X = |0\rangle\langle 1| + |1\rangle\langle 0|$$

o lo que es lo mismo, pero más intuitivo:

$$\begin{array}{rccc} X: & |0\rangle & \mapsto & |1\rangle \\ & |1\rangle & \mapsto & |0\rangle, \end{array}$$

lo cual especifica de manera explícita el resultado de la transformación sobre los vectores base.

En todo momento la superposición de valores asumidos por un *qubit* puede ser considerada como una combinación lineal de $|0\rangle$ y $|1\rangle$, como $a|0\rangle + b|1\rangle$, donde a y b son números complejos y cumplen con $|a|^2+|b|^2 = 1$, lo cual conlleva a un manejo simbólico propio de espacios vectoriales complejos para una base particular, de hecho en *espacios*

de Hilbert. En cada instante de cómputo un *qubit* asume un *estado intermedio* [32], el cual es un vector en el espacio vectorial complejo generado por los dos estados $|0\rangle$ y $|1\rangle$. El espacio de Hilbert complejo de dimensión 2, generado por $\beta = [|0\rangle, |1\rangle]$, tiene a β como una base ortonormal y cada uno de sus vectores unitarios es precisamente un estado intermedio.

El mecanismo de *medición* (ver Sección 1.2.3) de qubits se hace con respecto a la base, en este caso β, la probabilidad de que el valor medido sea $|0\rangle$ es $|a|^2$ y la probabilidad de que el valor medido sea $|1\rangle$ es $|b|^2$.

Aunque un qubit puede asumir una superposición infinita de estados, sólo es posible extrar un simple bit clásico y esto se hace mediante el proceso de medición. Cuando un qubit es medido, éste asume un solo estado de la base vectorial. Una vez que es medido un qubit, no puede ser medido una vez más.

1.2.2. Registros cuánticos

Pensemos en un objeto macroscópico dividido en piezas, digamos un rompecabezas gigante, donde cada una de sus piezas se encuentra volando en el espacio. El estado de este sistema puede ser descrito completamente con la explicación de cada una de sus piezas por separado. Sin embargo, no siempre se cumple ésto.

Como vimos un qubit puede ser representado por un vector en un espacio vectorial complejo de dos dimensiones. Un sistema cuántico de $k-$*qubits*, donde $k \in \mathbb{N}$, cuenta con un espacio de estados de 2^k dimensiones[2].

Así como varios bits pueden ser concatenados para formar un registro, se tiene la noción de *quregistro* o bien k*-qubit*, donde k es el número de bits.

Dado un entero $k \geq 1$, denotemos por n a la k-ésima potencia de 2, $n = 2^k$. Entonces cualquier vector de la forma ψ con entradas complejas y $\sum_{i=0}^{n-1} |\mu_i|^2 = 1$ es un *quregistro*, donde $\mu_i \in \mathbb{C}$.

$$\psi = \sum_{i=0}^{n-1} \mu_{i_2} |i_2\rangle = \mu_{00\ldots0}|00\ldots0\rangle + \ldots + \mu_{11\ldots1}|11\ldots1\rangle \tag{1.1}$$

La relación entre k y n es lo que determina el poder real de la computación cuántica: un *quregistro* puede asumir un número exponencial de estados, es decir, puede estar en una superposición de $2^n - 1$ posibles valores. Así tenemos que la concatenación de k *qubits*, produce vectores de dimensión 2^k. Los estados cuánticos se combinan a través del *producto tensorial* ($\otimes$) y de hecho el producto tensorial de k qubits es un *quregistro*. A continuación veremos la definición del produto tensorial para entender un poco más sobre computación cuántica.

Sean V y W dos espacios de vectores complejos con bases $\{v_1, v_2\}$ y $\{w_1, w_2\}$ respectivamente. El producto tensorial $V \otimes W$ tiene como base $\{v_1 \otimes w_1, v_1 \otimes w_2, v_2 \otimes w_1, v_2 \otimes w_2\}$. El espacio de estados para dos qubits, cada uno con base $\{|0\rangle, |1\rangle\}$, tiene la base $\{|0\rangle \otimes |0\rangle, |0\rangle \otimes |1\rangle, |1\rangle \otimes |0\rangle, |1\rangle \otimes |1\rangle\}$, o también $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. De manera general, podemos escribir $|x\rangle$, lo cual significa $|b_k b_{k-1} \ldots b_0\rangle$ donde b_i son dígitos binarios del número x. El producto tensorial $X \otimes Y$ tiene como dimensión $dim(X) * dim(Y)$.

[2]El espacio de estados es el conjunto de vectores normalizados en un espacio de dimensión 2^k, como el estado $a|0\rangle + b|1\rangle$ de un qubit es normalizado como $|a|^2 + |b|^2 = 1$.

Como se mencionó anteriormente, existen estados cuánticos que no pueden ser descritos en términos del estado de cada uno de sus componentes por separado. Es decir, no se pueden hallar los coeficientes y estados que factoricen dicho estado cuántico. Por ejemplo, en el estado $|00\rangle + |11\rangle$, no se pueden encontrar los coeficientes a_1, a_2, b_1, b_2, tal que, $(a_1|0\rangle + b_1|1\rangle) \otimes (a_2|0\rangle + b_2|1\rangle) = |00\rangle + |11\rangle$, ya que no es factorizable. Los estados que no pueden ser descompuestos de esta manera son llamados *entrelazados*.

El origen del poder del cómputo cuántico se deriva a partir de la posibilidad de explotar la evolución de las configuraciones de estados como un mecanismo computacional.

1.2.3. Toma de mediciones

La *medición de qubits* constituye otra gran diferencia entre el modelo clásico y el modelo cuántico, puesto que se puede medir (leer) el valor que tiene un bit, sin embargo, no se puede leer el valor de un qubit sin alterarlo, ya que éste se convierte en un estado determinista. La medición de un *qubit* proporciona, de manera probabilista un solo estado, ya sea $|0\rangle$ o bien $|1\rangle$. Dado el qubit

$$\psi = \mu_1|0\rangle + \mu_2|1\rangle \tag{1.2}$$

donde μ_1, $\mu_2 \in \mathbb{C}$, $|\mu_1|^2 + |\mu_2|^2 = 1$, entonces la medición proporcionará el estado $|0\rangle$ con probabilidad $|\mu_1|^2$ o $|1\rangle$ con probabilidad $|\mu_2|^2$.

Veamos un ejemplo para la medición de un sistema 2-qubit. Cualquier estado 2-qubit puede ser expresado de la forma $a|00\rangle + a|01\rangle + a|10\rangle + a|11\rangle$, donde a, b, c y d $\in \mathbb{C}$ y por lo tanto se cumple $|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$. Si queremos medir el primer qubit habrá que representar el 2-qubit como un producto tensorial de 2 qubits (si fuese factorizable), es decir, $(a_{00}|0\rangle + a_{01}|1\rangle) \otimes (a_{10}|0\rangle + a_{11}|1\rangle) \otimes \ldots \otimes (a_{n-1,0}|0\rangle + a_{n-1,1}|1\rangle$, tal que, $|a_{j0}|^2 + |a_{j1}|^2 = 1$. Dado $j = 0 \in \{0, \ldots, n-1\}$, procederemos a medir el primer qubit, es decir, a determinar su nuevo estado. En este caso, el primer qubit es $(a_{00}|0\rangle + a_{01}|1\rangle)$ donde la medición proporcionará el estado $|0\rangle$ con probabilidad $|a_{00}|^2$ o $|1\rangle$ con probabilidad $|a_{01}|^2$. Finalmente si suponemos que el resultado de la medición del primer qubit fue $|1\rangle$, entonces la medición de ese 2-qubit quedaría:

$$|1\rangle \otimes (a_{10}|0\rangle + a_{11}|1\rangle) \otimes \ldots \otimes (a_{n-1,0}|0\rangle + a_{n-1,1}|1\rangle$$

Cuando un quregistro se puede factorizar en k qubits, entonces la medida no tiene efecto sobre los otros qubits, únicamente sobre el cual se realizar la medición. Por otro lado, cuando un quregistro es entrelazado, se dice que la medición de un qubit afecta las mediciones de los qubits restantes, lo cual está directamente relacionado con la definición previa, en donde mencionamos que los estados entrelazados no pueden ser escritos como un producto tensorial de estados individuales.

1.2.4. Compuertas cuánticas

Así como una computadora clásica utiliza circuitos lógicos que permiten la manipulación de información mediante compuertas, en el cómputo cuántico existen mecanismos para manipular *quregistros*. El análogo a los circuitos lógicos son las *compuertas cuánticas* o bien *qucompuertas*. Las compuertas cuánticas y las composiciones de ellas

transforman linealmente un estado inicial ψ_1 en un estado final ψ_2, conservando siempre las normas de los vectores, es decir, su ortogonalidad. Cualquier transformación lineal en un espacio vectorial complejo puede ser descrita por una matriz. Sea A^* la conjugada transpuesta de la matriz A. Una matriz M es unitaria si se cumple que $AA^* = I$. Las compuertas son *transformaciones unitarias* en espacios de Hilbert [25].

Dado este hecho, una de las consecuencias más importantes de las compuertas cuánticas es que son *reversibles*. Para una mayor referencia sobre reversibilidad de compuertas véase [16].

Transformaciones lineales sobre qubits

Para entender mejor el concepto de compuerta cuántica, veremos algunos ejemplos muy sencillos que operan sobre qubits. A continuación, mostramos las matrices de Pauli, las cuales pueden generar cualquier transformación lineal de dimensión 2, a partir de una combinación lineal de ellas con coeficientes complejos, en la primera columna aparece cada matriz y en la segunda la transformación lineal correspondiente:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad I: \begin{array}{ccc} |0\rangle & \mapsto & |0\rangle \\ |1\rangle & \mapsto & |1\rangle \end{array}$$

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad X: \begin{array}{ccc} |0\rangle & \mapsto & |1\rangle \\ |1\rangle & \mapsto & |0\rangle \end{array}$$

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad Z: \begin{array}{ccc} |0\rangle & \mapsto & |0\rangle \\ |1\rangle & \mapsto & -|1\rangle \end{array}$$

$$Y = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \quad Y: \begin{array}{ccc} |0\rangle & \mapsto & -|1\rangle \\ |1\rangle & \mapsto & |0\rangle \end{array}$$

Estas transformaciones tienen un nombre convencional. I es la transformación identidad, X es la negación, $Y = ZX$ es la combinación de Z y de X, y Z es un corrimiento de fase. Cualquiera de éstas cumple la propiedad de ser transformaciones unitarias.

Otra compuerta muy común es la compuerta C_{not} (*Controlled-NOT gate*), la cual aplica la compuerta de negación al segundo qubit siempre que el primero se encuentre en estado $|1\rangle$,

$$\text{Cnot}: \begin{array}{ccc} |0x\rangle & \rightarrow & |0\rangle \otimes |x\rangle \\ |1x\rangle & \rightarrow & |1\rangle \otimes X|x\rangle, \end{array}$$

la cual tiene como matriz

$$\text{Cnot} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

es decir, $\text{Cnot} = \begin{pmatrix} I & 0 \\ 0 & X \end{pmatrix}$ donde X es la compuerta de negación

$$X: \begin{array}{ccc} |0\rangle & \rightarrow & |1\rangle \\ |1\rangle & \rightarrow & |0\rangle \end{array}$$

En la literatura el primer qubit es conocido como bit de control y el segundo como un qubit de salida o bien *target*. La transformación *Cnot* es unitaria, ya que se cumple que $C^*_{not} = C_{not}$ y además $C_{not}C_{not} = I$. Esta compuerta no puede ser descompuesta en el producto tensorial de dos transformaciones a bits.

Otra transformación importante que opera sobre un bit es la compuerta de *Hadamard* definida por

$$\mathrm{H}: \begin{array}{lcl} |0\rangle & \rightarrow & \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \\ |1\rangle & \rightarrow & \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle). \end{array}$$

La compuerta *H* tiene importantes aplicaciones. Cuando ésta es aplicada a $|0\rangle$, *H* crea una superposición del estado $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Aplicada a n bits genera una superposición de 2^n posibles estados.

$$\begin{array}{rcl} (H \otimes H \otimes \ldots \otimes H)|00\ldots0\rangle & = & \frac{1}{\sqrt{2^n}}((|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle) \otimes \ldots \otimes (|0\rangle + |1\rangle)) \\ & = & \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle. \end{array}$$

Esta última transformación, la que se aplica a n bits es llamada *Walsh*, o *Walsh-Hadamard*, transformación *W*. Y puede ser definida como una descomposición recursiva de la forma:

$$W_1 = H, W_{n+1} = H \otimes W_n.$$

Por otro lado, en la computación clásica se puede realizar la copia de información sin ningún problema, sin embargo, en la computación cuántica la accesabilidad a la información tiene una limitante. De acuerdo al Teorema 1, la mecánica cuántica no permite la copia exacta de estados cuánticos desconocidos, lo que implica tener ciertas limitaciones en la aproximación de copias.

Teorema 1 (No clonación) *De acuerdo a las propiedades unitarias de la mecánica cuántica no es posible hacer una copia o clonación de un estado cuántico.*

Probaremos un ejemplo muy sencillo utilizando la linealidad de las transformaciones unitarias.

Supongamos que U es una transformación unitaria cuya función es clonar estados, es decir $U(|a0\rangle) = |aa\rangle$ cualquiera que sea $|a\rangle$. Si en particular se considera un par de estados cuánticos ortogonales, digamos $|a\rangle$ y $|b\rangle$, para el estado $|c\rangle = \frac{1}{\sqrt{2}}(|a\rangle + |b\rangle)$ resulta, por un lado

$$U(|c\rangle) = U(\tfrac{1}{\sqrt{2}}(|a0\rangle + |b0\rangle)) = \tfrac{1}{\sqrt{2}}(U(|a0\rangle) + U(|b0\rangle)) = \tfrac{1}{\sqrt{2}}(|aa\rangle + |bb\rangle),$$

y por otro lado, siendo U de clonación

$$U(|c\rangle) = |cc\rangle = (|a\rangle + |b\rangle) \otimes (|a\rangle + |b\rangle) = \tfrac{1}{\sqrt{2}}(|aa\rangle + |bb\rangle)$$

lo que implica $\frac{1}{\sqrt{2}}(|ab\rangle + |ba\rangle) = 0$, lo cual no es posible.

Esto prueba que es imposible clonar perfectamente un estado cuántico desconocido usando transformaciones unitarias. Además, la clonación no es posible mediante la toma de mediciones, ya que ésta última es probabilista y destruye estados.

1.2.5. Algoritmos cuánticos

Un algoritmo, en general, es una sucesión finita de pasos para resolver una tarea. Un *algoritmo cuántico* consiste, así mismo, en la ejecución de una serie de compuertas cuánticas (operadores o transformaciones unitarias) aplicado a entidades, que pueden ser *qubits* o *quregistros*, seguido de una posterior toma de mediciones para recuperar los estados resultantes.

La figura 1.1 muestra un diagrama de un algoritmo cuántico básico [34]. Luego de una inicialización de la máquina de estados y de los *qubits* y *quregistros*, se aplican las transformaciones unitarias indicadas. Dado que la *toma de mediciones* es probabilista se ha de disponer de un generador de números aleatorios. Dependiendo de la *toma de mediciones* se decide si acaso se ha encontrado una solución, o bien, se reinicia el ciclo. En el rectángulo derecho aparecen concentradas las diferentes operaciones cuánticas y en el lado izquierdo una estructura típica de control (estructura selectiva).

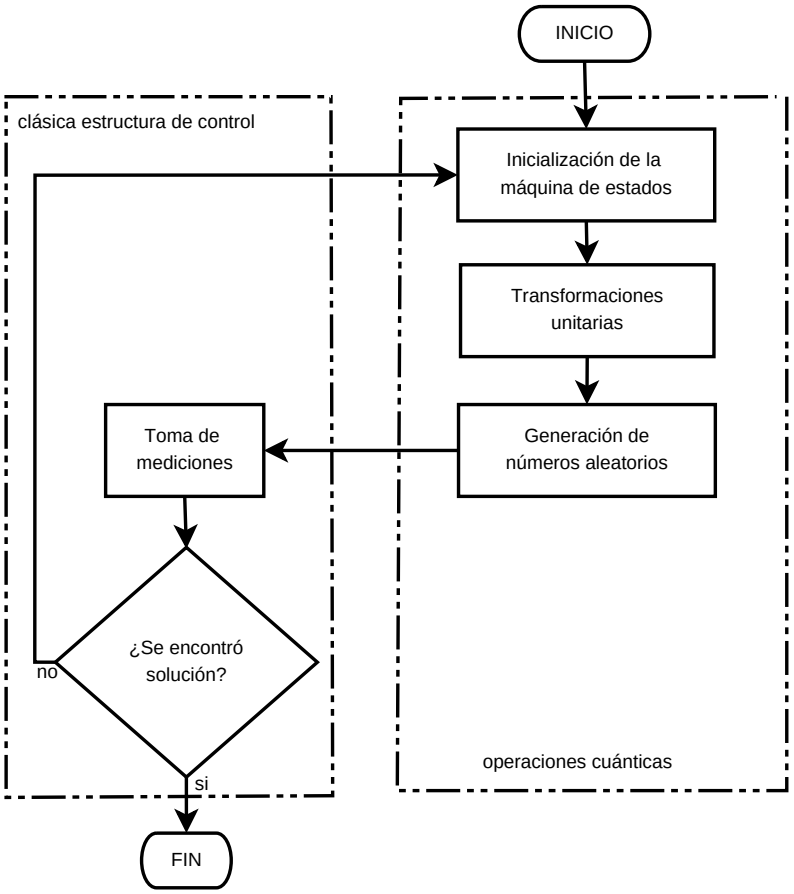

Figura 1.1: Flujo de un algoritmo cuántico básico.

1.2.6. Paralelismo cuántico

El poder de los algoritmos cuánticos a la computación cuántica lo ocasiona el *paralelismo cuántico* y del entrelazamiento de qubits. El paralelismo cuántico permite la

evaluación simultánea de una función $f(x)$ para muchos valores de x en una simple aplicación de una compuerta cuántica.

Supongamos la existencia de una serie de compuertas cuánticas U_f que implementa a una función f. Una forma cuántica de llevar a cabo esta implementación es considerar un computadora de dos qubits que empieza en el estado $|x, y\rangle$. Aplicando U_f a este estado podemos llegar al estado $|x, y \oplus f(x)$, donde $\oplus$ indica la operación módulo; el primer registro x denota el registro de datos de entrada, el segundo registro y es conocido como *target* o bien de salida. Note que si $y = 0$ la operación módulo $\oplus$ regresa solo el valor de $f(x)$. Ahora supongamos que el registro de datos de entrada es inicializado en la superposición $\frac{(|0\rangle+|1\rangle)}{\sqrt{2}}$, la cual es creada aplicando la compuerta de Hadamard al estado $|0\rangle$. Finalmente aplicamos U_f a esta superposición y resulta en el siguiente estado:

$$\frac{|0,f(0)\rangle+|1,f(1)\rangle}{\sqrt{(2)}}.$$

Este estado contiene información de las dos evaluaciones $f(0)$ y $f(1)$, y en cierto sentido, es como si hubiéramos evaluado $f(x)$ de dos valores de x simultáneamente, esta característica es conocida como *paralelismo implícito*. Sin embargo, esta propiedad no es muy útil debido a la forma de extracción de los valores. Como hemos visto, la medición (ver Sección 1.2.3) sólo arroja un valor de toda la superposición de valores que evaluamos, sin siquiera poder escogerlos. Actualmente se trata de encontrar nuevas formas de extracción de datos para que esta propiedad sea realmente efectiva.

1.3. Algoritmo cuántico de Deutsch-Jozsa

La idea de los algoritmos cuánticos es almacenar información en una superposición de estados cuánticos, manipularlos mediante transformaciones unitarias y extraer información útil del estado resultante.

Veamos el clásico algoritmo de *Deutsch-Jozsa*, el cual fue el primer algoritmo cuántico descrito para la computación de una tarea [12]. Éste puede resolverla exponencialmente más rápido que una computadora clásica, usando efectos cuánticos [13]. Aunque este algoritmo no tiene ninguna aplicación, muestra el poder de la computación cuántica y dió pauta a la creación de nuevos algoritmos cuánticos.

Sea D el conjunto de todas las funciones booleanas, *constantes* o *balanceadas*, de n-bits de entrada y de 1 bit de salida,

$$f : \mathbb{Z}_2^n \to \mathbb{Z}_2 = \{0, 1\}. \tag{1.3}$$

Una función es *balanceada* si tenemos $f(x) = 0$ para la mitad de los valores, $x \in \mathbb{Z}_2^n$ y $f(x) = 1$ para la otra mitad de valores de entrada. En otras palabras f es balanceada si toma el valor 0 , 2^{n-1} veces y es valor 1 también 2^{n-1} veces.

f es *constante* si para todos los valores de x, $f(x)$ es el mismo (ya sea 0 o 1).

El propósito del algoritmo de Deutsch-Jozsa es decidir, para una función f dada, si acaso es *constante* o *balanceada* "utilizando un solo paso de cómputo". Clásicamente, habría que evaluar los valores de $f(x)$ para todas las entradas x. En el mejor de los casos se tendría que evaluar dos veces y en el peor tendríamos que recorrer todas las

posibilidades, es decir, al menos $2^{n-1}+1$ veces. Veamos el procedimiento del algoritmo cuántico, donde con una sola evaluación de f determinaremos si es constante o balanceada.

Utilizando las propiedades de *superposición* e *interferencia* [3], es posible obtener la respuesta con una simple llamada a un *operador unitario* U_f, el cual podemos ver como una caja negra, como se muestra en la figura 1.2.

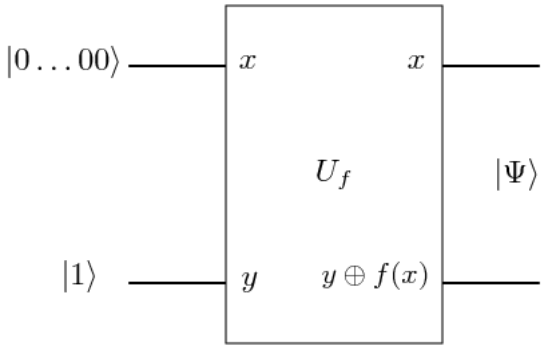

Figura 1.2: Circuito cuántico para la evaluación de $f(x)$ simultáneamente. Acepta entradas como $|x, y\rangle$ y las transforma al estado $x, y \oplus f(x)$

.

U_f ilustra un circuito cuántico [4], donde x son los n qubits de entrada y se le conoce como *registro de control* y se utiliza para guardar los argumentos de la función, el bit de entrada y para evaluar la función y es llamado *registro de función*. Este circuito cuántico actúa como sigue:

$$Uf : |xy\rangle = |x\rangle \otimes |y\rangle \mapsto |x\rangle \otimes |y \oplus f(x)\rangle \tag{1.4}$$

Supongamos el estado de entrada

$$\underbrace{|0\rangle \otimes \ldots \otimes |0\rangle}_{n\ veces} \otimes |1\rangle = |\underbrace{0\ldots0}_{n\ veces}1\rangle. \tag{1.5}$$

A cada qubit se le aplica la compuerta de Hadamard para obtener la superposición de todos los valores.

$$\underbrace{U_H \otimes \ldots \otimes U_H}_{n+1\ veces}|0\ldots01\rangle = \frac{1}{2^{n/2}} \sum_{x \in \mathbb{Z}_2^n} |x\rangle \otimes U_H|1\rangle. \tag{1.6}$$

Ahora procederemos a aplicarle el circuito U_f a la superposición de valores previa. Sin embargo, primero observemos que en la ecuación 1.4 si $y = 0$ queda solamente $f(x)$, como sigue:

$$|x0\rangle \longrightarrow |xf(x)\rangle$$

y si $y = 1$, de la siguiente manera:

[3] Característica de las partículas subatómicas cuando se comportan como ondas y se refiere a cuando dos o más se solapan o se entrelazan.

[4] Secuencia de transformaciones unitarias que ejecutan una acción.

$$|x1\rangle \longrightarrow |x(1 \oplus f(x))\rangle.$$

Enfoquémonos al término derecho de la ecuación 1.6 donde:

$$|x\rangle \otimes U_H|1\rangle = |x\rangle \otimes \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$

por la propiedad distributiva del producto tensorial se cumple que:

$$\tfrac{1}{\sqrt{2}}\begin{cases} |x0\rangle - |x1\rangle \text{ si } f(x) = 0 \\ |x1\rangle - |x0\rangle \text{ si } f(x) = 1 \end{cases}$$

o lo que es lo mismo

$$|x\rangle \otimes U_H|1\rangle = (-1)^{f(x)}|x\rangle \otimes U_H|1\rangle.$$

Por la propiedad de linealidad de U_f, podemos calcular la superposición 1.6,

$$\frac{1}{2^{n/2}} \sum_{x\in\mathbb{Z}_2^n} |x\rangle \otimes U_H|1\rangle \longrightarrow \frac{1}{2^{n/2}} \sum_{x\in\mathbb{Z}_2^n} (-1)^{f(x)}|x\rangle \otimes U_H|1\rangle. \tag{1.7}$$

Por último, aplicamos nuevamente la compuerta de Hadamard a los primeros n qubits, de lo que se obtiene:

$$\underbrace{U_H \otimes \ldots \otimes U_H}_{n \ veces} \frac{1}{2^{n/2}} \sum_{x\in\mathbb{Z}_2^n} (-1)^{f(x)}|x\rangle = \frac{1}{2^n} \sum_{x\in\mathbb{Z}_2^n} (-1)^{f(x)} \sum_{y\in\mathbb{Z}_2^n} (-1)^{x.y}|y\rangle \tag{1.8}$$

$$= \sum_{y\in\mathbb{Z}_2^n} c_y|y\rangle, \quad donde \quad c_y = \frac{1}{2^n} \sum_{x\in\mathbb{Z}_2^n} (-1)^{f(x)+x.y}. \tag{1.9}$$

El producto escalar módulo 2, $x.y$, está definido como:

$$x.y = (x_1y_1 + x_1y_1 + \ldots + x_ny_n) \ mod \ 2 \tag{1.10}$$

$$x.y = x_1y_1 \otimes x_1y_1 \otimes \ldots \otimes x_ny_n. \tag{1.11}$$

La probabilidad de que el valor final de la medición de los primeros n qubits del estado $|y\rangle = |0\ldots0\rangle$ es

$$\|c_{0\ldots0}\|^2 = \left\| \frac{1}{2^n} \sum_{x\in\mathbb{Z}_2^n} (-1)^{f(x)} \right\|^2 = \begin{cases} 1 \ si \ f \ es \ constante, \\ 0 \ si \ f \ es \ balanceada. \end{cases} \tag{1.12}$$

Con la medición de los primeros n-qubits podemos saber si f se trata de una función balanceada o constante. También se pueden consultar [45], [32] y [13] con mayor detalle.

Hay que notar que para llevar a cabo esta comparación, el tiempo que se necesita para ejecutar la función f no es tomada en cuenta.

Los algoritmos cuánticos muestran que el procedimiento para encontrar la solución de algunos problemas con computadoras cuánticas, es más eficiente que con las clásicas.

Aunque en la actualidad no existen computadoras cuánticas, y en efecto estamos muy lejos de que éste sea un hecho, no existe ninguna ley física que contradiga la posible realización de las computadoras cuánticas. Además, existen algunas especulaciones sobre si efectivamente la computación cuántica será más poderosa que la computadora clásica.

Capítulo 2

Diseño de un lenguaje de programación cuántica: GAMA

Los lenguajes de programación permiten la abstracción de problemas. Por ejemplo, el lenguaje ensamblador es una abstracción del código máquina, aunque cuando se quiere resolver un problema en este último, es necesario concentrarse tanto en la arquitectura de la computadora como en la estructura del problema, provocando así, problemas difíciles de entender y de mantener. Los lenguajes de alto nivel buscan crear una abstracción superior y tratan de expresar en código la descripción natural de la solución, y al mismo tiempo del problema [15].

Desde el surgimiento de los lenguajes de alto nivel, se les ha clasificado a grandes rasgos en los de propósito específico y en los de propósito general. Una ventaja que ofrecen los lenguajes de propósito específico es que ellos incorporan elementos y comportamientos propios de los problemas para los cuales fueron desarrollados, ejemplos de éstos son los de manipulación simbólica, de procesamiento de texto, y de consultas en bases de datos, por mencionar sólo algunos.

Contar con un lenguaje de alto nivel y de propósito específico nos permitirá tener mayor abstracción sobre los algoritmos cuánticos. Además, las entidades cuánticas incorporadas permitirán describir el problema en términos del propio problema y no en los de la máquina en los que está corriendo la aplicación, haciendo de esto una programación flexible.

La meta principal de nuestro proyecto es diseñar e implantar un lenguaje de programación orientado hacia la computación cuántica. Éste deberá permitir la especificación y depuración de algoritmos, esto último es, que establezca un mecanismo para el monitoreo de variables en tiempo de ejecución. Este monitoreo deberá ejecutar el programa instrucción por instrucción, mostrando los cambios en el contenido de las variables del programa.

El lenguaje de programación está diseñado de tal manera que resulte sencillo implementar cualquier algoritmo cuántico, además cuenta con la definición de una gramática sencilla, la cual permite al diseñador de algoritmos enfocarse en el comportamiento del algoritmo y no en el de la programación. Estas facilidades permiten enfocarlo hacia el aprendizaje.

Las principales aplicaciones del lenguaje para la simulación de algoritmos cuánticos serían en el campo de Teoría de Computabilidad, debido al interés en el estudio de algoritmos cuánticos por su capacidad de reducir la complejidades de problemas. Por

otro lado, se puede utilizar como un simulador de sistemas cuánticos en las áreas tanto de Física [46] como de Química [4].

En este capítulo mencionamos las características del lenguaje de programación GAMA (sección 2.1). Además, en la sección 2.2 presentamos un estudio de los requerimientos básicos de las entidades cuánticas para la definición adecuada de su gramática y de algunos elementos auxiliares. Por último, en la sección 2.3 presentamos los componentes necesarios en la definición del lenguaje GAMA.

2.1. Características de *GAMA*

GAMA[1]. es un lenguaje de alto nivel, imperativo, de propósito específico, procedimental y estructurado. A continuación detallamos cada una de sus características.

- Alto nivel: Es considerado un lenguaje de alto nivel, debido a que las operaciones se expresan de manera formal, parecidas al lenguaje matemático con uso de palabras reservadas, que siguiendo una tendencia mundial están en Inglés.
- Propósito específico: El manejo de operaciones sobre entidades cuánticas es lo que lo hace clasificarse como uno de este tipo. En la sección 2.2 presentamos la descripción y la notación de estas entidades, así como sus operaciones. Esto permite identificar las necesidades principales en la gramática de GAMA.
- Procedimiento e Instrucciones: El lenguaje se basa en la asignación de valores y en la utilización de variables para su almacenamiento, asimismo, en la ejecución de operaciones sobre los datos almacenados. Está dotado también de operadores como composición, iteración, recursión y de alternancia.
- Estructura: El lenguaje maneja tres estructuras de control: la secuencial, la selectiva y la iterativa.

2.2. Entidades cuánticas

En el capítulo 1 presentamos un panorama general de los conceptos fundamentales asociados a la computación cuántica. Entre los que se cuenta el bit cuántico (*qubit*), el registro cuántico (*quregistro*), la compuerta cuántica (*qucompuerta*) y la toma de mediciones. Para nuestro fin, consideramos las tres primeras como las entidades fundamentales en la simulación de algoritmos cuánticos. Por otro lado, la toma de mediciones es un mecanismo de lectura para la extracción de información sobre qubits y quregistros. Retomando estos conceptos, presentamos tanto la definición como la notación necesarias para el diseño de algoritmos cuánticos. Además, elaboramos un sumario de todas las operaciones realizables sobre estas entidades. Para obtener una mejor representación sobre las entidades cuánticas aprovechamos su descripción matemática, lo cual nos permite trabajar a los conjuntos de enteros, de reales y de complejos como escalares; a los qubits y quregistros como vectores y a las qucompuertas como matrices.

[1] *GAMA* son las siglas que hacen referencias a las iniciales de los nombres de las personas que colaboraron en el desarrollo del lenguaje: Guillermo, Amilcar y Mireya respectivamente. Y se completó con *A* para que tuviera referencia con la letra matemática Γ.

2.2.1. Escalares

Denotemos por $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$ al conjunto de los números enteros, reales y complejos, respectivamente. $\mathbb{Z}$ es un anillo y tanto $\mathbb{R}$ como $\mathbb{C}$ son campos. Dado que la computación cuántica se lleva a cabo en espacios de Hilbert, vale decir vectoriales, sobre $\mathbb{R}$ o $\mathbb{C}$ nos referiremos a los elementos de estos campos como escalares. Los espacios vectoriales sobre anillos se dicen ser módulos, por lo que también decimos que elementos de $\mathbb{Z}$ son reales. Uno de los requerimientos iniciales en nuestro lenguaje de programación es la inclusión de una aritmética entera y la de una compleja. Esta última incluye a la aritmética de punto flotante. La representación de cada número entero se realiza con una secuencia de dígitos, precedida por un signo "-", cuando se trata de enteros negativos. La de los números reales puede manejarse como una representación entera, con la única diferencia que ésta incluye un punto decimal. Con lo que respecta a la de los complejos, consideramos un punto (x, y) en el plano complejo. Para nuestro propósito es pertinente representar el punto (x, y) por el número complejo $z = x + iy$, donde $i = \sqrt{-1}$. Llamamos $x = Re(z)$ la parte real y a $y = Im(z)$ la parte imaginaria de z [41].

Operaciones entre escalares

Las operaciones binarias definidas entre escalares son la suma, la resta, la multiplicación y la división. Todas las operaciones obtienen como resultado un escalar del mismo tipo que el de los operandos involucrados, excepto por la división, la cual no garantiza este comportamiento debido a que los números enteros no son un campo. Otro operador binario básico es el módulo. Éste opera únicamente sobre enteros. Por otro lado, dos operadores unarios incluidos son, el valor absoluto y el conjugado, los cuales operan sobre cualquier escalar.

2.2.2. Vectores

Hemos mencionado en la sección 1.2.1 que los qubits y los quregistros pueden ser representados en términos de vectores, cuyos componentes son escalares. La notación adecuada para ello es la "bra-ket" de Dirac para describir estados cuánticos. Para contextualizar esta representación en GAMA, utilizaremos los símbolos semejantes "<", ">" y " | ", para la descripción de estados cuánticos.

Operaciones entre vectores

La dimensión juega un rol importante en operaciones vectoriales. Por esto, pondremos especial cuidado en su manejo. Las operaciones binarias entre vectores son la suma, la resta, el produto tensorial, el producto interior y el producto exterior. Para evitar confusiones únicamente trataremos las operaciones cuyo resultado sea otro vector, esto es, la suma, la resta y el producto tensorial. La suma y la resta operan únicamente entre vectores de igual dimensión cuyo resultado es de la misma dimensión. Por otro lado, el producto tensorial no tiene restricción alguna referente a su dimensión, por lo que cualquier par de vectores pueden ser utilizados para esta operación. La dimensión del vector resultante es la multiplicación de cada una de las dimensiones asociadas a los operandos. Las operaciones unarias definidas son para obtener el vector conjugado y el vector transpuesto. En el primer caso, se determinan el conjugado de cada uno

de los componentes del vector y no hay cambio en su dimensión. En el segundo caso, se busca cambiar un vector columna a un vector renglón o viceversa. Una operación interesante por las propiedades físicas que involucra, es la toma de medición sobre vectores. Ésta se encuentra explicada en la sección 3.8. La toma de medición necesita de dos partes: el vector que presenta el estado actual del sistema y el estado por el cual será medido. El vector resultante tiene la misma dimensión del vector medido.

2.2.3. Matrices

Como ya se mencionó en la sección 1.2.4, las compuertas cuánticas pueden ser representadas mediante matrices de números complejos. Estas matrices deben ser unitarias y sus órdenes de la forma $2^n \times 2^n$, donde $n \in \mathbb{N}$. Las matrices más usuales en cualquier algoritmo cuántico son las matrices de Pauli *X, Y, I y Z* y la de *Hadamard* (H) de dimensión 2; la matriz *Cnot* de dimensión 2^2 y la de *Toffoli* (T) de dimensión 2^3. Por esto, necesitamos incorporar estas compuertas básicas como primitivas propias de GAMA. Los operadores que actúan sobre quregistros de dimensión menor que 1, deben de tener la dimensión de su dominio explícitamente.

En el caso de que las matrices *I*, *H* y *T* necesiten operar con una dimensión mayor a la de su definición, deberá indicarse explícitamente. Este tipo de compuertas son generadas aplicando el operador producto tensorial a algunas de las compuertas básicas mencionadas.

Operaciones entre matrices

Las operaciones binarias definidas entre matrices son la suma, la resta, la multiplicación y el producto tensorial. Análogo a las operaciones vectoriales, la suma, la resta y la multiplicación requiere de operandos de igual dimensión, cuyo resultado la preserva, siendo que en la del producto tensorial no es necesario la verificación de dimensiones, donde la dimensión de la matriz cuadrada resultante es la multiplicación de las dimensiones de ambos operandos. Las operaciones unarias para obtener la matriz conjugada y la transpuesta mantienen la misma dimensión de sus operandos en la matriz resultante.

2.2.4. Operaciones mixtas

Consideramos la muliplicación de escalares con vectores o bien con matrices, la multiplicación de matrices con vectores, el producto interior y el producto exterior como operaciones mixtas, debido a que los elementos del dominio no corresponden a los de su contradominio. La multiplicación de escalares con vectores o con matrices produce un vector o una matriz de igual dimensión, respectivamente. En el caso de la multiplicación de matrices con vectores, por omisión los vectores transpuestos son considerados como vectores renglones. Así, se debe cumplir que el número de columnas del operando izquierdo corresponda con el número de renglones del operando derecho. Y el resultado es un vector que mantiene la dimensión del vector con el que se operó. Por otro lado, el producto interior multiplica un vector renglón con un vector columna con el mismo número de elementos y produce un escalar. En la notación de Dirac, el

producto interior se denota por $\langle \mid \rangle$. De manera análoga, el producto exterior multiplica un vector columna por un vector renglón con el mismo número de elementos y produce una matriz cuadrada de dimensión igual al número de elementos de los vectores.

2.3. Definición del lenguaje de programación

Describiremos el lenguaje de programación GAMA (Γ) por medio de su sintaxis [2] y de su semántica [3] [3]. Para la especificación sintáctica del lenguaje usaremos una notación muy común conocida como *BNF* (por su abreviatura en inglés *Backus Normal Form*), la cual es una gramática libre de contexto y se compone de una serie de reglas de derivación.

En la figura 2.1 mostramos los componentes principales para la definición de GAMA. Primero se encuentra el analizador léxico, el cual convierte la cadena de caracteres del programa de entrada en una cadena de componentes léxicos. Posteriormente, el analizador sintáctico proporciona todas las reglas de sintaxis que se aplicarán a dichos componentes. Esta última genera una representación intermedia explícita del programa fuente. El análisis semántico se lleva a cabo durante la ejecución del código intermedio generado por el análisis sintáctico, sin embargo, con las notaciones que existen en la actualidad, es mucho más difícil la descripción del análisis semántico. Así que lo explicaremos por medio de descripciones informales o bien mediante ejemplos. A continuación, veremos cada uno de estos componentes a detalle.

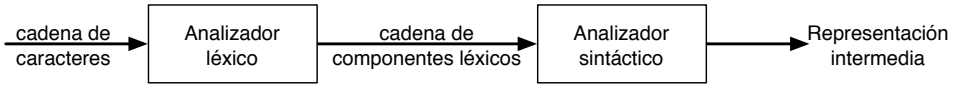

Figura 2.1: Componentes principales en la definición de GAMA.

2.3.1. Analizador léxico

Un analizador léxico se encarga de agrupar en componentes léxicos, o bien *tokens* (en inglés), la secuencia de caracteres del programa fuente, de acuerdo a ciertos patrones. Con ellos se busca la especificación y diseño de programas que ejecuten ciertas acciones activadas. Particularmente, en GAMA, buscamos la identificación de las entidades de origen cuántico, con sus respectivas operaciones, vistas en la sección 2.2. Además, claro está, de la identificación de todos los componentes que conforman un lenguaje de programación imperativo, estructurado y de alto nivel, como lo es el lenguaje C .

La herramienta de desarrollo para el análisis léxico que utilizamos es `LEX`, el cual, es un lenguaje basado en la ejecución de una acción dado un patrón (patrón - acción), donde los patrones se especifican por medio de expresiones regulares. El reconocimiento de tales expresiones regulares se realiza mediante un autómata finito eficiente [29].

[2] La sintaxis se refiere a la estructura o aspecto de los programas.
[3] La semántica es lo que proporciona significado de los programas.

Existe una interacción fundamental entre el analizador léxico y el analizador sintáctico. Mientras el léxico genera la secuencia de componentes léxicos, el sintáctico los obtiene para llevar a cabo el análisis. En la figura 2.1 mostramos esta interacción.

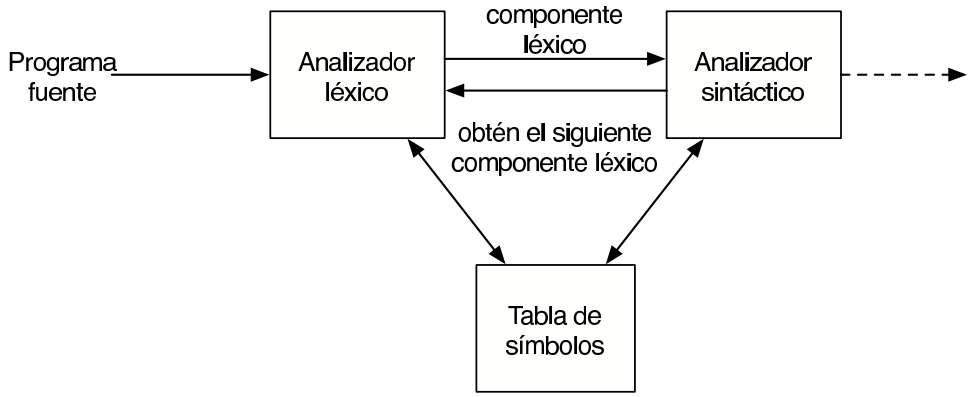

Figura 2.2: Interacción del analizador léxico con el sintáctico.

Los componentes léxicos de GAMA se conforman de constantes, identificadores, palabras reservadas, operadores y algunos otros símbolos separadores. En el Apéndice A encontramos el listado de las expresiones regulares que generan estos componentes léxicos.

Constantes

Éstas se refieren a las literales que representan valores numéricos ya sean enteros, complejos o con valores booleanos. Las constantes incluidas en GAMA son las que se encuentran en la tabla C.5.

PI	$3{,}14159265358979323846$
E	$2{,}71828182845904523536$
GAMMA	$0{,}57721566490153286060$
DEG	$57{,}29577951308232087680$
PHI	$1{,}61803398874989484820$
i	$\sqrt{-1}$
TRUE	1
FALSE	0

Tabla 2.1: Constantes de GAMA.

Identificadores

Un identificador es una secuencia de letras y dígitos. El primer carácter debe ser una letra; el subguión _ cuenta como una letra. Existe diferencia entre letras mayúsculas y minúsculas. GAMA utiliza identificadores para los nombres de variables, nombres de matrices y nombres de funciones.

Palabras reservadas

Éstas se utilizan para reconocer ciertas construcciones en el programa. Por ejemplo, `if, else, while, for`, para estructuras de control, `main, return`, para la estructura del programa, `H, I, Cnot` y `T`, para las compuertas cuánticas básicas. Y las etiquetas `Int`, `Bool`, `Complex`, `Qbit`, `Qreg` y `Qgate` corresponden a los diferentes tipos de datos. Algunas funciones matemáticas están incluidas: `sin, cos, atan, log, log10, exp, sqrt y norm`. Además, `display, exit, run, debugger, reset` y `pause` son algunas de las primitivas del depurador.

Int	while	sin	measure
Bool	for	cos	display
Complex	main	atan	exit
Qbit	return	log	run
Qreg	H	log10	debugger
Qgate	I	exp	reset
if	Cnot	sqrt	pause
else	T	norm	

Tabla 2.2: Palabras reservadas.

Operadores y símbolos especiales

Se consideran operadores aritméticos, lógicos, relacionales y de asignación para sus respectivas operaciones. Además, la eliminación de espacios en blanco provocan que en el análisis sintáctico no se tomen en cuenta, proporcionando sencillez a la gramática. Los símbolos especiales *(,), [,], {, }*, son reconocidos para diferentes funciones, ya sea para la declaración de matrices o bien para la definición de funciones, entre otras. Los blancos, tabuladores y nueva línea, son ignorados, excepto por aquellos que separan componentes.

Otra de las funciones del analizador léxico es la detección de cierto tipo de errores. Estos generalmente son generados por descuidos del programador, pueden ser debidos a diferentes razones, entre las cuales están: nombres ilegales de identificadores, números incorrectos, errores de ortografía en palabras reservadas, etc. En la sección 3.6 tenemos la descripción del manejo de errores.

2.3.2. Analizador sintáctico

Durante esta etapa, utilizamos la herramienta de UNIX, `LALR YACC`, para el desarrollo del análisis sintáctico. La sintaxis se puede describir por medio de gramáticas libres de contexto o bien mediante la notación BNF. La gramática de GAMA está basada en la del lenguaje C [23] y en la de HOC (lenguaje de programación prototipo desarrollado por Kernighan y Pike [22]). La definición de una gramática libre de contexto consiste de un conjunto de símbolos no terminales, un conjunto de símbolos terminales, un conjunto de producciones y un axioma inicial. La notación asociada a cada uno de estos componentes es la siguiente: Los símbolos no terminales serán representados

por una palabra, la cual podrá ser escrita tanto en letras mayúsculas como en minúsculas. Por otro lado, los símbolos terminales serán representados únicamente con letras mayúsculas. Una producción está formado por un símbolo no terminal en el lado izquierdo, y las posibles secuencias de símbolos terminales y de no terminales del lado derecho. En el texto, trataremos a los símbolos incluidos en una producción como variables gramaticales y la separación entre una secuencia y otra estará denotada por el símbolo "|". Por último, la manera de denotar el axioma inicial en el texto será con el símbolo no terminal *Start*. A continuación, describiremos la sintaxis de la estructura general de un programa cuántico, así como de los componentes más utilizados. El listado de las producciones se encuentra en el Apéndice B.

Estructura de un programa cuántico

Básicamente, GAMA acepta secuencias de instrucciones separadas por el símbolo ";", éstas también pueden ser compuestas con el uso de "{", "}". La estructura de un algoritmo cuántico está dividida en tres partes principales: la declaración de variables globales, la definición de funciones y el cuerpo del programa. La producción es la siguiente:

$$\begin{aligned} \textit{Start} &\longrightarrow \textit{File} \\ \textit{File} &\longrightarrow \textit{Declaration} \mid \textit{Functions} \end{aligned}$$

El axioma inicial *Start* invoca a la variable gramatical *File*, ésta última a su vez genera la estructura de un programa GAMA como lo ilustra el algoritmo 1. La sección de declaración de variables globales está dada por *Declaration* (las producciones se encuentran en inglés, debido a la analogía con la del lenguaje C). Por otro lado, la variable gramatical *Functions* describe la secuencia de definiciones de funciones y finaliza con la función principal *main*. Veamos las producciones correspondientes a la declaración de variables y de funciones.

Algoritmo 1: Estructura general de un algoritmo cuántico en GAMA.

```
Declaración de variables globales
Int a;
Definición de funciones
Int suma(Int a, Int b){
if a >b then
  return a;
else
  return b;
end if
}
Cuerpo del programa
main(){
a=suma(3,4);
}
```

Declaración de variables

El nombre de las variables de GAMA está asociado a un identificador. Los identificadores son reconocidos por el analizador léxico por medio de una expresión regular. La variable gramatical asociada a este identificador es *Identifier*. La declaración de una variable consiste del tipo de dato, del nombre y del valor. Las siguientes producciones generan la declaración de una o varias variables con el mismo tipo. Esta declaración consta del tipo de dato denotado por la variable gramatical *TypeSpecifier* y de la lista de variables denotadas por *InitDeclarationList*. Por un lado, apreciamos que *TypeSpecifier* deriva al símbolo terminal *TYPE*. Este símbolo terminal tiene asociados los diferentes tipos de símbolos en GAMA (`Int`, `Bool`, `Complex`, `Qbit`, `Qreg`, `Qgate`, *FUNCTION* y *CONST*). Por otro lado, la variable gramatical *InitDeclaratorList* produce una secuencia separada por comas de símbolos no terminales *InitDeclarator*. El *InitDeclarator* se encarga de reconocer a la variable gramatical *Declarator*, la cual es asociada con el identificador *Identifier*.

$$
\begin{array}{lcl}
\textit{Declaration} & \longrightarrow & \textit{TypeSpecifier InitDeclaratorList ';'} \\
\textit{TypeSpecifier} & \longrightarrow & \textit{TYPE} \\
\textit{InitDeclaratorList} & \longrightarrow & \textit{InitDeclarator} \\
 & & \mid \textit{InitDeclaratorList ',' InitDeclarator} \\
\textit{InitDeclarator} & \longrightarrow & \textit{Declarator} \\
\textit{Declarator} & \longrightarrow & \textit{Identifier}
\end{array}
$$

En esta lista pueden existir variables inicializadas. Para ello, la variable gramatical *InitDeclarator* agrega la siguiente producción. *Initializer* se encarga de validar la forma de inicialización de variables. La inicialización de variables de tipo booleano, entero, complejo, qubit, quregistro se realiza mediante una expresión con la variable gramatical *Expr*. Las producciones de la variable gramatical *Expr* las veremos con mayor detalle mas adelante.

$$
\begin{array}{lcl}
\textit{InitDeclarator} & \longrightarrow & \textit{Declarator} \\
 & & \textit{Declarator '=' Initializer} \\
\textit{Initializer} & \longrightarrow & \textit{Expr}
\end{array}
$$

Para la declaración e inicialización de arreglos, agregamos las siguientes producciones correspondientes a las variables gramaticales *Declarator* e *Initializer*. Mientras *Declarator* genera la definición de arreglos con ayuda del terminal *TINT*, el cual sirve para denotar su dimensión; *Initializer* genera la inicialización de éstos. La inicialización consiste de una serie de expresiones *Expr* agrupadas con los símbolos "{" y "}" y separadas por comas.

$$
\begin{array}{lcl}
\textit{Declarator} & \longrightarrow & \textit{Identifier} \\
 & & \textit{Identifier '[' TINT ']'} \\
\textit{Initializer} & \longrightarrow & \textit{Expr} \\
 & & \textit{Array} \\
\textit{Array} & \longrightarrow & \textit{'\{' ArgumentExprList '\}'} \\
\textit{ArgumentExprList} & \longrightarrow & \epsilon \\
 & & \textit{Expr} \\
 & & \textit{ArgumentExprList ',' Expr}
\end{array}
$$

Como la inicialización de variables de tipo de dato `Qgate` no se coteja mediante *Expr*, ya que se trata de una sintaxis muy particular agregamos las siguientes producciones. Observemos que la nueva variable gramatical *ArgExprMetalist* genera una lista de la misma producción para la inicialización de arreglos agrupados por "{" y "}".

$$
\begin{array}{lcl}
\textit{Initializer} & \longrightarrow & \textit{Expr} \\
 & & \textit{Array} \\
 & & \textit{'\{' ArgExprMetalist '\}'} \\
\textit{ArgExprMetalist} & \longrightarrow & \epsilon \\
 & & \textit{Array} \\
 & & \textit{ArgExprMetalist ',' Array}
\end{array}
$$

Tipos de datos

`Int`**:** Este tipo de datos es utilizado para operaciones aritméticas entre valores enteros. El valor máximo que puede tomar un entero depende de la arquitectura de la computadora.

`Bool`**:** Éste corresponde a términos para operaciones lógicas, las cuales la mayoría de las veces se utilizan para controlar el flujo del algoritmo. Estos datos pueden tomar valores de `TRUE` (1) o bien `FALSE` (0).

`Complex`**:** Los coeficientes de los vectores que representan *qubits* (sección 1.2.1), los cuales son llamados amplitudes, se encuentran en el espacio $\mathbb{C}$, por lo que es necesario contar con este tipo de datos, incluida su aritmética.

`Qbit`**:** Los *qubits*, los *quregistros* y las *qucompuertas* vienen a ser las principales entidades de GAMA. Cada qubit es una combinación lineal de dos estados determinados $|0\rangle$, $|1\rangle$, y es un punto en la esfera unitaria del espacio generado por $|0\rangle$ y $|1\rangle$. Recordemos que por cuestiones de la medición probabilista, el cuadrado de la suma de las amplitudes de los *qubits*, debe ser igual a uno.

Por ejemplo, $a = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$ es un qubit. Una operación muy común entre *qubits* es el producto tensorial. Así, si $b = a$ entonces $a \otimes b = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle$.

Este tipo de datos permite la definición de vectores en el espacio complejo, definidos con una sintaxis peculiar. La notación *bra-ket* de Dirac; donde la expresión $Qubit = 0{,}707106|0\rangle + 0{,}707106|1\rangle$ representa el $Qubit = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$, o lo que es lo mismo, $Qubit = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$.

Así la definición de un qubit se refiere a la secuencia de 2^n estados denotados por el *ket* $|x\rangle$, cada uno con su amplitud correspondiente, donde la suma de los cuadrados de las amplitudes debe ser igual a uno: $|\frac{1}{\sqrt{2}}|^2 + |\frac{1}{\sqrt{2}}|^2 = 1$. Particularmente para un qubit, la dimensión n es igual a 1. La sintaxis del qubit permite la notación *bra*, es decir, $\langle x|$, la cual denota el vector transpuesto. Este tipo de datos tiene que cumplir con las propiedades de qubits y además cuenta con las operaciones básicas sobre qubits, como lo es el producto tensorial.

`Qreg`**:** Un quregistro es una concatenación de qubits, así que prácticamente las mismas operaciones se aplican, pero componente a componente. La concatenación de n qubits forma un quregistro de longitud n. Formalmente la concatenación se interpreta como un producto tensorial, por lo cual un quregistro de longitud n es asimismo un vector de dimensión 2^n. Consecuentemente todo qubit puede ser visto como un quregistro de longitud 1 y éste es, como ya vimos un vector de dimensión $2 = 2^1$.

`Qgate`**:** Las qucompuertas son operadores matriciales unitarios. Distinguimos a algunas de éstas como funciones primitivas que forman parte del lenguaje. Cualquier otra compuerta cuántica se ha de realizar como una composición de una o varias primitivas. Por ejemplo, una compuerta primitiva es la llamada *Cnot* la cual aplica la compuerta de negación al segundo qubit siempre que el primero se encuentre en estado $|1\rangle$,

$$\text{Cnot}: \begin{array}{lcl} |0x\rangle & \mapsto & |0\rangle \otimes |x\rangle \\ |1x\rangle & \mapsto & |1\rangle \otimes X|x\rangle \end{array},$$

la cual tiene como matriz

$$\text{Cnot} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

es decir, Cnot $= \begin{pmatrix} I & 0 \\ 0 & X \end{pmatrix}$ y X es la compuerta de negación

$$\text{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

o sea $X: \begin{array}{lcl} |0\rangle & \mapsto & |1\rangle \\ |1\rangle & \mapsto & |0\rangle \end{array}$. El primer qubit puede verse como un bit de control y el segundo como un qubit de argumento.

Declaración de funciones

Las definiciones de funciones de GAMA constan del tipo de dato del valor de retorno *TypeSpecifier*, del nombre de la función seguida por sus parámetros, dada por *Declarator* y del cuerpo de la función *FunctionBody*. El cuerpo de la función puede estar vacío o puede contener una secuencia de sentencias *StatementList* o una de declaraciones de variables *DeclarationList* o bien de ambas, las declaraciones de variables primero y después la de instrucciones. Estas secuencias están agrupadas mediante los símbolos "{" y "}".

FunctionDefinition ⟶ *TypeSpecifier Declarator FunctionBody*
Declarator ⟶ *Identifier*
Identifier '[' TINT ']'
| *Declarator '(' ')'*
| *Declarator '(' ParameterTypeList ')'*
ParameterTypeList ⟶ *ParameterList*
ParameterList ⟶ ϵ
| *ParameterDeclaration*
| *ParameterList ',' ParameterDeclaration*
ParameterDeclaration ⟶ *TypeSpecifier Declarator*

La variable gramatical *Declarator* genera la producción para identificar el nombre de la función, el cual se trata de un identificador denotado por *Identifier*. Además tiene asociado una lista de cero o más parámetros dada por *ParameterTypeList*, separados por comas y delimitados por paréntesis. Cada parámetro en la lista consiste de un tipo de dato generado por *TypeSpecifier* y de un identificador (*Identifier*).

FunctionBody ⟶ *CompoundStatement*
CompoundStatement ⟶ *'{' '}'*
| *'{' StatementList '}'*
| *'{' DeclarationList '}'*
| *'{' DeclarationList StatementList '}'*
StatementList ⟶ *Statement*
| *StatementList Statement*
DeclarationList ⟶ *Declaration*
| *DeclarationList Declaration*

Statement ⟶ *StatementPair*
StatementUnpaired
StatementPair ⟶ *CompoundStatement*
| *ExpressionStatement*
| *IterationStatement*
| *RETURN Expr*

Una secuencia de instrucciones *Statement* dadas por *StatementList*, la cual consiste de una serie de producciones generadas por las variables *CompoundStatement*, *ExpressionStatement*, *IterationStatement* o bien por el símbolo terminal *RETURN* seguido por la expresión *Expr* que regresará el valor de la función. La variable gramatical *ExpressionStatement* genera todo tipo de expresiones que reconoce GAMA. Por otro lado, la variable *IterationStatement* produce las estructuras de control `while` y `for`, las cuales veremos más adelante. Las variables gramaticales *StatementPair* y *StatementUnpaired* fueron agregadas para el manejo de la estructura de control `if-else`.

Una vez que hemos definido las reglas gramaticales para la declaración de variables y de funciones, recordemos que la estructura de un programa GAMA está dada por una lista de declaración de variables, por una de declaración de funciones, seguida de la declaración de la función principal denotada por *MainFunction*, la cual indica el punto a partir del cual empezará la ejecución del programa. Únicamente con esta

estructura, es decir, con el listado de la declaración de variables globales, la definición de funciones y al final la función principal `main`, resultará en una sintaxis válida, en otro caso, se muestran los errores sintácticos correspondientes.

$$
\begin{array}{lcl}
\textit{Functions} & \longrightarrow & \textit{ListFunctionDefinition MainFunction} \\
\textit{ListFunctionDefinition} & \longrightarrow & \epsilon \\
 & & \mid \textit{FunctionDefinition} \\
 & & \mid \textit{ListFunctionDefinition FunctionDefinition}
\end{array}
$$

La función principal reconocida por el símbolo terminal *MAIN* no cuenta con valor de retorno ni argumentos de entrada, como se aprecia en la siguiente producción.

$$\textit{MainFunction} \longrightarrow \textit{MAIN '(' ')' FunctionBody}$$

Estructuras de control

Las sentencias de control de flujo que maneja *GAMA*, utilizan palabras reservadas tales como: `for`, `while`, `if`, `else`, etc, y determinan el orden en el que se han de ejecutar las instrucciones. El control de flujo puede ser de tipo secuencial, selectivo o iterativo. El secuencial pasa de manera incondicional de una instrucción a la siguiente, el selectivo escoge la instrucción siguiente en función de los valores que tome una prueba y el iterativo ejecuta la repetición de instrucciones, en tanto no se cumplan algunas condiciones de paro.

Cada una de estas estructuras tiene un cuerpo, el cual llamaremos *bloque*, éste es un conjunto de sentencias consecutivas y se encuentran delimitadas por los símbolos "{", "}".

Para evitar ambigüedades en la implementación de la estructura de control `if` en el reconocimiento de `if` y de `if-else`, utilizamos la estrategia discutida por Aho y Ullman [2], donde se establece una producción para declaraciones parejas y otra para declaraciones disparejas. De tal forma que la producción de la declaración de esta estructura de control se estableció así:

$$
\begin{array}{lcl}
\textit{Statement} & \longrightarrow & \textit{StatementPair} \\
 & & \mid \textit{StatementUnpaired} \\
\textit{StatementPair} & \longrightarrow & \textit{IF '(' Expr ')' StatementPair} \\
 & & \textit{ELSE StatementUnpair} \\
 & & \mid \textit{other statement ...} \\
 & & \\
\textit{StatementUnpaired} & \longrightarrow & \textit{IF '(' Expr ')' Statement} \\
 & & \mid \textit{IF '(' Expr ')' StatementPair} \\
 & & \textit{ELSE StatementUnpaired}
\end{array}
$$

Para los ciclos tenemos las estructuras `for` y `while`, las cuales ejecutan un bloque de sentencias de acuerdo a una condición lógica dada por su prueba. En la instrucción `for` puede existir la inicialización de contadores, que se vayan incrementando y sean factores de las condiciones.

$$
\begin{array}{lcl}
\textit{IterationStatement} & \longrightarrow & \textit{WHILE '(' Expr ')' Statement} \\
 & & \mid \textit{FOR '(' Expr ';' Expr ';' Expr ')' Statement}
\end{array}
$$

Operadores
~
( )
- (menos unitario)
$\hat{}t$ $\hat{}*$
$* \; / \; \% \; \&$
$+ \; -$
$< \; <= \; > \; >=$
$== \; != $
$\&\&$
$\|\|$
=

Tabla 2.3: Precedencia de operadores de GAMA.

La variable gramatical *Expr* genera diferentes tipos de expresiones, entre ellos están las condiciones utilizadas en las estructuras de control. Estas expresiones las presentamos a continuación.

Expresiones

La variable gramatical *Expr* genera expresiones. Una expresión puede definirse como "conjunto de términos que representa una cantidad", así podemos considerar a las constantes numéricas, a las variables y a las operaciones sobre ellas como expresiones. Estas expresiones se clasifican en aritméticas, lógicas, relacionales y cuánticas.

Las expresiones pueden ser binarias o unarias. Una expresión binaria tiene la forma

$$\langle \text{Expresión} \rangle \langle operador \rangle \langle \text{Expresión} \rangle,$$

y sintácticamente se resuelve de izquierda a derecha. En la representación de expresiones unarios, el operador puede ir tanto en la izquierda como en la derecha del operando. La precedencia de operadores especifica como se agrupan las expresiones. En la tabla 2.3 presentamos la precedencia entre los operadores de GAMA. Los operadores que están en la misma línea tienen igual precedencia; los renglones están en orden de precedencia decreciente. Los operadores $\hat{}t$ y $\hat{}*$ denotan a la transpuesta y a la conjugada respectivamente. El símbolo $\&$ está asociado al operador producto tensorial. Todos los demás operadores son equivalentes a los utilizados en el lenguaje C.

Expresiones simples Una expresión se genera a partir de las siguientes producciones.

Expr $\longrightarrow$ *'(' Expr ')'*
| *Cnumber*
| *VAR*
| *Identifier '(' ')'*
| *Identifier '(' ArgumentExprList ')'*
| *BLTIN '(' Expr ')'*

Son clasificadas como simples, debido a que pueden ser utilizadas como expresiones iniciales para formar expresiones más complejas. La variable gramatical *Cnumber* genera las reglas de producción de valores numéricos enteros y complejos. El terminal *VAR* está asociado al nombre de una variable. Las producciones en las que está involucrada la variable gramatical *Identifier* reconocen los llamados a funciones definidas en el programa. Por otro lado, el símbolo terminal *BLTIN* está asociado con el nombre de las funciones incluidas en la gramática de GAMA.

Expresiones aritméticas, lógicas, relacionales y la de asignación Entre las operaciones binarias generadas por las siguientes producciones se encuentran aquellas que resultan en un valor numérico o en uno booleano. Las expresiones aritméticas incluidas son la suma, la resta, la multiplicación, la división y el módulo, las cuales están asociadas a los siguientes operadores +, -, *, / y $\%$. Las lógicas binarias son la conjunción y la disyunción, denotadas por los símbolos $\&\&$ y $||$. Finalmente, las relacionales son el menor que, el mayor que, menor o igual que, mayor o igual que, igual que, y diferente que; sus operadores son $<$, $>$, $<=$, $>=$, $==$ y $!=$. En general las producciones binarias pueden representarse como:

$$\begin{aligned} \mathit{Expr} \quad &\longrightarrow \quad \mathit{Expr}\ '+'\ \mathit{Expr} \\ &\quad\ \mid \mathit{Expr}\ '-'\ \mathit{Expr} \\ &\quad\ \mid \ldots \end{aligned}$$

Las operaciones conjugado, norma, menos unitario y negación, denotados por los operadores $\hat{}*$, *norm*, $-$ y $\sim$; constituyen el conjunto de expresiones unarias. Y las producciones para reconocerlas son:

$$\begin{aligned} \mathit{Expr} \quad &\longrightarrow \quad \mathit{Expr}\ \mathit{CONJ} \\ &\quad\ \mid \mathit{NORM}\ '('\ \mathit{Expr}\ ')' \\ &\quad\ \mid\ '-'\ \mathit{Expr} \\ &\quad\ \mid\ \sim\ \mathit{Expr} \end{aligned}$$

La operación de asignación se trata de una expresión binaria con asociatividad de derecha a izquierda y el operador asociado a ella es el símbolo de igualdad "=". La parte izquierda al operador siempre consiste en una variable.

$$\begin{aligned} \mathit{Expr} \quad &\longrightarrow \quad \mathit{Asgn} \\ \mathit{Asgn} \quad &\longrightarrow \quad \mathit{VAR}\ '='\ \mathit{Expr} \end{aligned}$$

Expresiones cuánticas Pondremos especial atención a las expresiones que involucran entidades cuánticas el qubit, el quregistro y la qucompuerta presentadas en la sección 2.2. Iniciaremos con las producciones que generan qubits y quregistros.

$$\begin{aligned} \mathit{Expr} \quad &\longrightarrow \quad \mathit{NKET} \\ &\quad\ \mid \mathit{Expr}\ \mathit{NKET} \\ &\quad\ \mid \mathit{Expr}\ '+'\ \mathit{Expr} \end{aligned}$$

El símbolo terminal *NKET* fue asociado por el analizador léxico a las siguientes producciones. Estas generan los estados cuánticos en notación "ket" de Dirac, cuyas cadenas están formadas por una secuencia de ceros y unos delimiitados por los símbolos "|" y ">".

$$
\begin{array}{lcl}
NKET & \longrightarrow & \text{'|' } Aux\ Digit \text{ '>'} \\
Digit & \longrightarrow & 0 \mid 1 \\
Aux & \longrightarrow & \epsilon \\
 & & \mid Digit\ Aux
\end{array}
$$

Las compuertas cuánticas primitivas de GAMA son también un tipo de expresión. Éstas son las de Pauli (X, Y, I y Z), las de Hadamard (H), las Cnot, y las de Toffoli (T) antes mencionadas. Enunciamos las producciones que las generan a continuación.

$$
\begin{array}{lcl}
Expr & \longrightarrow & Qgate \\
Qgate & \longrightarrow & HAD \\
 & & \mid IDEN \text{ '(' } Expr \text{ ')'} \\
 & & \mid CNOT \\
 & & \mid TOFF \\
 & & \mid TOFF \text{ '(' } Expr \text{ ')'}
\end{array}
$$

Las operaciones binarias de las expresiones cuánticas son la multiplicación entre compuertas, el producto interior, el producto exterior y el producto tensorial. Para los tres primeros casos, se utiliza el símbolo "*" como su operador y para el último caso, el símbolo "&". Y son generadas a partir de las producciones de una expresión binaria. Por otro lado, las operaciones unarias que trabajan con elementos cuánticos son la transpuesta, la conjugada y la toma de medición, y sus operadores son $\hat{}t$, $\hat{}*$ y *measure*. Las producciones que cotejan este tipo de expresiones son:

$$
\begin{array}{lcl}
Expr & \longrightarrow & Expr\ TRANS \\
 & & \mid Expr\ CONJ \\
 & & \mid MEASURE \text{ '(' } Expr \text{ ',' } Expr \text{ ')'}
\end{array}
$$

La toma de mediciones, identificada por el símbolo terminal *MEASURE* requiere de dos parámetros de entrada. El primero es el qubit o quregistro sobre el cual se efectuará la operación. El segundo indica el estado cuántico por el cual se medirá el qubit o quregistro.

Operadores Ahora veamos los operadores incluidos en GAMA. En las tablas 2.4, 2.5 y 2.6 presentamos los diferentes operadores aritméticos, relacionales y lógicos, su nombre y los tipos de datos para los cuales opera.

Llamadas a funciones

Cada una de éstas es una expresión posfija, seguida de una lista de expresiones, las cuales son los argumentos de entrada de la función. En la gramática de GAMA, las llamadas a funciones constituyen otro tipo de expresiones. La sintaxis es la siguiente:

$$
\begin{array}{lcl}
Expr & \longrightarrow & Identifier \text{ '(' ')'} \\
 & & \mid Identifier \text{ '(' } ArgumentExprList \text{ ')'} \\
ArgumentExprList & \longrightarrow & Expr \\
 & & \mid ArgumentExprList \text{ ',' } Expr
\end{array}
$$

Operadores aritméticos		
Operador	*Nombre*	*Tipo*
+	suma	`Int, Complex, Qbit, Qreg, Qgate`
-	resta	`Int, Complex, Qbit, Qreg, Qgate`
*	multiplicación	`Int, Complex`
*	producto interior	`Qbit, Qreg`
*	producto exterior	`Qbit, Qreg`
/	división	`Int, Complex`
&	producto tensorial	`Qbit, Qreg, Qgate`
^t	transpuesta	`Qbit, Qreg, Qgate`
^*	conjugada	`Complex, Qbit, Qreg, Qgate`
`norm`	norma	`Complex, Qbit, Qreg`
`measure`	toma de medición	`Qbit, Qreg`

Tabla 2.4: Listado de operadores aritméticos en GAMA.

Operadores relacionales		
Operador	*Nombre*	*Tipo*
<	menor	`Int, Complex`
>	mayor	`Int, Complex`
>=	menor o igual	`Int, Complex`
<=	mayor o igual	`Int, Complex`
==	igualdad	`Int, Complex, Qbit, Qreg, Qgate`
!=	desigualdad	`Int, Complex, Qbit, Qreg, Qgate`
=	asignación	`Int, Complex, Qbit, Qreg, Qgate`

Tabla 2.5: Listado de operadores relacionales en GAMA.

Como podemos ver, el llamado a funciones se hace mediante el nombre de la función *Identifier*, seguido por la secuencia de argumentos de entrada delimitada por paréntesis. Los argumentos de entrada pueden ser nombres de variables, constantes numéricas o el llamado a funciones.

Los llamados a funciones se ejecutan desde cualquier otra función, sin problemas, siempre y cuando se hayan definido previamente.

2.3.3. Análisis semántico

Los analizadores léxico y sintáctico validan sentencias de acuerdo a las reglas gramaticales de un lenguaje, en los cuales los errores detectados corresponden, por un lado, a la identificación de símbolos y por otro lado a la sintaxis del programa [3]. Sin embargo, esto no garantiza la ejecución de proposiciones válidas, debido a la posible incoherencia en su significado. El análisis semántico se encarga de detectar errores semánticos, entre los cuales podemos mencionar la incompatibilidad de tipos de datos y

Operadores lógicos		
Operador	*Nombre*	*Tipo*
&&	conjunción	`Bool`
||	disyunción	`Bool`
~	negación	`Bool`

Tabla 2.6: Listado de operadores lógicos en GAMA.

la verificación de la declaración de variables antes de ser utilizadas [42].

El analizador semántico trata de llegar a la etapa de ejecución sin errores, sin embargo, esto es casi imposible. Son pocos los errores semánticos detectados durante la generación de código intermedio. La mayoría de los errores semánticos se centran durante la ejecución. En la generación de código intermedio se recopila información para la detección de errores semánticos en tiempo de ejecución.

Un componente principal del análisis semántico es el verificador de tipos, el cual puede admitir conversión de tipos si está definido en la especificación del lenguaje. Para ello, recopila información relacionada con los tipos de datos.

Para la definición de la semántica de un lenguaje de programación no existe una notación formal con la que se pueda describir fácilmente, debido a que tiene lugar a una relación imprescindible con la sintaxis. A partir de esto, describiremos las reglas semánticas más significativas de GAMA: declaraciones, asignaciones y proposiciones de flujo de control, auxiliándonos mediante ejemplos.

Declaraciones

La propiedad de *unicidad* consiste en evitar la duplicidad en la definición de un símbolo [2]. Ésta es necesaria en la declaración de variables y funciones para excluir la inconsistencia en los datos. Un ejemplo de esto es la declaración de dos variables con el mismo nombre como se muestra a continuación.

```
Int a;
Complex a;
```

Estos tipos de errores se detectan durante la ejecución. Ésta se realiza mediante un intérprete, el cual mantiene una tabla de símbolos. El manejo de tal tabla cuenta con mecanismos de búsqueda para garantizar la unicidad de los datos.

Asignaciones

En las expresiones matemáticas, la asignación funge como el mecanismo de cambio del valor de una variable. Análogo a esto, en los lenguajes de programación, las sentencias hacen las veces de una expresión matemática. Así, el conjunto de instrucciones de asignación modifican el estado de las variables del programa. Semánticamente, una asignación necesita compatibilidad entre los tipos de datos de ambos operandos. Sin embargo, la especificación de GAMA puede admitir ciertas coerciones a los operandos.

En la tabla 2.7 presentamos las conversiones de tipos de datos que admite el operador de asignación. Los renglones representan los tipos de datos del operando izquierdo

y las columnas los tipos del operando derecho. Algunos ejemplos son: la asignación a una variable de tipo `Complex` de datos complejos o enteros, y la asignación a una variable con un tipo de dato `Qbit` de datos con un tipo `Qbit` y `Qreg`, en esta última, siempre que se cumpla que la dimensión del quregistro sea igual a la del qubit.

	Int	Bool	Complex	Qreg	Qbit	Qgate
Int	✔	✔	✗	✗	✗	✗
Bool	✔	✔	✗	✗	✗	✗
Complex	✔	✗	✔	✗	✗	✗
Qreg	✗	✗	✗	✔	✔	✗
Qbit	✗	✗	✗	✔	✔	✗
Qgate	✗	✗	✗	✗	✗	✔
UNDEF	✔	✔	✔	✔	✔	✔

Tabla 2.7: Conversiones de tipos para la asignación.

De manera general, el operando derecho de la asignación es generado a partir de la evaluación de una expresión, la cual puede operar con tipos de datos compatibles para una posible conversión de tipo. Las operaciones que admiten esto, son: la suma, la resta, la multipllicación, la división y el producto tensorial. En el apéndice C presentamos la compatibilidad entre tipos de datos, usando tablas similares al de la asignación.

Proposiciones de flujo de control

Las expresiones booleanas se utilizan como condicionales de las estructuras de control: `if-else`, `for` y `while`. Cada una de estas, consiste de una condición y de un cuerpo. La semántica de estas estructuras está asociada a su flujo de ejecución, en el cual el resultado de la condición determina la ejecución del cuerpo. Ilustramos la estructura de la ejecución de estas proposiciones en la sección 3.4.

El único error semántico detectado consiste en la verificación de tipo booleano del condicional.

Capítulo 3

El intérprete

La ejecución de un programa GAMA se lleva a cabo mediante un intérprete, el cual se encarga de realizar las operaciones implícitas en un programa fuente. Generalmente, los intérpretes son más lentos en comparación con los compiladores, debido a la continua traducción y ejecución del programa. Sin embargo, una de sus principales ventajas es la flexibilidad para desarrollar entornos de programación y de depuración. Además, proporciona un entorno independiente de la arquitectura de la máquina, a lo cual se le conoce como *máquina virtual* [42].

En este capítulo describimos los componentes necesarios para la ejecución de programas, así como la interacción que existe entre ellos. En la sección 3.1 detallamos la *máquina de instrucciones*, la cual es el componente principal del intérprete y la interacción con los demás componentes, tales como la tabla de símbolos, el área de memoria de instrucciones, el área de memoria de pila. Posteriormente, en la sección 3.2 describimos el esquema general del intérprete, el cual consiste de las fases de análisis y de ejecución (véanse secciones 3.3 y 3.4), de la tabla de símbolos (véase sección 3.5) y del manejador de errores (véase sección 3.6). Además, en la sección 3.7 describimos las clases *CQregister* y *CQgate*, haciendo hincapié en el mecanismo de toma de mediciones detallado en la sección 3.8.

3.1. La máquina de instrucciones

La máquina de instrucciones es una representación intermedia del programa fuente. Generalmente, esta representación es el código para una *máquina de pila abstracta* [2]. Ésta posee memorias independientes, una para las instrucciones y otra para los datos. Las instrucciones están limitadas a operaciones definidas en GAMA, descritas en la sección 2.3; a manipulación de la pila y al control de flujo. El código intermedio de una expresión simula la evaluación de una representación postfija, utilizando una pila. Cuando encontramos un operando se inserta en la pila. Por lo que se ha de generar el código para insertarlo en la pila.

A continuación, describimos sus componentes: la tabla de símbolos, la memoria de instrucciones y la memoria de pila. En las secciones posteriores explicamos con mayor detalle su funcionamiento.

3.1.1. Componentes

Tabla de símbolos

Ésta se trata de una estructura de datos que se encarga del almacenamiento y de la representación de la información. El intérprete la utiliza para llevar un registro de la información sobre el ámbito y el enlace de los símbolos [2].

Para el manejo de la tabla es necesario contar con mecanismos eficientes para el acceso a la información, los cuales deben permitir la inserción y búsqueda de nuevos símbolos. El mecanismo para el manejo de la tabla de símbolos que presentamos en la sección 3.5 es una lista lineal. Aunque la implementación de tal lista es sencilla, su rendimiento es pobre cuando el número de símbolos aumenta considerablemente.

Área de memoria de instrucciones

La memoria de instrucciones almacena tanto las instrucciones como los datos del código intermedio. Las instrucciones constituyen una secuencia de operadores y operandos que se evaluarán en la fase de ejecución. Estas operaciones rigen el flujo de la máquina, el cual está generado a partir de primitivas que permitan la ejecución de la siguiente instrucción o bien el salto hacia una instrucción particular en la memoria. El código de máquina se maneja mediante apuntadores a funciones. En la ejecución de algunas instrucciones existe una interacción imprescindible entre la memoria de instrucciones, la memoria de pila y la tabla de símbolos. Con esto nos ajustamos a los procedimientos convencionales de Compiladores e Intérpretes [3].

Área de memoria de pila

Esta memoria es dinámica y sirve para llevar a cabo la evaluación de operaciones aritméticas, lógicas, relacionales y de asignación. Aumenta y disminuye su tamaño dependiendo del número de operandos, los cuales indican la posición del dato en la tabla de símbolos.

3.2. Esquema general del intérprete

El intérprete está dividido en dos fases principales, la fase de análisis y la de ejecución. La de análisis está compuesta por los analizadores léxico y sintáctico (véanse secciones 2.3.1, 2.3.2). Ambos mantienen una interacción continua, mientras el léxico se encarga de enviarle componentes léxicos al sintáctico, éste los toma, los coteja y genera la máquina de instrucciones, la cual contiene el código intermedio generado a partir de la interpretación del fuente. Durante la fase de ejecución se lleva a cabo la evaluación de instrucciones mediante un análisis semántico, como lo detallamos en la sección 2.3.3. Dos componentes esenciales en ambas fases son la tabla de símbolos y el manejador de errores.

En la figura 3.1 ilustramos estas fases y los componentes necesarios para la ejecución de un programa. Del lado izquierdo se encuentra la fase de análisis, la cual tiene como entrada el programa fuente y arroja como salida el código intermedio de la máquina

de instrucciones, mismo que entra a la fase de ejecución. En la fase de análisis son insertadas funciones, constantes y variables globales a la tabla de símbolos. Los errores pueden ser tanto léxicos como sintácticos. Por otro lado, durante la ejecución se realizan modificaciones a los elementos en la tabla de símbolos y los errores producidos son de tipo semántico o bien de ejecución.

A continuación, explicamos con mayor detenimiento las fases y los componentes del intérprete.

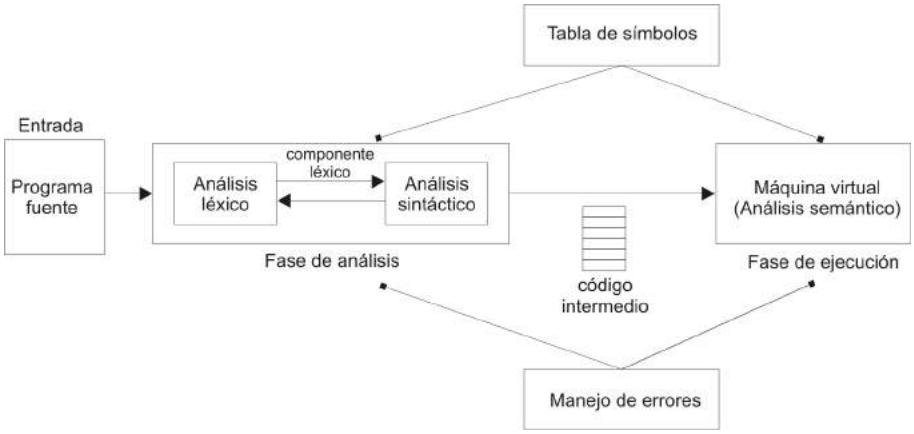

Figura 3.1: Esquema general del intérprete.

3.3. Fase de análisis

Esta fase consiste de la aplicación tanto del analizador léxico como del sintáctico que explicamos en la sección 2.3. Sin embargo, repasemos el concepto principal. El programa fuente es un archivo de texto con comandos en lenguaje GAMA, el cual es enviado al analizador léxico a través de la entrada estándar `stdin`. Éste divide el texto en componentes léxicos de acuerdo a la gramática, los cuales son enviados al analizador sintáctico. Así, mientras el léxico reconoce componentes léxicos, el sintáctico reconoce producciones e inserta las instrucciones correspondientes en el siguiente espacio libre de la máquina. Mediante esta interacción se va generando la representación del código intermedio en vez de calcular las respuestas de inmediato.

3.4. Fase de ejecución

Ésta se encarga de la ejecución o "interpretación" del código intermedio para calcular el resultado deseado. Veamos el código y la ejecución de la máquina de instrucciones para la siguiente asignación.

```
x = 2 * y
```

En el lado izquierdo de la tabla 3.1, apreciamos la lista de instrucciones que se utilizan para evaluar tal expresión y en el lado derecho explicamos el funcionamiento de

`constpush`	Inserta una constante a la pila.
`2`	... la constante `2`
`varpush`	Inserta el apuntador de la tabla de símbolos a la pila.
`y`	... para la variable `y`.
`eval`	Reemplaza el apuntador por el valor.
`mul`	Multiplica los dos operandos en el tope.
	El producto queda en el tope.
`varpush`	Inserta el apuntador de la tabla de símbolos a la pila.
`x`	... para la variable `x`.
`assign`	Guarda el valor en una variable.
`pop`	Borra el valor que se encuentra en el tope de la pila.
`STOP`	Fin de la secuencia de instrucciones.
...	

Tabla 3.1: Código intermedio y modo de operar de la expresión `x = 2*y`.

cada una de éstas. Para llevar a cabo la evaluación de esta expresión es necesario el uso de una pila, la cual almacena el resultado en la variable `x`.

Las máquinas de pila usualmente resultan en intérpretes sencillos, adecuados para lenguajes libres de contexto, y ésta no es la excepción. Se trata únicamente de un arreglo que contiene operadores y operandos. Los operadores son las instrucciones de la máquina; y cada una de ellas es la llamada a una función cuyos argumentos son los operandos.

La ejecución de la máquina es simple: consiste de varios ciclos. Cada ciclo ejecuta la función apuntada por la instrucción, a su vez apuntada por el contador de programa `pc`, e incrementa `pc` para que esté listo para la siguiente instrucción. Una instrucción con código de operación `STOP` termina la iteración. Algunas instrucciones como `constpush` y `varpush` también incrementan `pc` para pasar sobre cualquier argumento que siga a la instrucción. La habilidad de C para manipular apuntadores hacia funciones propicia el desarrollo de un código compacto y eficiente.

Las estructuras de control que reconoce GAMA son las iterativas tales como `for` y `while`, y la selectiva `if-else`. Veamos el estado de la máquina de instrucciones para `while`. Cuando se encuentra la palabra clave `while`, se genera la operación `whilecode`. Al mismo tiempo, las dos posiciones que siguen en la máquina se reservan para llenarlas posteriormente. Una vez que se ha reconocido toda la sentencia `while`, las dos posiciones extras reservadas después de la instrucción `whilecode` se llenan con las posiciones del cuerpo de la iteración y de la sentencia que les sigue. La figura 3.2 ilustra con mayor claridad este concepto.

La situación para un `if-else` es parecida, excepto que se reservan tres localidades de la máquina, una para la sentencia si se cumple la condición, otra para la sentencia si no se cumple y la sentencia que le sigue. La figura 3.3 muestra esta situación.

Tomando en cuenta que la rutina de ejecución de la máquina se mueve a lo largo de una serie de instrucciones hasta encontrar una sentencia de paro `STOP`, explicaremos el funcionamiento de las dos rutinas `whilecode` e `ifcode`. La generación de código durante el análisis sintáctico ha determinado cuidadosamente que un `STOP` termine cada secuencia de instrucciones. El cuerpo de un `while` y las partes condicionales de la

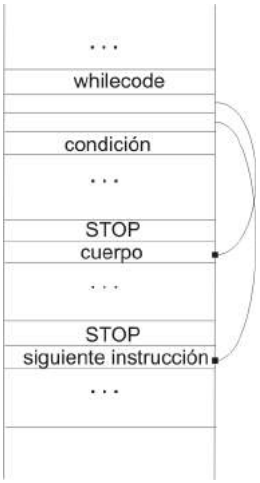

Figura 3.2: Máquina de instrucciones para la estructura `while`.

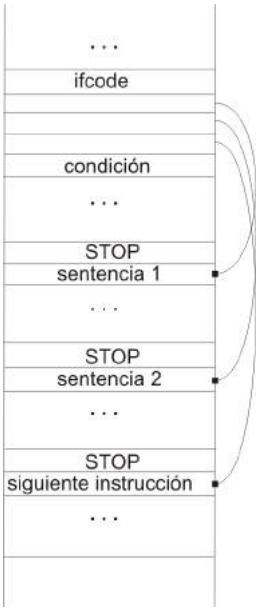

Figura 3.3: Máquina de instrucciones para la estructura `if-else`.

estructura `if-else` se manejan mediante llamadas recursivas a la rutina de ejecución, la cual regresa al nivel principal cuando han terminado su tarea. El control de estas tareas recursivas se hace mediante código en las rutinas `whilecode` e `ifcode`.

Para añadir el uso de funciones fue necesario incorporar una nueva pila para llevar el registro de funciones anidadas. Esta pila contiene toda la información de la función en ejecución; su entrada en la tabla de símbolos, la dirección de retorno después de ser llamada, la dirección de los argumentos en la pila de expresiones y el número de argumentos con el que se llamó.

Durante esta fase se realiza una comprobación dinámica de tipos, es denominada *dinámica* porque se lleva a cabo en tiempo de ejecución. Esta comprobación debe garantizar la detección y comunicación de algunas clases de errores de programación. Además, debe asegurar que los operandos de un operador tengan tipos compatibles. Un tipo compatible es aquel que es legal para el operador o bien que puede ser convertido de manera implícita por el intérprete. A esto último también se le conoce como *coerción* [2].

Antes de establecer las reglas semánticas aplicadas a la comprobación de tipos fue necesario obtener la información acerca de las construcciones sintácticas del lenguaje, la noción de tipos y las reglas para asignar tipos a las construcciones de lenguaje. Un ejemplo es 'si ambos operandos de los operadores aritméticos de suma, sustracción y multiplicación son de tipo entero, entonces el resultado es de tipo entero."

Para llevar a cabo esta tarea utilizamos un conjunto de matrices de funciones, las cuales corresponden a operaciones binarias, donde los operadores establecen la localización de la función en la matriz de acuerdo a su tipo. En caso de que no exista ninguna operación en tal localización, se trata de una no válida.

3.5. Tabla de símbolos

Como mencionamos anteriormente, en la implementación utilizamos una lista ligada dinámica de nodos, donde cada nodo es una estructura contenedora de atributos con los que se identifica a los símbolos.

3.5.1. Tipos de símbolos

Los diferentes tipos de símbolos pueden ser variables, funciones o constantes. La identificación de cada uno de éstos se realiza en la fase del análisis léxico, en la que también son insertados en la tabla de símbolos.

Los atributos asociados a cada símbolo son nombre, tipo, valor, liga al siguiente símbolo y una bandera para saber si se trata de un elemento o de un arreglo. Estos se muestran en la figura 3.4. En el atributo "tipo" se describe el tipo de dato en el caso de ser una variable o bien el tipo de símbolo para funciones o constantes. La asignación de valores a los símbolos depende de los diferentes tipos de datos. Por ejemplo, para los valores con tipos de datos enteros y booleanos, la asignación es directa sobre un entero del lenguaje `C`. Para los complejos utilizamos la biblioteca `complex` de `C++` y para las entidades cuánticas (qubit, quregistro y qugate) desarrollamos una biblioteca en `C++`, con programación orientada a objetos, para sus operaciones. A continuación,

veremos con mayor detalle la descripción de los diferentes tipos de datos asociados a las variables y la de los tipos de símbolos restantes (funciones y constantes).

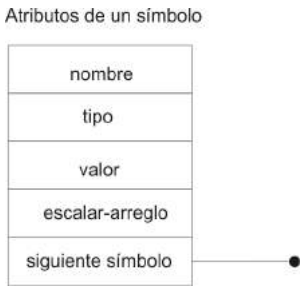

Figura 3.4: Estructura de un símbolo.

Variables

Los diferentes tipos de datos para las variables son `Int`, `Bool`, `Complex`, `Qbit`, `Qreg` y `Qgate`. Los tres últimos representan entidades cuánticas. La biblioteca que maneja éstas consta de dos clases principales *CQregister* y *CQgate*, las cuales están descritas más adelante en la sección 3.7.

Enteros y booleanos El tipo `Int` se utiliza para guardar números enteros y `Bool` para booleanos. Ambos son tipos de datos fundamentales en GAMA. Para el caso de tipo booleano utilizamos el 1 y 0 para denotar los valores lógicos *TRUE* y *FALSE* obviamente con connotación de *VERDADERO* y *FALSO*. Ambos son almacenados como un tipo `int` del lenguaje `C`, por lo que su tamaño depende de la arquitectura de la máquina. Dentro de la estructura de la tabla de símbolos sus valores son almacenados directamente, a diferencia de los demás tipos.

Complejos El tipo `Complex` se ocupa para el álgebra de números complejos esencial en las operaciones con qubits y quregistros. Utilizamos la biblioteca *complex* de `C++`, en donde un se trata un complejo como un objeto. En la tabla de símbolos únicamente se almacena la dirección de la localización de estos objetos.

Bits y registros cuánticos Los tipos de datos de ambas entidades `Qbit` y `Qreg` son objetos de la clase *CQregister*, la cual consiste de la manipulación vectorial mediante arreglos de lenguaje `C`. Básicamente, la diferencia entre ambas entidades radica en la dimensión. Mientras la dimensión de un qubit es dos, la de un quregistro es 2^n, con $n \geq 1$. En la tabla de símbolos se guarda únicamente la dirección de la localización de estos objetos.

Compuertas cuánticas El tipo de datos `Qgate` está asociado a un objeto de la clase *CQgate*. La definición de una variable tipo `Qgate` es similar a la definición de matrices cuadradas en lenguaje `C`.

Análogo a los dos casos anteriores, la tabla de símbolos almacena únicamente la dirección de la localización de cada objeto.

Funciones

Las funciones son otro tipo de símbolos dentro de GAMA, su reconocimiento es detectado en la fase de análisis sintáctica y son identificadas mediante el tipo de símbolo `FUNCTION`. Las funciones matemáticas: `sin`, `cos`, `atan`, `log`, `log10`, `exp` y `sqrt` se encuentran predefinidas en la gramática de GAMA, las cuales se almacenan en la tabla de símbolos durante la carga inicial.

Constantes

Éstas también se insertan en la tabla de símbolos durante la carga inicial. El tipo de símbolo que las identifica es `CONST`. Sus valores pueden ser enteros, booleanos o complejos y son almacenados directamente cuando se trata de enteros o booleanos y como una dirección cuando son complejos, qubits, quregistros o qugates.

3.5.2. Manejo de la tabla de símbolos

Para el acceso a los símbolos, utilizamos funciones tales como inserción, búsqueda, modificación y eliminación, las cuales fueron desarrolladas por medio de una búsqueda secuencial a lo largo de la lista. Esta técnica fue elegida debido a su sencillez, sin embargo, por depender linealmente del número de elementos en la tabla de símbolos, su rendimiento empieza a disminuir. Este manejo podría ser mejorado en el futuro.

3.5.3. Ámbito de las variables

Básicamente, las variables son clasificadas en *globales* y *locales*. Las locales son declaradas dentro de cada función y su significado está limitado a dicha función y a sus módulos. Las globales son declaradas al inicio del programa y todos los módulos pueden tener acceso a ellas.

El ambiente inicial de la ejecución de un programa tiene asociada, entre otros elementos, la tabla de símbolos global. En ésta se almacenan variables, funciones y constantes. Las variables pueden ser accesadas desde cualquier módulo del programa. En los llamados a funciones se genera un ambiente por cada uno de ellos. Éste tiene asociada una nueva tabla de símbolos válida en el ámbito de la función para el manejo de variables locales. Así, un llamado a función tiene un nuevo ambiente, de manera tal, que el llamado recursivo de funciones genera diferentes tablas de símbolos. La liberación de la memoria que ocupa una tabla se realiza automáticamente al terminar el llamado a la función. La figura 3.5 ilustra el alcance de las variables con respecto a la tabla de símbolos asociada a la función. Los diversos ambientes se manejan mediante una pila. El primer espacio pertenece al ambiente inicial, el cual contiene la tabla de símbolos global. Los demás pertenecen al llamado a la función `main` y a la función `factorization` de manera recursiva. Donde, cada nuevo ambiente tiene asociado una tabla de símbolos local diferente.

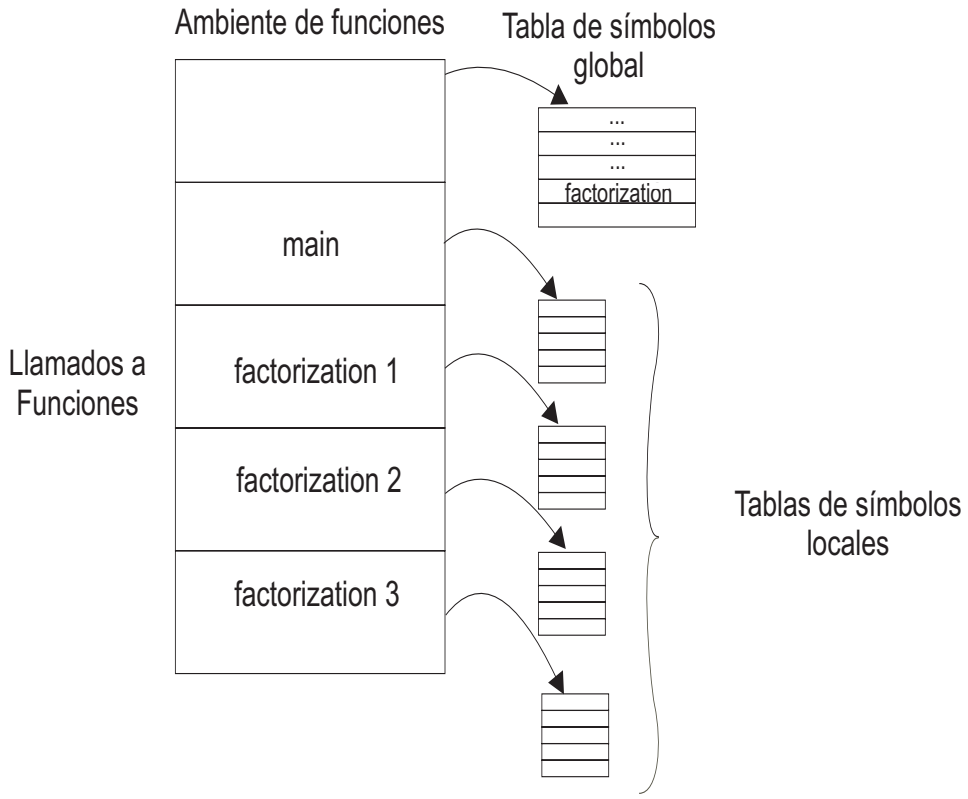

Figura 3.5: Ámbito de variables globales y locales.

3.6. Manejador de errores

Los errores pueden ser producidos en cualquiera de las fases del intérprete y son generados por diversas causas. Entre ellas están los errores léxicos, por ejemplo, escribir mal un identificador, palabra clave u operador, los sintácticos, tales como una expresión aritmética con paréntesis no equilibrados, los semánticos, como un operador aplicado a un operando incompatible, y los lógicos, como una llamada infinitamente recursiva. La mayoría de errores son detectados en la fase de análisis. Una razón es que muchos errores son de naturaleza sintáctica o se manifiestan cuando la cadena de componentes léxicos que proviene del analizador léxico desobedece las reglas gramaticales que definen a GAMA.

Aunque algunos intérpretes y compiladores realizan la recuperación de errores, GAMA no lo hace. Éste únicamente avisa al usuario la posible causa de error y si es de tipo `WARNING` o `EXCEPTION`. Cuando se trata de `WARNING` la ejecución continúa normalmente, sin embargo cuando se trata de un error tipo `EXCEPTION` se le avisa al usuario del error, y se procede con la eliminación de las tablas de símbolos, la inicialización de la máquina de instrucciones, dejando el intérprete listo para una nueva ejecución.

3.7. Descripción de las clases *CQregister* y *CQgate*

Para realizar las operaciones fundamentales de una computadora cuántica descritas en el capítulo 1, introducimos las clases *CQregister* y *CQgate*. Están descritas en `C++` y se basan en los principios de la mecánica cuántica. Por un lado, la clase `CQregister` se utiliza para simular sistemas n-quregistros y por el otro, la clase `CQgate` se utiliza para representar operadores cuánticos. Veamos la descripción de cada una de ellas.

3.7.1. La clase *CQregister*

Para modelar un registro cuántico utilizamos la clase *CQregister*. Su representación es mediante vectores de dimensión potencia de 2, como lo expusimos en la sección 1.2.2. Así, el vector se maneja como un arreglo de números complejos, el cual contiene las amplitudes asociadas a los estados cuánticos por medio de sus índices. La única restricción que existe en la definición de este vector es que la suma cuadrada de sus elementos sea igual a uno.

Los atributos incorporados en esta clase son dimensión, número de elementos, número de columnas, número de renglones y el arreglo de objetos complejos. Los vectores columna tienen un número de columnas igual a uno y los vectores transpuestos tienen un número de renglones igual a uno.

Las operaciones implantadas sobre quregistros son suma, resta, multiplicación por un escalar, multiplicación con compuertas cuánticas, multiplicación entre quregistros, producto tensorial, transpuesta, conjugado y el mecanismo para la toma de mediciones. Obviando las operaciones vectoriales implícitas en la definición de quregistros, daremos mayor énfasis a la toma de mediciones. Esta operación constituye uno de los problemas abiertos de la computación cuántica y la abordaremos con detall en la sección 3.8.

3.7.2. La clase *CQgate*

La clase *CQgate* abstrae el concepto de transformación unitaria, entidad cuántica descrita en la sección 1.2.4. Básicamente, manipula un arreglo de números complejos que representan una matriz cuadrada. Sus restricciones son que el orden de la matriz, debe ser potencia de dos, y que únicamente se trabaja con matrices unitarias.

Además, cuenta con la definición de algunas compuertas cuánticas básicas, por ejemplo, las matrices de Pauli I, Y, Z y X (véase sección 1.2.4), las cuales pueden generar todo el espacio de matrices unitarias de dimensión dos mediante combinaciones lineales. Así como la de Hadamard (H), la *Controlled-Not* (Cnot) y la de Toffoli (T) son generadas a partir de las de Pauli. Veamos la definición de cada una de ellas.

$$H = \frac{1}{\sqrt{2}}X + \frac{1}{\sqrt{2}}Z \tag{3.1}$$

$$Cnot = |0\rangle\langle 0| \otimes I + |1\rangle\langle 1| \otimes X \tag{3.2}$$

$$T = |0\rangle\langle 0| \otimes I \otimes I + |1\rangle\langle 1| \otimes Cnot \tag{3.3}$$

Las matrices de Pauli son matrices inicializadas estáticamente. Todas las demás, se producen a partir de las combinaciones que se muestran en las ecuaciones 3.1, 3.2 y 3.3. Esta propiedad facilita y simplifica el uso de compuertas cuánticas.

Sus operaciones son suma, resta, multiplicación por un escalar, multiplicación entre matrices, multiplicación por vectores, producto tensorial, transpuesta de una matriz y conjugada de un matriz.

3.8. Toma de medición

Dado el k-quregistro $\Psi = \sum_{i=0}^{2^k-1} a_i|i_2\rangle$, donde i_2 es la representación en base 2 de i, utilizando k bits. La medición de Ψ se hace respecto al número de qubit $j \in [0, k-1]$ y se trata básicamente de una proyección del j-ésimo qubit a sus estados básicos, $|0\rangle$ o $|1\rangle$. Podemos denotar a la función M_k como la medición de un k-quregistro, definida así

$$M_k : (\Psi, j) \mapsto M(\Psi, i) \in \{|0\rangle, |1\rangle\},$$

tal que la probabilidad de que $M_k(\Psi, j)$ sea uno de los estados básicos es:

$$Prob(M_k(\Psi, j) = \epsilon) = \sum_{\substack{i=0 \\ [(i_2)]_j = \epsilon}}^{2^k-1} |a_i|^2 \tag{3.4}$$

con $\epsilon = \{0, 1\}$ e $[(i_2)]_j = \epsilon$ indica que el j-ésimo bit de i_2 es ϵ. El resultado de esta proyección sustituye al j-ésimo qubit, dando lugar a la generación de un nuevo quregistro. Por ejemplo, para el estado

$$\Psi = \frac{1}{2}|00\rangle + \frac{1}{2}|01\rangle + \frac{1}{2}|10\rangle + \frac{1}{2}|11\rangle,$$

$Prob(M_2(\Psi, 0) = |0\rangle) = \frac{1}{2}$ y $Prob(M_2(\Psi, 0) = |1\rangle) = \frac{1}{2}$.

Al realizar $M_k(\Psi, j)$ se puede afectar a la parte entrelazada del j-ésimo qubit en Ψ. Un ejemplo de esto, lo muestra el estado Φ.

$$\Phi = \frac{1}{\sqrt{6}}|000\rangle - \frac{1}{\sqrt{6}}|011\rangle + \frac{1}{\sqrt{3}}|100\rangle - \frac{1}{\sqrt{3}}|111\rangle,$$

Al medir este estado, utilizando el segundo qubit la probabilidad de que sea igual a cero es $P(M_3(\Psi, 2) = |0\rangle) = |\frac{1}{\sqrt{6}}|^2 + |\frac{1}{\sqrt{3}}|^2 = \frac{1}{2}$ y la de que sea igual a uno es

$$P(M_3(\Psi, 2) = |1\rangle) = 1 - P(M_3(\Psi, 2) = |0\rangle).$$

Aunque estamos midiendo el segundo qubit, el tercero se ve afectado indirectamente. Podemos apreciar este fenómeno mediante la siguiente factorización

$$\Phi = \left(\frac{1}{\sqrt{3}}|0\rangle + \sqrt{\frac{2}{3}}|1\rangle\right) \otimes \left(\frac{1}{\sqrt{2}}|00\rangle - \frac{1}{\sqrt{2}}|11\rangle\right),$$

en la cual observamos que el segundo y el tercer qubit están acoplados, de tal forma que la medición de uno incide en la del otro, provocando que la proyección sea a uno de los estados compuestos de ambos sistemas, en este caso $|00\rangle$ y $|11\rangle$.

En conclusión, la medición para los quregistros *entrelazados* afecta a todos los qubits involucrados, a diferencia de los quregistros *no-entrelazados*, donde se afecta a uno solo. La descomposición de tales quregistros se realiza mediante una factorización de subsistemas, tratando de encontrar el j-ésimo qubit o el subsistema que lo contenga. Esta tarea no es fácil, ya que no existe un método directo para decidir si un quregistro es factorizable o no, por un número de qubit particular. Por lo que, los únicos métodos que existen se enfocan a casos particulares. Tratar su generalización es de nuestro interés.

3.8.1. Discusión de la factorización

Veamos algunos de los problemas implicados en el proceso de factorización. Uno de ellos es muy básico y consiste en decidir si un k-quregistro es factorizable, ya sea completa o parcialmente, o bien no factorizable. Otro problema inherente a éste, es la obtención del quregistro factorizado. La búsqueda particular de una factorización entre todas las posibles combinaciones hace del método uno del tipo exhaustivo. Además, de que el proceso para factorizar un quregistro conlleva a la evaluación de diferentes pruebas de consistencia para determinar los coeficientes de los factores.

Es importante considerar tanto estados entrelazados como no-entrelazados en la descripción de la factorización de quregistros. Para ello, es necesario saber si un quregistro es factorizable o no, ya sea de modo parcial o completo. Existen pocos métodos para conocer esta información, de los cuales la mayoría trabaja para casos muy particulares. Por ejemplo, el algoritmo que presentamos en la sección 3.8.2 determina si un quregistro es factorizable o no solamente mediante qubits. A continuación, mostramos todas las posibles combinaciones de agrupaciones en las que se puede factorizar un quregistro.

Árbol de posibilidades

Teorema 1 *Un k-quregistro puede factorizarse de 2^{k-1} maneras distintas, siempre y cuando sea factorizable a nivel de qubits.*

Demostración: Sea Υ un k-quregistro factorizable en l quregistros de menor o igual longitud:

$$\Upsilon = \Upsilon_0 \otimes \cdots \otimes \Upsilon_{l-1} \tag{3.5}$$

tal que $k_0 + \cdots + k_{l-1} = k$, con $k_i = longitud(\Upsilon_i)$.

Así, el número de posibilidades para factorizar el k-quregistro mediante l factores es igual al número de posibilidades de representar a k como la suma de l sumandos enteros no nulos. Por lo que, ambos son igual a la siguiente expresión:

$$\begin{pmatrix} k-1 \\ l \end{pmatrix}$$

la cual se trata del coeficiente binomial, e indica el número de maneras de escoger l factores de $k-1$ posibilidades. Por tanto, el número total de factorizaciones del k-quregistro como producto de quregistros de acuerdo a [11] es:

$$\sum_{l=0}^{k-1} \begin{pmatrix} k-1 \\ l \end{pmatrix} = 2^{k-1}$$

Con lo que queda demostrado.
■

Nuestro algoritmo contempla el caso más básico: factorización mediante qubits, ya que de antemano sabemos que se trata de un quregistro no-entrelazado. Todos los demás casos, caen en quregistros que contienen estados entrelazados. Por ejemplo, en la ecuación 3.6 mostramos uno de ellos, el cual se trata de un 3-quregistro Ψ, compuesto tanto de estados entrelazados y no-entrelazados . Así su única forma de factorización es como la ecuación 3.7 lo indica. Donde el sistema izquierdo es un estado entrelazado y el derecho un qubit.

$$\Psi = \frac{1}{2}|000\rangle + \frac{1}{2}|001\rangle + \frac{1}{2}|110\rangle + \frac{1}{2}|111\rangle \tag{3.6}$$

$$\Psi = (\frac{1}{2}|00\rangle + \frac{1}{2}|11\rangle) \otimes (\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle) \tag{3.7}$$

Podemos numerar todas las posibles factorizaciones de este tipo de quregistros de acuerdo a su longitud. Para ello, primero denotemos a un estado entrelazado de longitud l como l-$qreg_{ent}$. Para un quregistro de longitud 3, existen tres combinaciones que involucran estados entrelazados. Éstas se aprecian en la tabla 3.2. En el primer renglón se trata del ejemplo anterior, en el cual tenemos la composición de un quregistro entrelazado por la izquierda y de un qubit por la derecha. El segundo es similar al primero, con la única diferencia de que el quregistro entrelazado se encuentra a la derecha. Finalmente, el tercero indica que el quregistro es no factorizable o bien entrelazado totalmente.

1.-	4-$qreg_{ent} \otimes (\frac{1}{2}\|0\rangle + \frac{1}{2}\|1\rangle)$
2.-	$(\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle)\otimes$4-$qreg_{ent}$
3.-	8-$qreg_{ent}$

Tabla 3.2: Numeración de las combinaciones para la factorización de un 3-quregistro con estados entrelazados.

Ahora veamos las combinaciones para un 4-quregistro descritas en la tabla 3.3. Así,

1.-	4-$qreg_{ent} \otimes (\frac{1}{2}\|0\rangle + \frac{1}{2}\|1\rangle) \otimes (\frac{1}{2}\|0\rangle + \frac{1}{2}\|1\rangle)$
2.-	4-$qreg_{ent}\otimes$4-$qreg_{ent}$
3.-	$(\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle)\otimes$4-$qreg_{ent} \otimes (\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle)$
4.-	$(\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle) \otimes (\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle)\otimes$4-$qreg_{ent}$
5.-	8-$qreg_{ent} \otimes (\frac{1}{2}\|0\rangle + \frac{1}{2}\|1\rangle)$
6.-	$(\frac{1}{2}|0\rangle + \frac{1}{2}|1\rangle)\otimes$8-$qreg_{ent}$
7.-	16-$qreg_{ent}$

Tabla 3.3: Numeración de las combinaciones para la factorización de un 4-quregistro con estados entrelazados.

el número de combinaciones para la factorización de quregistros con estados entrelazados tiene un crecimiento exponencial $2^{k-1} - 1$. Una técnica para encontrar este tipo de factorizaciones, es probar cada caso y finalizar cuando sea factorizable. Sin embargo, esto tiene una complejidad exponencial $\Theta(2^k)$.

3.8.2. Algoritmo de medición mediante la factorización de qubits

El algoritmo que presentamos a continuación hace las veces de $M_k(\Psi, i)$ describa en la sección 3.8. Está restringido a la factorización de quregistros a nivel de qubits, es decir, que se puedan poner totalmente en subsistemas representados por qubits. Por lo que, al detectar quregistros compuestos por estados entrelazados se termina el proceso de búsqueda de factores.

Consideremos el quregistro Ψ de k-qubits, definido como

$$\Psi = \sum_{i=0}^{2^k-1} b_i |i_2\rangle, \tag{3.8}$$

donde $i_2 \in \mathbb{Z}_2^n$ es la representación binaria de i. Y además los coeficientes b_i cumplen con

$$\sum_{i=0}^{2^k-1} |b_i|^2 = 1. \tag{3.9}$$

Sabemos que si el quregistro fuese factorizable al nivel de qubits, entonces Ψ, puede expresarse como

$$\Psi = \bigotimes_{i=0}^{k-1} (a_{i0}|0\rangle + a_{i1}|1\rangle), \tag{3.10}$$

es decir, el k-quregistro, puede descomponerse como el producto de k qubits, donde los coeficientes a_{i0} y a_{i1} corrresponden a los estados $|0\rangle$ y $|1\rangle$ del i-ésimo qubit.

Una vez que se obtenga esta descomposición se procede a medir el i-ésimo qubit. Tal medición se trata básicamente de una proyección del qubit a sus estados básicos, $|0\rangle$ o $|1\rangle$, de acuerdo a la función de probabilidad asociada a las amplitudes de cada estado. Para ello, generamos un número aleatorio $p \in [0, 1]$ y decidimos su valor determinista conforme a la regla:

$$b_i = \begin{cases} 0 & si \quad p < |a_{i0}|^2 \\ 1 & si \quad p \geq |a_{i0}|^2 \end{cases} \tag{3.11}$$

El resultado de esta proyección sustituye al i-ésimo qubit, dando lugar a la generación de un nuevo quregistro.

El algoritmo 2 describe las distintas tareas que se llevan a cabo para realizar la medición de quregistros no-entrelazados. Éste tiene como elementos de entrada al quregistro y al índice del qubit por el cual se realizará la medición. Primero intenta factorizar el quregistro haciendo un llamado a la función `Factorization()`. Por un lado, si el quregistro es no factorizable, genera un error y finaliza. Por otro lado, si el quregistro es factorizable, regresa el arreglo de qubits y continúa con las tareas restantes, las cuales consisten en la medición del i-ésimo qubit, la sustitución del resultado en el arreglo de qubits y la reconstrucción del quregistro de salida.

Algoritmo 2: MeasureQuregister

Entrada: Un quregistro `qA`, índice i del qubit a ser medido.
Salida: Un quregistro `qO`, tal que `qO` es obtenido del quregistro `qA` después de agregar el valor medido en el i-qubit.

```
qB = Factorization(qA);
if qB == FACTORIZABLE then
  qM = MeasureQubit(qB, i);
  ReplaceQubit(qB, qM, i);
  qO = MakeQuregister(qB);
  return qO;
else
  Error();
  Exit();
end if
```

La función `MeasureQubit()` realiza la medición del i-ésimo qubit en el arreglo de factorización y tiene como parámetros de entrada el arreglo de qubits y el índice. Primero, extrae el qubit indizado y procede a realizar la medición de tal qubit. El algoritmo 3 muestra este procedimiento. La función `GetCoefficient()` obtiene la amplitud del estado $|0\rangle$, y `Random()` genera un número aleatorio entre 0 y 1. La salida del `MeasureQubit()` es cualquiera de los valores $|0\rangle$ y $|1\rangle$, la probabilidad de que sea el primero es $|a|^2$ y la de que sea el segundo es la complementaria $1 - |a|^2$.

Algoritmo 3: Función `MeasureQubit`

Entrada: Arreglo de qubits `qA`, índice i del qubit a medir.
Salida: Qubit medido.

```
qB = qA[i];
a = GetCoefficient(qB);
p = Random();
if p < |a|² then
  return |0⟩
else
  return |1⟩
end if
```

La función `ReplaceQubit()` reemplaza el i-ésimo qubit de un arreglo.

Algoritmo 4: Función `ReplaceQubit`

Entrada: Arreglo de qubits `qA`, nuevo qubit `qB`, índice i del qubit a reemplazar.

```
qA[i] = qB;
```

La función `MakeQuregister()` construye un quregistro a partir de una serie de productos tensoriales sobre qubits los cuales están almacenados en un arreglo.

Algoritmo 5: Función `MakeQuregister`

Entrada: Arreglo de n qubits `qA`.
Salida: Un quregistro `qO`.

```
qO = qA[0];
for i=1 to n-1 do
  qO = qO ⊗ qA[i];
end for
return qO
```

El algoritmo de medición 2 parece sencillo. Sin embargo la complicación de éste radica en el proceso de factorización del quregistro, del cual se encarga la función `Factorization()`.

La factorización

Como se explicó anteriormente, el problema principal en la medición subyace en la descomposición del quregistro Ψ en sus k-qubits. Para resolverlo, replantearemos la factorización de un quregistro mediante qubits definida en la ecuación 3.10. Ésta puede verse como la factorización de un polinomio P en varias variables que no conmutan. Donde P consiste de 2^k términos, los cuales tienen asociados un coeficiente b_i y un producto de las variables x_{ij}, de la forma

$$P = b_0 x_{00} x_{10} \dots x_{(n-1)0} + b_1 x_{00} x_{10} \dots x_{(n-1)1} + \dots + b_{n-1} x_{01} x_{11} \dots x_{(n-1)1,} \tag{3.12}$$

tal que P puede expresarse como el producto de k binomios:

$$P = \prod_{i=0}^{k-1} (a_{i0} x_{i0} + a_{i1} x_{i1}) \tag{3.13}$$

Por ejemplo, cuando $k = 2$, P se escribe como

$$P = b_0 x_{00} x_{01} + b_1 x_{10} x_{11} + b_2 x_{20} x_{21} + b_3 x_{30} x_{31}. \tag{3.14}$$

y se busca factorizarlo en los siguientes binomios

$$P = (a_{00} x_{00} + a_{01} x_{01})(a_{10} x_{10} + a_{11} x_{11}) \tag{3.15}$$

En la ecuación 3.13 apreciamos que el problema es encontrar los coeficientes a_{i0} y a_{i1} de tales binomios, los cuales están relacionados con los coeficientes b_i's de la ecuación 3.12. A continuación, veremos un método de como obtener los coeficientes y las condiciones necesarias para determinar si el polinomio es factorizable mediante binomios. Primero, presentamos un ejemplo sencillo, para que después nos resulte fácil concluir tales condiciones.

Supongamos que deseamos encontrar el coeficiente a_{00} de la ecuación 3.15. Utilizaremos la derivada parcial y la propiedad de normalización de binomios para llegar a nuestro objetivo.

La derivada parcial de P con respecto a x_{00}, denotada como $P_{x_{00}}$, es

$$\frac{\partial P}{\partial x_{00}} = a_{00}(a_{10} x_{10} + a_{11} x_{11}), \tag{3.16}$$

al dividir P entre este resultado, obtenemos

$$\frac{P}{P_{x_{00}}} = x_{00} + \frac{a_{01}}{a_{00}} x_{01} = x_{00} + \gamma_{00} x_{01}, \tag{3.17}$$

es decir, $\gamma_{00} = \frac{a_{01}}{a_{00}}$, .

Por otro lado, sabemos que

$$a_{00}^2 + a_{01}^2 = 1 \tag{3.18}$$

entonces si expresamos a a_{01} en términos de a_{00} y de γ_{00} tenemos que $a_{01} = a_{00}\gamma_{00}$, y sustituyendo en 3.18, y despejando a_{00} tenemos la siguiente expresión

$$a_{00} = \frac{1}{\sqrt{1 + \gamma_{00}^2}}. \tag{3.19}$$

Podemos proceder de igual forma para encontrar a_{01}, es decir, dividir P entre la derivada parcial de P con respecto a x_{01}, para encontrar $\gamma_{01} = \frac{a_{00}}{a_{01}}$, y posteriormente despejar a_{01} de la ecuación 3.18 para encontrar

$$a_{01} = \frac{1}{\sqrt{1 + \gamma_{01}^2}}.$$

De la misma forma, hallamos los coeficientes a_{10} y a_{11}.

$$a_{10} = \frac{1}{\sqrt{1 + \gamma_{10}^2}}, \quad a_{11} = \frac{1}{\sqrt{1 + \gamma_{11}^2}}.$$

Así, observamos que el problema se reduce a encontrar los valores γ_{ij}, donde $i \in [0, n-1]$ y $j \in [0, 1]$. Además, comparando las expresiones 3.14 y 3.15 de P, podemos ver que los coeficientes b_i de P son productos de los coeficientes a_{ij} de los binomios de la siguiente forma.

$$b_0 = a_{00}a_{10},\ b_1 = a_{00}a_{11},\ b_2 = a_{01}a_{10},\ b_3 = a_{10}a_{11}$$

Lo que nos permite obtener estos valores a partir de los coeficientes b_i's del polinomio. Generando las siguientes relaciones, con las que se establecen las condiciones necesarias para probar si un polinomio es factorizable mediante binomios.

$$\begin{aligned}
\gamma_{00} &= \tfrac{a_{01}}{a_{00}} = \tfrac{b_2}{b_0} = \tfrac{b_3}{b_1} \\
\gamma_{01} &= \tfrac{a_{00}}{a_{01}} = \tfrac{b_0}{b_2} = \tfrac{b_1}{b_3} \\
\gamma_{10} &= \tfrac{a_{11}}{a_{10}} = \tfrac{b_1}{b_0} = \tfrac{b_3}{b_2} \\
\gamma_{11} &= \tfrac{a_{10}}{a_{11}} = \tfrac{b_0}{b_1} = \tfrac{b_2}{b_3}
\end{aligned} \tag{3.20}$$

De este ejemplo podemos observar lo siguiente:

1. Es posible que las igualdades entre las fracciones de los coeficientes b_i no se cumplan, en cuyo caso tenemos un criterio para decidir cuando P no es factorizable mediante binomios.

2. Debido a $\gamma_{i0} = \gamma_{i1}^{-1}$, es suficiente con encontrar un valor γ_{ij} para obtener los coeficientes a_{ij} del i-ésimo binomio.

3. Cada γ_{ij} puede ser generada a partir de la división de diferentes combinaciones de los coeficientes b_i's, para conocerlas es necesario establecer una función que determine la relación entre las parejas.

Al generalizar el procedimiento (partiendo del polinomio 3.12) para obtener una factorización de la forma 3.13, tenemos que encontrar las distintas γ_{ij} asociadas a los coeficientes de cada binomio en la factorizacion de P.

Computación de las variables γ_{ij}

Hemos establecido que para encontrar los coeficientes del i-ésimo binomio de P, $(a_{i0}x_{i0} + a_{i1}x_{i1})$, es suficiente con encontrar γ_{i0} o γ_{i1}.

La computación de estas variables depende del número n de binomios que sería posible obtener de P, y del i-ésimo binomio que se desea obtener. Para el caso que nos ocupa, podemos generalizar a γ_{ij}, donde $i \in [0, n-1]$ y $j \in [0, 1]$, como

$$\gamma_{i0} = \frac{a_{i1}}{a_{i0}} \tag{3.21}$$

$$\gamma_{i1} = \frac{a_{i0}}{a_{i1}}. \tag{3.22}$$

Además, como las variables γ_{ij} resultan de la división de dos coeficientes b_i del polinomio original P, entonces debemos buscar las combinaciones de los b_i cuya división sea igual a la división de los coeficientes a_{i0} y a_{i1}.

Al observar las relaciones entre los coeficientes b_i y las γ_{ij} de la ecuación 3.20 tenemos una primera aproximación, de esta relación. Si consideramos el caso $n = 3$, entonces el polinomio P tendrá 8 coeficientes b_k, con $k \in [0, 7]$, donde los b_k's están relacionados con los productos de los coeficientes $a'_{ij}s$ como

$$b_0 = a_{00}a_{10}a_{20},\ b_1 = a_{00}a_{10}a_{21},\ b_2 = a_{00}a_{11}a_{20},\ b_3 = a_{00}a_{11}a_{21}, \tag{3.23}$$

$$b_4 = a_{01}a_{10}a_{20},\ b_5 = a_{01}a_{10}a_{21},\ b_6 = a_{01}a_{11}a_{20},\ b_7 = a_{01}a_{11}a_{21} \tag{3.24}$$

Análogo al caso $n = 2$, los valores de γ_{i0} están determinados como

$$\begin{aligned}
\gamma_{00} &= \frac{a_{01}}{a_{00}} = \frac{b_4}{b_0} = \frac{b_5}{b_1} = \frac{b_6}{b_2} = \frac{b_7}{b_3} \\
\gamma_{10} &= \frac{a_{11}}{a_{10}} = \frac{b_2}{b_0} = \frac{b_3}{b_1} = \frac{b_6}{b_4} = \frac{b_7}{b_5} \\
\gamma_{20} &= \frac{a_{21}}{a_{20}} = \frac{b_1}{b_0} = \frac{b_3}{b_2} = \frac{b_5}{b_4} = \frac{b_7}{b_6}
\end{aligned}$$

La figura 3.6 nos ayuda a ver la distribución de los b_k para cada γ_{ij}, y se aprecia una estructura de mariposa.

Podemos ver que los coeficientes utilizados para calcular el valor de γ_{0j} del primer binomio tienen un salto en su índice k de $2^{n-1} = 4$. Para el segundo binomio, los saltos de los índices son $2^{n-2} = 2$, pero se aprecia que esta relación se realiza por un bloque de 4 elementos (que es el caso del primer binomio cuando $n = 2$), por lo que los demás

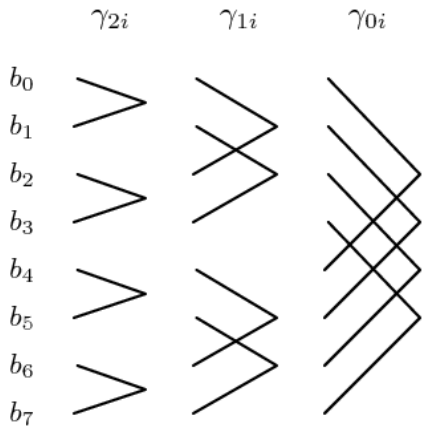

Figura 3.6: Relación de los coeficientes b_k's del polinomo P con γ_{ij}.

coeficientes se relacionan en el segundo bloque con saltos de 2. Y para el tercer binomio, los saltos del índice se realizan en $2^{n-2} = 2$, y se relacionan en parejas, o bloques de 2.

Podemos obtener el comportamiento de la relación de los coeficientes con estas características. Entonces si tenemos un arreglo B de tamaño 2^n, donde se almacenen los coeficientes b_k, y si conocemos el número total de binomios n y el binomio i asociado a la variable, digamos γ_{i0}, entonces podemos obtener los coeficientes b_k's que pueden dividirse para obtener el valor de γ con el algoritmo 6:

Algoritmo 6: Función `ComputeGamma`

Entrada: $n > 0$, $B[2^n]$, y $i \in [0, n-1]$.
Salida: γ_{i0} o un mensaje de error.

```
lim = 2^n
jump = 2^(dim-i-1)
inc = 0
for k=0 to lim-1 do
  for j=0 to jump-1 do
    num = j + inc
    den = j + jump + inc
    if i = 0 then
      γ_i0 = B[num]/B[den]
    else if γ_i0 ≠ B[num]/B[den] then
      return NO_FACTORIZABLE
    end if
  end for
  inc = den + 1
end for
return γ_i0
```

Si regresamos al problema de descomposición del quregistro, las amplitudes de $|\psi\rangle$ se almacenan en el vector A, y podemos utilizar este algoritmo para calcular los valores γ que se utilizarán para formar las amplitudes de los n qubits. Note también que este algoritmo cuenta con un criterio para decidir si P es factorizable, lo que se traduce en

si $|\psi\rangle$ está en un estado puro [45].

La función `Factorization()`

Una vez que hemos establecido que los coeficientes a_{ij} de los binomios pueden calcularse a partir de las variables γ_{ij}, podemos escribir a la función `Factorization()` presentada en el algoritmo 7. La entrada de esta función es un quregistro `qA` de n qubits y esperamos como resultado el arreglo de qubits en el que se factoriza si acaso se trata de uno factorizable, en caso de no serlo regresará un mensaje de error. La función `CoefficientArray(qA)` se encarga de tomar los coeficientes del quregistro `qA`. Una vez que obtenemos tales coeficientes determinamos las diferentes γ_{ij}'s con la función `ComputeGamma()` descrita anteriormente. Si el resultado es factorizable entonces los qubits en los que se factoriza el quregistro `qA` se depositan en el arreglo de salida `qB`.

Algoritmo 7: Factorization

Entrada: Un quregistro `qA` de n qubits.
Salida: Un arreglo `qB` de qubits cuyo producto es el quregistro `qA` o mensaje de que `qA` no es factorizable.

```
A = CoefficientArray(qA);
Inicializar el arreglo qB de n qubits.
for i = 0 to n − 1 do
  γ_i0 = ComputeGamma(A, n, i)
  if γ_i0 == NO_FACTORIZABLE then
    return NO_FACTORIZABLE
  else
    a0 = 1/ Sqrt(1+γ_i0^2 );
    a1 = 1/ Sqrt(1+(1/γ_i0)^2 );
    qA[i] = MakeQubit(a0, a1);
    qB[i] = qA[i];
  end if
end for
return qB
```

3.8.3. Conclusiones de la medición

1. El algoritmo que presentamos es muy sencillo, ya que resuelve la factorización de qubits para un caso particular. Cuando se trata de un quregistro factorizable a nivel de qubits. Sin embargo, es de gran ayuda en la inicialización de este problema.

2. Actualmente la investigación de estados entrelazados es un problema abierto, por lo que es necesario enfocarse a estos problemas, partiendo desde lo más básico.

3. El problema de descomposición de quregistros puede plantearse como una factorización de binomios, en la cual buscamos sus respectivos coeficientes binomiales a partir de los coeficientes de un polinomio de 2^n términos.

4. Los coeficientes se obtienen a partir de la división de distintos coeficientes del polinomio original. Estas combinaciones nos ayudan a determinar si el polinomio es factorizable o no.

5. Desde este enfoque, el problema se complica cuando se desea generalizar el mecanismo para tratar con estados entrelazados durante el proceso de factorización, ya que el problema cambia a encontrar una factorización de 2^n-nomios.

Capítulo 4

Entorno de simulación

Una herramienta de programación es considerada un sistema que asiste al programador en la programación. Actualmente es muy común encontrar este tipo de herramientas con un módulo de depuración integrado y la razón es aumentar la productividad del programador. Además, es mejor para los programadores contar con una sola herramienta de programación en vez de varias. Las ventajas que éstas ofrecen son una integración consistente, facilidad de uso y funcionalidades de altos niveles. Estas herramientas son conocidas como *Entornos de Desarrollo Integrado* (EDI), los cuales permiten la edición, la compilación y la depuración de programas de una forma práctica [38]. Además, la integración de una *Interfaz Gráfica de Usuario* (IGU) ofrece un entorno más natural para el programador, permitiendo insertar puntos de ruptura directamente sobre el código fuente y conocer el estado de la ejecución del programa. Por otro lado, la incorporación del depurador sobre el compilador proporciona al usuario el control de la ejecución del programa, el manejo de puntos de ruptura y la inspección de variables.

Este capítulo lo hemos dividido en dos secciones principales. En la primera definimos la arquitectura de nuestro entorno de simulación para el lenguaje GAMA, la cual consiste de la integración de tres componentes: el de edición (sección 4.1.1), el de ejecución (sección 4.1.2) y el de depuración (sección 4.1.3). Debido a la importancia del depurador para el proyecto, decidimos realizar una extensión del tema en la sección 4.2, haciendo énfasis en las vistas de las tareas básicas de depuración 4.2.1, en el núcleo del depurador 4.2.2 y la forma de comunicación entre ambos 4.2.3.

4.1. Integración del entorno

Las funcionalidades incorporadas en el entorno de simulación consisten de la edición, de la ejecución y de la depuración de programas GAMA. Éstas hacen de él uno del tipo EDI. La interfaz gráfica asociada al entorno de simulación está basada en una arquitectura multidocumentos, lo que permite crear, editar, guardar y ejecutar diferentes programas simultáneamente. Mientras la creación, la edición y el almacenamiento, son tareas que corresponden únicamente a la interfaz gráfica, la ejecución necesita de una interacción continua con el intérprete. Así, la arquitectura multidocumentos asigna la ejecución de un intérprete a cada uno de los programas en la sesión, de manera concurrente.

El depurador mantiene una interacción con el intérprete y la interfaz gráfica. Por un lado, participa en la generación del código intermedio por medio de la inserción de primitivas que permiten manejar tanto la definición de puntos de ruptura como la ejecución de una instrucción a la vez. Además, tiene acceso a la tabla de símbolos para la inspección del espacio de configuraciones. Por otro lado, utiliza la interfaz gráfica para mostrar diferentes vistas de la información del programa. En las siguientes secciones detallamos tanto los componentes del entorno como la estrategia de integración.

4.1.1. La edición

La interfaz gráfica de usuario presenta al programador la información y las posibles acciones a realizar, mediante el uso de imágenes y objetos gráficos (íconos, ventanas, botones). Esto permite que el usuario tenga una interacción afable con la computadora.

El *Modelo-Vista-Controlador* (MVC) es un patrón de arquitectura de software que permite separar la funcionalidad de una aplicación en tres clases principales: la que modela los datos, la que los despliega y la que media la interacción entre ambas. La figura 4.1 ilustra el patrón MVC, donde apreciamos la relación directa mediante las flechas punteadas y la indirecta por las líneas sólidas. La clase modeladora manipula el conjunto de datos. La clase de vista presenta los datos al usuario. Por último, la clase controladora funge como mediadora entre el usuario y la vista, convirtiendo las acciones del usuario en peticiones de navegar o editar datos, los cuales, de ser necesario, son transmitidos a la modeladora. Más información de este modelo se puede encontrar en [17] y [8].

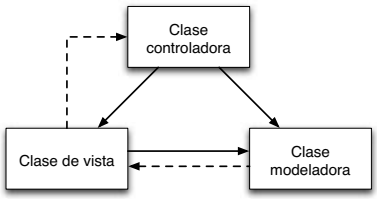

Figura 4.1: Modelo MVC, mostrando la interacción entre la clase modeladora, la clase de vista y la clase controladora.

El entorno de simulación de GAMA está basado en la arquitectura de aplicaciones multidocumentos [27]. Además, elegimos como plataforma de ejecución a `Linux` y a `Qt` como base de desarrollo de la interfaz gráfica de usuario. `Qt` es un conjunto de bibliotecas orientadas al desarrollo de interfaces gráficas para `C++` y su principal ventaja de `Qt` es la portabilidad de sus aplicaciones a diferentes manejadores de ventanas en sistemas tipo UNIX. Entre los manejadores más comunes están *Gnome, KDE, X11*, e incluso *Aqua* de Mac OS X. Otra de sus ventajas es que ofrece facilidades para trabajar con una interfaz multidocumentos. De manera tal, que el desarrollo de la interfaz gráfica se reduce a la programación del objeto gráfico de edición.

Por su parte, `Qt` proporciona un modelo *Modelo-Vista* [8] basado en el MVC clásico, en donde las clases modeladora y de vista tienen la misma función, y son nombradas Modelo y Vista. Sin embargo, la clase controladora es reemplazada por un nuevo

concepto conocido como *Delegador*, el cual se encarga de regular los elementos de edición. Qt provee un Delegador por cada tipo de vista, lo cual es suficiente para muchas aplicaciones. La figura 4.2 muestra la arquitectura de Qt.

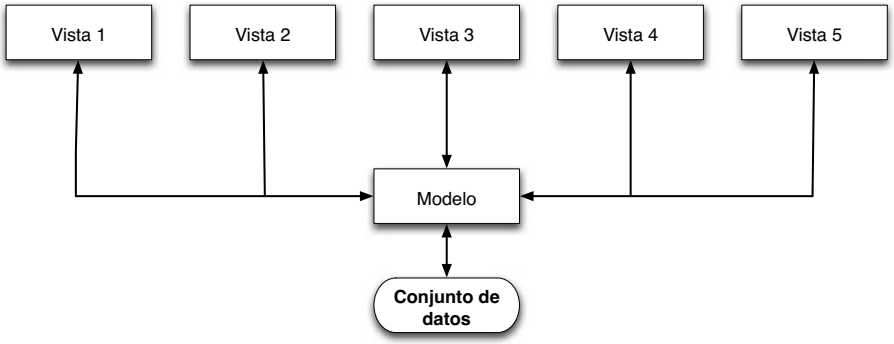

Figura 4.2: Arquitectura *Modelo-Vista* de Qt.

Usando la arquitectura de Qt, se pueden usar modelos que sólo actualicen la información que esté siendo modificada en la vista. Añadiendo dos o más vistas a la aplicación, el usuario tiene la oportunidad de ver e interactuar con los datos de diferentes maneras, cuando se tenga en condiciones de sobrecarga. Qt automáticamente mantiene múltiples vistas en sincronización y refleja sus cambios inmediatamente.

Una aplicación multidocumento consiste en una ventana principal con una barra de menú, una barra de herramientas, una barra de estado y un área de trabajo. Las opciones disponibles en la barra de menú permiten realizar tareas generales en el entorno. La barra de herramientas contiene botones distinguidos por íconos. Estos botones están asociados a la ejecución de las tareas usuales. La barra de estado muestra mensajes del estado del programa. Por último, el área de trabajo puede desplegar una o varios ventanas de documentos. La zona principal de la ventana muestra el documento activo. Particularmente, la interfaz gráfica del entorno de GAMA es una del tipo multidocumentos. Y la característica principal de ésta es que cada ventana de documento consta de un área de edición de programas y un área de visualización del espacio de configuraciones, con esto último nos referimos a los valores de las variables incluidas en el programa. En la figura 4.3 presentamos la interfaz gráfica del entorno de GAMA, donde se pueden apreciar las partes que conforman a una aplicación MDI.

4.1.2. La ejecución

La arquitectura *MDI* de la interfaz gráfica permite la ejecución simultánea de diferentes procesos. Originalmente el controlador de ventanas de la interfaz gráfica tiene

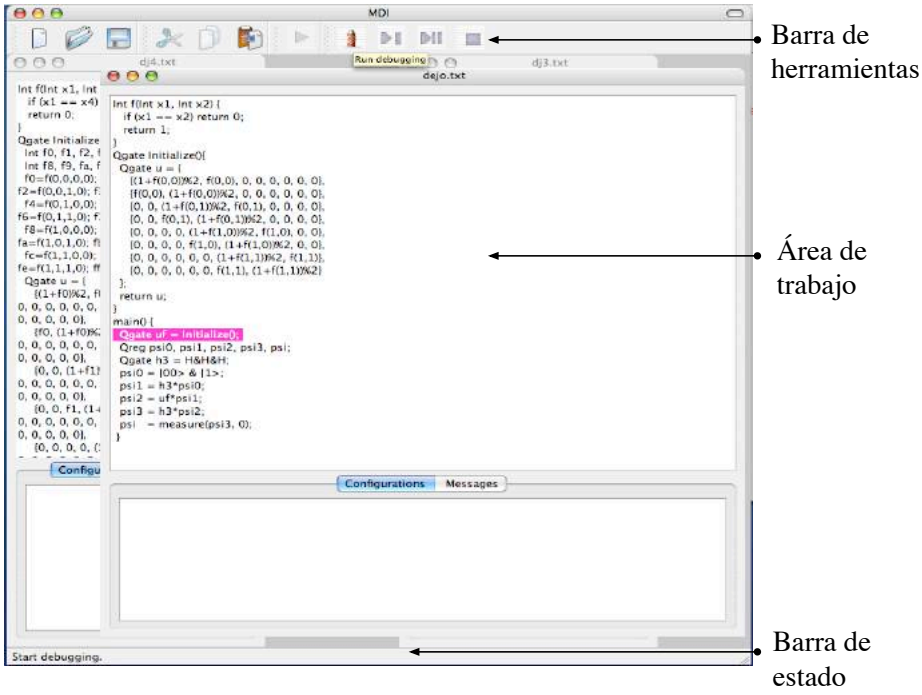

Figura 4.3: Interfaz gráfica multidocumentos.

asociado un hilo de ejecución principal (proceso padre), el cual se encarga de aceptar los eventos del usuario. Cada vez que se crea o se carga un archivo, éste genera una nueva ventana y crea un nuevo proceso *intérprete* (hijo). La comunicación entre ambos procesos se realiza mediante tuberías bidireccionales asociadas a su salida y a la entrada estándar [28]. El enfoque multitarea del entorno proviene de la ejecución simultánea de procesos intérprete asociados a cada documento abierto.

En la figura 4.4 mostramos mediante un diagrama el mecanismo de ejecución multitarea. Inicialmente, el proceso *padre* (controlador de ventanas de la interfaz gráfica) crea el canal de comunicación, en este caso, una tubería bidireccional. Mediante la función `fork()` se genera un proceso intérprete (hijo) de GAMA y se asocia la tubería a ambos procesos (padre-hijo). Una vez que se establece el canal de comunicación, se procede al envío del programa GAMA que permitirán tanto la evaluación de expresiones como el despliegue de información asociada a las variables.

El intérprete de GAMA considera dos modos de ejecución y para ello utiliza los comandos `run` y `debugger`. Por un lado, `run` consiste de la ejecución en segundo plano de un programa. Por otro lado, el modo `debugger` realiza la ejecución de acuerdo a lo que el usuario indique (ya sea mediante ejecución de puntos de ruptura o ejecución por paso). Ambos modos necesitan del envío inicial del programa fuente para su posterior interpretación. Por omisión, el intérprete se encuentra en modo `run`.

4.1.3. La depuración

La depuración (*debugging*) en cualquier lenguaje de programación permite acceder, en cualquier instante de ejecución, a la configuración interna de la plataforma en la que se esté corriendo un programa: valores internos de variables e instrucciones actuales

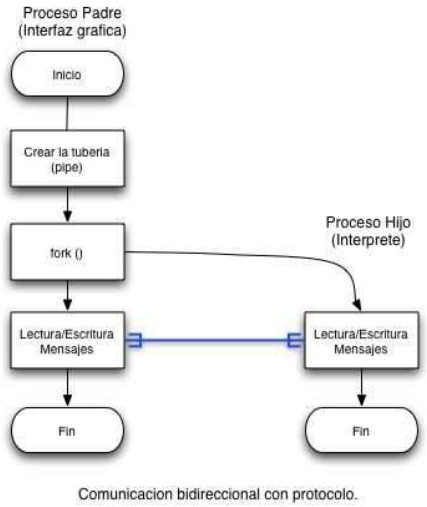

Figura 4.4: Proceso de comunicación entre la interfaz gráfica y el proceso intérprete.

[36] [1].

Las funciones de depuración nos permiten rastrear las configuraciones que se están originando con el algoritmo cuántico. El depurador proporciona la posibilidad de ejecutar el algoritmo, instrucción por instrucción, que permite visualizar las variables en todo momento y establecer puntos de ruptura.

Para llevar a cabo la depuración es necesaria la interacción de la interfaz gráfica y del intérprete. Por un lado, la interfaz gráfica es la que permite la interacción directa con el usuario, quien establece el control de la ejecución mediante puntos de ruptura o bien de la ejecución instrucción por instrucción. Por otro lado, el intérprete cuenta con primitivas de depuración que hacen posible la evaluación de expresiones, el acceso a la tabla de símbolos y la interrupción de la ejecución en los puntos indicados por el usuario.

Esta herramienta permite al usuario conocer el estado actual de las variables en la ejecución de un algoritmo cuántico, lo cual le permite verificar los resultados de las operaciones efectuadas y considerar una posible alteración en el código. En la siguiente sección explicaremos a detalle cada una de las tareas que conlleva la interacción de la depuración tanto con la interfaz gráfica como con el intérprete de comandos.

4.2. El depurador

La herramienta de programación que nos ayudará a llevar a cabo la tarea de depuración es conocido como *depurador*. Existen diferentes tipos de depuradores, entre los cuales destacan: los depuradores a nivel de código máquina (*Machine-level Debugger*) y los depuradores simbólicos (*Source-level Debugger*) [24] que trabajan con lenguajes de alto nivel. La diferencia principal entre ellos radica en la naturaleza del código que manejan. Así, mientras el código máquina es abstracto, el código en lenguaje de alto nivel resulta más natural para el usuario. Los depuradores simbólicos permiten al

usuario interactuar directamente con el código fuente, debido a que tienen la capacidad de abstraer diferentes conceptos como variables, funciones y expresiones. El punto de ruptura (*breakpoint*) es el concepto fundamental para este tipo de depuradores, el cual consiste en la definición de un punto en el código fuente donde se suspenderá la ejecución. Particulamente, GAMA simplifica la interacción con el depurador y ayuda a visualizar el comportamiento del programa a través de la interfaz gráfica de usuario.

4.2.1. Interfaz de usuario

Es necesario proporcionar al usuario una interfaz que facilite las tareas de depuración. Para ello, incorporamos los componentes gráficos (botones de control, indicador de los puntos de ruptura, iluminación de la instrucción en ejecución (*highlighting*) y ventana de visualización de variables). A continuación, explicaremos la funcionalidad de cada uno de estos componentes.

Vista del código

Para facilitar al usuario el acceso a las funcionalidades de depuración se utiliza la *vista del código*, la cual proporciona un medio transparente para el usuario. De esta manera, el usuario percibe únicamente la ejecución del código fuente y desconoce los mecanismos implicados en el proceso, lo que resulta en un depurador de alto nivel. Esta vista está ligada directamente con el editor del programa y permite que el usuario conozca la información asociada de las instrucciones a ejecutar.

Entre los principales componentes gráficos se encuentran los botones de control, el indicador de los puntos de ruptura, la iluminación de la instrucción en ejecución y la ventana de visualización de variables. Los botones de control están asociados a las funciones de depuración de: inicio de depuración (`StartDebugging`), ejecución de la siguiente instrucción (`NextStep`), ejecución hasta el siguiente punto de ruptura `NextBreakpoint` y fin de la depuración (`StopDebugging`), estos se encuentran en la barra de herramientas reconocidos mediante íconos. El indicador de los puntos de ruptura se trata de una señal colocada a la izquierda del código fuente, el cual apunta a las posibles pausas de la ejecución, dadas por el usuario. El establecimiento de puntos de ruptura permite definir el control de la ejecución, el cual es reflejado en la especificación de pausas para examinar las variables. Para llevar a cabo la iluminación de la instrucción actual en ejecución es necesaria la ayuda del componente gráfico (*highlighting*) que para resaltar la instrucción en actual ejecución en el área de trabajo. Por último, la ventana de visualización de variables se trata de una tabla que muestra el estado de las variables involucradas en el programa. La figura 4.5 ilustra esta vista, indicando cada uno de sus componentes.

Espacio de configuraciones

El espacio de configuraciones de un algoritmo cuántico está conformado por todas las posibles instanciaciones que pueden ser asumidas por todas las variables involucradas, ya sea de manera implícita o explícita, en el algoritmo. Una ejecución del algoritmo cuántico es propiamente una trayectoria en el espacio de configuraciones, en la cual el paso de una configuración a la siguiente, está dada por la aplicación de una

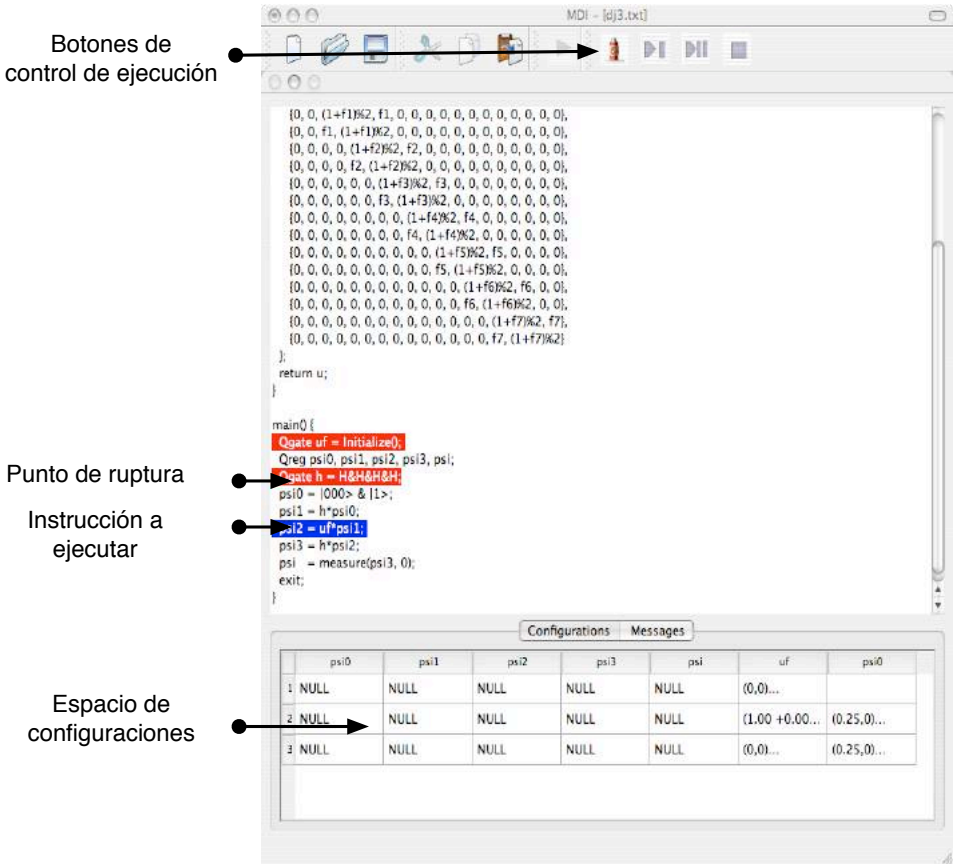

Figura 4.5: Vista de un programa en depuración.

instrucción. La evolución de las configuraciones nos permite visualizar el funcionamiento interno del algoritmo y al revisar la configuración final [33] [24], determinar si efectivamente el algoritmo es correcto.

De manera específica, dada una función f de un dominio en su contradominio, cada punto del dominio determina una instancia inicial y ésta se codifica como una configuración inicial. Al final de la ejecución en las variables de salida comprobaremos que los valores de la función están siendo asumidos con probabilidad 1 de acuerdo con la función evaluada f.

4.2.2. El núcleo del depurador

La facilidad con la que se adaptó el depurador al código máquina es una de las ventajas de utilizar un depurador a nivel de código. Entre los elementos que se agregaron están las primitivas `reset`, `debugger`, `run` y `pause`, las banderas `FLAGEXE`, `FLAGCOM` y `FLAGSYM`, y el control sobre el número de línea de la instrucción a ejecutar. En la tabla 4.1 describimos el uso de dichas banderas y en la tabla 4.2 la funcionalidad de las primitivas y su relación con las banderas que modifican.

Como mencionamos anteriormente, la interacción entre la interfaz gráfica y el depurador es esencial, misma que describiremos a continuación. Para ello, supondremos un ambiente inicial, donde el usuario ha escrito el programa que desea depurar y ha se-

Bandera	Uso	Valores
FLAGEXE	Modo de ejecución	RUNMODE/DEBUGMODE
FLAGCOM	Control de ejecución	STEP/NEXTBP
FLAGSYM	Envío de variables	ENABLE/DISABLED

Tabla 4.1: Funcionalidad de las banderas y los valores que pueden tomar.

Primitiva	Funcionalidad
reset	Inicializa la máquina de instrucciones y la tabla de símbolos.
debugger	Indica al intérprete la ejecución en modo de depuración poniendo FLAGEXE = DEBUGMODE.
run	Indica al intérprete la ejecución en segundo plano poniendo FLAGEXE = RUNMODE.
pause	Espera el siguiente comando de ejecución del usuario.

Tabla 4.2: Primitivas de depuración.

ñalado los puntos de ruptura. En este momento, se ha establecido una conexión entre el documento y el proceso intérprete-depurador. La comunicación entre ellos se realiza mediante el envío y la recepción de mensajes, detallando los tipos de tales mensajes, ya sean de conexión, de visualización de variables, de control de ejecución, de control del número línea o bien para reiniciar el programa. La especificación de estos mensajes será descrita en la sección 4.2.3.

Una vez establecido un ambiente inicial, el usuario presiona el botón de control *StartDebugging*, mismo momento en el que se le agrega la primitiva `debugger` al inicio del código fuente, también se traducen los puntos de ruptura indicados por el usuario a la primitiva `pause`. El código fuente modificado se envía al proceso intérprete-depurador, quien lo recibe para generar el código máquina. Durante la generación del código máquina se introduce el número de línea de cada instrucción en el programa. Inmediatamente después, empieza la ejecución del programa y se detiene en la primera instrucción, esperando a que el usuario presione alguno de los botones de control de ejecución ya sea `NextStep` o bien `NextBreakpoint`. Para el primer caso, la ejecución continúa con la siguiente instrucción. Sin embargo, cuando se trata de un `NextBreakpoint` se ejecutan todas las instrucciones involucradas hasta la siguiente primitiva `pause`. Para ambos casos, la ejecución se detiene al alcanzar ese punto, esperando la siguiente acción del usuario. La visualización del flujo de ejecución se realiza mediante la iluminación de la instrucción en curso, para ello, existe una sincronización entre la línea iluminada que percibe el usuario y el número de línea de la instrucción próxima a ejecutarse. La actualización del espacio de configuraciones se realiza únicamente cuando hay cambios, estos ocurren principalmente en las instrucciones de asignación, en las de definición de variables y en el paso de parámetros de una función y son controladas con la bandera `FLAGSYM`. Todo este proceso continúa hasta terminar la ejecución del programa y se procede a reinicializar tanto la máquina de instrucciones como la tabla de símbolos.

El modo `RUNMODE` se encarga de la ejecución en segundo plano del proceso intérprete. El usuario utiliza este modo de ejecución cuando necesita observar únicamente los

resultados finales. De esta manera, tiene la posibilidad de ejecutar varios programas a la vez. En este modo no existe el intercambio continuo de mensajes entre el editor y el intérprete. Los únicos mensajes que maneja son de conexión y de resultados.

4.2.3. Formato de mensajes

El envío y la recepción de mensajes entre el editor y el proceso intérprete-depurador es de suma importancia, ya que es el enlace entre lo que el usuario está viendo y lo que el intérprete está ejecutando. De tal manera, debe existir una sincronización y consistencia entre ellos.

La identificación de los tipos de mensajes es mediante etiquetas, las cuales son nombradas de acuerdo a su función. En la tabla 4.3 mostramos los mensajes que envía el editor al proceso intérprete-depurador y en la 4.4 los que envía el intérprete-depurador al editor.

`CONNECTION`	Petición de conexión.
`NEXT`	Envía siguiente instrucción a ejecutar.
`EXIT`	Finalización del programa y cierre de conexión.

Tabla 4.3: Tipos de mensajes enviados del editor al proceso intérprete-depurador.

`CONNECTION`	Establecimiento de conexión.
`LINE`	Petición de la siguiente instrucción mediante el número de línea y envío de resultados de la instrucción previa.
`EXCEPTION`	Envío del tipo de error que finaliza la ejecución.
`WARNING`	Envío del tipo de error que solo avisan al usuario y no afectan a la ejecución del programa.

Tabla 4.4: Tipos de mensajes enviados del proceso intérprete-depurador al editor.

Cada mensaje contiene una etiqueta seguida del símbolo ":" y la información necesaria correspondiente a cada tipo de mensaje. En la tabla 4.5 presentamos el formato de los mensajes que contienen información extra.

`NEXT`	:	Siguiente instrucción.
`LINE`	:	El número de línea y el resultado de la instrucción previa.
`EXCEPTION`	:	Tipo de error que finaliza la ejecución.
`WARNING`	:	Tipo de error de aviso.

Tabla 4.5: Formato de mensajes.

Capítulo 5

Pruebas de efectividad

En los capítulos anteriores hemos presentado el lenguaje de programación GAMA, su intérprete y su entorno de simulación. Esto se resume en una herramienta para la simulación de algoritmos cuánticos en una computadora clásica. En este capítulo tomamos como caso de estudio, como prueba de la efectividad del simulador, al algoritmo de Deutsch-Jozsa.

En la sección 5.1 retomamos el algoritmo de Deutsch-Jozsa y presentamos el pseudocódigo de un caso particular. En la sección 5.2 describimos el programa en GAMA para simular este algoritmo y en la sección 5.3 elaboramos una discusión de este caso de estudio.

5.1. Algoritmo de Deutsch-Jozsa

Como mencionamos en el capítulo 1, la principal ventaja de los algoritmos cuánticos es la reducción de su complejidad con respecto a la de los algoritmos clásicos. Uno de los algoritmos cuánticos que más se ha estudiado para explicar esta reducción es el algoritmo de Deutsch-Jozsa [14, 21, 9], en donde el aspecto fundamental es el uso del operador U_f. Presentamos este operador con el propósito de obtener la simulación de tal algoritmo [26].

El problema de Deutsch-Jozsa se considera entre los problemas con promesa [9], en el cual una caja negra identifica si acaso la función que se está evaluando es o no balanceada. De acuerdo a la descripción hecha previamente, el algoritmo de Deutsch-Jozsa considera el dominio de funciones

$$f : \mathbb{Z}_2^n \to \mathbb{Z}_2 = \{0, 1\},$$

es decir, funciones cuyo dominio es un conjunto de n números binarios y su contradominio es un número binario $\{0,1\}$.

Estas funciones pueden ser tanto balanceadas como constantes. Las funciones balanceadas consisten de igual cantidad de unos y ceros en su imagen; mientras que las funciones constantes tienen sólo ceros o sólo unos. El objetivo del algoritmo cuántico de Deutsch-Jozsa es identificar el tipo de función utilizando un solo paso de cómputo, lo que reduce el número de evaluaciones respecto al algoritmo clásico, el cual necesita evaluar entre 2 y $2^{n-1} + 1$ entradas para verificar el tipo de función.

x_1	x_2	f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7
0	0	0	0	0	0	1	1	1	1
0	1	0	0	1	1	0	0	1	1
1	0	0	1	0	1	0	1	0	1
1	1	0	1	1	0	1	0	0	1

Tabla 5.1: Funciones constantes o balanceadas de dos entradas

En la tabla 5.1 se muestra el conjunto de funciones constantes (funciones f_0 y f_7) o balanceadas $f : \mathbb{Z}_2^2 \to \mathbb{Z}_2$.

La figura 5.1 muestra esquemáticamente la serie de transformaciones que se le aplican al estado inicial $|\psi_0\rangle$ para resolver el problema de Deutsch-Jozsa. Básicamente las tres transformaciones que generan los estados intermedios $|\psi_1\rangle$, $|\psi_2\rangle$ y $|\psi_3\rangle$ son las que constituyen el algoritmo.

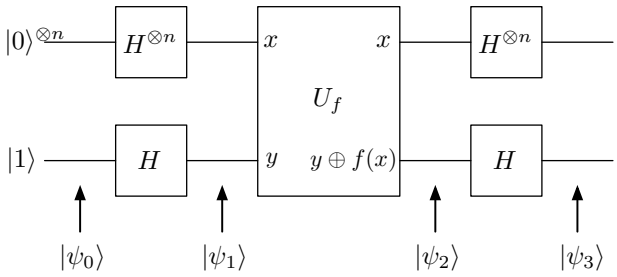

Figura 5.1: Algoritmo de Deutsch-Jozsa para un quregistro de control $|0\rangle^{\otimes n}$ y un qubit de función $|1\rangle$.

Para describir la serie de transformaciones es necesario primero establecer el estado inicial $|\psi_0\rangle$. Este estado es generado a partir del producto tensorial entre un quregistro de control (usado para almacenar los argumentos de la función) $|0\rangle^{\otimes n}$ y un qubit de función $|1\rangle$ (utilizado para evaluar la función) [10]:

$$|\psi_0\rangle = |0\rangle^{\otimes n} \otimes |1\rangle.$$

La primera transformación a $|\psi_0\rangle$ consiste en aplicarle la compuerta cuántica de Hadamard,

$$\begin{aligned} |\psi_1\rangle &= H^{\otimes n+1}|\psi_0\rangle \\ &= H^{\otimes n}|0\rangle^{\otimes n} \otimes H|1\rangle \\ &= \sum_{x\in\mathbb{Z}_2^n} \frac{1}{\sqrt{2^n}}|x\rangle \otimes \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}}\right], \end{aligned} \tag{5.1}$$

donde x, es la representación en base dos de un número entero entre $[0, 2^{n-1}]$, utilizando n bits.

La segunda transformación consiste en aplicar la compuerta U_f a $|\psi_1\rangle$. El operador U_f se encarga de realizar el mapeo de un quregistro de la siguiente manera:

$$U_f : |xa\rangle = |x\rangle \otimes |a\rangle \to |x\rangle \otimes |a \oplus f(x)\rangle$$

donde $\oplus$ denota la suma entera módulo 2, es decir:

$$0 \oplus 0 = 0, \;\; 0 \oplus 1 = 1, \;\; 1 \oplus 0 = 1, \;\; 1 \oplus 1 = 0.$$

El operador U_f tiene el efecto de introducir la fase $(-1)^{f(x)}$ al estado $|x\rangle$. Así al aplicar U_f al estado $|\psi_1\rangle$, de la ecuación 5.1 tenemos:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x \in \mathbb{Z}_2^n} (-1)^{f(x)} |x\rangle \otimes \left[\frac{|0\rangle - |1\rangle}{\sqrt{2}} \right], \tag{5.2}$$

Finalmente, la tercera transformación consiste en aplicar el operador de Hadamard a $|\psi_2\rangle$, de lo que obtenemos:

$$|\psi_3\rangle = \frac{1}{\sqrt{2^n}} \sum_{x,z \in \mathbb{Z}_2^n} (-1)^{f(x)+x\cdot z} |z\rangle \otimes |1\rangle, \tag{5.3}$$

donde $x \cdot z$ se refiere al producto escalar módulo dos, entre cadenas de bits, denotado de la siguiente forma:

$$x \cdot z = x_0 z_0 \odot x_1 z_1 \odot \cdots \odot x_{n-1} z_{n-1}.$$

El estado resultante es $|\psi_3\rangle$. Para conocer si la función es balanceada o constante, es necesario interpretar la medición del estado $|\psi_3\rangle$. Si el resultado de la medición es el estado determinista $|0^{\otimes n} 1\rangle$ entonces sabemos que se trata de una función constante, en cualquier otro caso la función es balanceada.

Antes de elaborar el pseudocódigo para su simulación en GAMA, explicaremos como generar el operador U_f.

5.1.1. El operador U_f

Como se aprecia en la ecuación 5.2, toda la información de la función a evaluar f se incluye en el quregistro $|\psi_2\rangle$ al aplicar el operador U_f. En esta sección veremos la manera expresar a U_f en forma matricial, con el propósito de tener una representación que se pueda utilizar en la simulación.

Retomemos brevemente el comportamiento del algoritmo. Notemos que el qubit de función debe ser $|1\rangle$, debido a que al aplicarle una compuerta de Hadamard a tal qubit aparece un signo negativo, como se aprecia en la ecuación 5.1. Este signo se distribuye hacia la mitad de los elementos del quregistro $|\psi_0\rangle$ durante la primera transformación. Este signo negativo es el que dispersa hacia los estados $|x\rangle$ de la ecuación 5.2 al aplicarle el operador U_f al estado $|\psi_1\rangle$ y que en la operación de medición ayuda a determinar si la función $f(x)$ es constante o balanceada.

Cuando el operador U_f se aplica al estado

$$|x\rangle \otimes H|1\rangle = |x\rangle \otimes \frac{1}{\sqrt{2}} \left(|0\rangle - |1\rangle \right),$$

se tiene

$$U_f \left(|x\rangle \otimes H|1\rangle \right) \;\; = \;\; U_f \left(|x\rangle \otimes \frac{1}{\sqrt{2}} \left(|0\rangle - |1\rangle \right) \right)$$

$$
\begin{aligned}
&= \frac{1}{\sqrt{2}} \begin{cases} |x0\rangle - |x1\rangle & \text{si } f(x) = 0 \\ |x1\rangle - |x0\rangle & \text{si } f(x) = 1 \end{cases} \\
&= \frac{1}{\sqrt{2}} \begin{cases} |x0\rangle - |x1\rangle & \text{si } f(x) = 0 \\ -(|x0\rangle - |x1\rangle) & \text{si } f(x) = 1 \end{cases} \\
&= (-1)^{f(x)}|x\rangle \otimes H|1\rangle.
\end{aligned}
$$

Y podemos ver que el signo negativo, para cada estado $|x\rangle$ dependerá del valor de $f(x)$.

Para expresar matricialmente el operador U_f, es necesario definirlo en términos de submatrices $U_{f(x)}$. Podemos escribir a la matriz $U_{f(x)}$ de tamaño 2×2 como:

$$
U_{f(x)} = \begin{pmatrix} 1 \oplus f(x) & f(x) \\ f(x) & 1 \oplus f(x) \end{pmatrix}. \tag{5.4}
$$

Y de manera general al operador U_f como una matriz $2^{n+1} \times 2^{n+1}$, definida como sigue

$$
U_f = \begin{pmatrix} U_{f(0)} & 0 & \cdots & 0 \\ 0 & U_{f(1)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_{f(2^n-1)} \end{pmatrix}. \tag{5.5}
$$

Entonces la ecuación 5.2 se escribe $U_f(2^n - 1)$

$$
|\psi_3\rangle = \begin{pmatrix} U_{f(0)} & 0 & \cdots & 0 \\ 0 & U_{f(1)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_{f(2^n-1)} \end{pmatrix} (H^{\otimes n}|0\rangle^{\otimes n} \otimes H|1\rangle) \tag{5.6}
$$

Por ejemplo, para el caso de $n = 2$, la matriz U_f se escribe como:

$$
U_f = \begin{pmatrix} 1 \oplus f_{00} & f_{00} & 0 & 0 & 0 & 0 & 0 & 0 \\ f_{00} & 1 \oplus f_{00} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 \oplus f_{01} & f_{01} & 0 & 0 & 0 & 0 \\ 0 & 0 & f_{01} & 1 \oplus f_{01} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \oplus f_{10} & f_{10} & 0 & 0 \\ 0 & 0 & 0 & 0 & f_{10} & 1 \oplus f_{10} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \oplus f_{11} & f_{11} \\ 0 & 0 & 0 & 0 & 0 & 0 & f_{11} & 1 \oplus f_{11} \end{pmatrix} \tag{5.7}
$$

donde

$$
\begin{aligned}
f_{00} &= f(0,0) \\
f_{01} &= f(0,1) \\
f_{10} &= f(1,0) \\
f_{11} &= f(1,1).
\end{aligned} \tag{5.8}
$$

Se puede ver en las matrices 5.5 y 5.7 que si la función f es constante, entonces se tiene a la matriz unitaria $2^{n+1} \times 2^{n+1}$ o a una matriz con submatrices $U_{f(x)}$ antidiagonales

en su diagonal principal. Por lo que al aplicar esta matriz a $|\psi_1\rangle$ se obtendrá a $\pm|\psi 1\rangle$, y al aplicar la compuerta de Hadamard, del último paso del algoritmo de Deutsch-Jozsa se tiene $\pm|0\rangle^{\otimes n} \otimes |1\rangle$. Entonces al realizar la medición sobre los primeros n-qbuits obtenemos que la amplitud del estado $|0\rangle^{\otimes n}$ es uno.

Si la función es balanceada, entonces U_f se expresa como una combinación de matrices $U_{f(x)}$ diagonales y antidiagonales. De esta forma, el estado resultante $|\psi_3\rangle = |z\rangle \otimes H|1\rangle$ contiene al elemento $|z\rangle$, el cual es cualquier estado distinto de $|0\rangle^{\otimes n}$. Así, la medición final dará como resultado que la amplitud de $|0\rangle^{\otimes n}$ es cero.

5.1.2. Pseudocódigo

Ahora explicaremos el pseudocódigo del algoritmo cuando $n = 2$ con el fin de entenderlo, implementarlo y probarlo en nuestro simulador de algoritmos cuánticos. El estado inicial $|\psi_0\rangle$ se expresa como:

$$|\psi_0\rangle = |00\rangle \otimes |1\rangle$$

y el estado final $|\psi_3\rangle$ se obtiene, como hemos mencionado, de la aplicacion de las transformaciones de Hadamard y U_f a dicho estado inicial como sigue:

$$|\psi_3\rangle = H^{\otimes 3} U_f H^{\otimes 3} |\psi_0\rangle$$

donde $H^{\otimes 3} = H \otimes H \otimes H$ y U_f está definido como en la ecuación 5.7.

El algoritmo 8 muestra los pasos necesarios para resolver este ejemplo. Primero se inicializa el operador U_f con la función `Initialize()` (algoritmo 9). Como hemos mencionado el operador U_f está conformado de evaluaciones de la función $f(x_1, x_2)$. Esta función se detalla en el algoritmo 10 y en este ejemplo es la función balanceada f_3 de la tabla 5.1. Los siguientes pasos corresponden a las transformaciones hechas al estado inicial ψ_0. La primera equivale a aplicarle la compuerta $H^{\otimes 3}$. La segunda consiste en la aplicación del operador U_f y finalmente la tercera en aplicarle la compuerta $H^{\otimes 3}$ nuevamente.

Algoritmo 8: Algoritmo cuántico de Deutsch-Jozsa para la función $f(x_1, x_2)$.

Salida: Un qureg istro ψ.
$U_f = Initialize()$;
$\psi_0 = |00\rangle \otimes |1\rangle$;
$H3 = H \otimes 3$;
$\psi_1 = H3 * \psi_0$;
$\psi_2 = U_f * \psi_1$;
$\psi_3 = H3 * \psi_2$;
$\psi = measure(\psi_3, 0)$;
return ψ;

El estado ψ es el estado resultante, al cual se le realizará una toma de medición, haciéndolo por cualquier qubit que corresponda al qureg istro de control, que en este caso tomamos el qubit 0 para medirlo.

Algoritmo 9: Función *Initialize()*.

Salida: Una qucompuerta `uf` que representa el operador U_f.

Qgate uf = {
$\{(1 + f(0,0))\,\%2, f(0,0), 0, 0, 0, 0, 0, 0\}$,
$\{f(0,0), (1 + f(0,0))\,\%2, 0, 0, 0, 0, 0, 0\}$,
$\{0, 0, (1 + f(0,1))\,\%2, f(0,1), 0, 0, 0, 0\}$,
$\{0, 0, f(0,1), (1 + f(0,1))\,\%2, 0, 0, 0, 0\}$,
$\{0, 0, 0, 0, (1 + f(1,0))\,\%2, f(1,0), 0, 0\}$,
$\{0, 0, 0, 0, f(1,0), (1 + f(1,0))\,\%2, 0, 0\}$,
$\{0, 0, 0, 0, 0, 0, (1 + f(1,1))\,\%2, f(1,1)\}$,
$\{0, 0, 0, 0, 0, 0, f(1,1), (1 + f(1,1))\,\%2\}$
};
return `uf;`

Algoritmo 10: Función $f(x_1, x_2)$.

Entrada: Dos valores de entrada x_1 y x_2.

Salida: El resultado de la función evaluada en los valores de entrada.

`Int f(Int` x_1`, Int` x_2`)` {
if $x_1 == x_2$ **then**
return `0;`
end if
return `1;` }

5.2. Simulación en GAMA

Presentamos el código GAMA para llevar a cabo la simulación del algoritmo, mismo que se muestra en la figura 5.2. Por ahora, estamos interesados únicamente en obtener el resultado de la simulación y no en el de estados intermedios durante una depuración. De tal manera que el estado resultante de cada una de las variables involucradas se puede observar en la pestaña inferior *Configurations*, en donde el quregistro `psi` contiene la información de si acaso la función se trata de una constante o bien de una balanceada. En la siguiente sección veremos la interpretación al resultado del algoritmo dado en tal quregistro.

5.3. Resultado

Como resultado de la simulación anterior, para el caso $n = 2$ y la función balanceada f_3 de la tabla 5.1, obtuvimos que la medición del quregistro ψ_3 generó el quregistro $\psi = (1{,}0 + 0{,}0i)|111\rangle$. Como explicamos en la sección 5.1, considerando que el quregistro ψ_0 es $|001\rangle$, diremos por un lado que si el quregistro resultante arroja el mismo valor del quregistro inicial se trata de una función constante y por otro lado, si se trata de cualquier otro estado sabemos que es una función balanceada. Bajo esta premisa observamos que el quregistro $\psi = |111\rangle$ indica que la función $f(x_1, x_2)$ es balanceada.

En el apéndice D presentamos la simulación del algoritmo de Deutsch-Jozsa para el

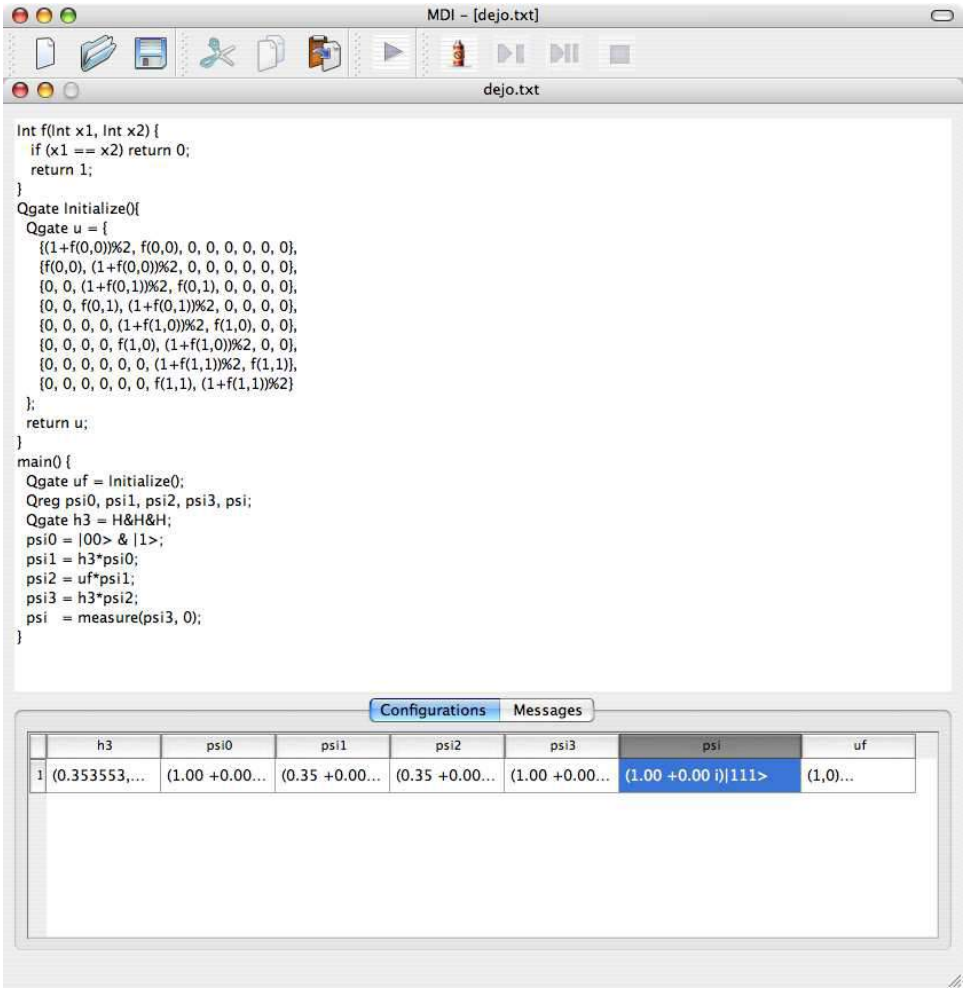

Figura 5.2: Simulación del algoritmo de Deutsch-Jozsa en GAMA.

caso $n = 2$ y una función constante. Se aprecia que el quregistro de salida $\psi = |001\rangle$, lo que indica que la función $f(x_1, x_2)$ es constante.

En este capítulo hemos realizado la simulación del algoritmo cuántico de Deutsch-Jozsa en el lenguaje GAMA. Para ello, hemos establecido un mecanismo para construir la compuerta cuántica U_f en forma matricial, que ayuda a obtener la simulación de su comportamiento. Se puede apreciar que el qubit de función $|1\rangle$ se utiliza para inducción la secuencia de signos negativos, necesarios para que el algoritmo funcione, y para que el registro de control alcance la dimensión apropiada para operarse con la compuerta cuántica U_f.

Ahora veamos la complejidad del algoritmo en la simulación. Uno de los principales procesos en que se concentra la complejidad del algoritmo es la inicialización de la compuerta U_f, la cual, además de estar relacionada directamente con el número de evaluaciones de la respectiva función f, incrementa exponencialmente su tamaño con respecto al número de qubits utilizados. Tomando en cuenta el tiempo de la inicialización de la compuerta U_f, la complejidad del algoritmo se concentra en este proceso y es exponencial. Por otro lado, si no consideramos tal inicialización, podemos suponer que la complejidad del algoritmo se reduciría. En efecto, la complejidad disminuye al no

considerar la inicialización de la compuerta, reduciéndose solamente al tiempo que se tarda en llevar a cabo las operaciones con dicha compuerta. En este caso, la mayoría de las operaciones son multiplicaciones de matrices y su complejidad es polinomial. Sin embargo, tal complejidad no se mantiene debido al crecimiento exponencial del tamaño de la compuerta, aún sin tomar en cuenta su inicialización. Esta observación es de gran importancia, ya que la complejidad de la simulación del algoritmo cuántico sigue siendo exponencial. De hecho para un número mayor de 8 qubits es necesario expandir la memoria de la computadora para que realizar las operaciones involucradas.

El desarrollo de la computadora cuántica física promete reducir las complejidades de algunos algoritmos, debido al número exponencial de operaciones que ejecutará en un sólo paso de cómputo. Sin embargo, en la actualidad todavía estamos a la expectativa de dicha computadora. Por lo pronto, la simulación de algoritmos cuánticos en computadoras clásicas realiza ese número exponencial de operaciones de manera secuencial.

En la figura 5.3 mostramos la gráfica de tiempos de las corridas del algoritmo de Deustch-Jozsa para funciones de dos a ocho qubits de entrada. Podemos observar que se trata de una función exponencial tomando en cuenta el tiempo de inicialización de la compuerta podemos apreciar un crecimiento exponencial. Posteriormente, en la figura 5.4 mostramos la misma gráfica pero tomando en consideración únicamente el tiempo de proceso del algoritmo, el cual también se comporta como una función exponencial.

Los diferentes ejemplos que se utilizaron para generar esta gráfica se encuentran en la carpeta `/ejemplos/` del disco de instalación anexo a este documento.

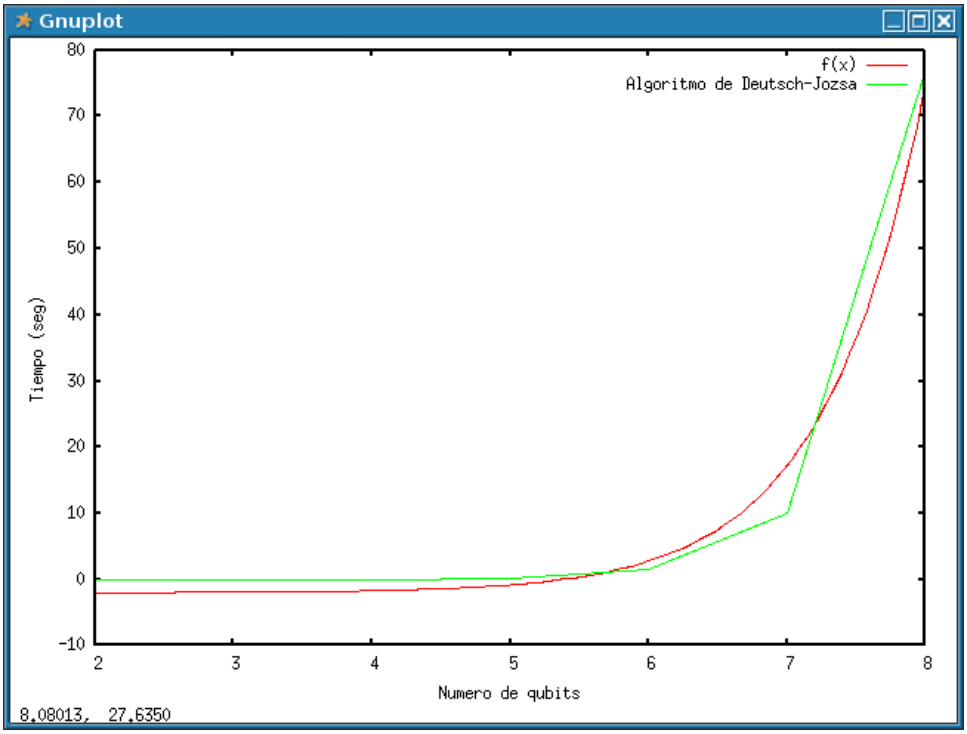

Figura 5.3: Gráfica de tiempos del algoritmo Deutsch-Jozsa.

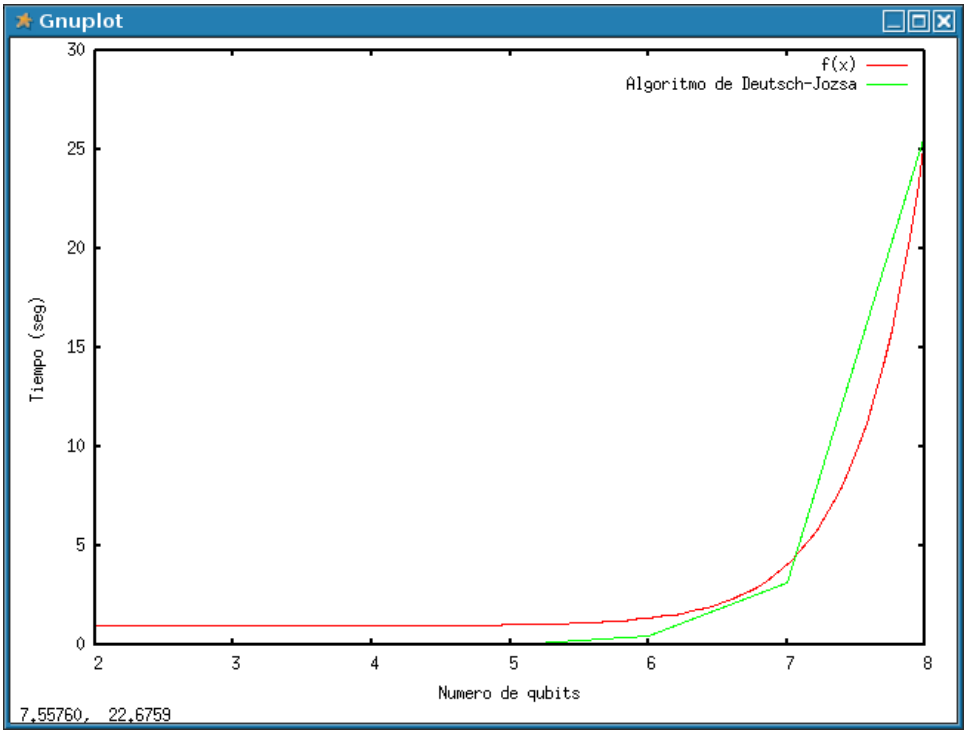

Figura 5.4: Gráfica de tiempos del algoritmo Deutsch-Jozsa sin tomar en cuenta la inicialización de la compuerta.

Capítulo 6

Resultados y conclusiones

El objetivo principal de este trabajo de tesis fue desarrollar un lenguaje de programación para la simulación de cómputo cuántico. Para ello se ha desarrollado un trabajo de diseño e implementación del lenguaje, así como un trabajo teórico para entender los fundamentos de la computación cuántica y poder plasmarlos en nuestro simulador. En este capítulo englobamos los resultados obtenidos y damos nuestras conclusiones.

6.1. Resultados

A continuación, listamos los resultados que se obtuvieron durante el desarrollo del simulador propuesto de algoritmos cuánticos. Todos ellos han sido explicados detalladamente a lo largo del documento, siendo este capítulo una breve recopilación.

Estudio de los lenguajes de programación para la simulación de cómputo cuántico: Al realizar este estudio, apreciamos que algunos de los lenguajes diseñados para este propósito no han llegado a la fase de implementación. De los lenguajes implementados la mayoría son interpretados y manejan los operadores cuánticos como llamadas a funciones. A continuación, sintetizamos las características de cada lenguaje. Dado que GAMA tiene un enfoque práctico mantiene dos importantes características que no son incluidos en ningún otro: definición dinámica de operadores cuánticos y el módulo de depuración.

Análisis de las entidades cuánticas: Realizamos un estudio de las principales entidades involucradas en el cómputo cuántico: qubit, quregistro y qucompuerta. Esto nos permitió conocer sus propiedades y operaciones básicas con el propósito de incorporarlas en nuestro lenguaje.

Biblioteca de primitivas cuánticas: Desarrollamos un módulo que contiene una biblioteca de objetos en C++ para el manejo de las entidades cuánticas qubit, quregistro y qucompuerta. Este módulo puede utilizarse de manera independiente en otros programas y aplicaciones que lo requieran.

Algoritmo para la toma de mediciones: En el desarrollo del lenguaje nos percatamos de los problemas involucrados en la toma de mediciones. Hemos desarrollado un algoritmo que permite hacer mediciones tomando como base la factorización de

qubits, como se explica a detalle en la sección 3.8. Este algoritmo funciona para casos muy particulares y es deseable tener un algoritmo más general.

Lenguaje de programación cuántica GAMA: Uno de los aspectos principales en el simulador es el lenguaje de programación. GAMA es el resultado de la investigación de diferentes lenguajes de programación de propósito específico y de propósito general general, así como del estudio realizado a las entidades básicas del cómputo cuántico. El resultado final es un lenguaje de propósito específico.

La definición de la gramática de GAMA hace de éste un lenguaje imperativo, procedimental y estructurado, como se discute a detalle en el capítulo 2. Estas características brindan las siguientes propiedades al lenguaje:

Imperativa : Incluye la concatenación de sentencias mediante el carácter ";". La declaración de variables. Estas incluyen los tipos para manejo datos enteros, complejos, booleanos, qubits, quregistros y qucompuertas. Definición dinámica de componentes cuánticos, particularmente en el caso de qucompuertas el lenguaje incluye a las matrices de Pauli, además de las matrices Hadamard y Toffoli como parte del lenguaje. Aritmética compleja y matricial. Primitivas para el manejo de operadores binarios entre diferentes tipos de datos, tales como: asignación, suma, resta, multiplicación, división y producto tensorial, por mencionar algunos. Y la incorporacion de funciones matemáticas básicas como seno, coseno, exponencial, logaritmo y raíz cuadrada, entre otras.

Procedimental: GAMA permite la definición de funciones y de recursividad en ellas.

Estructurado: El control del flujo del programa se realiza con estructuras de control condicionales (`if-else`) e iterativas (`while` y `for`). Para el manejo apropiado de dichas estructuras GAMA incluye la prueba de enunciados booleanos.

Intérprete: La ejecución de GAMA se realiza mediante el uso de un intérprete, si bien es cierto que el rendimiento baja, se obtienen beneficios con este enfoque, los principales son: la portabilidad y la facilidad de incorporarle el módulo de depuración. Se ha procurado que el intérprete sea modular para desarrollar ambientes de ejecución de GAMA en distintas plataformas de trabajo. Actualmente a nivel terminal GAMA se ejecuta en sistemas tipo UNIX, pricipalmente Linux y OS X.

Entorno de simulación: Uno de los componentes tecnológicos que facilitan el uso de un lenguaje de programación es su ambiente de desarrollo. En el caso de GAMA se ha procurado tener un ambiente gráfico que funciona como interfaz entre el desarrollador de programas GAMA y el intérprete. Se ha procurado que esta interfaz sea un ambiente integral de desarrollo que incluye un módulo de edición, un módulo de ejecución y un módulo de depuración. Además el entorno de simulación permite el manejo de varios documentos al mismo tiempo con lo que el usuario puede editar, ejecutar y depurar distintos programas GAMA en una misma sesión de trabajo. El desarrollo de este entorno de simulación se ha realizado sobre `Qt`, lo que ha permitido que en este momento se cuenten con

versiones probadas del entorno de simulación para GAMA en sistemas Linux y OS X.

Simulación del algoritmo Deutsch-Jozsa: Probamos nuestro simulador con el algoritmo cuántico de Deutsch-Jozsa. Aunque hubiésemos querido abordar algunos algoritmos con mayor complejidad. Sin embargo, sabemos que el algoritmo de Deutsch-Jozsa es un buen parámetro para probar la efectividad de la herramienta.

Análisis de simulación cuántica: La simulación del algoritmo cuántico de Deutsch-Jozsa en GAMA nos permitió corroborar la ventaja de tener la definición dinámica de los operadores cuánticos, ya que simplemente se inicializan las variables correspondientes y se aplican como multiplicaciones a los vectores que representan los estados cuánticos del sistema. Esto también nos permitió conocer el comportamiento total de este algoritmo al establecer la forma matricial del operador U_f. Además, la definicón de este operador refleja que la disminución en la complejidad del algoritmo cuántico radica en que la matriz U_f contiene todas las evaluaciones de la función que se resuelve con este algoritmo, y es precisamente la superposición la que incluye esta información al sistema cuántico durante la ejecución del algoritmo.Naturalemente la inicialización de la matriz U_f produce que la complejidad del algoritmo de Deutsch-Jozsa sea 2^n, sin embargo, como en la mayoria de los experimentos físicos realizados para probar este algoritmo, si no se incluye el procedimiento de inicialización, la complejidad se reduce a una sola evaluación.

6.2. Conclusiones

1. Efectivamente fue posible la simulación de algoritmos cuánticos mediante un lenguaje de programación de propósito específico en una computadora clásica, aunque lo hace de manera ineficiente.

2. La programación de un algoritmo cuántico en GAMA permite que el *espacio problema* sea equivalente al *espacio solución*, debido a la incorporación de las entidades cuánticas qubit, quregistro y qucompuerta en la gramática.

3. Aunque el algoritmo para la toma de mediciones es básico, probamos que es suficiente para aplicarlo en el algoritmo cuántico de Deutsch-Jozsa.

4. GAMA es considerada una herramienta de aprendizaje y experimentación para personas interesadas en conocer el paradigma del cómputo cuántico.

5. La inicialización del operador U_f es la clave para la reducción de la complejidad en el algoritmo cuántico de Deutsch-Jozsa.

6.3. Trabajo a futuro

El desarrollo de este proyecto requirió de 1 año, por lo que tratamos de obtener el simulador de cómputo cuántico lo más completo posible. Sin embargo, se trata de

una versión inicial, que a pesar de ser la primera está estructurada lo suficientemente modular para posibles modificaciones y/o mejoras. Enumeramos algunas de estas mejorías por orden de importancia.

1. Incorporar un algoritmo de medición que trabaje de manera más general, trabajo que actualmente realiza el estudiante de maestría William De la Cruz De los Santos en el Departamento de Computación del CINVESTAV, IPN, bajo la dirección del Dr. Guillermo Morales Luna.

2. Realizar un análisis más elaborado de la efectividad de la herramienta, mediante la simulación de otros algoritmos cuánticos más complejos.

3. Agregar características del lenguaje de programación C a GAMA, con el objetivo de hacerlo más robusto. Algunas de ellas son: manejo de arreglos bidimensionales, manejo de apuntadores, ampliar el manejo de estructuras de control para el `for` y el `while`, entre otros.

4. Añadir un mecanismo robusto para el manejo de errores, el cual actualmente es muy básico.

5. Ampliar la biblioteca de primitivas cuánticas con nuevas operaciones de álgebra lineal y tensorial.

6. Mejorar el entorno de simulación con funciones como: enumeración de líneas, nuevas funciones de depuración, y mejorar el formato de los datos que se presentan.

Apéndice A

Listado de componentes léxicos

```
%{
/* Declaraciones */
#include <stdlib.h>
#include <math.h>
int cont_line;   //contador de lineas
%}

/* Definiciones regulares */
integer    [0-9]+
dreal      ([0-9]*"."[0-9]+)
ereal      ([0-9]*"."[0-9]+[eE][+-]?[0-9]+)
real       {dreal}|{ereal}
nl         \n
%%

{integer}  return TINT;
{real}     return REAL;
{nl}      {cont_line++;}

//Funciones matematicas
sin        return(BLTIN);
cos        return(BLTIN);
atan       return(BLTIN);
log        return(BLTIN);
log10      return(BLTIN);
exp        return(BLTIN);
sqrt       return(BLTIN);
abs        return(BLTIN);

//Constantes matematicas
"PI"         return(CONSTS);
"E"          return(CONSTS);
"GAMMA"      return(CONSTS);
"DEG"        return(CONSTS);
"PHI"        return(CONSTS);
"i"          return(CMP);
```

```
//Tipos de datos
Int         { yylval.integer = 0; return(TIPO); }
Bool        { yylval.integer = 1; return(TIPO); }
Complex     { yylval.integer = 2; return(TIPO); }
Qreg        { yylval.integer = 3; return(TIPO); }
Qbit        { yylval.integer = 4; return(TIPO); }
Qgate       { yylval.integer = 5; return(TIPO); }

//Constantes booleanas
TRUE         return TBOOL;
FALSE        return(TBOOL);

//Operadores
"&&"        return(AND);
"||"        return(OR);
"=="        return(EQ);
"!="        return(NE);
"<="        return(LE);
">="        return(GE);
"^t"        return(TRANS);
"^*"        return(CONJ);

//Palabras reservadas
if          return(IF);
else        return(ELSE);
while       return(WHILE);
for         return(FOR);
return      return(RETURN);
main        return MAIN;

//Compuertas cuanticas
H           return(HAD);
I           return(IDEN);
Cnot        return(CNOT);
T           return(TOFF);
norm        return(NORM);
measure     return(MEASURE);

//Comandos del interprete
display     return(DISPLAY);
exit        return(EXIT);
run         return(RUN);
debugger    return(DEBUGGER);
reset       return(RESET);
pause       return(PAUSE);

//Identificadores
[A-Za-z_]([A-Za-z0-9_])*  return VAR;
```

```
//Constantes para qubits y quregistros
"|"[0-1][0-1]+">" {return(NKET);}
"|"[0-1]">"       {return(NKET);}

//Caracteres especiales
;           return(';');
=           return(ASIG);
"{"         return '{';
"}"         return '}';
[\[]        return '[';
[\]]        return ']';
[ \t]       ;
.           return(yytext[0]);
%%
```

Apéndice B

Gramática de GAMA

A continuación se presenta el conjunto de reglas sintácticas de GAMA. Las producciones se encuentran en letras cursivas, los símbolos terminales se encuentra en tipo de letra `typescript` y las palabras reservadas con **negritas**.

Start	$\longrightarrow$	*File*
File	$\longrightarrow$	*Declaration*
		| *Functions*
		| *Stat*
Functions	$\longrightarrow$	*ListFunctionDefinition MainFunction*
ListFunctionDefinition	$\longrightarrow$	ϵ
		| *FunctionDefinition*
		| *ListFunctionDefinition FunctionDefinition*
MainFunction	$\longrightarrow$	*MAIN '(' ')' FunctionBody*
FunctionDefinition	$\longrightarrow$	*TypeSpecifier Declarator FunctionBody*
TypeSpecifier	$\longrightarrow$	`TYPE`
FunctionBody	$\longrightarrow$	*CompoundStatement*
Stat	$\longrightarrow$	`DISPLAY`
		| `RESET`
		| `DEBUGGER`
		| `RUN`
		| `Error`
Statement	$\longrightarrow$	*StatementPair*
		| *StatementUnpaired*

StatementPair ⟶ *If '(' Expr ')' StatementPair* **ELSE** *StatementUnpaired*
| *CompoundStatement*
| *ExpressionStatement*
| *IterationStatement*
| **RETURN** *Expr*
| `DISPLAY`
| `EXIT`
| `PAUSE`

StatementUnpaired ⟶ *If '(' Expr ')' Statement*
| *If '(' Expr ')' StatementPair* **ELSE** *StatementUnpaired*

CompoundStatement ⟶ *'{' '}'*
| *'{' StatementList '}'*
| *'{' DeclarationList '}'*
| *'{' DeclarationList StatementList '}'*

StatementList ⟶ *Statement*
| *StatementList Statement*

DeclarationList ⟶ *Declaration*
| *DeclarationList Declaration*

Declaration ⟶ *TypeSpecifier InitDeclaratorList ';'*

Asgn ⟶ `VAR` *ASIG Expr*
| `VAR` *'[' Expr ']' '=' Expr*

InitDeclaratorList ⟶ *InitDeclarator*
| *InitDeclaratorList ',' InitDeclarator*

InitDeclarator ⟶ *Declarator*
| *Declarator '=' Initializer*

Initializer ⟶ *Expr*
| *Array*
| *'{' ArgExprMetalist '}'*

ArgExprMetalist ⟶ ϵ
| *Array*
| *ArgExprMetalist ',' Array*

Array ⟶ *'{' ArgumentExprList '}'*

ArgumentExprList ⟶ ϵ
| *Expr*
| *ArgumentExprList ',' Expr*

Declarator ⟶ *Identifier*
| *Identifier '[' `TINT` ']'*
| *Declarator '(' ')'*
| *Declarator '(' ParameterTypeList ')'*

ParameterTypeList ⟶ *ParameterList*

ParameterList ⟶ ϵ
| *ParameterDeclaration*
| *ParameterList ',' ParameterDeclaration*

ParameterDeclaration ⟶ *TypeSpecifier Declarator*
Identifier ⟶ `VAR`
| `VARFUNC`
ExpressionStatement ⟶ *';'*
| *Expr ';'*
IterationStatement ⟶ *While '(' Expr ')' Statement*
| *For '(' ';' Expr ';' ')' Statement*
| *For '(' ';' Expr ';' Expr ')' Statement*
| *For '(' Expr ';' Expr ';' ')' Statement*
| *For '(' Expr ';' Expr ';' Expr ')' Statement*
While ⟶ **WHILE**
If ⟶ **IF**
For ⟶ **FOR**
Expr ⟶ *'(' Expr ')'*
| *Cnumber*
| `TBOOL`
| `VAR`
| `VAR` *'[' Expr ']'*
| *qgate*
| *Expr* `NKET`
| *Expr* `TRANS`
| *Expr* `CONJ`
| *Asgn*
| *Identifier '(' ')'*
| *Identifier '(' ArgumentExprList ')'*
| `BLTIN` *'(' Expr ')'*
| *Expr '+' Expr*
| *Expr '-' Expr*
| *Expr '*' Expr*
| *Expr '/' Expr*
| *Expr* `AND` *Expr*
| *Expr* `OR` *Expr*
| *Expr* `EQ` *Expr*
| *Expr* `NE` *Expr*
| *Expr* `LE` *Expr*
| *Expr* `GE` *Expr*
| *Expr '<' Expr*
| *Expr '>' Expr*
| *Expr '&' Expr //tensor product*
| **NORM** *'(' Expr ')'*
| **MEASURE** *'(' Expr ')'*
| *' ' Expr*
| *'-' Expr*
| `NKET`

Cnumber ⟶ `TINT`
| `CMP`
| *Number*
| *Number* `CMP`
| `CONSTS`
| `CONSTS CMP`
| `TINT CMP`

Number ⟶ `REAL`

Qgate ⟶ `HAD`
| `IDEN` *'(' Expr ')'*
| `CNOT`
| `TOFF`
| `TOFF` *'(' Expr ')'*

Apéndice C

Tablas de conversiones de tipos

	Int	Bool	Complex	Qreg	Qbit	Qgate
Int	✔	✗	✔	✗	✗	✗
Bool	✗	✗	✗	✗	✗	✗
Complex	✔	✗	✔	✗	✗	✗
Qreg	✗	✗	✗	✔	✔	✗
Qbit	✗	✗	✗	✔	✔	✗
Qgate	✗	✗	✗	✗	✗	✔

Tabla C.1: Conversiones entre tipos para la suma.

	Int	Bool	Complex	Qreg	Qbit	Qgate
Int	✔	✗	✔	✗	✗	✗
Bool	✗	✗	✗	✗	✗	✗
Complex	✔	✗	✔	✗	✗	✗
Qreg	✗	✗	✗	✔	✔	✗
Qbit	✗	✗	✗	✔	✔	✗
Qgate	✗	✗	✗	✗	✗	✔

Tabla C.2: Conversiones entre tipos para la resta.

	Int	Bool	Complex	Qreg	Qbit	Qgate
Int	✔	✗	✔	✔	✔	✔
Bool	✗	✗	✗	✗	✗	✗
Complex	✔	✗	✔	✔	✔	✔
Qreg	✔	✗	✔	✔	✔	✔
Qbit	✔	✗	✔	✔	✔	✔
Qgate	✔	✗	✔	✔	✔	✔

Tabla C.3: Conversiones entre tipos para la multiplicación.

	Int	Bool	Complex	Qreg	Qbit	Qgate
Int	✔	✗	✔	✗	✗	✗
Bool	✗	✗	✗	✗	✗	✗
Complex	✔	✗	✔	✗	✗	✗
Qreg	✔	✗	✔	✗	✗	✗
Qbit	✔	✗	✔	✗	✗	✗
Qgate	✔	✗	✔	✗	✗	✗

Tabla C.4: Conversiones entre tipos para la división.

	Int	Bool	Complex	Qreg	Qbit	Qgate
Int	✗	✗	✗	✗	✗	✗
Bool	✗	✗	✗	✗	✗	✗
Complex	✗	✗	✗	✗	✗	✗
Qreg	✗	✗	✗	✔	✔	✗
Qbit	✗	✗	✗	✔	✔	✗
Qgate	✗	✗	✗	✗	✗	✔

Tabla C.5: Conversiones entre tipos para el producto tensorial.

Apéndice D

Prueba del algoritmo de Deutsch-Jozsa

Para el caso $n = 2$ y la función f_0 de la tabla 5.1, la simulación del algoritmo de Deutsch-Jozsa en GAMA queda como:

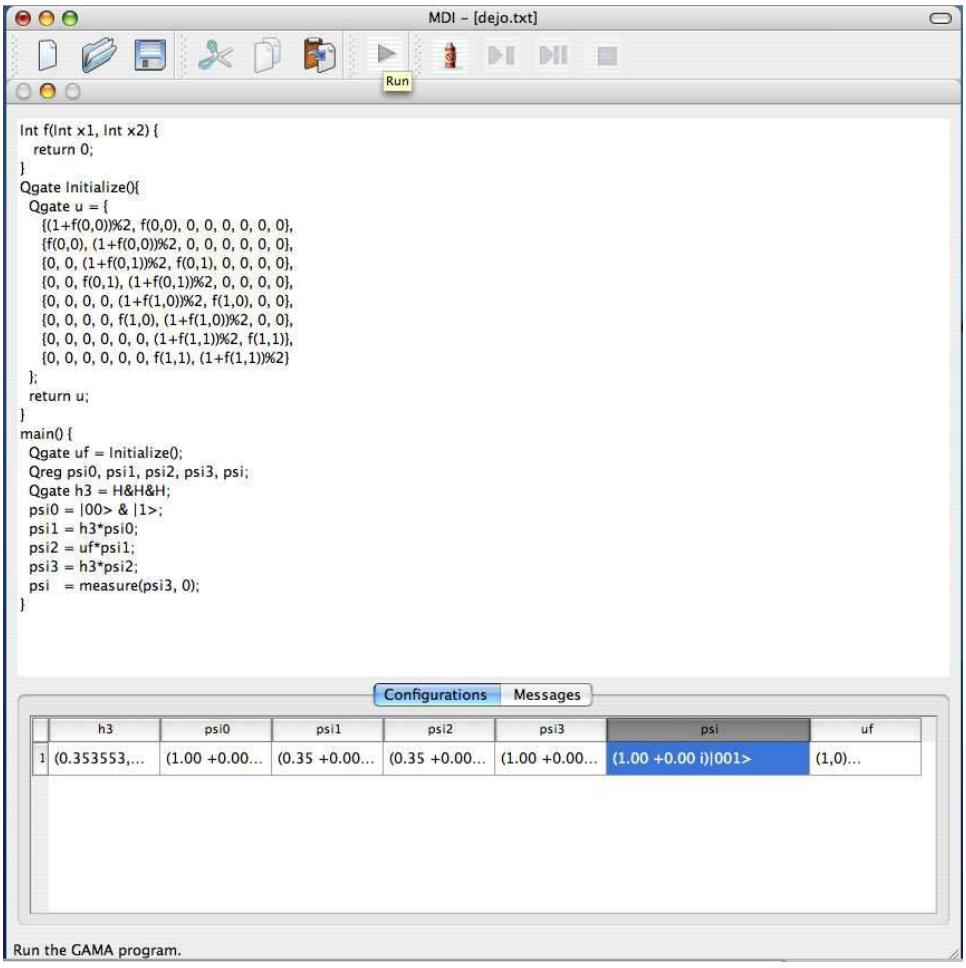

Figura D.1: Simulación del algoritmo de Deutsch-Jozsa en GAMA.

Bibliografía

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Softw.*, 8(3):21–26, 1991.

[2] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume I: Parsing of *Series in Automatic Computation*. Prentice Hall, Englewood Cliffs, New Jersey, 1972.

[3] Ravi Sethi y Jeffrey D. Ullman Alfred V. Aho. *Compiladores: Principios, técnicas y herramientas*. Addison Wesley Longman, México, 1998.

[4] Alan Aspuru-Guzik, Anthony D. Dutoi, Peter J. Love, and Martin Head-Gordon. Simulated quantum computation of molecular energies. *Science*, 309:1704, 2005.

[5] Gregory David Baker. *"Qgol". A system for simulation quantum computations: Theory, Implementation and Insights*. PhD thesis, Department of Computing, School of Mathematics, Physics, Computing and Electronics, Macquarie University, October 1996.

[6] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM J. Comput.*, 26(5):1411–1473, 1997.

[7] Stefano Bettelli. *Toward an architecture for quantum programming*. PhD thesis, Universita di Trento, February 2002.

[8] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[9] Adam Brazier and Martin B. Plenio. Broken promises and quantum algorithms, 2003.

[10] David Collins, K. W. Kim, and W. C. Holton. Deutsch-jozsa algorithm as a test of quantum computation. *Phys. Rev. A*, 58(3):R1633–R1636, Sep 1998.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press/McGraw-Hill, 2003.

[12] D Deutsch and R Jozsa. Rapid solution of problems by quantum computation. *Proc Roy Soc Lond A*, 439:553–558, October 1992.

[13] Artur Ekert Dirk Bouwmeester and Anton Zeilinger. *The Physics of Quantum Information.* Springer, Berlin, Germany.

[14] Ping Dong, Zheng-Yuan Xue, Ming Yang, and Zhuo-Liang Cao. Scheme for implementing the deutsch-jozsa algorithm via atomic ensembles, 2005.

[15] Bruce Eckel and Chuck Allison. *Thinking in C++: Practical Programming*, volume Two. Prentice Hall, December 2003.

[16] Richard P. Feynman. *Conferencias sobre computación. Prólogos de Alberto Galindo.* Crítica, 2003.

[17] Simson Garfinkel and Michael K. Mahoney. *Building Cocoa Applications: A Step-by-Step Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[18] Simon J. Gay. Quantum programming languages: survey and bibliography. *Mathematical. Structures in Comp. Sci.*, 16(4):581–600, 2006.

[19] Ian Glendinning. Quantum programming languages and tools, 2005. Online catalogue.

[20] Jonathan Grattage and Thorsten Altenkirch. A compiler for a functional quantum programming language. Under revision, January 2005.

[21] E. S. Guerra. A cavity qed implementation of deutsch-jozsa algorithm, 2004.

[22] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice-Hall, Englewood Cliffs, NJ, 1984.

[23] Brian W. Kernighan and Dennis M. Ritchie. *El lenguaje de programación C, segunda edición*. Prentice Hall, 1991.

[24] Naveen Kumar, Bruce R. Childers, and Mary Lou Soffa. Tdb: a source-level debugger for dynamically translated programs. In *AADEBUG'05: Proceedings of the sixth international symposium on Automated analysis-driven debugging*, pages 123–132, New York, NY, USA, 2005. ACM Press.

[25] Samuel J. Lomonaco. A rosetta stone for quantum mechanics with an introduction to quantum computation, 2000.

[26] A. Meneses Viveros M. Paredes López. La transformación u_f del algoritmo de deutsch-jozsa, 2007.

[27] G. Morales Luna M. Paredes López. Ambiente para la simulación de algoritmos cuánticos, 2006.

[28] Francisco M. Márquez. *UNIX, Programación Avanzada*. Alpha Omega., 2004.

[29] Tony Mason and Doug Brown. *Lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1990.

[30] Tim McDonald. Ibm claims historic test-tube quantum computer, 2001.

[31] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood, CliffS, N. J., 1967.

[32] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information.* CUP, 2000.

[33] Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho, and Christopher E. Wee. Sequential debugging at a high level of abstraction. *IEEE Softw.*, 8(3):27–36, 1991.

[34] Bernhard Ömer. Classical concepts in quantum programming. arXiv:quant-ph/0211100, 2002.

[35] Bernhard Ömer. *Structured Quantum Programming*. PhD thesis, Technical University of Vienna, Department of Theoretical Physics, May 2003.

[36] Steven P. Reiss. Software tools and environments. *ACM Comput. Surv.*, 28(1):281–284, 1996.

[37] Eleanor Rieffel. An introduction to quantum computing for non-physicists. *ACM Comput. Surv.*, 32(3):300–335, 2000.

[38] Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[39] Roland Rüdiger. Quantum programming languages - a survey -, 2003. Seminar talks given at Institut fur Mathematische Physik, TU Braunschweig.

[40] L. I. Schiff. *Quantum Mechanics.* World Scientific, New York, USA.

[41] Hans Schwerdtfeger. *Geometry of Complex Numbers : Circle Geometry, Moebius Transformation, Non-Euclidean Geometry*. Dover, New York, 1979. [first published 1962].

[42] Robert W. Sebesta. *Concepts of programming languages, 6th ed.* Addison Wesley, 2004.

[43] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004.

[44] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[45] Bernd Thaller. Advanced visual quantum mechanics, 2004.

[46] Xiang-Bin Wang, J. Q. You, and Franco Nori. Measurement-based quantum computation with superconducting charge qubits, 2006.

[47] Paolo Zuliani. *Quantum Programming*. PhD thesis, University of Oxford, 2001.