Compilación del lambda cálculo con matrices densidad en la máquina cuántica IBM-Q

Tesina de Grado para la obtención del grado de Licenciado en Ciencias de la Computación

Autor: Martín Villagra
V-2719/7

Director: Alejandro Díaz-Caro
Co-director: Pablo E. Martínez López

20 de diciembre de 2023

El cálculo λ_ρ introducido por Díaz-Caro en 2017 es un lenguaje basado en el lambda cálculo con extensiones para la computación cuántica donde se utiliza un modelo de control clásico y datos cuánticos. Los estados cuánticos se describen mediante matrices de densidad, que permite operar con estados cuánticos mixtos.

En esta tesina se provee un algoritmo para tipar λ_ρ. Utilizando este algoritmo probamos que el tipado es NP-completo bajo condiciones de minimización de los tipos. Seguidamente analizamos la relación entre λ_ρ y las aplicaciones actuales de la computación cuántica al definir una traducción de una versión modificada de λ_ρ a Python y haciendo uso de la biblioteca de Qiskit. Además de probar su correctitud, implementamos estos algoritmos en el lenguaje funcional Haskell.

Palabras claves: Lambda cálculo, computación cuántica, matrices de densidad, control clásico, Qiskit, Python, traducción, inferencia de tipos, Haskell, compilador, IBM, cubrimiento de vértices mínimo, NP-completo, NP-hard, NP-difícil, Simplex, Ramificación y Poda.

The λ_ρ calculus introduced by Díaz-Caro in 2017 is a language based on the lambda calculus with extensions for quantum computing where a classical control with quantum data model is used. Quantum states are described by density matrices, which allow mixed quantum states to be used.

In this thesis we first provide a type inference algorithm to type λ_ρ. Using this algorithm, we then prove that typing this language is NP-complete under minimization of the size of the types. Then we analyse the relationship between λ_ρ and current applications of quantum computing by defining a translation between a modified version of λ_ρ and Python, while making use of the Qiskit library. In addition to proving its correctness, we implement these algorithms using the Haskell functional language.

Keywords: Lambda calculus, quantum computing, density matrices, classic control, Qiskit, Python, translation, type inference, Haskell, compiler, IBM, minimum vertex cover, NP-complete, NP-hard, NP-hardness, Simplex, Branch and Bound.

Agradecimientos

Quiero expresar mi más profundo agradecimiento a mi familia, amigos y profesores por su apoyo y orientación inquebrantables a lo largo de mi trayectoria académica.

A mi familia y mascotas, les agradezco por su apoyo incondicional y sus constantes preguntas. A mis amigos, les doy las gracias por estar siempre allí para ofrecer un oído atento. Y a mis profesores, les estoy en deuda por su experiencia, paciencia y mentoría, que me han inspirado a estar en una búsqueda de aprendizaje de por vida.

También quisiera reconocer a las innumerables otras personas que han contribuido a mi crecimiento académico y personal en la UNR. Incluyendo todos mis compañeros de la universidad y en especial los que formaron parte del mundo de la ICPC. Los viajes, las simulaciones, y su amistad han tenido un impacto profundo en mi vida, no solo me han motivado mucho hasta hoy en día, sino que también fueron un punto clave para mejorar como persona. Estoy agradecido por la oportunidad de haber aprendido de ustedes y con ustedes.

Finalmente, quiero dedicar esta tesis a mis seres queridos, cuyo apoyo y aliento inquebrantables han sido la fuerza motriz detrás de mi éxito. Gracias por creer en mí y por ser mi fuente constante de inspiración y motivación.

Índice general

Introducción

La computación cuántica, un campo de estudio que ha surgido recientemente en la intersección de la física cuántica y la informática, ha capturado la atención de la comunidad científica y tecnológica en las últimas décadas. La promesa de revolucionar la forma en que abordamos problemas computacionales complejos es una de las razones que han impulsado un crecimiento vertiginoso en esta área.

A medida que la computación cuántica ha ido madurando, hemos sido testigos de un hito significativo en su evolución: la disponibilidad de hardware cuántico que ahora se encuentra al alcance de investigadores, científicos y programadores en todo el mundo. Este avance ha sido impulsado por la creciente inversión en investigación y desarrollo por parte de empresas líderes en tecnología, así como por la creación de plataformas de acceso público que permiten ejecutar experimentos cuánticos y desarrollar algoritmos en la nube. Hoy en día, no es necesario contar con un laboratorio de física cuántica de última generación para experimentar con la computación cuántica, ya que se ha vuelto accesible a través de servicios en línea que proporcionan acceso a hardware cuántico remoto. Este acceso democrático a la computación cuántica marca un emocionante capítulo en el avance de esta tecnología y amplía las posibilidades de investigación y desarrollo en este campo fascinante.

Lambda cálculo cuántico

Aplicar el cálculo lambda, una abstracción fundamental de la teoría de la computación, en este contexto se presenta como un campo de estudio relevante y de gran potencial. En las últimas décadas, ha habido una abundancia de investigaciones en torno a extensiones cuánticas del lambda cálculo, i.e. [Ton04; SV05; PSV14; Zor16; AD05; ADV17; DD17]. En estos trabajos, la representación seleccionada para el estado cuántico fueron los vectores en el espacio de Hilbert. Sin embargo, existe una formulación alternativa para la mecánica cuántica que hace uso de las matrices de densidad. Estas matrices de densidad proporcionan una forma de describir un sistema cuántico de estado mixto; es decir, un conjunto probabilístico de varios estados posibles. Todos los postulados de la mecánica cuántica pueden describirse a través

de dicho formalismo y, por lo tanto, toda la computación cuántica también puede describirse a través de él.

El formalismo utilizando matrices de densidad se ha empleado ampliamente en lenguajes de programación cuántica, p. ej. [Sel04; DP06; FDY11; YYW17; FYY13; YYW17]. Además, el libro "Fundamentos de la programación cuántica" [Yin16] está escrito íntegramente en el lenguaje de las matrices de densidad. Sin embargo, hasta donde sabemos, el único cálculo lambda con matrices de densidad es el introducido en [Día17].

Además de la distinción de lenguajes en función de cómo manejan los estados cuánticos (vectores en un espacio de Hilbert versus matrices de densidad), también podemos diferenciarlos por cómo consideran el control, que puede ser cuántico o clásico. La arquitectura de un modelo de datos cuánticos/control clásico se basa en una computadora cuántica ejecutándose dentro de un dispositivo especializado conectado a una computadora clásica, y que la computadora clásica le indica a la computadora cuántica qué operaciones realizar y lee el resultado después de las mediciones. Muchos estudios se han desarrollado siguiendo este paradigma, p. ej. [AG05; SV05; Gre+13; PSV14; Zor16]. La idea de tener un lenguaje cuántico donde el control sea clásico y los datos sean cuánticos se describió en el modelo QRAM de Knill [Kni96] en 1996. Esto inspiró el trabajo de Selinger sobre lenguajes de programación cuánticos [Sel04], que más tarde introdujo la creación de un cálculo lambda cuántico en este paradigma [SV05].

En [Día17] se propone una extensión cuántica del cálculo lambda, llamada λ_ρ (pronunciado: *lambda rho*), en el paradigma de datos cuánticos/control clásico, donde los datos cuánticos están representados por matrices de densidad. Si bien este lenguaje puramente teórico es sumamente simple, hasta ahora, nunca ha sido ejecutado en una máquina cuántica real.

Como antecedente se dispone del trabajo presentado por Borgna [Bor19], que provee una traducción de λ_ρ al lenguaje de Selinger [SV05]. En este se utiliza una técnica para convertir entre los dos modelos de estados de estos lenguajes conocida como purificación.

Python y Qiskit

Python [Fou21], un lenguaje de programación de alto nivel, se ha convertido en una de las herramientas más populares y versátiles en el mundo de la informática y la ciencia de datos. Su sintaxis clara y legible, su amplia comunidad de desarrolladores y su capacidad para integrarse con diversas tecnologías lo convierten en una elección ideal para una variedad de aplicaciones. En este contexto, Python no solo se utiliza para el desarrollo de software convencional, sino que también se ha convertido en un

recurso esencial en la vanguardia de la computación cuántica gracias a bibliotecas como Qiskit.

Qiskit, desarrollada por IBM Quantum [IBM21], es una poderosa y versátil biblioteca de código abierto que permite a los programadores y científicos de datos acceder al mundo de la computación cuántica por medio de Python como lenguaje principal. Esta biblioteca se ha convertido en un componente esencial para aquellos interesados en experimentar y desarrollar algoritmos cuánticos en hardware real o simuladores cuánticos de alta fidelidad. Con Qiskit, los usuarios pueden aprovechar las capacidades emergentes de la computación cuántica, experimentar con algoritmos cuánticos y explorar el fascinante mundo de los qubits y la superposición.

Enfoque y objetivos

En el presente trabajo de investigación, presentamos una contribución al campo de la computación cuántica y el cálculo lambda. Queremos establecer un vínculo entre una idea que inicialmente fue puramente teórica, el lenguaje λ_ρ, con las aplicaciones prácticas de la computación cuántica que están disponibles desde estos últimos años.

Nuestra propuesta se centra en la traducción de λ_ρ a un lenguaje de programación ampliamente empleado en la informática clásica, Python. Sin embargo, el objetivo no es simplemente realizar una traducción, sino habilitar la ejecución de código de λ_ρ en hardware cuántico real.

Para probar la validez de dicha traducción debemos formalizar su definición y probar propiedades sobre ella. A su vez, cómo la traducción funciona exclusivamente para programas válidos de λ_ρ, se define un algoritmo de tipado de λ_ρ.

Aportes de este trabajo

Las contribuciones de este trabajo son las siguientes:

Capítulo 2 Tipado de λ_ρ

- Proveemos un algoritmo de inferencia de tipos en dos etapas para λ_ρ.
- Probamos la completitud y correctitud de la primera etapa.
- Usando este algoritmo probamos que el tipado es NP-completo cuando se minimiza la suma del tamaño de los tipos.

Capítulo 3 Qiskit con Python

- Presentamos una versión simplificada de Python con Qiskit para poder probar la correctitud de la traducción.

Capítulo 4 Traducción de λ_ρ^* a Qiskit

- Demostramos propiedades sobre el método de la purificación y presentamos una técnica de purificación novedosa que preserva la composición.
- A base de las limitaciones impuestas por Qiskit proponemos una versión modificada de λ_ρ, λ_ρ^*.
- Definimos la traducción de λ_ρ^* a Python, con prueba de correctitud y retracción a izquierda.

Capítulo 5 Implementación

- Implementamos los algoritmos presentados (tipado y traducción) en Haskell.

Capítulo 1

Preliminares

Este capítulo contiene dos partes muy importantes para entender la base teórica del trabajo. En la primera parte se dan definiciones generales sobre el álgebra lineal, las cuales son necesarias para la segunda parte en la que se introduce la mecánica cuántica en términos de matrices de densidad. Al final se detallan algunos conceptos adicionales que resultan útiles más adelante.

1.1. Álgebra Lineal

El álgebra lineal es el estudio de espacios vectoriales y operaciones lineales sobre esos espacios. Para entender mecánica cuántica es necesario un conocimiento firme en álgebra lineal. En esta sección se definen algunos conceptos y notaciones que serán útiles más adelante, inspirados en los libros [YM08] y [NC10].

1.1.1. Vectores

Los objetos elementales del álgebra lineal son los *espacios vectoriales*. El espacio que más nos interesa es el de $\mathbb{C}^n$, el espacio de todas las n-tuplas de números complejos, $(z_1, \ldots, z_n)$. Los elementos de este espacio son *vectores*, con sus operaciones de adición y multiplicación por escalar habituales. Suelen representarse también cómo una matriz de $n \times 1$, es decir una columna de n filas, en cuyo caso se denominan vector columna. La notación estándar en mecánica cuántica de un vector en un espacio vectorial es la siguiente:

$$|\phi\rangle$$

ϕ es una etiqueta del vector y la notación $|\cdot\rangle$ es usada para indicar que el objeto es un vector columna. El objeto $|\phi\rangle$ es conocido también como *ket*.

El espacio vectorial contiene también un elemento especial $\vec{0}$, el vector cero. El mismo cumple con la propiedad $|v\rangle + \vec{0} = |v\rangle \ \forall |v\rangle$. Es importante no confundir este vector con $|0\rangle$ el cual se reserva para significar algo completamente distinto (en la Sección 1.2.1).

1.1.2. Bases e independencia lineal

Un *generador* es un conjunto de vectores $|v_1\rangle, \ldots, |v_n\rangle$ de un espacio vectorial V tales que cualquier vector $|v\rangle \in V$ puede ser escrito como una combinación lineal $|v\rangle = \sum_i a_i |v_i\rangle$ de los vectores del conjunto.

Ejemplo 1.1. Un generador de $\mathbb{C}^2$ es el conjunto:

$$|v_1\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}; |v_2\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Decimos que $|v_1\rangle$ y $|v_2\rangle$ generan el espacio vectorial $\mathbb{C}^2$. Otro ejemplo de generador para $\mathbb{C}^2$ es:

$$|v_1\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}; |v_2\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ -1 \end{bmatrix},$$

puesto que cualquier vector $|v\rangle = (a_1, a_2)$ puede ser escrito como una combinación lineal de $|v_1\rangle$ y $|v_2\rangle$:

$$|v\rangle = \frac{a_1 + a_2}{\sqrt{2}} |v_1\rangle + \frac{a_1 - a_2}{\sqrt{2}} |v_2\rangle.$$

Definición 1.2. Un conjunto de vectores no nulos $|v1\rangle, \ldots, |v_n\rangle$ es *linealmente dependiente* si existe un conjunto de números complejos $a_1, \ldots, a_n$ con $a_i \neq 0$ para al menos un valor de i, tal que $a_1 |v_1\rangle + \cdots + a_n |v_n\rangle = 0$.

Se dice que un conjunto de vectores es linealmente independiente si no es linealmente dependiente. Se puede demostrar que cualquier conjunto de vectores independientes que generan un espacio vectorial V contiene la misma cantidad de elementos. Llamamos a tal conjunto una base de V. Además, tal base siempre existe. El número de elementos de la base se define como la dimensión de V. En este trabajo estamos interesados solamente en espacios vectoriales con dimensiones finitas.

1.1.3. Operadores lineales

Un operador lineal entre los espacios vectoriales V y W se define como una función $A : V \to W$ que es lineal en su entrada, es decir:

$$A\left(\sum_i a_i \left|v_i\right\rangle\right) = \sum_i a_i A\left(\left|v_i\right\rangle\right). \tag{1.1}$$

La forma más conveniente de entender los operadores lineales es en términos de sus representaciones matriciales. De hecho, el paradigma de los operadores lineales y las matrices resultan ser equivalentes. Para ver la conexión, es útil comprender primero que una matriz A compleja de $m \times n$ con entradas A_{ij} es, de hecho, un operador lineal que envía vectores en el espacio vectorial $\mathbb{C}^n$ al espacio vectorial $\mathbb{C}^m$, utilizando la multiplicación entre A y vectores en $\mathbb{C}^n$. Con esta definición es fácil ver como A cumple con la definición de operador lineal de la Ecuación 1.1.

Habiendo visto que las matrices pueden ser vistas como operadores lineales, ¿pueden los operadores lineales ser vistos como matrices? Suponga que $A : V \to W$ es un operador lineal entre los espacios vectoriales V y W. Suponga que $\left|v_1\right\rangle, \ldots, \left|v_n\right\rangle$ es una base de V y $\left|w_1\right\rangle, \ldots, \left|w_m\right\rangle$ es una base de W. Entonces por cada j en el rango $1, \ldots, m$ sabemos que $A\left|v_j\right\rangle \in W$, por lo que existe una descomposición en la base $\{w_i\}$. Es decir, existe una serie de números complejos $A_{1j}, \ldots, A_{nj}$ tales que:

$$A\left|v_j\right\rangle = \sum_i A_{ij}\left|w_i\right\rangle.$$

La matriz cuyos valores son compuestos por los valores A_{ij} se la conoce como la *representación matricial* del operador A. Esta matriz es equivalente al operador A y ambas abstracciones son intercambiables.

1.1.4. Producto interno

El productor interno es una función que toma dos vectores $\left|v\right\rangle$ y $\left|w\right\rangle$ de un espacio vectorial y devuelve un número complejo. La notación para esta operación en mecánica cuántica es $\langle v|w\rangle$. Una función $\langle\cdot|\cdot\rangle$ de $V \times V \to \mathbb{C}$ es un producto interno si satisface que:

1. Es lineal en su segundo argumento:

$$\left\langle v\right| \sum_i \lambda_i \left|w_i\right\rangle\rangle = \sum_i \lambda_i \left\langle v|w_i\right\rangle. \tag{1.2}$$

2. $\langle v|w\rangle = \langle w|v\rangle^*$, donde $*$ denota el conjugado del número complejo.

3. $\langle v|w\rangle \geq 0$, con la igualdad dándose si y solo si $|v\rangle = 0$.

El productor interno utilizado para $\mathbb{C}^n$ se define como:

$$\langle \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} | \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix} \rangle = \sum_i y_i^* z_i = \begin{bmatrix} y_1^* & \dots & y_n^* \end{bmatrix} \begin{bmatrix} z_1 \\ \vdots \\ z_n \end{bmatrix}. \tag{1.3}$$

Donde el primer $*$ denota el transpuesto conjugado de la representación matricial del vector.

Llamamos como *espacio prehilbertiano* a un espacio vectorial equipado con un producto interno.

Notación Bra

La notación $\langle v|$ es usada para denotar el *vector dual* de $|v\rangle$. El dual de un vector es un operador lineal que se define sobre un espacio prehilbertiano de la siguiente manera: $\langle v| \left(|w\rangle\right) = \langle v|w\rangle$. En la Sección 1.1.5 se muestra que para el espacio $\mathbb{C}^n$, el vector dual es simplemente un vector fila.

Vectores normalizados

Los vectores $|w\rangle$ y $|v\rangle$ son ortogonales si su producto interno es cero. Por ejemplo, $|w\rangle = [\,1\;0\,]^*$ y $|v\rangle = [\,0\;1\,]^*$ son ortogonales con respecto al producto interno definido en 1.3. Definimos la norma del vector $|v\rangle$ como

$$\| \, |v\rangle \, \| = \langle v|v\rangle \, . \tag{1.4}$$

Un vector unitario es un vector $|v\rangle$ tal que $\| \, |v\rangle \, \| = 1$. También decimos que $|v\rangle$ está normalizado si es unitario. Un conjunto de vectores $|i\rangle$ con índice i es ortonormal si cada vector es unitario y son ortogonales entre sí, es decir, $\langle i|j\rangle = \delta_{ij}$, donde i y j pertenecen al conjunto y $\delta_{ij} = \begin{cases} 1 & \text{si } i \neq j \\ 0 & \text{si } i = j \end{cases}$.‘

Definición 1.3 (Adjunto)**.** El adjunto de un operador A se nota por $A^\dagger$ y se define como el operador transpuesto y conjugado de A. Es decir, si $\alpha_{ij} = \langle u_i|A|u_j\rangle$ son las componentes de A, las componentes de $A^\dagger$ son $\alpha_{ij}^* = \langle u_j|A|u_i\rangle^* = \langle u_i|A^\dagger|u_j\rangle$.

Definición 1.4 (Operador Hermitiano)**.** Un operador A cuyo adjunto es A se lo llama operador *Hermitiano*.

Definición 1.5 (Operador normal)**.** Un operador A es normal si $AA^\dagger = A^\dagger A$. Claramente, un operador Hermitiano es también normal.

1.1.5. Espacio de Hilbert

La mecánica cuántica gira en torno a los *espacios de Hilbert* porque son utilizados para describir el estado de los sistemas cuánticos. En el espacio dimensional finito y complejo con el que se trabaja en computación e información cuántica, un espacio prehilbertiano es automáticamente un espacio de Hilbert. Se prefiere el término espacio de Hilbert, pero basta con saber que si se trata de espacios prehilbertianos con infinitas dimensiones, el mismo debe satisfacer requerimientos adicionales para ser considerado espacio de Hilbert.

Con estas convenciones, al producto interno de un espacio de Hilbert se le puede asignar una conveniente representación matricial. Siendo $|w\rangle = \sum_i w_i |i\rangle$ y $|v\rangle = \sum_i v_i |i\rangle$ representaciones de dos vectores bajo una base ortonormal $|i\rangle$. Entonces, como $\langle i|j\rangle = \delta_{ij}$,

$$\langle v|w\rangle = \sum_{ij} v_i^* w_j \delta_{ij} = \sum_i v_i^* w_i. \tag{1.5}$$

De esta manera el producto interno de dos vectores es igual al producto vectorial interno entre las dos representaciones matriciales de esos vectores, asumiendo que ambas representaciones son sobre la misma base ortonormal. También vemos como el vector dual $\langle v|$ se puede representar como el vector fila donde sus componentes son el conjugado complejo de las componentes del vector columna de la representación de $|v\rangle$.

Una manera útil de representar operadores lineales es a través del producto interno, con un método conocido como representación del producto externo. Sea $|v\rangle$ un vector en un espacio prehilbertiano V y $|w\rangle$ un vector en un espacio prehilbertiano W. Definimos $|w\rangle\langle v|$ como el operador lineal de V a W cuya acción es definida por:

$$(|w\rangle\langle v|)(|v'\rangle) = |w\rangle\langle v|v'\rangle = \langle v|v'\rangle |w\rangle . \tag{1.6}$$

Esta ecuación encaja maravillosamente en nuestras convenciones de notación, según la que la expresión $|w\rangle\langle v|v'\rangle$ podría tener dos significados: el resultado de operar $|w\rangle\langle v|$ sobre $|v'\rangle$, o el resultado de multiplicar $|w\rangle$ por el número complejo $\langle v|v'\rangle$.

¡Nuestras definiciones fueron elegidas de manera que estás dos interpretaciones son equivalentes!

1.1.6. Autovectores y autovalores

Un *autovector* de un operador lineal A (dentro de un espacio vectorial) es un vector $|v\rangle$ no nulo tal que $A|v\rangle = \lambda|v\rangle$, donde λ es un número complejo conocido como el autovalor de A asociado a $|v\rangle$. Se asume que el lector está familiarizado con las propiedades elementales de los autovalores y autovectores, en particular, cómo obtenerlos vía la ecuación característica. La *ecuación característica* se define como $c(\lambda) = \det\|A - \lambda I\|$, donde det es la función determinante de las matrices. Se puede demostrar que la función característica depende solo del operador A, y no de la representación matricial que tenga A. Las soluciones de la ecuación $c(\lambda) = 0$ son los autovalores del operador A. Por el teorema fundamental del álgebra, todo polinomio tiene por lo menos una raíz compleja, por lo que todo operador A tiene por lo menos un autovalor, y su autovector correspondiente. El autoespacio correspondiente al autovalor λ es el conjunto de vectores que tienen a λ como su autovalor.

Definición 1.6 (Representación diagonal)**.** Una *representación diagonal* del operador A en un espacio vectorial V es una representación $A = \sum_i \lambda_i |i\rangle\langle i|$, donde los vectores $|i\rangle$ forman una base ortonormal de autovalores para A, siendo λ_i sus autovalores asociados. Un operador es *diagonalizable* si tiene una representación diagonal.

Definición 1.7 (Operador positivo)**.** Un operador A se dice *positivo* si para todo vector $|\psi\rangle$, $\langle\psi|A|\psi\rangle \geq 0$. Si $\langle\psi|A|\psi\rangle > 0$ para todo $|\psi\rangle \neq 0$, decimos que A es *definido positivo*.

Teorema 1.8 (Descomposición espectral)**.** Todo operador normal M en un espacio vectorial V es diagonal con respecto a una base ortonormal de V. A su vez, todo operador diagonalizable es normal.

Demostración. Se encuentra demostrado por inducción en la dimensión de V en la página 72 del libro [NC10]. □

Teorema 1.9 (Descomposición del operador positivo)**.** Se puede demostrar que todo operador positivo es automáticamente Hermitiano y, por lo tanto, posee por la descomposición espectral una representación diagonal $\sum_i \lambda_i |i\rangle\langle i|$. Además, sus autovalores son no negativos ($\lambda_i \in \mathbb{R}_0^+$). □

Definición 1.10 (Operador unitario)**.** Un operador A es unitario si $AA^\dagger = A^\dagger A = I$. Claramente, un operador unitario es también normal.

1.1.7. Producto tensorial

El producto tensorial es una forma de combinar espacios vectoriales para formar espacios vectoriales más grandes. Esta construcción es fundamental para comprender la mecánica cuántica de sistemas compuestos.

Sean V y W dos espacios vectoriales con bases canónicas $B = \{b_i | i \in I\}$ y $C = \{c_j | j \in J\}$ respectivamente. El producto tensorial $V \otimes W$ de V y W es el espacio vectorial de base canónica $\{b_i \otimes c_j | i \in I, j \in J\}$, donde $b_i \otimes c_j$ es el par ordenado formado por el vector b_i y el vector c_j. La operación $\otimes$ se extiende a vectores de V y W bilinealmente:

$$(\sum_i \alpha_i b_i) \otimes (\sum_j \beta_j c_j) = \sum_{ij} \alpha_i \beta_j (b_i \otimes c_j). \tag{1.7}$$

Observación. Notar que $V \times W \neq V \otimes W$. Por ejemplo, si $V = W = \mathbb{C}^2$, con base canónica $\{i, j\}$, entonces $V \times W$ contiene a $i \otimes i$ y a $j \otimes j$, pero no a $i \otimes i + j \otimes j$, que no es producto de dos vectores de $\mathbb{C}^2$.

Propiedades (del producto tensorial)**.** Sean A, B, C, D operadores y $|a\rangle, |b\rangle, |c\rangle, |d\rangle$ vectores de un espacio de Hilbert.

1. $(A \otimes B)(C \otimes D) = AC \otimes BD$, si AC y BD son multiplicaciones válidas.
2. $(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger$
3. $\langle |a\rangle \otimes |b\rangle \,|\, |c\rangle \otimes |d\rangle \rangle = \langle a|c\rangle \langle b|d\rangle$
4. $(A \otimes B)(|a\rangle \otimes |b\rangle) = A|a\rangle \otimes B|b\rangle$

1.2. Computación cuántica

Si bien la computación cuántica es un campo muy interesante y en rápida evolución, no entraremos en gran detalle respecto a su historia en esta tesina. Existen numerosos libros que ofrecen una gran introducción al tema, tales como los libros [NC10] o [YM08]. En su lugar resulta útil establecer una pequeña base teórica inspirada en dichas fuentes con la cual es posible entender este trabajo.

1.2.1. Postulados de la mecánica cuántica

La mecánica cuántica es un marco matemático para el desarrollo de teorías físicas. Por sí sola, la mecánica cuántica no revela qué leyes debe obedecer un sistema físico,

pero sí proporciona un marco matemático y conceptual para el desarrollo de dichas leyes. En las próximas secciones, se ofrece una descripción completa de los postulados básicos de la mecánica cuántica y como se aplican a este trabajo. Estos postulados establecen una conexión entre el mundo físico y el formalismo matemático de la mecánica cuántica.

Los postulados de la mecánica cuántica se derivaron después de un largo proceso de prueba y error, que implicó una cantidad considerable de conjeturas y errores por parte de los creadores de la teoría. No se sorprenda si la motivación detrás de los postulados no siempre está clara; incluso para los expertos, los postulados básicos de la mecánica cuántica parecen sorprendentes. Lo que se espera es poder entender su definición desde el punto de vista matemático.

Espacio de estados

El primer postulado de la mecánica cuántica establece el escenario en el que se desarrolla la mecánica cuántica. El escenario es nuestro amigo familiar de álgebra lineal, el espacio de Hilbert.

> **Postulado 1**: A cualquier sistema físico aislado le corresponde un espacio de Hilbert complejo, conocido como el *espacio de estados* del sistema. El sistema se describe completamente mediante su *vector de estado*, que es un vector unitario en el espacio de estados del sistema.

El postulado no indica, para cierto sistema físico, cuál es exactamente su espacio de estados ni su vector. Descubrir eso para un sistema específico es un problema dificultoso para el cual los físicos desarrollaron teorías armoniosas e intrincadas.

Para nuestros fines, será suficiente hacer suposiciones muy simples (y razonables) sobre los estados de los sistemas que nos interesan, y mantenerlas a lo largo de todo el trabajo.

El sistema cuántico más simple, y el cual más nos importa, es el *qubit*. Un qubit tiene un espacio de estados de dos dimensiones (es decir existe en un espacio vectorial de dos dimensiones). Asumiendo que los vectores $|0\rangle$ y $|1\rangle$ forman una base ortonormal para ese espacio, entonces un estado vectorial arbitrario $|\psi\rangle$ de ese espacio puede ser escrito como:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle \tag{1.8}$$

donde a y b son números complejos. La condición dada por el **Postulado 1** es que $|\psi\rangle$ sea un vector unitario, que se traduce en que $|a|^2 + |b|^2 = 1$. La condición $\langle\psi|\psi\rangle = 1$ es comúnmente llamada la *condición de normalización* para estados vectoriales.

Usaremos el qubit como nuestro sistema cuántico fundamental. Para este trabajo es suficiente considerar a los qubits en términos de su representación matemática, sin

tener en cuenta su realización física. La base ortonormal $|0\rangle$ y $|1\rangle$ debe ser pensada como fija y establecida de antemano. Intuitivamente, los estados $|0\rangle$ y $|1\rangle$ son análogos a los valores 0 y 1 que un bit clásico puede tener. La diferencia está en que un qubit puede existir en una *superposición* de estos estados, donde no es posible decir que el qubit está completamente en el estado $|0\rangle$ ni en el $|1\rangle$.

Ejemplo 1.11. Un estado puede ser:

$$\frac{|0\rangle - |1\rangle}{\sqrt{2}}.$$

El mismo cumple con la definición dada en la Ecuación 1.8 con $a = \frac{1}{\sqrt{2}}, b = -\frac{1}{\sqrt{2}}$ y se encuentra normalizado.

Evolución

¿Cómo cambia con el tiempo el estado de un sistema cuántico? El siguiente postulado da una descripción de tales cambios en el sistema.

> **Postulado 2** La evolución de un sistema cuántico *cerrado* se describe mediante una transformación unitaria. Es decir, el estado $|\psi\rangle$ del sistema en el tiempo t_1 está relacionado con el estado $|\psi'\rangle$ del sistema en el tiempo t_2 mediante un operador unitario U que depende solo de los tiempos t_1 y t_2,
>
> $$|\psi'\rangle = U\,|\psi\rangle\,. \tag{1.9}$$

Así como el **Postulado 1** no nos dice cuál es el espacio de estados, este postulado tampoco revela qué transformaciones físicas corresponden a cada operador posible. Por ejemplo, si tenemos solo un qubit, resulta ser que cualquier operador unitario puede ser implementado físicamente. Algunos ejemplos de operadores son:

- La matriz $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$, también conocida como el NOT cuántico, por su similitud al NOT clásico. Notar que $X\,|0\rangle = |1\rangle$ y $X\,|1\rangle = |0\rangle$.
- La compuerta de *Hadamard*, $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.
- Las matrices de *Pauli* que a pesar de su simplicidad son extensivamente usadas en computación cuántica y en este trabajo.

$$\sigma_0 = I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \sigma_1 = \sigma_x = X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$
$$\sigma_2 = \sigma_y = Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \quad \sigma_3 = \sigma_z = Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Es posible dar una versión continua del postulado utilizando las *ecuaciones de Schrödinger*, pero este trabajo se restringe a las transformaciones discretas.

Medición

Postulado 3 Las medidas cuánticas se describen mediante una colección $\{M_m\}$ de operadores de medición. Estos son operadores que actúan sobre el espacio de estados del sistema que está siendo medido. El índice m se refiere a los resultados de medición que pueden ocurrir en el experimento. Si el estado del sistema cuántico es $|\psi\rangle$ inmediatamente antes de la medición, entonces la probabilidad de que ocurra el resultado m está dada por:

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle\,, \tag{1.10}$$

y el estado del sistema luego de la medición es

$$\frac{M_m\,|\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}}. \tag{1.11}$$

Los operadores de medición satisfacen la *ecuación de completitud*,

$$\sum_m M_m^\dagger M_m = I. \tag{1.12}$$

La ecuación de completitud expresa el hecho de que las probabilidades suman uno:

$$\begin{aligned} \sum_m p(m) &= \sum_m \langle\psi|M_m^\dagger M_m|\psi\rangle \\ &= \langle\psi| \sum_m M_m^\dagger M_m|\psi\rangle \\ &= \langle\psi|I|\psi\rangle = \langle\psi|\psi\rangle = 1. \end{aligned}$$

Un ejemplo simple pero importante es la *medición de un qubit en la base computacional*. Esta es una medición sobre un solo qubit con dos posibles resultados definidos por dos operadores de medición $M_0 = |0\rangle\langle 0|\,, M_1 = |1\rangle\langle 1|$. Dado un estado $|\phi\rangle = a\,|0\rangle + b\,|1\rangle$, la probabilidad de obtener el resultado de medición 0 es

$$p(0) = \langle\phi|M_0^\dagger M_0|\phi\rangle = \langle\phi|M_0|\phi\rangle = |a|^2$$

.

El estado luego de obtener esta medición es

$$\frac{M_0\,|\phi\rangle}{|a|} = \frac{a}{|a|}\,|0\rangle$$

Los multiplicadores que tienen módulo 1, como $\frac{a}{|a|}$, pueden ser ignorados porque representan solo un cambio de fase del sistema. Se puede probar [NC10] que un cambio de fase no altera los posibles resultados de las mediciones en el sistema. En definitiva, el estado resultante luego de la medición es $|0\rangle$.

Sistemas compuestos

> **Postulado 4** El espacio de estados de un sistema físico compuesto es el producto tensorial de los espacios de estados de los sistemas físicos componentes. Además, si los sistemas se numeran del 1 al n, y el sistema número i está dado por el estado $|\psi_i\rangle$, entonces el estado conjunto del sistema total es $|\psi_1\rangle \otimes \cdots \otimes |\psi_n\rangle$.

Es posible pensar en este postulado de manera intuitiva. Si el sistema A está en cierto estado con probabilidad a y el sistema B (independiente a A) está en otro estado con probabilidad b, entonces la probabilidad de que el sistema A y B esté en esos estados es $a * b$. Dado que un sistema cuántico existe en varios estados con distintas probabilidades a la vez, resulta lógico que la composición asigne una probabilidad a cada posible combinación de estados. El producto tensorial realiza matemáticamente este concepto.

1.2.2. Matrices de densidad

Hasta el momento hemos examinado la mecánica cuántica utilizando vectores de estados. Sin embargo, existe una formulación alternativa que implica el uso del operador densidad (o matriz densidad). Aunque es matemáticamente equivalente, esta presentación ofrece un lenguaje más práctico para razonar en algunos escenarios comunes que se presentan en la mecánica cuántica. Este formalismo es esencial para entender el lenguaje λ_ρ que se presentará más adelante.

Definición 1.12 (Traza)**.** La *traza* de una matriz es la suma de sus elementos

diagonales. Así, si $A = \sum_i \sum_j \alpha_{ij} |u_i\rangle \langle u_j|$, la traza se define por

$$\mathsf{tr}(A) = \sum_i \alpha_{ii}$$

Teorema 1.13. Sea $A = |\psi\rangle \langle\varphi|$. Entonces, $\mathsf{tr}(A) = \langle\varphi|\psi\rangle$.

Demostración. Sean $|\psi\rangle = \sum_i a_i |u_i\rangle$ y $|\varphi\rangle = \sum_j b_j |u_j\rangle$. Entonces,

$$\mathsf{tr}(A) = \mathsf{tr}(|\psi\rangle \langle\varphi|) = \mathsf{tr}(\sum_i a_i |u_i\rangle \sum_j b_j^* \langle u_j|) \tag{1.13}$$

$$= \mathsf{tr}(\sum_{ij} a_i b_j^* |u_i\rangle \langle u_j|) = \sum_i a_i b_i^* = \langle\varphi|\psi\rangle \qquad \square$$

Ejemplo 1.14. Sea $A = |1\rangle \langle -|$. Entonces, $A = \frac{1}{\sqrt{2}}(|1\rangle \langle 0| - |1\rangle \langle 1|)$ y $\mathsf{tr}(A) = \frac{1}{\sqrt{2}}$.

Por otro lado, siguiendo el teorema, $\mathsf{tr}(A) = \langle 1|-\rangle = \frac{1}{\sqrt{2}}(\langle 1|0\rangle - \langle 1|1\rangle) = \frac{1}{\sqrt{2}}$.

El siguiente corolario es muy útil para evaluar la traza de un operador.

Corolario 1.15. Sea $|\psi\rangle$ un vector normalizado y A un operador cuántico. Entonces

$$\mathsf{tr}(A |\psi\rangle \langle\psi|) = \langle\psi| A |\psi\rangle$$

Ejemplo 1.16. Siendo $X = |0\rangle \langle 1| + |1\rangle \langle 0|$, $\mathsf{tr}(X |1\rangle \langle 1|) = \langle 1| X |1\rangle = \langle 1| (|0\rangle \langle 1| + |1\rangle \langle 0|) |1\rangle = \langle 1|0\rangle \langle 1|1\rangle + \langle 1|1\rangle \langle 0|1\rangle = 0 + 0 = 0$.

Propiedades (de la traza de una matriz)**.** Sean A y B matrices de la misma dimensión, U un operador unitario y $\lambda \in \mathbb{C}$. Entonces:

1. $\mathsf{tr}(AB) = \mathsf{tr}(BA)$
2. $\mathsf{tr}(A + B) = \mathsf{tr}(A) + \mathsf{tr}(B)$
3. $\mathsf{tr}(\lambda A) = \lambda \mathsf{tr}(A)$
4. $\mathsf{tr}(UAU^\dagger) = \mathsf{tr}(A)$

1.2.3. Conjuntos de estados cuánticos

El operador densidad provee una manera conveniente de describir un sistema cuántico en el cual el estado no se conoce del todo.

Definición 1.17 (Operador o matriz densidad)**.** Supongamos que un sistema cuántico está en uno de un número de estados $|\psi_i\rangle$, donde la probabilidad de que el estado sea $|\psi_i\rangle$ viene dada por p_i.

Decimos que el conjunto $\{p_i, |\psi_i\rangle\}$ es el *conjunto de estados puros*. El *operador*

densidad o *matriz densidad* para este estado viene dado por la ecuación

$$\rho = \sum_i p_i \left|\psi_i\right\rangle \left\langle\psi_i\right|$$

Ejemplo 1.18. El operador densidad del conjunto $\{(1/3, |+\rangle); (2/3, |1\rangle)\}$ es:

$$\rho = 1/3 \left|+\right\rangle \left\langle+\right| + 2/3 \left|1\right\rangle \left\langle1\right| = 1/6 \left|0\right\rangle \left\langle0\right| + 1/6 \left|0\right\rangle \left\langle1\right| + 1/6 \left|1\right\rangle \left\langle0\right| + 5/6 \left|1\right\rangle \left\langle1\right|$$

Es decir:

$$\rho = \begin{bmatrix} 1/6 & 1/6 \\ 1/6 & 5/6 \end{bmatrix}$$

Observación. Todos los postulados de la mecánica cuántica se pueden reformular en términos del operador densidad, y haremos eso más adelante en esta sección.

Evolución

Supongamos que la evolución de un sistema cuántico cerrado se describe por el operador unitario U. Si el sistema estaba inicialmente en el estado $|\psi_i\rangle$ con probabilidad p_i, entonces, luego de la evolución el sistema estará en estado $U|\psi_i\rangle$ con probabilidad p_i. Por lo tanto, la evolución del operador densidad se describe por

$$\rho = \sum_i p_i \left|\psi_i\right\rangle \left\langle\psi_i\right| \xrightarrow{U} \sum_i p_i U \left|\psi_i\right\rangle \left\langle\psi_i\right| U^\dagger = U\rho U^\dagger$$

Ejemplo 1.19. Siguiendo con el Ejemplo 1.18, tomemos $U = H$, entonces el conjunto de estados puros original $\{(1/3, |+\rangle); (2/3, |1\rangle)\}$ evolucionará a $\{(1/3, |0\rangle); 2/3, |-\rangle)\}$ y su matriz densidad puede calcularse de dos maneras equivalentes:

1. A partir del conjunto de estados puros:

$$\begin{aligned} \rho' &= 1/3 \left|0\right\rangle \left\langle0\right| + 2/3 \left|-\right\rangle \left\langle-\right| \\ &= 4/6 \left|0\right\rangle \left\langle0\right| - 3/8 \left|0\right\rangle \left\langle1\right| - 3/8 \left|1\right\rangle \left\langle0\right| + 3/8 \left|1\right\rangle \left\langle1\right| \\ &= \begin{bmatrix} 5/8 & -3/8 \\ -3/8 & 3/8 \end{bmatrix} \end{aligned}$$

2. O utilizando la igualdad dada más arriba: $\rho' = H\rho H^\dagger = H\rho H$.

Medición

Supongamos que realizamos una medición descrita por las matrices $\{M_m\}$. Si el estado inicial era $|\psi_i\rangle$, entonces la probabilidad condicional de obtener el resultado m es

$$p(m|i) = \langle\psi_i| M_m^\dagger M_m |\psi_i\rangle \overset{\text{Cor.1.15}}{=} \mathsf{tr}(M_m^\dagger M_m |\psi_i\rangle \langle\psi_i|)$$

Usando la ley de probabilidades totales, la probabilidad de obtener el resultado m es

$$\begin{aligned} p(m) &= \sum_i p(m|i)p_i \\ &= \sum_i p_i \mathsf{tr}(M_m^\dagger M_m |\psi_i\rangle \langle\psi_i|) \\ &= \mathsf{tr}(\sum_i p_i M_m^\dagger M_m |\psi_i\rangle \langle\psi_i|) \\ &= \mathsf{tr}(M_m^\dagger M_m \sum_i p_i |\psi_i\rangle \langle\psi_i|) \\ &= \mathsf{tr}(M_m^\dagger M_m \rho) \end{aligned}$$

Si el estado inicial era $|\psi_i\rangle$, el estado luego de obtener el resultado m será

$$|\psi_i^m\rangle = \frac{M_m |\psi_i\rangle}{\sqrt{\langle\psi_i| M_m^\dagger M_m |\psi_i\rangle}}$$

Por lo tanto, luego de una medición que dé resultado m tendremos el conjunto de estados $|\psi_i^m\rangle$, con probabilidades $p(i|m)$ respectivamente. En consecuencia, el operador densidad ρ_m correspondiente es

$$\rho_m = \sum_i p(i|m) |\psi_i^m\rangle \langle\psi_i^m| = \sum_i p(i|m) \frac{M_m |\psi_i\rangle \langle\psi_i| M_m^\dagger}{\langle\psi_i| M_m^\dagger M_m |\psi_i\rangle} \tag{1.14}$$

Pero, usando teoría de probabilidad condicional,

$$p(i|m) = \frac{p(m \cap i)}{p(m)} = \frac{p(m|i)p_i}{p(m)} = p_i \frac{\mathsf{tr}(M_m^\dagger M_m |\psi_i\rangle \langle\psi_i|)}{\mathsf{tr}(M_m^\dagger M_m \rho)} = p_i \frac{\langle\psi_i| M_m^\dagger M_m |\psi_i\rangle}{\mathsf{tr}(M_m^\dagger M_m \rho)}$$

Sustituyendo en (1.14), obtenemos

$$\rho_m = \sum_i p_i \frac{M_m |\psi_i\rangle \langle\psi_i| M_m^\dagger}{\mathsf{tr}(M_m^\dagger M_m \rho)} = \frac{M_m \rho M_m^\dagger}{\mathsf{tr}(M_m^\dagger M_m \rho)}$$

Ejemplo 1.20. Volviendo al conjunto de estados del Ejemplo 1.18, tenemos

$$\rho = 1/8 |0\rangle \langle 0| + 1/8 |0\rangle \langle 1| + 1/8 |1\rangle \langle 0| + 7/8 |1\rangle \langle 1| = \begin{bmatrix} 1/8 & 1/8 \\ 1/8 & 7/8 \end{bmatrix}$$

que corresponde a la matriz densidad del conjunto de estados $\{(\frac{1}{4}, |+\rangle); (\frac{3}{4}, |1\rangle)\}$.

Utilizaremos la medición $\{P_0, P_1\}$ con $P_0 = |0\rangle\langle 0|$ y $P_1 = |1\rangle\langle 1|$.

Entonces, la probabilidad de medir 0 viene dada por

$$\begin{aligned}\mathsf{tr}(P_0^\dagger P_0 \rho) &= |0\rangle\langle 0|\,(1/8\,|0\rangle\langle 0| + 1/8\,|0\rangle\langle 1| + 1/8\,|1\rangle\langle 0| + 7/8\,|1\rangle\langle 1|) \\ &= \mathsf{tr}(1/8\,|0\rangle\langle 0| + 1/8\,|0\rangle\langle 1|) \\ &= 1/8\,\mathsf{tr}(|0\rangle\langle 0|) + 1/8\,\mathsf{tr}(|0\rangle\langle 0|) \\ &= 1/8\end{aligned}$$

Similarmente, la probabilidad de medir 1 viene dada por

$$\mathsf{tr}(|1\rangle\langle 1|\,\rho) = 1/8\,\mathsf{tr}(|1\rangle\langle 0|) + 7/8\,\mathsf{tr}(|1\rangle\langle 1|) = 7/8$$

Podemos ver que el conjunto de estados está en el estado $|1\rangle$ con probabilidad $3/4$. Si ese es efectivamente el estado inicial, la probabilidad de medir 1 sería 1. Por otro lado, en el estado $|+\rangle$, la probabilidad de medir 1 es $1/2$. De ahí que la probabilidad de medir 1 es

$$3/4 \times 1 + 1/4 \times 1/2 = 7/8$$

tal y como dedujimos con la traza.

Luego de realizar la medición, si se midió 1, el estado del sistema podrá ser descrito por el operador siguiente:

$$\rho_1 = \frac{P_1 \rho P_1^\dagger}{7/8} = \frac{7/8\,|1\rangle\langle 1|}{7/8} = |1\rangle\langle 1|$$

Efectivamente, si se midió 1 y el estado inicial era $|+\rangle$, el estado final será $|1\rangle$, pero lo mismo pasa si el estado inicial era $|1\rangle$, por lo que la matriz densidad es la matriz densidad del conjunto de estados $\{1, |1\rangle\}$.

Definición 1.21. Un sistema cuántico donde el estado $|\psi\rangle$ se conoce exactamente se dice que está en un *estado puro*. En este caso, el operador densidad es simplemente $\rho = |\psi\rangle\langle\psi|$.
Si no es un estado puro, ρ está en un *estado mixto* (o mezcla), o que es una mezcla de diferentes estados puros.

Teorema 1.22. Para todo operador densidad ρ se tiene $\mathsf{tr}(\rho^2) \leq 1$.

Más aún, la igualdad se cumple si y solo si ρ está en un estado puro □

Teorema 1.23. Un estado cuántico que está en estado ρ_i con probabilidad p_i, puede ser descrito por la matriz densidad $\sum_i p_i \rho_i$.

Demostración. Supongamos que ρ_i viene de un conjunto $\{p_{ij}, |\psi_{ij}\rangle\}$ de estados puros (con i fijo). Por lo tanto, la probabilidad de estar en el estado $|\psi_{ij}\rangle$ viene dada por $p_i p_{ij}$. Es decir que la matriz densidad es $\rho = \sum_i \sum_j p_i p_{ij} |\psi_{ij}\rangle \langle\psi_{ij}| = \sum_i p_i \rho_i$. □

1.2.4. Propiedades generales del operador densidad

Teorema 1.24 (Caracterización de operadores densidad)**.** Un operador ρ es el operador densidad de un conjunto $\{p_i, |\psi_i\rangle\}$ si y solo si satisface las siguientes condiciones:

1. $\mathsf{tr}(\rho) = 1$
2. ρ es un operador positivo

Demostración.
⇒) Sea $\rho = \sum_i p_i |\psi_i\rangle \langle\psi_i|$ un operador densidad. Entonces,

1. $\mathsf{tr}(\rho) = \sum_i p_i \mathsf{tr}(|\psi_i\rangle \langle\psi_i|) = \sum_i p_i \langle\psi_i|\psi_i\rangle = \sum_i p_i = 1.$
2. Sea $|\varphi\rangle$ un vector arbitrario en el espacio de estados. Entonces,

$$\begin{aligned}\langle\varphi| \rho |\varphi\rangle &= \langle\varphi| \left(\sum_i p_i |\psi_i\rangle \langle\psi_i| \right) |\varphi\rangle \\ &= \sum_i p_i \langle\varphi|\psi_i\rangle \langle\psi_i|\varphi\rangle \\ &= \sum_i p_i |\langle\varphi|\psi_i\rangle|^2 \geq 0\end{aligned}$$

⇐) Sea ρ cualquier operador positivo con traza igual a 1. Como ρ es positivo, usando el Teorema 1.9 tenemos $\rho = \sum_j \lambda_j |j\rangle \langle j|$, donde los vectores $|j\rangle$ son ortogonales y $\lambda_j \in \mathbb{R}_0^+$ son autovalores de ρ. Por la condición de traza 1, tenemos $\sum_j \lambda_j = 1$. Por lo tanto, un sistema en el estado $|j\rangle$ con probabilidad λ_j tendrá un operador de densidad ρ. □

El Teorema 1.24 nos permite reformular el **Postulado 1** para no depender de vectores, y podemos entonces escribir todos los postulados en términos del operador densidad.

> **Postulado 1** A cualquier sistema físico aislado le corresponde un espacio de Hilbert complejo, conocido como el *espacio de estados* del sistema. El sistema se describe completamente mediante su *operador densidad*, que

es un operador positivo ρ con traza 1, que actúa en el espacio de estados del sistema. Si un sistema cuántico está en estado ρ_i con probabilidad p_i, entonces el operador densidad del sistema es $\sum_i p_i \rho_i$.

Postulado 2 La evolución de un sistema físico cuántico aislado se describe por una *transformación unitaria*. Es decir, el estado ρ del sistema en el tiempo t_1 se relaciona con el estado ρ' del sistema en el tiempo t_2 a través del operador unitario U, el cual solo depende de los tiempos t_1 y t_2.

$$\rho' = U \rho U^\dagger$$

Postulado 3 La medición cuántica se describe por una colección $\{M_m\}$ de *matrices de medición*. Dichas matrices actúan en el espacio de estados del sistema que se mide. El índice m refiere a los resultados posibles de la medición. Si el estado del sistema es ρ, inmediatamente antes de la medición, entonces la probabilidad de que el resultado m ocurra viene dada por

$$p(m) = \mathsf{tr}(M_m^\dagger M_m \rho) \tag{1.15}$$

y el estado del sistema luego de la medición es

$$\frac{M_m \rho M_m^\dagger}{\mathsf{tr}(M_m^\dagger M_m \rho)} \tag{1.16}$$

Las matrices satisfacen la ecuación de completitud,

$$\sum_m M_m^\dagger M_m = I$$

Postulado 4 El espacio de estados de un sistema físico compuesto es el producto tensorial de los espacios de estados de los componentes. Más aún, si tenemos sistemas enumerados de 1 a n, donde el sistema i está en el estado ρ_i, el estado conjunto del sistema total es $\rho_1 \otimes \rho_2 \otimes \cdots \otimes \rho_n$.

1.2.5. El operador densidad reducido

Uno de los usos más interesantes del operador densidad es para describir subsistemas de un sistema cuántico compuesto. Tal descripción viene dada por el *operador densidad reducido*.

Definición 1.25 (Traza parcial)**.** Sean A y B dos sistemas físicos. La traza parcial sobre el sistema B (tr_B) es un operador lineal definido por

$$\mathsf{tr}_B\left(|a_1\rangle\langle a_2| \otimes |b_1\rangle\langle b_2|\right) = |a_1\rangle\langle a_2|\,\mathsf{tr}(|b_1\rangle\langle b_2|) = \langle b_2|b_1\rangle\,|a_1\rangle\langle a_2|$$

para todo $|a_1\rangle, |a_2\rangle$ en el espacio de estados de A y $|b_1\rangle$, $|b_2\rangle$ en el espacio de estados de B.

Definición 1.26 (Operador densidad reducido)**.** Sean A y B dos sistemas físicos tales que su estado es descrito por el operador densidad ρ^{AB}. El operador densidad reducido para A se define por

$$\rho^A = \mathsf{tr}_B(\rho^{AB})$$

Ejemplos 1.27. Supongamos que tenemos un sistema cuántico en el estado $\rho^{AB} = \rho \otimes \sigma$, donde ρ es el operador densidad del sistema A y σ el del sistema B. Entonces,

$$\rho^A = \mathsf{tr}_B(\rho \otimes \sigma) = \rho\,\mathsf{tr}(\sigma) = \rho$$

Similarmente, $\rho^B = \sigma$.

Un ejemplo menos trivial es el estado de Bell $\beta_{00} = 1/\sqrt{2}(|00\rangle + |11\rangle)$, que tiene el siguiente operador densidad

$$\rho = \left(\frac{|00\rangle + |11\rangle}{\sqrt{2}}\right)\left(\frac{\langle 00| + \langle 11|}{\sqrt{2}}\right) = \frac{|00\rangle\langle 00| + |11\rangle\langle 00| + |00\rangle\langle 11| + |11\rangle\langle 11|}{2}$$

Haciendo la traza sobre el segundo qubit obtenemos el operador densidad del primer qubit:

$$\begin{aligned}
\rho^1 &= \mathsf{tr}_2(\rho) \\
&= \left(\mathsf{tr}_2(|00\rangle\langle 00|) + \mathsf{tr}_2(|11\rangle\langle 00|) + \mathsf{tr}_2(|00\rangle\langle 11|) + \mathsf{tr}_2(|11\rangle\langle 11|)\right)/2 \\
&= \left(\mathsf{tr}_2(|0\rangle\langle 0| \otimes |0\rangle\langle 0|) + \mathsf{tr}_2(|1\rangle\langle 0| \otimes |1\rangle\langle 0|)\right. \\
&\qquad \left. + \mathsf{tr}_2(|0\rangle\langle 1| \otimes |0\rangle\langle 1|) + \mathsf{tr}_2(|1\rangle\langle 1| \otimes |1\rangle\langle 1|)\right)/2 \\
&= \left(\langle 0|0\rangle\,|0\rangle\langle 0| + \langle 0|1\rangle\,|1\rangle\langle 0| + \langle 1|0\rangle\,|0\rangle\langle 1| + \langle 1|1\rangle\,|1\rangle\langle 1|\right)/2 \\
&= \left(|0\rangle\langle 0| + |1\rangle\langle 1|\right)/2 \\
&= I/2
\end{aligned}$$

Notar que este es un estado mixto, ya que $\mathsf{tr}((I/2)^2) = 1/2 < 1$. Es decir que, al estar enredados, por más que el estado de dos qubits sea un estado puro, el primer qubit solo está en un estado mixto: es decir, un estado que no conocemos completamente.

1.2.6. Purificación

Modelar sistemas cuánticos utilizando matrices de densidad puede resultar muy conveniente dependiendo del contexto. Como se verá más adelante estas representan la parte central del lenguaje λ_ρ. Sin embargo, hay que tener en cuenta que las máquinas cuánticas y sus lenguajes operan al nivel de qubits, es decir únicamente con estados puros.

Afortunadamente, existe una técnica descrita por Borgna [Bor19] (Sección 3.1.1, que fue basada en [NC10], Parte I-Sección 2.5) para convertir un estado mixto dado por una matriz de densidad en un estado puro. Esta operación se denomina purificación y la idea es agregar algunos qubits al sistema para codificar el estado mixto como parte de un estado puro.

Definición 1.28 (Purificación)**.** Sea ρ^A un estado correspondiente a un sistema cuántico A de n-qubits.

Como ρ^A es una matriz positiva, por el Teorema 1.9, tiene una descomposición en valores propios $\rho^A = \sum_{i=1}^{2^n} \lambda_i \left|v_i\right\rangle \left\langle v_i\right|$, donde $\{\left|v_i\right\rangle\}$ es una base ortonormal de autovectores [KC91].

Sea B otro sistema cuántico de n-qubits y $\{\left|e_i\right\rangle\}_{i=1,\ldots,2^n}$ una base ortonormal de B. Definimos la purificación de ρ^A como el estado cuántico del sistema $A \otimes B$ de la siguiente forma:

$$\text{pur}(\rho^A) = \sum_{i}^{2^n} \sqrt{\lambda_i} \left|v_i\right\rangle \otimes \left|e_i\right\rangle$$

1.3. Clases de complejidad

En esta sección vamos a presentar los conceptos básicos de la teoría de la intratabilidad: las clases P y NP de problemas. Estas son clasificaciones fundamentales para comprender la dificultad computacional de los problemas. Para hablar de complejidad de un problema siempre nos referimos a la máxima cantidad de instrucciones a ejecutar con una máquina de Turing determinista en función del tamaño de la entrada n para poder obtener el resultado del problema. Estas nociones están basadas a las explicadas en el libro "Introducción a la teoría de autómatas, lenguajes y computación" [HMU02].

También vamos a definir el concepto de "NP-completo", una propiedad que tienen determinados problemas de NP. Se trata de problemas como mínimo tan complejos (salvo diferencias polinómicas de tiempo) como cualquier problema de NP.

Clase P La clase P (Polinomial) está compuesta por problemas que pueden resolverse eficientemente en tiempo polinomial en función del tamaño de la entrada. Es decir, si un problema pertenece a P, existe un algoritmo que puede resolverlo en tiempo $O(n^k)$, donde n es el tamaño de la entrada y k es una constante. Ejemplos de problemas en P incluyen la suma de números enteros y la ordenación de una lista de elementos.

Clase NP La clase NP (No Determinista Polinomial) está formada por problemas para los cuales, dado un posible "certificado" o solución, se puede verificar en tiempo polinomial si es una solución válida. Aunque encontrar la solución puede ser difícil, comprobarla es fácil. Los problemas de NP se caracterizan por una complejidad temporal de $O(2^n)$, donde n es el tamaño de la entrada. Un ejemplo icónico de problema en NP es el problema del viajante de comercio.

Clase NP-Hard Un problema se considera NP-hard (o NP-difícil) si todos los problemas en NP pueden ser transformados en tiempo polinomial a una instancia del mismo. En otras palabras, si se pudiera encontrar un algoritmo eficiente para resolver un problema NP-hard, se podría encontrar un algoritmo eficiente para resolver todos los problemas en NP. Ejemplos de problemas NP-hard incluyen el problema de la mochila y el problema de la satisfacción booleana (SAT).

Clase NP-Completo Los problemas NP-completos son una clase especial de problemas que son tanto NP como NP-hard. El problema SAT es un ejemplo de un problema NP-completo. Resolver un problema NP-completo en tiempo polinomial demostraría que P = NP, lo cual es uno de los mayores misterios sin resolver en la teoría de la computación.

La relación entre estas clases se puede ilustrar en la Figura 1.1.

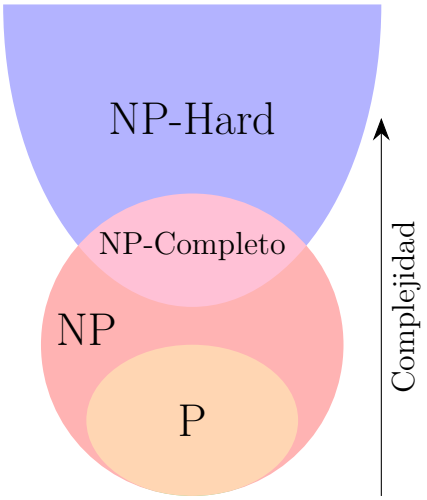

Figura 1.1: Diagrama de Venn mostrando la relación entre las clases de complejidad, asumiendo P $\neq$ NP.

1.4. Cubrimiento por vértices mínimo

El cubrimiento por vértices mínimo es un problema clásico de la teoría de grafos. A continuación, se enuncian definiciones basadas en el libro [Vaz08] que serán útiles en el Capítulo 2.

Definición 1.29 (Cubrimiento mínimo)**.** Un cubrimiento de vértices C para un grafo $G = (V, E)$ es un subconjunto de V tal que para cada arco $uv \in E$ se cumple que $u \in C$ o $v \in C$. Es decir, es un conjunto de vértices C donde cada arco tiene al menos uno de sus extremos en C. Tal conjunto se dice que cubre las aristas de g. Un cubrimiento mínimo es un cubrimiento con la menor cantidad de vértices.

Teorema 1.30. Sea un grafo $G = (V, E)$. El problema de encontrar el cubrimiento mínimo de vértices G es equivalente a resolver el siguiente problema de $|V|$ incógnitas:

$$\text{minimizar} \sum_{v \in V} x_v \tag{1.17}$$

$$\text{sujeto a } x_p + x_q \geq 1 \quad \forall pq \in E \tag{1.18}$$

$$x_v \in \{0, 1\} \quad \forall v \in V \tag{1.19}$$

□

Este teorema no se va a demostrar, pero se puede entender intuitivamente. A cada vértice se le asigna una variable que por la línea 1.19 debe ser cero o uno. Si es uno el vértice pertenece al cubrimiento mínimo y si es cero, no. La línea 1.17 establece que el cubrimiento debe ser el mínimo. La línea 1.18 se cerciora de que cada arco tenga al menos uno de sus extremos cubiertos.

Por último, aseguramos que la clase de complejidad de este problema es *NP-completo*, es decir que se encuentra en la clase *NP-hard* y a su vez posee una verificación polinomial.

Teorema 1.31. Encontrar el cubrimiento por vértices mínimo es NP-completo.

Este teorema fue probado por primera vez en 1972 por Richard Karp en su famosa publicación "Reducibility Among Combinatorial Problems" [Kar72], donde se prueba que 21 problemas diferentes son NP-completos.

Observación. Lo que es NP-completo es la versión decidible del problema. Es decir aquella que devuelve sí o no. En este caso el problema sería dado un grafo $G = (V, E)$ y un número k, responder si existe un cubrimiento de E, $S \subseteq V$ tal que $|S| \leq k$. Cabe destacar que si se puede responder esta pregunta en tiempo polinomial, entonces es posible reconstruir el cubrimiento mínimo en tiempo polinomial.

1.5. El Método Simplex

La *programación lineal* es un extenso campo del álgebra lineal dedicado a maximizar o minimizar (optimizar) una función lineal, de tal forma que las variables estén sujetas a una serie de restricciones expresadas mediante un sistema de ecuaciones también lineales. La técnica tradicionalmente usada para resolver problemas de programación lineal es el método Simplex. El mismo se encuentra explicado en el Capítulo 4 de "Operations Research: Applications and Algorithms" [Win22]. Este algoritmo resulta útil más adelante en el tipado de λ_ρ y se enuncia a continuación.

Definición 1.32 (Método Simplex)**.** El algoritmo de Simplex soluciona el problema de maximizar $c^T x$ sujeto a las restricciones $Ax = b$ y $x \geq 0$. Donde $x \in \mathbb{R}^n, c \in \mathbb{R}^n, b \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$.

No vamos a ahondar en su funcionamiento, pero sí vamos a utilizarlo para resolver la última etapa del tipado.

1.6. Funcionalidades de Haskell poco conocidas

Dado que en este trabajo elegimos Haskell como lenguaje de implementación, discutimos algunos detalles del mismo que resultan relevantes y suelen ser poco comprendidos. Entender estos conceptos es esencial para poder leer el código que se expone en el Capítulo 5.

1.6.1. Transformadores de Mónadas

La posibilidad de combinar mónadas resulta atractiva para implementar la traducción presentada en este trabajo, puesto que varios algoritmos necesitan un estado y a su vez tener la posibilidad de producir errores. Si bien podría declararse una mónada que tenga esas dos propiedades, combinar la mónada `State` y `Except` es un enfoque más conciso y claro.

Haskell tiene la capacidad de hacer esto empleando los transformadores de mónadas, que son variantes de las mónadas tradicionales. Sus constructores de tipo están parametrizados sobre un constructor de tipo de mónada y producen tipos monádicos combinados [New03].

1.6.2. Constructores de tipo transformador

Los constructores de tipos juegan un papel fundamental en el soporte de mónadas de Haskell. En particular, el tipo `Reader r a` representa valores de tipo `a` dentro de una mónada `Reader` con un entorno de tipo `r`. El constructor de tipos `Reader r` es una instancia de la clase `Monad`, y la función `runReader::(r->a)` ejecuta una computación en la mónada Reader y devuelve el resultado del tipo `a`.

Existe una versión transformadora de la mónada Reader, llamada `ReaderT`, que agrega un constructor de tipo de mónada como parámetro adicional. `ReaderT r m a` representan los valores de la mónada combinada en la que Reader es la mónada base y `m` es la mónada interna. `ReaderT r m` es una instancia de la clase `Monad`, y la función `runReaderT::(r -> m a)` realiza un cálculo en la mónada combinada y devuelve un resultado de tipo `m a`.

Utilizando las versiones transformadoras de las mónadas, podemos producir mónadas combinadas de forma muy sencilla. `ReaderT r IO` es una mónada combinada de Reader+IO. También podemos generar la versión sin transformador de una mónada a partir de la versión transformadora aplicándola a la mónada de identidad. Entonces `ReaderT r Identity` es la misma mónada que `Reader r`.

1.6.3. Lifting

Cuando usamos mónadas combinadas creadas por los transformadores de mónadas, evitamos tener que administrar explícitamente los tipos de mónadas internas, lo que da como resultado un código más claro y simple. En lugar de escribir código adicional dentro de las implementaciones para manipular valores del tipo de la mónada interna, podemos recurrir a las operaciones de lifting para traer funciones de la mónada interna a la mónada combinada. Esto quiere decir que sabiendo como manipular cada mónada involucrada es posible escribir código que opere sobre ambas mónadas.

La familia de funciones `liftM` se utilizan para elevar funciones no monádicas a una mónada. Cada transformador de mónada proporciona una función de elevación que se emplea para elevar un cálculo monádico a una mónada combinada. Los transformadores también proporcionan una función `liftIO`, que es una versión de lift que está optimizada para hacer lifting de la mónada IO.

1.7. λ_ρ, un cálculo lambda con matrices de densidad

En 2017 Díaz-Caro presenta λ_ρ [Día17], un cálculo lambda cuántico que usa matrices de densidad para representar estados cuánticos. El mismo se caracteriza por codificar los estados directamente sobre los términos, a diferencia de otros lenguajes como el descrito por Selinger [Sel04] que recurre a las matrices de densidad únicamente para su interpretación. Esto genera lenguajes más simples, ya que no se necesita una clausura para hacer referencia el estado cuántico.

El cálculo λ_ρ utiliza un sistema de reducción probabilístico para modelar la operación de medición. Es importante resaltar que, si bien podemos codificar estados mixtos, la medición probabilística va a generar un estado puro. En las siguientes secciones se detalla la definición, el tipado y las reglas de reducción del lenguaje.

1.7.1. Definición de λ_ρ

Se puede entender a λ_ρ como una extensión del lambda cálculo simplemente tipado con términos que representan los cuatro postulados cuánticos y un término (letcase) para el control clásico basado en el resultado de las mediciones. Siguiendo esta idea se define el conjunto de términos de λ_ρ (Λ_ρ) en la Definición 1.33.

Definición 1.33 (Gramática de λ_ρ)**.**

$$\begin{aligned} t ::= {} & x \mid \lambda x.t \mid t\ t && \text{(Lambda cálculo estándar)} \\ & \mid \rho^n \mid U^n t \mid \pi^m t \mid t \otimes t && \text{(Postulados cuánticos)} \\ & \mid (b^m, \rho^n) \mid \mathsf{letcase}\ x = t\ \mathsf{in}\ \{t, \ldots, t\} && \text{(Control clásico)} \end{aligned}$$

donde:

- $n, m \in \mathbb{N}, m \leq n$.
- ρ^n es una matriz de densidad de n-qubits, es decir, una matriz positiva de $2^n \times 2^n$ con traza 1.
- $b^m \in \mathbb{N}_0, 0 \leq b^m < 2^m$.
- $t, \ldots, t$ contiene 2^m términos
- U^n es un operador unitario de dimensión $2^n \times 2^n$, es decir, una matriz $2^n \times 2^n$ tal que $(U^n)^\dagger = (U^n)^{-1}$.
- $\pi^n = \{\pi_0, \ldots, \pi_{2^n-1}\}$, describe una medición cuántica en la base computacional, donde cada π_i es un operador proyector de dimensión 2^n que proyecta a un vector de la base canónica.

1.7.2. Tipado de λ_ρ

El sistema de tipos de λ_ρ es afín, por lo que las variables pueden ser usadas a lo sumo una sola vez. Otra característica peculiar es que contiene enteros. Por ejemplo, el tipo '3' representa un estado cuántico de 3 qubits. Mientras que el tipo '$(3, 5)$' es el resultado de una medición de 3 qubits sobre un estado de 5 qubits. Sobre estos valores se agregan las funciones de alto orden del cálculo lambda afín.

Para formalizar el tipado introducimos contextos que son un mapeo entre variables y sus tipos. Por ejemplo $\Delta = x : A$ es un contexto que le asigna el tipo A a la variable x. Dados dos contextos Δ y Γ, denotamos Δ, Γ como la unión de ambos contextos:

$$(\Delta, \Gamma)(x) = \begin{cases} \Delta(x) \text{ si } x \in \mathrm{dom}(\Delta) \\ \Gamma(x) \text{ si } x \in \mathrm{dom}(\Gamma) \end{cases}$$

Definición 1.34 (Tipado de λ_ρ)**.** Siendo $n, m \in \mathbb{N}, m \leq n$,

$$A := n \mid (m, n) \mid A \multimap A$$

$$\frac{}{\Gamma, x : A \vdash x : A}\ \mathtt{ax} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \multimap_i \qquad \frac{\Gamma \vdash t : A \multimap B \quad \Delta \vdash r : A}{\Gamma, \Delta \vdash t\ r : B}\ \multimap_e$$

$$\frac{}{\Gamma \vdash \rho^n : n}\ \mathtt{ax}_\rho \qquad \frac{\Gamma \vdash t : n}{\Gamma \vdash U^m\ t : n}\ \mathtt{u} \qquad \frac{\Gamma \vdash t : n}{\Gamma \vdash \pi^m t : (m, n)}\ \pi$$

$$\frac{\Gamma \vdash t : n \quad \Delta \vdash r : m}{\Gamma, \Delta \vdash t \otimes r : n + m}\ \otimes \qquad \frac{}{\Gamma \vdash (b^m, \rho^n) : (m, n)}\ \mathtt{ax_{am}}$$

$$\frac{x : n \vdash t_0 : A \quad \ldots \quad x : n \vdash t_{2^m-1} : A \quad \Gamma \vdash r : (m, n)}{\Gamma \vdash \mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\} : A}\ \mathtt{lc}$$

1.7.3. Reglas de reducción

Las reglas de reducción dadas en la Definición 1.35, son descritas por la relación $\xrightarrow{\lambda_\rho}_p$, que es una relación probabilística donde p es la probabilidad de que tal reducción ocurra.

Si U^m es aplicado a ρ^n, con $m \leq n$, definimos $\overline{U^m} = U^m \otimes I^{n-m}$. Similarmente, cuando π^m es aplicado a ρ^n, con $m \leq n$, definimos $\overline{\pi^m}$ como el conjunto de operadores de medición $\{\pi_0 \otimes I^{n-m}, \ldots, \pi_{2^m-1} \otimes I^{n-m}\}$.

Antes de dar las reglas es conveniente dejar en claro la notación para las sustituciones.

Sustitución de expresiones

La sustitución se define de la forma usual. Escribimos A en lugar de α como $[A/\alpha]$ y la sustitución nula como $[-]$. Aplicar la sustitución σ a la expresión E se denota σE. Las sustituciones σ y μ se dice que son iguales si y solo sí $\forall x\ \sigma(x) = \mu(x)$. Al ser funciones, las sustituciones pueden ser compuestas y aplicadas. El dominio y rango de una sustitución se definen de la siguiente manera:

$$\mathrm{dom}(\sigma) = \{\alpha/\alpha \neq \sigma(\alpha)\} \tag{1.20}$$
$$\mathrm{rng}(\sigma) = \{\sigma(\alpha)/\alpha \in \mathrm{dom}(\sigma)\} \tag{1.21}$$

Estas definiciones permiten probar que $\sigma = \sigma' \implies \mathrm{dom}(\sigma) = \mathrm{dom}(\sigma')$ y $\mathrm{rng}(\sigma) = \mathrm{rng}(\sigma')$.

Finalmente, se enuncian las reglas de reducción de λ_ρ.

Definición 1.35 (Reglas de reducción de λ_ρ)**.**

$$(\lambda x.t)\ r \xrightarrow{\lambda_\rho}_1 [r/x]t$$

$$U^m\ \rho^n \xrightarrow{\lambda_\rho}_1 \rho'^n \qquad \text{con } \rho'^n = \overline{U^m}\ \rho^n\ \overline{U^m}^\dagger$$

$$\pi^m\ \rho^n \xrightarrow{\lambda_\rho}_{p_i} (i^m, \rho_i^n) \quad \text{con} \begin{cases} p_i = \mathsf{tr}(\overline{\pi_i^m}^\dagger\ \overline{\pi_i^m}\ \rho^n) \\ \rho_i^n = (\overline{\pi_i^m}\ \rho^n\ \overline{\pi_i^m}^\dagger)/p_i \end{cases}$$

$$\rho \otimes \rho' \xrightarrow{\lambda_\rho}_1 \rho'' \qquad \text{con } \rho'' = \rho \otimes \rho'$$

$$\mathsf{letcase}\ x = (b^m, \rho^n)\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\} \xrightarrow{\lambda_\rho}_1 [\rho^n/x]t_{b^m}$$

$$\frac{t \xrightarrow{\lambda_\rho}_p r}{\lambda x.t \xrightarrow{\lambda_\rho}_p \lambda x.r} \qquad \frac{t \xrightarrow{\lambda_\rho}_p r}{t\ s \xrightarrow{\lambda_\rho}_p r\ s} \qquad \frac{t \xrightarrow{\lambda_\rho}_p r}{s\ t \xrightarrow{\lambda_\rho}_p s\ r} \qquad \frac{t \xrightarrow{\lambda_\rho}_p r}{U^m\ t \xrightarrow{\lambda_\rho}_p U^m\ r}$$

$$\frac{t \xrightarrow{\lambda_\rho}_p r}{\pi^n\ t \xrightarrow{\lambda_\rho}_p \pi^n\ r} \qquad \frac{t \xrightarrow{\lambda_\rho}_p r}{t \otimes s \xrightarrow{\lambda_\rho}_p r \otimes s} \qquad \frac{t \xrightarrow{\lambda_\rho}_p r}{s \otimes t \xrightarrow{\lambda_\rho}_p s \otimes r}$$

$$\frac{t \xrightarrow{\lambda_\rho}_p r}{\mathsf{letcase}\ x = t\ \mathrm{in}\ \{s_0, \ldots, s_n\} \xrightarrow{\lambda_\rho}_p \mathsf{letcase}\ x = r\ \mathrm{in}\ \{s_0, \ldots, s_n\}}$$

La siguiente notación es útil para cuando queremos hacer varios pasos de la reducción de forma sucesiva.

Definición 1.36 ($\to_p^*$)**.** Para cualquier relación probabilística $\to_p$, denotamos $\to_p^*$ a su cierre reflexivo y transitivo. Es decir:

$$\frac{}{x \to_1^* x} \qquad \frac{x \to_p^* y \quad y \to_q z}{x \to_{pq}^* z}$$

Capítulo 2

Tipado de λ_ρ

Antes de traducir una expresión dada de λ_ρ, resulta conveniente verificar si es válida. Esto se puede hacer realizando una derivación de tipos utilizando las reglas dadas en la Definición 1.34.

Esto no es trivial, ya que λ_ρ es un lenguaje de tipado implícito, es decir, el tipo de las variables no está indicado en las expresiones. Por ejemplo, la expresión $\lambda x.x$ tiene infinitos tipos porque x puede ser de cualquier tipo. Otras expresiones tienen solo un tipo: $|0\rangle\langle 0|$, puesto que es una matriz de densidad de un solo qubit. Dada una expresión basta con encontrar un tipo para demostrar que es válida.

La forma más simple de resolver este problema es modificar λ_ρ para que sea de tipado explícito, es decir que la misma expresión provea los tipos de las subexpresiones. En este caso el tipado se vuelve un problema de chequeo casi trivial. Sin embargo, para este trabajo se optó por el tipado implícito original debido a la interesante complejidad que surge a partir del sistema de tipos relativamente simple de λ_ρ, que veremos en más profundidad en la siguientes secciones.

2.1. Inferencia de tipos simples

El proceso de asignarle un tipo a una expresión se conoce como inferencia de tipos. Se habla de tipos simples debido a que los tipos de λ_ρ no son polimórficos. Existen dos algoritmos que en conjunto logran esta tarea: el algoritmo de Hindley y el de unificación de Robinson. Inicialmente, Hindley procesa la expresión y genera una serie de ecuaciones de tipos que el tipado final debe satisfacer para ser válida. A continuación, el algoritmo de Robinson resuelve estas ecuaciones utilizando un método de unificación para llegar al tipado final.

En el capítulo de 6 del libro [Gil11] se explican ambas etapas para un lenguaje gené-

rico similar al cálculo lambda (PCF). Aquí se presentan dichos algoritmos adaptados para λ_ρ.

Observación. Es posible combinar las etapas de Hindley y Robinson en una sola, sin embargo, no es aplicable para el caso de λ_ρ debido a las inecuaciones involucradas, como se verá más adelante.

2.1.1. Dificultades al tipar λ_ρ

Las adaptaciones necesarias a los algoritmos de inferencia clásicos no son triviales, puesto que nuestro lenguaje tiene naturales como parte del tipo. Veamos como esta característica tiene el efecto de producir inecuaciones.

Ejemplo 2.1. Dada la siguiente expresión: $\lambda y.\lambda x.\pi^5(x \otimes y)$, podemos hacerle la siguiente derivación de tipo:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x:1 \vdash x:1}\,\mathtt{ax} \quad \dfrac{}{y:4 \vdash y:4}\,\mathtt{ax}}{y:4, x:1 \vdash x \otimes y:5}\,\otimes}{y:4, x:1 \vdash \pi^5(x \otimes y):(5,5)}\,m}{y:4 \vdash \lambda x.\pi^5(x \otimes y):1 \multimap (5,5)}\,\multimap_i}{\vdash \lambda y.\lambda x.\pi^5(x \otimes y):4 \multimap (1 \multimap (5,5))}\,\multimap_i$$

Sin embargo, este no es el único tipo posible. Intuitivamente, observamos que únicamente se requiere que el tamaño de $x \otimes y$ sea mayor o igual a 5. En este caso otros tipos válidos son:

- $3 \multimap (2 \multimap (5,5))$
- $1 \multimap (100 \multimap (5,101))$

O más generalmente $X \multimap (Y \multimap (5, X+Y)), X+Y \geq 5$.

Una posible solución a este problema sería dar un tipado polimórfico (es decir con variables de tipo) y tipar la expresión anterior como $X \multimap (Y \multimap (5, X+Y))$. El problema con esto es que hay expresiones que son inválidas porque no existe ninguna asignación a las variables de tipo que las satisfagan. Esto se ejemplifica a continuación.

Ejemplo 2.2. Intentemos tipar la siguiente expresión:

$$\mathsf{letcase}\ x = \pi^2 \left|00\right\rangle \left\langle 00\right| \ \text{in}$$
$$\{\lambda a.\lambda b.\lambda c.a \otimes b,\ \lambda a.\lambda b.\lambda c.b \otimes c,\ \lambda a.\lambda b.\lambda c.a \otimes c,\ \lambda a.\lambda b.\lambda c.\left|00000\right\rangle \left\langle 00000\right|\}$$

Debido a la regla `lc`, todos los casos deben tener el mismo tipo. Como son funciones, esto significa que tanto los argumentos recibidos como el tipo de retorno deben ser

qubits del mismo tamaño. El cuarto caso fija el valor de retorno en 5. Si el tipo de los casos son de la forma $a \multimap b \multimap c \multimap 5$. Se puede ver que un tipado válido debe satisfacer las siguientes ecuaciones:

$$\begin{cases} a + b = 5 \\ b + c = 5 \\ a + c = 5 \end{cases}$$

Este sistema no tiene soluciones enteras, por lo que la expresión es inválida.

Observación (Equivalencia con sistemas de ecuaciones lineales). Sea un sistema de ecuaciones de la forma $Ax = b$ con $x, b \in \mathbb{N}^n, x_i \geq 1, b_i \geq 1$, donde A está formado por ceros o unos. Es posible construir una expresión de λ_ρ de manera similar al Ejemplo 2.2 cuyo tipado sea equivalente a encontrar una solución de tal sistema. Más aún, el sistema tiene solución si y solo si tal expresión tiene un tipo. Esta afirmación no se va a demostrar, ya que vamos a enfocarnos en un resultado más importante con el Teorema 2.14, pero sirve para ilustrar la potencial complejidad que es tipar λ_ρ.

Basándonos en estos ejemplos observamos que el algoritmo de Hindley para λ_ρ tiene que producir inecuaciones también. Y sucesivamente en el algoritmo de Robinson debemos resolverlas.

2.2. Algoritmo de Hindley

El algoritmo de Hindley es un algoritmo recursivo sobre la expresión a tipar, que lleva también un contexto a medida que se ejecuta. El resultado devuelto es una tupla compuesta por el tipo para la expresión y un conjunto de ecuaciones. Escribimos $\Gamma \vdash t \rightsquigarrow A, E$ para simbolizar que dado un término t bajo un entorno Γ, el algoritmo de Hindley devuelve el tipo A y las ecuaciones E que debe satisfacer.

Podemos describir esta primera etapa usando un conjunto de reglas en el estilo de una semántica grandes pasos. Las premisas de cada regla representan los subproblemas de la recursión y la conclusión define el resultado de esta recursión.

Nótese que para poder expresar las ecuaciones debemos admitir temporalmente variables de tipo. Es decir, el tipo A devuelto por el algoritmo aún no está totalmente resuelto. En la Sección 2.3 se reemplazan estas variables por tipos concretos.

Definición 2.3 (Variables de tipo libres)**.** Definimos la función ftv sobre un tipo de λ_ρ como el conjunto de variables de tipo que contiene. Esta función se extiende de manera intuitiva a los contextos ($\text{ftv}(\Gamma) = \text{ftv}(a : A, b : B, c :$

$C, \dots) = \text{ftv}(A) \cup \text{ftv}(B) \cup \text{ftv}(C) \cup \dots)$, ecuaciones y sustituciones. También utilizamos la notación $\text{ftv}(A; B; C; \dots) = \text{ftv}(A) \cup \text{ftv}(B) \cup \text{ftv}(C) \cup \dots$.

Cuando sea necesario vamos a distinguir entre la igualdad entre tipos $=_\rho$ y la igualdad entre los números naturales $=_\mathbb{N}$, se sobreentiende que el operador $\geq$ aplica solo para los naturales. Para la igualdad entre ecuaciones usamos $\equiv$, dado que las ecuaciones mismas ya contienen $=$.

Definición 2.4 (Algoritmo de Hindley para λ_ρ)**.**

$$\frac{}{\Gamma, x : A \vdash x \rightsquigarrow A, \emptyset}\ \text{Hax} \qquad \frac{\Gamma, x : A \vdash t \rightsquigarrow B, E}{\Gamma \vdash \lambda x.t \rightsquigarrow A \multimap B, E}\ \text{H}\multimap_i$$

$$\frac{\Gamma \vdash t \rightsquigarrow C, E \quad \Delta \vdash r \rightsquigarrow A, F}{\Gamma, \Delta \vdash t\ r \rightsquigarrow B, E \cup F \cup \{C =_\rho A \multimap B\}}\ \text{H}\multimap_e$$

$$\frac{}{\Gamma \vdash \rho^n \rightsquigarrow n, \emptyset}\ \text{Hax}_\rho \qquad \frac{\Gamma \vdash t \rightsquigarrow A, E}{\Gamma \vdash U^m\ t \rightsquigarrow A, E \cup \{A \geq m\}}\ \text{Hu}$$

$$\frac{\Gamma \vdash t \rightsquigarrow A, E}{\Gamma \vdash \pi^m t \rightsquigarrow (m, A), E \cup \{A \geq m\}}\ \text{H}m$$

$$\frac{\Gamma \vdash t \rightsquigarrow A, E \quad \Delta \vdash r \rightsquigarrow B, F}{\Gamma, \Delta \vdash t \otimes r \rightsquigarrow C, E \cup F \cup \{C =_\mathbb{N} A + B\}}\ \text{H}\otimes$$

$$\frac{}{\Gamma \vdash (b^m, \rho^n) \rightsquigarrow (m, n), \emptyset}\ \text{Hax}_{\texttt{am}}$$

$$\frac{\begin{array}{c} x : X \vdash t_0 \rightsquigarrow A_0, E_0 \\ \dots \\ x : X \vdash t_{2^m-1} \rightsquigarrow A_{2^m-1}, E_{2^m-1} \qquad \Gamma \vdash r \rightsquigarrow B, F \end{array}}{\Gamma \vdash \mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_0, \dots, t_{2^m-1}\} \rightsquigarrow A_0, \bigcup \begin{cases} E_0,\ \dots,\ E_{2^m-1} \\ F \\ \{A_0 =_\rho A_1\} \\ \quad \vdots \\ \{A_0 =_\rho A_{2^m-1}\} \\ \{B =_\rho (m, X)\} \\ \{X \geq m\} \end{cases}} \texttt{Hlc}$$

donde:

- $B \notin \mathrm{ftv}(\Gamma; \Delta; A; C; E; F)$ para la regla H $\multimap_e$.
- $C \notin \mathrm{ftv}(\Gamma; \Delta; A; B; E; F)$ para la regla H$\otimes$.
- $X \notin \mathrm{ftv}(\Gamma; B; E)$ para la regla Hlc.

Ejemplo 2.5. Se presenta el árbol de derivación del algoritmo para la expresión del Ejemplo 2.1: $\lambda y.\lambda x.\pi^5(x \otimes y)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : X \vdash x : X, \emptyset}\,\texttt{Hax} \quad \dfrac{}{y : Y \vdash y : Y, \emptyset}\,\texttt{Hax}}{y : Y, x : X \vdash x \otimes y : Z, \{Z = X + Y\}}\,\mathrm{H}\otimes}{y : Y, x : X \vdash \pi^5(x \otimes y) \rightsquigarrow (5, Z), \{Z = X + Y, Z \geq 5\}}\,\mathrm{H}m}{y : Y \vdash \lambda x.\pi^5(x \otimes y) \rightsquigarrow X \multimap (5, Z), \{Z = X + Y, Z \geq 5\}}\,\mathrm{H} \multimap_i}{\vdash \lambda y.\lambda x.\pi^5(x \otimes y) \rightsquigarrow X \multimap Y \multimap (5, Z), \{Z = X + Y, Z \geq 5\}}\,\mathrm{H} \multimap_i$$

Para demostrar la correctitud de este algoritmo resulta útil primero definir que se considera como una solución del sistema de ecuaciones.

Definición 2.6 (Solución)**.** Sea E un sistema de ecuaciones con variables de tipo $X_0, \dots, X_{n-1}$. Una sustitución $\sigma = [V_0/X_0, \dots, V_{n-1}/X_{n-1}]$ es una *solución* de E si por cada ecuación $C = D$ en E, se cumple que $\sigma C = \sigma D$, análogamente por cada ecuación $C \geq D$ en E, se cumple que $\sigma C \geq \sigma D$. Podemos también

decir que σE se satisface (es decir todas sus ecuaciones son trivialmente ciertas).

Observación. Si σ satisface la ecuación $C =_{\mathbb{N}} A + B$, es decir, se cumple $\sigma C =_{\mathbb{N}} \sigma A + \sigma B$, queda implícito que $\sigma A, \sigma B, \sigma C \in \mathbb{N}$. De igual manera, si la ecuación es de la forma $A \geq B$, $\sigma A, \sigma B \in \mathbb{N}$.

En el siguiente teorema vamos a demostrar la correctitud probando que, si una solución verifica las ecuaciones generadas por el algoritmo de Hindley para un término t, entonces tal solución se puede usar para dar un tipo válido a t.

Teorema 2.7 (Correctitud de Hindley)**.** Si tenemos que $\Gamma \vdash t \rightsquigarrow A, E$ y σ es una solución de E, entonces $\sigma\Gamma \vdash t : \sigma A$.

Demostración. Se procede por inducción en el árbol de derivación de $\rightsquigarrow$.

Caso Hax

Dado $\Gamma, x : A \vdash x \rightsquigarrow A, \emptyset$. Sea σ la sustitución vacía que satisface trivialmente $\emptyset$. Por la regla ax de la Definición 1.34 sobre $\sigma\Gamma, x : \sigma A \vdash x$, se tiene:

$\sigma\Gamma, x : \sigma A \vdash x : \sigma A$.

Caso H $\multimap_i$

Dado $\Gamma \vdash \lambda x.t \rightsquigarrow A \multimap B, E$. Por hipótesis de inducción: $\sigma\Gamma, x : \sigma A \vdash t : \sigma B$ y σ satisface E. Aplicando la regla $\multimap_i$ de la Definición 1.34 se tiene que $\sigma\Gamma \vdash \lambda x.t : \sigma A \multimap \sigma B$.

Caso H $\multimap_e$

Dado $\Gamma, \Delta \vdash t\ r \rightsquigarrow B, E \cup F \cup \{C =_\rho A \multimap B\}$. Sea σ una solución de $E \cup F \cup \{C =_\rho A \multimap B\}$.

Como σ satisface E y F, por HI: $\sigma\Gamma \vdash t : \sigma C = \sigma A \multimap \sigma B$ y $\sigma\Delta \vdash r : \sigma A$. Aplicando la regla $\multimap_e$ de la Definición 1.34 se tiene que $\sigma\Gamma, \sigma\Delta \vdash t\ r : \sigma B$.

Caso Hax_ρ

Dado $\Gamma \vdash \rho^n \rightsquigarrow n, \emptyset$. Sea σ la sustitución vacía. Por ax_ρ de la Definición 1.34 sobre $\sigma\Gamma \vdash \rho^n$, se tiene:

$\sigma\Gamma \vdash \rho^n : n = \sigma n$.

Caso Hu

Dado $\Gamma \vdash U^m\ t \rightsquigarrow A, E \cup \{A \geq m\}$. Sea σ una solución de $E \cup \{A \geq m\}$.

Como σ es solución de E, por HI: $\sigma\Gamma \vdash t : \sigma A$. Como σ es solución de $A \geq m$, sabemos que $\sigma A \geq m$, por lo que se puede aplicar la regla $\mathtt{u}$ de la Definición 1.34 obteniendo $\sigma\Gamma \vdash U^m\ t : \sigma A$.

Caso Hm

Dado $\Gamma \vdash \pi^m t \rightsquigarrow (m, A), E \cup \{A \geq m\}$. Sea σ una solución de $E \cup \{A \geq m\}$.

Como σ satisface E, por HI: $\sigma\Gamma \vdash t : \sigma A$. Puesto que $\sigma A \geq m$ (y a su vez $\sigma A \in \mathbb{N}$), se puede aplicar la regla m de la Definición 1.34 llegando a $\sigma\Gamma \vdash \pi^m\ t : (m, \sigma A) = \sigma(m, A)$.

Caso H$\otimes$

Dado $\Gamma, \Delta \vdash t \otimes r \rightsquigarrow C, E \cup F \cup \{C =_{\mathbb{N}} A + B\}$. Sea σ una solución de $E \cup F \cup \{C =_{\mathbb{N}} A + B\}$.

Como σ satisface E y F, por HI: $\sigma\Gamma \vdash t : \sigma A$ y $\sigma\Delta \vdash r : \sigma B$. A su vez como σ satisface $C =_{\mathbb{N}} A + B$, $\sigma C =_{\mathbb{N}} \sigma A + \sigma B$ y $\sigma A, \sigma B, \sigma C \in \mathbb{N}$.

Aplicando la regla $\otimes$ de la Definición 1.34 se tiene que $\sigma\Gamma, \sigma\Delta \vdash t \otimes r : \sigma A + \sigma B =_{\mathbb{N}} \sigma C$.

Caso H$\mathtt{ax}_{\mathtt{am}}$

Dado $\Gamma \vdash (b^m, \rho^n) \rightsquigarrow (m, n), \emptyset$. Sea σ la sustitución vacía. Por la regla $\mathtt{ax}_{\mathtt{am}}$ de la Definición 1.34 sobre $\sigma\Gamma \vdash (b^m, \rho^n)$ se tiene que $\sigma\Gamma \vdash (b^m, \rho^n) : (m, n)$.

Caso H$\mathtt{lc}$

Dado $\Gamma \vdash \mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\} \rightsquigarrow A_0, \bigcup_{i=0}^{2^m-1} E_i \cup F \cup \{A_0 = A_1, \ldots, A_0 =_\rho A_{2^m-1}, B =_\rho (m, X), X \geq m\}$. Sea σ una solución de $\bigcup_{i=0}^{2^m-1} E_i \cup F \cup \{A_0 =_\rho A_1, \ldots, A_0 =_\rho A_{2^m-1}, B =_\rho (m, X), X \geq m\}$.

Para cada $i \in \{0 \ldots 2^m - 1\}$, tenemos que:

- Como σ satisface E_i, por HI: $x : \sigma X \vdash t_i : \sigma A_i$.
- Como σ satisface $A_0 =_\rho A_i \implies \sigma A_0 =_\rho \sigma A_i \implies x : \sigma X \vdash t_i : \sigma A_0$.

A su vez:

- Como σ satisface F, por HI: $\Gamma \vdash r : \sigma B$.
- Como σ satisface $B =_\rho (m, X) \implies \sigma B =_\rho \sigma(m, X) = (m, \sigma X)$.

Finalmente, resulta posible aplicar la regla $\mathtt{lc}$ de la Definición 1.34:

$$\frac{x : \sigma X \vdash t_0 : \sigma A_0 \quad \ldots \quad x : \sigma X \vdash t_{2^m-1} : \sigma A \quad \sigma\Gamma \vdash r : (m, \sigma X)}{\sigma\Gamma \vdash \mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\} : \sigma A_0}$$

□

Lema 2.8. Podemos asumir que las variables de tipos en los distintos contextos, conjuntos y ecuaciones en las premisas de las reglas de Hindley son disjuntos. Es decir:

- Para la regla H $\multimap_e$: $\text{ftv}(\Gamma; C; E) \cap \text{ftv}(\Delta; A; F) = \emptyset$.
- Para la regla H$\otimes$: $\text{ftv}(\Gamma; A; E) \cap \text{ftv}(\Delta; B; F) = \emptyset$.
- Para la regla H`lc`: $(\bigcup_i \text{ftv}(A_i; E_i) \cup \{X\}) \cap \text{ftv}(\Gamma; B; F) = \emptyset$.

□

El siguiente teorema resulta imprescindible para probar la validez del algoritmo. Más aún será utilizado para probar más adelante que el tipado de λ_ρ es un problema NP-hard (Teorema 2.14), puesto que todo tipado provee implícitamente una solución para las ecuaciones de Hindley.

Teorema 2.9 (Completitud de Hindley)**.** Si $\Gamma \vdash t : A$, entonces $\exists \Gamma', A', E, \sigma$ tal que:

1. $\Gamma' \vdash t \rightsquigarrow A', E$.
2. σ satisface E.
3. $\sigma A' = A \wedge \sigma\Gamma' = \Gamma$.
4. $\text{dom}(\sigma) \subseteq \text{ftv}(\Gamma'; A'; E)$

Demostración. Se procede por inducción en la derivación del tipo $t : A$.

Caso $\dfrac{}{\Gamma, x : A \vdash x : A}\ \texttt{ax}$

Por H`ax`, tenemos que $\Gamma, X : A \vdash x \rightsquigarrow A, \emptyset$. Sea $\sigma = \emptyset$. Trivialmente se da que $\sigma A = A$ y $\sigma(\Gamma, x : A) = \sigma\Gamma, x : \sigma A = \Gamma, x : A$. A su vez $\text{dom}(\sigma) \subseteq S\ \forall S$.

Caso $\dfrac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \multimap B}\ \multimap_i$

Por hipótesis de inducción tenemos que: $\Gamma', x : A' \vdash t \rightsquigarrow B', E$ y $\exists \sigma / \sigma$ satisface las ecuaciones de E, $\sigma B' = B, \sigma\Gamma' = \Gamma, \sigma A' = A, \text{dom}(\sigma) \subseteq \text{ftv}(\Gamma'; A'; E)$.

Aplicando la regla H $\multimap_i$, $\Gamma' \vdash \lambda x.t \rightsquigarrow A' \multimap B', E$.

A su vez $\sigma(A' \multimap B') = \sigma A' \multimap \sigma B' = A \multimap B$ y $\sigma\Gamma' = \Gamma$.

Caso $\dfrac{\Gamma \vdash t : A \multimap B \quad \Delta \vdash r : A}{\Gamma, \Delta \vdash t\ r : B}\ \multimap_e$

Por hipótesis de inducción tenemos que:

- $\Gamma' \vdash t \rightsquigarrow C', E$ y $\exists \sigma / \sigma$ satisface las ecuaciones de E y $\sigma\Gamma' = \Gamma, \sigma C' = A \multimap B, \mathrm{dom}(\sigma) \subseteq \mathrm{ftv}(\Gamma'; C'; E)$.
- $\Delta' \vdash r \rightsquigarrow A', F.$ y $\exists \mu / \mu$ satisface las ecuaciones de F y $\mu\Delta' = \Delta, \mu A' = A, \mathrm{dom}(\mu) \subseteq \mathrm{ftv}(\Delta'; A'; F)$.

Aplicando la regla H $\multimap_e$:

$$\Gamma', \Delta' \vdash t\ r \rightsquigarrow B', E \cup F \cup \{C' =_\rho A' \multimap B'\}$$

donde $B' \notin \mathrm{ftv}(\Gamma'; \Delta'; E; F; A'; C')$.

Sea $\tau = [B/B'] \circ \mu \circ \sigma$ (composición de las sustituciones). Se tiene que:

- Aplicando el Lema 2.8, $\mathrm{dom}(\mu) \cap \mathrm{ftv}(E) = \emptyset$ y $B' \notin \mathrm{ftv}(E)$ por lo que $\tau E = ([B/B'] \circ \mu \circ \sigma)E = \sigma E$. De esta manera como σE se satisface, τE se satisface.
- Aplicando el Lema 2.8, $\mathrm{dom}(\sigma) \cap \mathrm{ftv}(F) = \emptyset$ por lo que $\sigma F = F$. Luego como $B' \notin \mathrm{ftv}(F)$ tenemos que $\tau F = \mu F$. De esta manera como μ satisface F, τ satisface F.
- $\tau(C' =_\rho A' \multimap B') \equiv (\tau C' =_\rho \tau A' \multimap \tau B') \equiv (A \multimap B =_\rho A \multimap B)$. Por lo que τ satisface $\{C' =_\rho A' \multimap B'\}$
- Aplicando el Lema 2.8, $\mathrm{dom}(\sigma) \cap \mathrm{ftv}(\Delta') = \emptyset \wedge \mathrm{dom}(\mu) \cap \mathrm{ftv}(\Gamma') = \emptyset \wedge B' \notin \mathrm{ftv}(\Gamma'; \Delta') \implies \tau(\Gamma', \Delta') = \tau\Gamma', \tau\Delta' = \sigma\Gamma', \mu\Delta' = \Gamma, \Delta$.
- Como $B' \notin \mathrm{dom}(\mu) \cup \mathrm{dom}(\sigma)$, $\tau B' = [B/B']B' = B$.
- $\mathrm{dom}(\tau) = \mathrm{dom}(\sigma) \cup \mathrm{dom}(\mu) \cup \{B'\} \subseteq \mathrm{ftv}(\Gamma'; C'; E) \cup \mathrm{ftv}(\Delta'; A'; F) \cup \{B'\} = \mathrm{ftv}(\Gamma'; \Delta'; B'; E; F; \{C' =_\rho A' \multimap B'\})$.

Caso $\dfrac{}{\Gamma \vdash \rho^n : n}\ \mathtt{ax}_\rho$

Por $\mathtt{Hax}_\rho$, tenemos que $\Gamma \vdash \rho^n \rightsquigarrow n, \emptyset$. Sea $\sigma = \emptyset$. Trivialmente se da que $\sigma n = n$ y $\sigma\Gamma = \Gamma$. A su vez $\mathrm{dom}(\sigma) \subseteq S\ \forall S$.

Caso $\dfrac{\Gamma \vdash t : n}{\Gamma \vdash U^m\ t : n}\ \mathtt{u}$

Por hipótesis de inducción: $\Gamma' \vdash t \rightsquigarrow A', E$ y $\exists \sigma / \sigma$ satisface las ecuaciones de $E, \sigma A' = n, \sigma\Gamma' = \Gamma, \mathrm{dom}(\sigma) \subseteq \mathrm{ftv}(\Gamma'; A'; E)$.

Aplicando la regla Hu tenemos que:

$$\Gamma' \vdash U^m\ t \rightsquigarrow A', E \cup \{A' \geq m\}$$

Ahora, $\sigma(A' \geq m) \equiv \sigma A' \geq m \equiv n \geq m$, lo cual es cierto porque es un requerimiento de la regla de tipado u. En consecuencia, σ satisface $E \cup \{A' \geq m\}$.

Caso $\dfrac{\Gamma \vdash t : n}{\Gamma \vdash \pi^m t : (m, n)}\ \mathtt{m}$

Por hipótesis de inducción: $\Gamma' \vdash t \rightsquigarrow A', E$ y $\exists \sigma / \sigma$ satisface las ecuaciones de $E, \sigma A' = n, \sigma\Gamma' = \Gamma, \mathrm{dom}(\sigma) \subseteq \mathrm{ftv}(\Gamma'; A'; E)$.

Aplicando la regla Hm tenemos que:

$$\Gamma' \vdash \pi^m t \rightsquigarrow (m, A'), E \cup \{A' \geq m\}$$

Notar que $\sigma(m, A') = (m, \sigma A') = (m, n)$. Ahora, $\sigma(A' \geq m) \equiv (\sigma A' \geq m) \equiv (n \geq m)$, lo cual es cierto porque es un requerimiento de la regla de tipado m. Por lo tanto, σ satisface $E \cup \{A' \geq m\}$.

Caso $\dfrac{\Gamma \vdash t : n \quad \Delta \vdash r : m}{\Gamma, \Delta \vdash t \otimes r : n + m}\ \otimes$

Por hipótesis de inducción tenemos que:

- $\Gamma' \vdash t \rightsquigarrow A', E$ y $\exists \sigma / \sigma$ satisface las ecuaciones de E y $\sigma\Gamma' = \Gamma, \sigma A' = n, \mathrm{dom}(\sigma) \subseteq \mathrm{ftv}(\Gamma'; A'; E)$.
- $\Delta' \vdash r \rightsquigarrow B', F.$ y $\exists \mu / \mu$ satisface las ecuaciones de F y $\mu\Delta' = \Delta, \mu B' = B, \mathrm{dom}(\mu) \subseteq \mathrm{ftv}(\Delta'; B'; F)$.

Aplicando la regla H$\otimes$:

$$\Gamma', \Delta' \vdash t \otimes r \rightsquigarrow C', E \cup F \cup \{C' =_{\mathbb{N}} A' + B'\}$$

donde $C' \notin \mathrm{ftv}(\Gamma'; \Delta'; E; F; A'; B')$.

Sea $\tau = [n + m / C'] \circ \mu \circ \sigma$ (composición de las sustituciones). Se tiene que:

- Aplicando el Lema 2.8, $\mathrm{dom}(\mu) \cap \mathrm{ftv}(E) = \emptyset$ y $C' \notin \mathrm{ftv}(E)$ por lo que $\tau E = ([n + m / C'] \circ \mu \circ \sigma)E = \sigma E$. De esta manera como σE se satisface, τE se satisface.

- Aplicando el Lema 2.8, $\mathrm{dom}(\sigma) \cap \mathrm{ftv}(F) = \emptyset$ por lo que $\sigma F = F$. Luego como $C' \notin \mathrm{ftv}(F)$ tenemos que $\tau F = \mu F$. De esta manera como μ satisface F, τ satisface F.
- $\tau(C' =_{\mathbb{N}} A' + B') \equiv (\tau C' =_{\mathbb{N}} \tau A' + \tau B') \equiv (n + m =_{\mathbb{N}} n + m)$. Por lo que τ satisface $\{C' =_{\mathbb{N}} A' + B'\}$
- Aplicando el Lema 2.8, $\mathrm{dom}(\sigma) \cap \mathrm{ftv}(\Delta') = \emptyset \wedge \mathrm{dom}(\mu) \cap \mathrm{ftv}(\Gamma') = \emptyset \wedge C' \notin \mathrm{ftv}(\Gamma'; \Delta') \implies \tau(\Gamma', \Delta') = \tau\Gamma', \tau\Delta' = \sigma\Gamma', \mu\Delta' = \Gamma, \Delta$.
- Como $C' \notin \mathrm{dom}(\mu) \cup \mathrm{dom}(\sigma)$, $\tau C' = [n + m/C']C' = n + m$.
- $\mathrm{dom}(\tau) = \mathrm{dom}(\sigma) \cup \mathrm{dom}(\mu) \cup \{C'\} \subseteq \mathrm{ftv}(\Gamma'; A'; E) \cup \mathrm{ftv}(\Delta'; B'; F) \cup \{C'\} = \mathrm{ftv}(\Gamma'; \Delta'; C'; E; F; \{C' =_{\mathbb{N}} A' + B'\})$.

Caso $\dfrac{}{\Gamma \vdash (b^m, \rho^n) : (m, n)}\ \mathtt{ax_{am}}$

Por $\mathtt{Hax_{am}}$, tenemos que $\Gamma \vdash (b^m, \rho^n) \rightsquigarrow (m, n), \emptyset$. Sea $\sigma = \emptyset$. Trivialmente se da que $\sigma(m, n) = (m, n)$ y $\sigma\Gamma = \Gamma$. A su vez $\mathrm{dom}(\sigma) \subseteq S\ \forall S$.

Caso $\dfrac{x : n \vdash t_0 : A \quad \ldots \quad x : n \vdash t_{2^m-1} : A \quad \Gamma \vdash r : (m, n)}{\Gamma \vdash \mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\} : A}\ \mathtt{lc}$

Por hipótesis de inducción tenemos que:

- $\Gamma' \vdash r \rightsquigarrow B', F$ y $\exists\sigma/\sigma$ satisface las ecuaciones de F y $\sigma\Gamma' = \Gamma, \sigma B' = (m, n), \mathrm{dom}(\sigma) \subseteq \mathrm{ftv}(\Gamma', B', F)$.
- $x : X'_i \vdash t_i \rightsquigarrow A'_i, E_i$ y $\exists\mu_i/\mu_i$ satisface las ecuaciones de E_i y $\mu_i X'_i = n, \mu_i A'_i = A, \mathrm{dom}(\mu_i) \subseteq \mathrm{ftv}(X'_i, A'_i, E_i) \quad \forall i \in [0, \ldots, 2^m - 1]$.

Como X'_i es solo una variable de tipo y $\mu_i X'_i = n\ \forall i$, podemos unificarlas en una sola: $X' \leftarrow X'_i\ \forall i$. Luego, aplicando la regla `Hlc`:

$$\Gamma' \vdash \mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\} \rightsquigarrow A'_0, G$$

donde $X' \notin \mathrm{ftv}(\Gamma'; B'; F)$ y $G = \bigcup \begin{cases} E_0, \ldots, E_{2^m-1} \\ F \\ \{A'_0 =_\rho A'_1\} \\ \quad \vdots \\ \{A'_0 =_\rho A'_{2^m-1}\} \\ \{B' =_\rho (m, X')\} \\ \{X' \geq m\} \end{cases}$.

Sea $\tau = \left(\bigcirc_{i=0}^{2^m-1} \mu_i\right) \circ \sigma$ (es decir τ aplica la sustitución σ, luego μ_{2^m-1}, μ_{2^m-2}, ... y así sucesivamente hasta aplicar μ_0). Primero veamos que τ satisface G:

- Para cada $E_j, j \in [0, \ldots, 2^m - 1]$ podemos aplicar el Lema 2.8 para saber que $\mathrm{dom}(\sigma) \cap \mathrm{ftv}(E_j) = \emptyset$. Entonces $\tau E_j = (\bigcirc_{i=0}^{2^m-1} \mu_i) E_j$. Ahora, aplicando nuevamente el Lema 2.8, $\mathrm{dom}(\mu_j) \cap \mathrm{ftv}(E_i) = \emptyset \ \forall i \neq j$. Por lo tanto, $\tau E_j = \mu_j E_j$. De esta manera como $\mu_j E_j$ se satisface, τE_j se satisface.
- Aplicando el Lema 2.8, $\mathrm{dom}(\mu_i) \cap \mathrm{ftv}(F) = \emptyset \ \forall i$ por lo que $\mu_i F = F$. En consecuencia, $\tau F = \sigma F$. De esta manera como σ satisface F, τ satisface F.
- Para cada $i \in [1, \ldots, 2^m - 1]$, $\tau(A'_0 =_\rho A'_i) \equiv (\tau A'_0 =_\rho \tau A'_i) \equiv (\mu_0 A'_0 =_\rho \mu_i A'_i) \equiv (A =_\rho A)$. Por lo que τ satisface $\{A'_0 =_\rho A'_i\}$.
- $\tau(B' =_\rho (m, X')) \equiv (\tau B' =_\rho (m, \tau X')) \equiv (\sigma B' =_\rho (m, \mu_{2^m-1} X')) \equiv ((m, n) =_\rho (m, n))$. Por lo que τ satisface $\{B' =_\rho (m, X')\}$.
- $\tau(X' \geq m) \equiv (\tau X' \geq m) \equiv (\mu_{2^m-1} X' \geq m) \equiv (n \geq m)$. Lo cual es cierto porque es un requerimiento para aplicar la regla de tipado `lc`. Por lo que τ satisface $\{X' \geq m\}$.

Y finalmente probamos el resto de las condiciones:

- Aplicando el Lema 2.8, $\mathrm{dom}(\mu_i) \cap \mathrm{ftv}(\Gamma') = \emptyset \ \forall i \implies \tau\Gamma' = \sigma\Gamma' = \Gamma$.
- Como $A'_0 \notin \mathrm{dom}(\sigma) \cup \mathrm{dom}(\mu_i) \ \forall i \geq 1$, $\tau A'_0 = \mu_0 A'_0 = A$.
- $$\begin{aligned} \mathrm{dom}(\tau) &= \bigcup_i^{2^m-1} \mathrm{dom}(\mu_i) \cup \mathrm{dom}(\sigma) \\ &\subseteq \bigcup_i^{2^m-1} \mathrm{ftv}(X'; A'_i; E_i) \cup \mathrm{ftv}(\Gamma'; B'; F) \\ &= \mathrm{ftv}(\Gamma'; A'_0; G). \end{aligned}$$

□

Observación. La demostración del Teorema 2.9 no solo prueba la existencia de la solución, sino que muestra además como construirla a partir del árbol de derivación del tipo.

2.3. Algoritmo de unificación de Robinson

El algoritmo de Robinson recibe las ecuaciones generadas por Hindley y las resuelve para producir una solución. La idea es resolver primero las ecuaciones basadas en $=_\rho$ y a continuación las que usen $=_\mathbb{N}$ y $\geq$. El algoritmo de Robinson se encuentra detallado en la Definición 2.11.

Lema 2.10. El algoritmo de Hindley no puede generar una ecuación de la forma $A = X$ o $X = A$, donde X ocurre en A y $A \neq X$. □

Definición 2.11 (Robinson para λ_ρ)**.** Dado el sistema de ecuaciones E para construir la solución σ, se debe aplicar de manera iterativa y mientras sea posible los siguientes pasos:

- Si una ecuación en el sistema es de la forma $A \multimap B =_\rho C \multimap D$, reemplazarla por las ecuaciones $A =_\rho C$ y $B =_\rho D$.
- Si una ecuación es de la forma $(m, A) =_\rho (m, B)$, reemplazarla por las ecuaciones $A =_\rho B$.
- Si una ecuación es de la forma $X = X$, remover esa ecuación.
- Si una ecuación es de la forma $V =_\rho X$ o $X =_\rho V$, donde X no ocurre en V, la X es sustituida por V en todas las ecuaciones.
- Dado $n, m, o \in \mathbb{N}$, si alguna ecuación es de la forma:
  - $n = A \multimap B$ o $A \multimap B = n$,
  - $(m, n) = A \multimap B$ o $A \multimap B = (m, n)$,
  - $n = (m, o)$,
  - $A + B = C$ o $A \geq B$, con algún $A, B, C \notin \mathbb{N}^+$,
  - $n = m$ o $(n, a) = (m, b)$, con $n \neq m$ o $a \neq b$;

  el algoritmo debe fallar.

Cuando no se puedan aplicar más reglas, quedan solo las ecuaciones de la forma $A + B =_\mathbb{N} C$ y $A \geq B$, cada uno de A, B, C o bien son naturales o variables de tipos por lo que pertenecen a $\mathbb{N}$. Por lo tanto, se puede resolver el sistema de ecuaciones con algún procedimiento de álgebra lineal. Si el sistema es indeterminado el algoritmo debe fallar.

Si el algoritmo termina sin fallar, entonces el sistema final es de la forma $X_0 = V_0, \ldots, X_{n-1} = V_{n-1}$, donde las X_i son variables diferentes y no ocurren en las

V_i. En este caso, la sustitución $\sigma = [V_0/X_0, \ldots, V_{n-1}/X_{n-1}]$ es una solución del sistema inicial.

La prueba de la correctitud y completitud de este algoritmo se deja como trabajo a futuro (ver la Sección 6.3.2).

Corolario 2.12. Es posible validar un término de λ_ρ aplicando el algoritmo modificado de Hindley para obtener un tipo con variables A y un conjunto de ecuaciones. El algoritmo de Robinson definido en 2.11 puede encontrar una solución σ a las ecuaciones. Si no existe solución el término no es válido. Utilizando el Teorema 2.7 con el entorno vacío, se puede obtener un tipo como resultado, validando el término.

2.4. Resolución del sistema de ecuaciones lineales

La Definición 2.11 no especifica qué algoritmo emplear para resolver el sistema de ecuaciones e inecuaciones lineales sobre enteros positivos generado en su última parte. La resolución de este sistema eficientemente no es simple, puesto que se requiere encontrar soluciones enteras.

Una forma fácil de resolver el sistema es usando un algoritmo de fuerza bruta que pruebe todas las posibles soluciones y verifique si satisfacen las ecuaciones. Esto es posible siempre y cuando las variables involucradas no sean de muchos qubits, puesto que en tal caso el tipado tardaría demasiado en ejecutarse debido a la inherente complejidad exponencial.

Una opción viable que existe es recurrir al algoritmo de Simplex, presentado en la Sección 1.5. A continuación, vemos cómo es posible aplicarlo a este problema.

2.4.1. Simplificación y aplicación de Simplex

Antes de poder aplicar Simplex es necesario adaptar el problema a un formato compatible con el mismo. Si el sistema generado por el algoritmo de Robinson genera m ecuaciones y p inecuaciones sobre n variables de tipo que son enteros positivos, podemos expresar tal sistema en formato matricial de la siguiente forma:

Encontrar x tal que:

$$\begin{aligned} Ax &= b \\ Cx &\geq d \\ x_i &\geq 1 \quad \forall i \end{aligned}$$

donde $x \in \mathbb{N}^n, b \in \mathbb{N}^m, d \in \mathbb{N}^p, A \in \mathbb{N}_0^{m \times n}, C \in \mathbb{N}_0^{p \times n}$.

En primer lugar, podemos resolver el sistema para $y_i \leftarrow x_i - 1$. $Ax = b \implies A(x-1) = b - A\overline{1}$. A su vez, $Cx \geq d \implies C(x-1) \geq d - C\overline{1}$. Haciendo $b' = b - A\overline{1},\ d' = d - C\overline{1}$ tenemos:

Encontrar y tal que:

$$\begin{aligned} Ay &= b' \\ Cy &\geq d' \\ y_i &\geq 0 \quad \forall i \end{aligned}$$

donde $y \in \mathbb{N}_0^n, b' \in \mathbb{N}_0^m, d' \in \mathbb{N}_0^p, A \in \mathbb{N}_0^{m \times n}, C \in \mathbb{N}_0^{p \times n}$.

Seguidamente, podemos reemplazar las inecuaciones por igualdades agregando p variables al sistema.

Para cada inecuación tenemos que $\sum_{j=0}^{n} C_{ij} y_j \geq d'_i$, entonces $\exists s_i \in \mathbb{N}_0^+ / \sum_{j=0}^{n} C_{ij} y_j - s_i = d'_i$.

Haciendo $A' = \begin{bmatrix} A & 0^{m \times p} \\ C & -I^{p \times p} \end{bmatrix}$, $b'' = (b' \ \ d')^T$ y $z = (y \ \ s)^T$. El problema se vuelve:

Encontrar z tal que:

$$\begin{aligned} A'z &= b'' \\ z_i &\geq 0 \quad \forall i \end{aligned}$$

donde $z \in \mathbb{N}_0^{n+p}, b'' \in \mathbb{N}_0^{m+p}, A' \in \mathbb{Z}^{(m+n) \times (n+p)}$.

Con este sistema finalmente podemos aplicar el algoritmo de Simplex de la Definición 1.32. Como el algoritmo va a maximizar $c^T x$, podemos tomar $c_i = -1 \ \ \forall i$, de esta manera minimizamos x y el algoritmo devuelve las asignaciones de tipos de menor tamaño.

Existe una diferencia, Simplex genera soluciones reales, no necesariamente enteras. Sabemos que si Simplex devuelve una solución entera entonces ya el algoritmo concluye. Si, en cambio, se tiene una solución no entera es necesario aplicar otra técnica conocida como Ramificación y Poda (del inglés "Branch and Bound"), presentada en la Sección 9.3 del libro [Win22]. Dado que no es común encontrarse con este caso, se deja como trabajo a futuro explorar e implementarlo (discutido en la Sección 6.3.3).

2.4.2. Equivalencia con el problema de cubrimiento de vértices mínimo

En la sección anterior vimos cómo más allá de encontrar un tipado, también puede resultar interesante la pregunta de cuál es el tipado que minimiza el tamaño total de los qubits. Por ejemplo, tiene mucho más sentido tipar la expresión $\lambda x.x$ como $1 \multimap 1$ que como $143 \multimap 143$. Extraordinariamente, se puede probar que bajo esta premisa el tipado se vuelve *NP-completo*. Para probar esta afirmación basta con ver que es equivalente a otro problema NP-hard, el de cubrimiento de vértices mínimo.

El siguiente lema nos sirve para poder introducir variables que no aparecen en el tipo y es útil para reducir la longitud de los árboles de derivaciones.

Lema 2.13. Si ninguna de las variables de un término t aparece en un contexto Δ:

$$\frac{\Gamma \vdash t \rightsquigarrow A, E}{\Delta, \Gamma \vdash t \rightsquigarrow A, E}\ \texttt{Hctx}$$

□

Teorema 2.14 (Complejidad del Tipado)**.** Tipar una expresión de λ_ρ minimizando la suma de los tamaños de los qubits es NP-completo.

Demostración. Dado un grafo $G = (V, E)$, vamos a construir una expresión de λ_ρ cuyo tipado mínimo sea equivalente a encontrar un cubrimiento de vértices mínimo de G en la forma dada por el Teorema 1.30. La idea de esta expresión es construir un letcase donde cada caso codifique la restricción 1.18 (del Teorema 1.30). Cada variable de tipo representa un vértice, el cual vale uno si no forma parte del cubrimiento o dos si lo es cubriendo la restricción 1.19. De esta manera minimizar la suma del tamaño de todas las variables equivale a la minimización 1.17.

Sea $e_0, \ldots, e_{|E|-1}$ un orden cualquiera del conjunto E y $v_0, \ldots, v_{|V|-1}$ un orden cualquiera del conjunto V. Necesitamos codificar al menos $|E|$ casos, por lo que necesitamos medir $m = \lceil \log_2 |E| \rceil$ qubits. Para el resto de los $2^m - |E|$ casos podemos usar una expresión que no genere ninguna ecuación.

Utilizando las definiciones auxiliares:

$$\lambda_{v\in V} = \lambda v_0.\ \lambda v_1.\ \lambda v_2.\ \ldots\ \lambda v_{n-2}.\ \lambda v_{n-1}$$

$$\multimap_{v\in V} T = V_0 \multimap V_1 \multimap V_2 \multimap \cdots \multimap V_{n-2} \multimap V_{n-1} \multimap T$$

$$cs_i = \lambda_{v\in V}.(\lambda y.\ |0\rangle\langle 0|)\ (\pi^3(v_p \otimes v_q)) \quad \text{siendo } e_i = pq$$

La expresión es:

$$\chi = \mathsf{letcase}\ x = \pi^m(|0\rangle\langle 0|)^m\ \mathsf{in}\ \{cs_0, \ldots, cs_{|E|-1}, \lambda_{v\in V}.\ |0\rangle\langle 0|, \ldots\} \tag{2.1}$$

A continuación, se muestra derivación del algoritmo de Hindley (Definición 2.4) para esta expresión. Comenzamos primero por sus subexpresiones:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : Y \vdash |0\rangle\langle 0| \rightsquigarrow 1, \emptyset}\ \mathtt{Hax}_\rho}{\vdash \lambda y.\ |0\rangle\langle 0| \rightsquigarrow Y \multimap 1, \emptyset}\ \mathrm{H}\multimap_i \quad \dfrac{\dfrac{}{v_p : V_p \vdash v_p \rightsquigarrow V_p, \emptyset}\ \mathtt{Hax} \quad \dfrac{}{v_q : V_q \vdash v_q \rightsquigarrow V_q, \emptyset}\ \mathtt{Hax}}{v_p : V_p, v_q : V_q \vdash v_p \otimes v_q \rightsquigarrow S_{pq}, \{S_{pq} =_{\mathbb{N}} V_p + V_q\}}\ \mathrm{H}\otimes}{v_p : V_p, v_q : V_q \vdash \pi^3(v_p \otimes v_q) \rightsquigarrow (3, S_{pq}), \{S_{pq} =_{\mathbb{N}} V_p + V_q, S_{pq} \geq 3\}}\ \mathtt{Hm}}{v_i : V_i\ \forall i \vdash (\lambda y.\ |0\rangle\langle 0|)\ (\pi^3(v_p \otimes v_q)) \rightsquigarrow 1, \{S_{pq} =_{\mathbb{N}} V_p + V_q, S_{pq} \geq 3, Y \multimap 1 =_\rho (3, S_{pq}) \multimap 1\}}\ \mathrm{H}\multimap_e}{\vdots}\ \mathrm{H}\multimap_i$$

$$\dfrac{\dfrac{\dfrac{}{v_i : V_i\ \forall i \vdash |0\rangle\langle 0| \rightsquigarrow 1, \emptyset}\ \mathtt{Hax}}{\vdots}\ \mathrm{H}\multimap_i}{x : X \vdash \lambda_{v\in V}.(\lambda y.\ |0\rangle\langle 0|)\ (\pi^3(v_p \otimes v_q)) \rightsquigarrow \multimap_{v\in V} 1, \{S_{pq} =_{\mathbb{N}} V_p + V_q, S_{pq} \geq 3, Y \multimap 1 =_\rho (3, S_{pq}) \multimap 1\}}\ \mathtt{Hctx} \tag{2.2}$$

$$\dfrac{\dfrac{\dfrac{}{v_i : V_i\ \forall i \vdash |0\rangle\langle 0| \rightsquigarrow 1, \emptyset}\ \mathtt{Hax}}{\vdots}\ \mathrm{H}\multimap_i}{x : X \vdash \lambda_{v\in V}.\ |0\rangle\langle 0| \rightsquigarrow \multimap_{v\in V} 1, \emptyset}\ \mathtt{Hctx} \tag{2.3}$$

$$\dfrac{\dfrac{}{\vdash (|0\rangle\langle 0|)^m \rightsquigarrow m, \emptyset}\ \mathtt{Hax}_\rho}{\vdash \pi^m(|0\rangle\langle 0|)^m \rightsquigarrow (m, m), \{m \geq m\}}\ \mathtt{Hm} \tag{2.4}$$

Finalmente, utilizando 2.2, 2.3 y 2.4 podemos concluir:

$$\frac{\vdots}{\vdash \mathsf{letcase}\ x = \pi^m(|0\rangle\langle 0|)^m\ \mathsf{in}\ \{cs_0, \ldots, cs_{|E|-1}, \lambda_{v\in V}.\, |0\rangle\langle 0|, \ldots\} \rightsquigarrow \multimap_{v\in V} 1,\ \bigcup \begin{cases} \{S_{pq} =_{\mathbb{N}} V_p + V_q, S_{pq} \geq 3, Y \multimap 1 =_\rho (3, S_{pq}) \multimap 1\}\ \forall pq \in E \\ \{\multimap_{v\in V} 1 =_\rho \multimap_{v\in V} 1\}, \\ \{m \geq m\}, \\ \{(m,m) =_\rho (m, X)\}, \\ \{X \geq m\} \end{cases}} \mathtt{Hlc} \tag{2.5}$$

Si fijamos $X = m, Y = (3, S_{pq})$, las únicas ecuaciones no triviales que quedan definen el siguiente sistema:

$$\begin{aligned} V_p + V_q &\geq 3 \quad \forall pq \in E \\ V_i &\geq 1 \quad \forall i \in [0, \ldots, |V|] \end{aligned}$$

Haciendo $x_i \leftarrow V_i - 1$

$$\begin{aligned} x_p + x_q &\geq 1 \quad \forall pq \in E \\ x_v &\geq 0 \quad \forall v \in V \end{aligned}$$

Si a su vez pedimos que la suma de los valores de x_v sean los menores posibles, ningún x_v va a ser mayor a uno nunca, y como es un natural se cumple que $x_v \in \{0, 1\}$ y el sistema es:

$$\begin{aligned} \text{minimizar} \sum_{v\in V} x_v & \\ \text{sujeto a } x_p + x_q &\geq 1 \quad \forall pq \in E \\ x_v &\in \{0, 1\} \quad \forall v \in V \end{aligned} \tag{2.6}$$

Que es igual a la versión de programación lineal del problema de cubrimiento de vértices mínimo presentada en el Teorema 1.30. De esta manera, si existe un algoritmo que encuentra el tipo que minimiza los tamaños de los qubits para la expresión $\chi(2.1)$, por el Teorema 2.9 de la completitud de Hindley, tal tipo define una solución al sistema 2.6, que es un problema NP-hard (y NP-completo). Estas observaciones también aplican para las versiones decidibles de los problemas. A su vez, dada una solución es trivial verificar en tiempo lineal que las ecuaciones se cumplen, por lo que el problema es NP-completo. □

Capítulo 3

Qiskit con Python

Qiskit [Dev21; Asf+21] es un kit de desarrollo de software libre fundado por IBM (especificado en [McK+18]) inicialmente orientado para su servicio de nube de computación cuántica. Provee herramientas para crear y manipular programas cuánticos, permitiendo simularlos localmente y ejecutarlos en dispositivos cuánticos. Puede ser utilizado potencialmente por cualquier hardware cuántico de propósito general.

La versión primaria de Qiskit es una biblioteca disponible para el lenguaje de programación Python. En este trabajo se usará para definir y ejecutar circuitos en términos de compuertas.

3.1. Arquitectura

Qiskit permite definir circuitos cuánticos dentro de Python. Una vez declarados es posible enviar estos circuitos a máquinas cuánticas (o simuladores) que devolverán el resultado de las mediciones realizadas. En este sentido Python es nuestro paradigma clásico y por medio de Qiskit podemos acceder a la parte cuántica. Por estas razones el modelo presentado en este trabajo consiste en dos partes, el lenguaje clásico de Python y la máquina abstracta de Qiskit. Debido a la complejidad de ambos sistemas en este trabajo se presentan versiones sumamente simplificadas con el fin de probar la correctitud de la traducción.

3.2. La máquina abstracta de Qiskit

Un programa en Qiskit es una lista de instrucciones a ejecutar sobre una lista de qubits que representan la memoria de la máquina cuántica. Contiene también un registro para almacenar el resultado de la medición. Formalmente, un estado del programa está representado por una tripleta $(|\phi\rangle, r, L) \in \Lambda_{qk}$, donde:

- Λ_{qk} es el conjunto de estados de Qiskit.
- $|\phi\rangle$ es un vector normalizado de $\bigotimes_{i=0}^{n-1}\mathbb{C}^2$ para algún $n > 0$.
- $r \in \mathbb{N}_0$ es el resultado de la última medición efectuada.
- L es una lista ordenada de instrucciones ($\subseteq \mathfrak{I}_{qk}$). Las instrucciones se muestran en la Definición 3.1.

Definición 3.1 (Instrucciones de Qiskit, $\mathfrak{I}_Q$)**.**

Init $[c_0, \ldots, c_{2^n-1}]$	Inicializa el estado cuántico al vector estado provisto.
Gate G $[q_0, \ldots, q_{n-1}]$	Aplica la compuerta G sobre los qubits $q_0, \ldots, q_{n-1}$.
Measure $[q_0, \ldots, q_{n-1}]$	Mide los qubits $q_0, \ldots, q_{n-1}$.

donde:

- $G \in \{U^{\theta,\phi,\lambda}, CU^{\theta,\phi,\lambda}, \text{SWAP}, \text{CSWAP}\}$, con $\theta, \phi, \lambda \in \mathbb{R}$.

3.3. Semántica operacional de Qiskit

La máquina abstracta ejecuta las instrucciones en orden, modificando su estado como se describe a continuación. La definición de cada compuerta es análoga a la provista en la Definición 4.2. Cabe destacar que Qiskit inicializa todos los qubits a $|0\rangle$ al inicio de toda ejecución. Definimos formalmente la ejecución a través de la función $\overset{qk}{\to}$: $\Lambda_{qk} \to \Lambda_{qk}$.

Definición 3.2 (Semántica operacional de Qiskit)**.**

Si $|\phi'\rangle = \left[\begin{smallmatrix} c_0 & \cdots & c_{2^n-1} \end{smallmatrix}\right]^*$:

$$(|\phi\rangle, r, \textbf{Init } [c_0, \ldots, c_{2^n-1}]; L) \overset{qk}{\to}_1 (|\phi'\rangle, r, L)$$

Si G' es la compuerta que aplica G a los qubits $q_0, \ldots, q_{n-1}$:

$$(|\phi\rangle, r, \textbf{Gate } G\ [q_0, \ldots, q_{n-1}]; L) \overset{qk}{\to}_1 (G'\,|\phi\rangle, r, L)$$

Si $\{\pi_0, \ldots, \pi_{2^n-1}\}$ son los operadores que miden los qubits $q_0, \ldots, q_{n-1}$:

$$(|\phi\rangle, r, \textbf{Measure } [q_0, \ldots, q_{n-1}]; L) \xrightarrow{qk}_{p_i} \left(\frac{\pi_i |\phi\rangle}{\sqrt{\langle\phi| \pi_i^\dagger \pi_i |\phi\rangle}}, i, L \right)$$

con $p_i = \langle\phi| \pi_i^\dagger \pi_i |\phi\rangle$

3.4. Python

Python [Fou21] es un lenguaje interpretado de alto nivel. Actualmente, se encuentra entre los más utilizados del mundo [Kuh11]. Su filosofía de diseño se centra en la legibilidad del código, y se caracteriza por ser dinámicamente tipado, con recolector de basura. Su paradigma es procedimental, orientado a objetos y con programación funcional.

Debido a la alta complejidad del lenguaje está fuera del alcance de este trabajo especificarlo en su totalidad. A su vez dicha especificación haría muy difícil probar cualquier propiedad sobre el mismo. En su lugar se define a continuación una versión reducida que contiene las características que resultan necesarias para este proyecto. Estas definiciones son esenciales para definir de manera concisa la traducción que se presentará en el capítulo 4 y probar propiedades sobre la misma.

3.4.1. Modelo de la biblioteca de Qiskit

Para modelar la biblioteca de Qiskit en Python se diseñó un wrapper 'Circuit' que simplifica lo más posible su interfaz. Las variables internas del wrapper se modelan como un par (I, L) donde I es una lista de números complejos representando el vector de estado inicial del circuito y L es la lista de instrucciones a ejecutar. De esta forma los métodos del wrapper son:

`Circuit(I, L)` Simplemente inicializa el estado interno a (I, L).

`gate` agrega una instrucción para aplicar una compuerta a la lista de instrucciones.

`compose` dado otro circuito devuelve la composición de ambos. El nuevo circuito generado contiene la concatenación de las listas de instrucciones de los circuitos iniciales.

`measure` utiliza el simulador/computadora cuántica de Qiskit para ejecutar la lista de instrucciones y devuelve el resultado de la medición.

3.4.2. Definición de Python

Nuestra simplificación contiene:

- Las funciones de lambda de Python, que se asemejan mucho a las expresiones lambda de λ_ρ, por lo que su traducción es casi directa. En nuestra simplificación, la diferencia es meramente sintáctica.
- La clase 'Circuit' y sus métodos explicados en la Sección 3.4.1.
- La expresión de `letcase`, que simplemente es un azúcar sintáctico para la indexación de listas.
- Utiliza $\mathbb{N}$, $\mathbb{R}$, $\mathbb{C}$ para los tipos `int`, `float` y `complex` respectivamente. Esta simplificación es necesaria puesto que modelar la representación interna de los números volvería la traducción sumamente compleja y no es relevante para el presente trabajo.

La especificación formal de los términos de Python (Λ_{py}) se muestra en la Definición 3.3.

Notar que no se va a dar un tipado para Python porque la traducción que se presenta en el Capítulo 4 ya valida el tipado de la expresión de λ_ρ. Resulta entonces innecesario volverla a tipar luego de haberla traducido.

Definición 3.3 (Gramática de Python)**.**

$t ::= v$
| `t(t)` (Aplicación de lambda)
| `t.gate(G, [r, ..., r])` (Aplicación de compuertas)
| `t.measure([n, ..., n])` (Medición de circuitos)
| `t.compose(t)` (Composición de circuitos)
| `t.size()` (Cantidad de qubits del circuito)
| `letcase(t, [t, ..., t])` (Letcase)
| `(t, t)` (Pares)
| `t+t` | `t-t` | `t*t` | `t/t` | `t**t` (Aritmética)

$v ::= b$ | `n` | `Circuit([c, ..., c], [l, ..., l])` (Valores)

$b ::=$ `x` | `lambda x: t` (Términos base)

donde

- $\mathtt{l} \in \mathfrak{I}_{qk}, \mathtt{n} \in \mathbb{N}_0, \mathtt{c} \in \mathbb{C}, \mathtt{r} \in \mathbb{R}$.
- $\mathtt{G} \in \{U^{\theta,\phi,\lambda}, UC^{\theta,\phi,\lambda}, \text{SWAP}, \text{CCNOT}, \text{CSWAP}\}$, con $\theta, \phi, \lambda \in \mathbb{R}$.

3.4.3. Estrategia de reducción de Python

El mecanismo básico de la reducción es que dado una expresión del tipo 'Circuit().compuerta()', la misma se reducirá a un 'Circuit()', donde el estado interno de la clase 'Circuit' fue modificado para agregar dicha compuerta. Las mediciones sobre un circuito devuelven un entero.

Para la composición de circuitos notar que además de concatenar las listas de instrucciones se tienen que incrementar las referencias a los qubits del segundo circuito, para ello se define la función *shift*.

El resto de las reglas son similares a la del lambda cálculo afín.

Definición 3.4 (Estrategia de reducción de Python)**.**

$$\texttt{Circuit(I, [l}_0\texttt{, ..., l}_{x-1}\texttt{]).gate(G, [a}_0\texttt{, ..., a}_{g-1}\texttt{])} \xrightarrow{py}_1 \texttt{Circuit(I, [l}_0\texttt{, ..., l}_{x-1}\texttt{, Gate G [a}_0\texttt{, ..., a}_{g-1}\texttt{]])} \quad (G_{py})$$

Si $|\mathtt{J}| = 2^m$, es decir J es un vector de estado de m qubits:

$$\texttt{Circuit(I, [l}_0\texttt{, ..., l}_{x-1}\texttt{]).compose(Circuit(J, [c}_0\texttt{, ..., c}_{y-1}\texttt{]))} \xrightarrow{py}_1 \texttt{Circuit(I} \otimes \texttt{J, [}\mathit{shift}\texttt{(l}_0\texttt{, m), ..., }\mathit{shift}\texttt{(l}_{x-1}\texttt{, m), c}_0\texttt{, ..., c}_{y-1}\texttt{])} \quad (\otimes_{py})$$

$$\texttt{Circuit(J, [...]).size()} \xrightarrow{py}_1 \log_2 |\mathtt{J}| = \mathtt{m} \quad (\text{sz}_{py})$$

Si $(|0\rangle^n, 0, \textbf{Init } \mathtt{I}; \mathtt{l}_0; \ldots; \mathtt{l}_{x-1}; \textbf{Measure } [\mathtt{q}_0, \ldots, \mathtt{q}_s]) \xrightarrow{qk}{}^*_p (|\phi\rangle, r, [\,])$: [1]

$$\texttt{Circuit(I, [l}_0\texttt{, ..., l}_{x-1}\texttt{]).measure([q}_0\texttt{, ..., q}_s\texttt{])} \xrightarrow{py}_p r \quad (\pi_{py})$$

$$\texttt{(lambda x.t)(v)} \xrightarrow{py}_1 [\mathtt{v}/\mathtt{x}]\mathtt{t}, \text{ donde v es un valor} \quad (\lambda_{py})$$

$$\texttt{letcase(((i, m), rho), [t}_0\texttt{, ..., t}_{m-1}\texttt{])} \xrightarrow{py}_1 \mathtt{t}_i\texttt{(rho)}, \text{ con } 0 \leq i < m \quad (\text{let}_{py})$$

$$\frac{\texttt{t} \overset{py}{\to}_p \texttt{r}}{\texttt{t(s)} \overset{py}{\to}_p \texttt{r(s)}}\ Lr_{py} \qquad \frac{\texttt{t} \overset{py}{\to}_p \texttt{r} \quad v \text{ es un valor}}{\texttt{v(t)} \overset{py}{\to}_p \texttt{v(r)}}\ Rr_{py}$$

$$\frac{\texttt{t} \overset{py}{\to}_p \texttt{r}}{\texttt{t.gate(G, A)} \overset{py}{\to}_p \texttt{r.gate(G, A)}}\ Gr_{py}$$

$$\frac{\texttt{t} \overset{py}{\to}_p \texttt{r}}{\texttt{t.measure(A)} \overset{py}{\to}_p \texttt{r.measure(A)}}\ \pi r_{py}$$

$$\frac{\texttt{t} \overset{py}{\to}_p \texttt{r}}{\texttt{t.compose(s)} \overset{py}{\to}_p \texttt{r.compose(s)}}\ L{\otimes}_{py}$$

$$\frac{\texttt{t} \overset{py}{\to}_p \texttt{r} \quad \texttt{v} \text{ es un valor}}{\texttt{v.compose(t)} \overset{py}{\to}_p \texttt{v.compose(r)}}\ R{\otimes}_{py}$$

$$\frac{t \overset{\lambda_\rho}{\to}_p r}{\texttt{letcase(t, TC)} \overset{\lambda_\rho}{\to}_p \texttt{letcase(r, TC)}}\ \text{letr}_{py}$$

$$\frac{\texttt{t} \overset{py}{\to}_p \texttt{r}}{\texttt{t} \oplus \texttt{s} \overset{py}{\to}_p \texttt{r} \oplus \texttt{s}}\ L{\oplus}_{py} \qquad \frac{\texttt{t} \overset{py}{\to}_p \texttt{r} \quad v \text{ es un valor}}{\texttt{v} \oplus \texttt{t} \overset{py}{\to}_p \texttt{v} \oplus \texttt{r}}\ R{\oplus}_{py}$$

$\texttt{a + b} \overset{py}{\to}_1 (a+b) \quad (+_{py})$ $\qquad$ $\texttt{a / d} \overset{py}{\to}_1 \lceil {}^{a}/_{d} \rceil \quad ({}/_{py})$

$\texttt{a - b} \overset{py}{\to}_1 (a-b) \quad (-_{py})$ $\qquad$ $\texttt{a ** b} \overset{py}{\to}_1 (a^c) \quad (**_{py})$

$\texttt{a * b} \overset{py}{\to}_1 (a*b) \quad (*_{py})$

donde:

- $\oplus \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{**}\}$.
- $a, b, d \in \mathbb{C}, d \neq 0$.
- La función *shift*($\cdot$, n) hace que todos los operadores actúen n posiciones a la derecha. Formalmente:

$$\textit{shift}(\textbf{Gate}\ G\ [a_0, \ldots, a_{m-1}], n) = \textbf{Gate}\ G\ [a_0 + n, \ldots, a_{m-1} + n]$$
$$\textit{shift}(\textbf{Measure}\ [a_0, \ldots, a_{m-1}], n) = \textbf{Measure}\ [a_0 + n, \ldots, a_{m-1} + n]$$

[1] es decir si ejecutar y medir el circuito Q en Qiskit devuelve la medición r con probabilidad p.

Capítulo 4

Traducción de λ_ρ^* a Qiskit

En la Sección 1.7 se presentó λ_ρ tal cual fue definido en [Día17]. En este capítulo primero veremos que resulta imposible hacer una traducción de este lenguaje y proponemos alteraciones mínimas a este, definiendo λ_ρ^*, de manera de que la traducción sea posible.

Luego vamos a presentar varios resultados respecto al método de la purificación, que resulta esencial para la traducción. Utilizando esta técnica definimos formalmente la traducción de λ_ρ^* a Python.

Finalmente, se demuestra su correctitud y la existencia de la retracción por izquierda.

4.1. Limitaciones de Qiskit e introducción de λ_ρ^*

El paradigma que nos presenta Python con Qiskit consiste en construir circuitos cuánticos dentro de Python (computadora clásica). Estos circuitos son compilados y encapsulados dentro de un "trabajo" (del inglés *job*), el cual es enviado a algún computador cuántico que los ejecuta, devolviendo los resultados clásicos. Dado que λ_ρ posee tanto lógica clásica, una traducción debe elegir entre implementar la lógica clásica dentro del trabajo o fuera de ella:

- Implementar la lógica clásica dentro de la **computadora cuántica**. Esto necesita que los trabajos generados tengan instrucciones condicionales o de salto. Desafortunadamente, Qiskit no soporta tales instrucciones aún por lo que se deja como trabajo a futuro (ver Sección 6.3.1).
- Implementar la lógica clásica dentro de la **computadora clásica**. Este es el enfoque elegido. Pero existe un inconveniente, en el tiempo que demora la transmisión de las mediciones a la máquina clásica y la preparación de otro

trabajo, los qubits pierden coherencia, es decir el estado cuántico se pierde. Bajo esta premisa resulta imposible implementar el lenguaje de λ_ρ tal cual es propuesto.

Nota: Qiskit puede devolver los qubits luego de la medición lo cual permitiría hacer la traducción completa, sin embargo, esto solo es posible en el caso que se lo configure para hacer una simulación y no con una computadora cuántica real.

4.1.1. Modificación del operador de medición

En relación con lo mencionado anteriormente, es necesario hacer una modificación al lenguaje definido por Díaz-Caro [Día17]. Originalmente la expresión $\pi^m \rho^n$ retorna el resultado de medir los primeros m qubits de ρ^n y la matriz de densidad resultante luego de realizar la medición. Como efectivamente el estado cuántico se pierde, vamos a modificar la medición para que mida siempre todos los qubits del estado. Tal modificación se puede apreciar en la siguiente regla de reducción:

$$\pi^m\ \rho^n \xrightarrow{\lambda_\rho}_{p_i} (\lceil i/2^{n-m} \rceil, \rho_i^n) \quad \text{con} \quad \begin{cases} p_i = \mathsf{tr}({\pi^n}_i^\dagger\ \pi_i^n\ \rho^n) \\ \rho_i^n = (\pi_i^n\ \rho^n\ {\pi^n}_i^\dagger)/p_i \end{cases} \qquad (\pi_\rho)$$

Notar como las matrices de medición utilizadas no son $\overline{\pi_i^m}$, sino π_i^n. A su vez el entero devuelto por la medición representa los m bits más significativos de la medición. De forma que $0 \leq \lceil i/2^{n-m} \rceil < 2^m$. Esta modificación no genera cambios en la gramática ni el tipado.

4.1.2. Especificación de compuertas de λ_ρ^*

λ_ρ permite definir cualquier compuerta unitaria, sin embargo, Qiskit tiene un conjunto más reducido de compuertas disponibles. Para modelar esto se va a remplazar la regla $U^n t$ por $G\ t$, donde G se especifica en la Definición 4.1. En general las compuertas G pueden recibir algunos parámetros: θ, ϕ, λ y p, siendo p el qubit menos significativo sobre el cual actúan. A continuación, se muestra la gramática de λ_ρ^*, que define los términos de ella (Λ_ρ^*).

Observación. Si bien estas compuertas son una selección de todas las compuertas soportadas por Qiskit, es posible extender la implementación con más compuertas fácilmente.

Definición 4.1 (Gramática de λ^*_ρ)**.**

$$t ::= G\ t \mid \dots$$

$$\begin{aligned} G ::= & \ U_p^{\theta,\phi,\lambda} && \text{(Compuerta unaria general)} \\ \mid & \ CU_p^{\theta,\phi,\lambda} && \text{(Compuerta unaria general controlada)} \\ \mid & \ \mathrm{SWAP}_p && \text{(Swap)} \\ \mid & \ \mathrm{CSWAP}_p && \text{(Compuerta de Fredkin)} \end{aligned}$$

donde $\theta, \phi, \lambda \in \mathbb{R}$ y $p \in \mathbb{N}_0$.

Definición 4.2 (Matrices de λ^*_ρ)**.** Las matrices presentadas se definen en su forma matricial de la siguiente manera:

$$G_p = I^p \otimes G_0$$

$$U_0^{\theta,\phi,\lambda} = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{bmatrix}$$

$$CU_0^{\theta,\phi,\lambda} = I^1 \otimes |0\rangle\langle 0| + U_0 \otimes |1\rangle\langle 1| = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta/2) & 0 & -e^{i\lambda}\sin(\theta/2) \\ 0 & 0 & 1 & 0 \\ 0 & e^{i\phi}\sin(\theta/2) & 0 & e^{i(\phi+\lambda)}\cos(\theta/2) \end{bmatrix}$$

$$\mathrm{SWAP}_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathrm{CSWAP}_0 = I^2 \otimes |0\rangle\langle 0| + \mathrm{SWAP}_0 \otimes |1\rangle\langle 1| = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

donde I^n es el operador identidad de n qubits.

Definición 4.3 ($|G|$)**.** A su vez dado un operador G, usamos $|G|$ para denotar la cantidad de qubits sobre los que opera. Formalmente, $|G| = N \iff G$ es de dimensión $2^N \times 2^N$.

4.1.3. Estrategia de reducción de λ^*_ρ

En la publicación original [Día17], no se da una estrategia en particular para la reducción por lo que se eligió una al estilo *Call-by-value* puesto que simplifica la traducción. Esta elección se ve reflejada en la Definición 4.5 donde algunas reglas aplican solamente si hay un valor involucrado. Vamos a definir formalmente el concepto de valor y presentar la estrategia de reducción, teniendo en cuenta las modificaciones mencionadas anteriormente.

Definición 4.4 (Valor)**.** Una expresión v de un lenguaje es un valor si y solo si $\nexists\, r / v \to r$, donde $\to$ es la relación de reducción de dicho lenguaje.

Observación. Sea $t, r \in \Lambda^*_\rho$. Si $t \xrightarrow{\lambda_\rho}_p r \implies t$ no es un valor.

Definición 4.5 (Estrategia de reducción de λ^*_ρ)**.**

$$(\lambda x.t)\ v \xrightarrow{\lambda_\rho}_1 [v/x]t, \text{ donde } v \text{ es un valor} \qquad (\lambda_\rho)$$

$$G\ \rho^n \xrightarrow{\lambda_\rho}_1 \rho'^n \qquad \text{con } \rho'^n = \overline{G}\ \rho^n\ \overline{G}^\dagger \qquad (G_\rho)$$

$$\pi^m\ \rho^n \xrightarrow{\lambda_\rho}_{p_i} (\lceil i/2^{n-m} \rceil, \rho^n_i) \quad \text{con} \begin{cases} p_i = \mathsf{tr}({\pi^n}_i^\dagger\ \pi^n_i\ \rho^n) \\ \rho^n_i = (\pi^n_i\ \rho^n\ {\pi^n}_i^\dagger)/p_i \end{cases} \qquad (\pi_\rho)$$

$$\rho \otimes \rho' \xrightarrow{\lambda_\rho}_1 \rho'' \qquad \text{con } \rho'' = \rho \otimes \rho' \qquad (\otimes_\rho)$$

$$\mathsf{letcase}\ x = (b^m, \rho^n)\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\} \xrightarrow{\lambda_\rho}_1 [\rho^n/x]t_{b^m} \qquad (\mathrm{let}_\rho)$$

$$\frac{t \xrightarrow{\lambda_\rho}_p r}{t\ s \xrightarrow{\lambda_\rho}_p r\ s}\ Lr_\rho \qquad \frac{t \xrightarrow{\lambda_\rho}_p r \quad v \text{ es un valor}}{v\ t \xrightarrow{\lambda_\rho}_p v\ r}\ Rr_\rho \qquad \frac{t \xrightarrow{\lambda_\rho}_p r}{G\ t \xrightarrow{\lambda_\rho}_p G\ r}\ Gr_\rho$$

$$\frac{t \xrightarrow{\lambda_\rho}_p r}{\pi^n\ t \xrightarrow{\lambda_\rho}_p \pi^n\ r}\ \pi r_\rho \qquad \frac{t \xrightarrow{\lambda_\rho}_p r}{t \otimes s \xrightarrow{\lambda_\rho}_p r \otimes s}\ L{\otimes_\rho} \qquad \frac{t \xrightarrow{\lambda_\rho}_p r \quad v \text{ es un valor}}{v \otimes t \xrightarrow{\lambda_\rho}_p v \otimes r}\ R{\otimes_\rho}$$

$$\frac{t \xrightarrow{\lambda_\rho}_p r}{\mathsf{letcase}\ x = t\ \text{in}\ \{s_0, \ldots, s_n\} \xrightarrow{\lambda_\rho}_p \mathsf{letcase}\ x = r\ \text{in}\ \{s_0, \ldots, s_n\}}\ \text{letr}_\rho$$

4.2. Purificación del estado mixto

Una de las diferencias fundamentales entre λ_ρ y Qiskit es que el primero representa el estado cuántico utilizando matrices de densidad con estados mixtos (matrices de $2^n \times 2^n$ denotados ρ^n). Qiskit se maneja únicamente con vectores de estados puros (vectores de 2^n denotados $|\phi\rangle$), por lo cual una conversión es necesaria. Para ello recurrimos a la técnica de purificación presentada en la Definición 1.28. Vamos a probar primero como esta purificación clásica ($\text{pur}(\rho^n) \to |\phi\rangle$) tiene la ventaja de que preserva la aplicación de compuertas y las mediciones, pero no así la composición. Para solventarlo, presentamos una novedosa alteración que preserva las tres operaciones.

4.2.1. Propiedades de la purificación clásica

Teorema 4.6 (Inversa de la purificación)**.** La inversa de la purificación está dada por:

$$\text{pur}^{-1}(|\phi\rangle) = \mathsf{tr}_B(|\phi\rangle \langle\phi|)$$

Demostración. Sea ρ^n una matriz de densidad y $\sum_i^{2^n} \lambda_i |v_i\rangle \langle v_i|$ su descomposición espectral. Se demuestra que $\text{pur}^{-1}(\text{pur}(\rho^n)) = \rho^n$.

$$
\begin{aligned}
\mathrm{pur}^{-1}(\mathrm{pur}(\rho^n)) &= \mathrm{pur}^{-1}(\sum_i^{2^n} \sqrt{\lambda_i}\,|v_i\rangle \otimes |e_i\rangle) \\
&= \mathsf{tr}_B \left(|\sum_i^{2^n} \sqrt{\lambda_i}\,|v_i\rangle \otimes |e_i\rangle\rangle \langle \sum_i^{2^n} \sqrt{\lambda_i}\,|v_i\rangle \otimes |e_i\rangle| \right) \\
&= \mathsf{tr}_B \left(\sum_i^{2^n} \sum_j^{2^n} \sqrt{\lambda_i}\sqrt{\lambda_j}\,|v_i\rangle \langle v_j| \otimes |e_i\rangle \langle e_j| \right) \\
&= \sum_i^{2^n} \sum_j^{2^n} \sqrt{\lambda_i}\sqrt{\lambda_j} \mathsf{tr}_B \left(|v_i\rangle \langle v_j| \otimes |e_i\rangle \langle e_j| \right) \\
&= \sum_i^{2^n} \sum_j^{2^n} \sqrt{\lambda_i}\sqrt{\lambda_j}\,\langle e_i|e_j\rangle\,|v_i\rangle \langle v_j| \\
&= \sum_i^{2^n} \sum_j^{2^n} \sqrt{\lambda_i}\sqrt{\lambda_j}\,\delta_{i,j}\,|v_i\rangle \langle v_j| \\
&= \sum_i^{2^n} \lambda_i\,|v_i\rangle \langle v_j| = \rho^n
\end{aligned}
$$

□

Teorema 4.7 (Evolución de la purificación)**.** Sea ρ^n una matriz de densidad y G una compuerta de dimensión $|G| = m$.

$$\mathrm{pur}(\overline{G}\rho^n\overline{G}^\dagger) = (\overline{G} \otimes I^n)\ \mathrm{pur}(\rho^n) \tag{4.1}$$

Demostración. Siendo $|w_i\rangle = \overline{G}\ \ |v_i\rangle$

$$
\begin{aligned}
\mathrm{pur}(\overline{G}\rho^n\overline{G}^\dagger) &= \mathrm{pur}\left(\overline{G}(\sum_i^{2^n} \lambda_i\,|v_i\rangle \langle v_i|)\overline{G}^\dagger \right) = \mathrm{pur}\left(\sum_i^{2^n} \lambda_i \overline{G}\,|v_i\rangle \langle v_i|\,\overline{G}^\dagger \right) \\
&= \mathrm{pur}\left(\sum_i^{2^n} \lambda_i\,|w_i\rangle \langle w_i| \right) = \sum_i^{2^n} \sqrt{\lambda_i}\,|w_i\rangle \otimes |e_i\rangle \\
&= \sum_i^{2^n} \sqrt{\lambda_i}(\overline{G}\ \ |v_i\rangle) \otimes (I^n\ \ |e_i\rangle) = \sum_i^{2^n} \sqrt{\lambda_i}(\overline{G} \otimes I^n)\ (|v_i\rangle \otimes |e_i\rangle) \\
&= (\overline{G} \otimes I^n)\ \left(\sum_i^{2^n} \sqrt{\lambda_i}\,|v_i\rangle \otimes |e_i\rangle \right) = (\overline{G} \otimes I^n)\ \mathrm{pur}(\rho^n)
\end{aligned}
$$

□

Teorema 4.8 (Medición de la purificación)**.** Sea $\phi = \text{pur}(\rho^n)$. Medir n qubits de ρ^n usando los operadores de medición de la base canónica $\pi = \{\pi_0, \ldots, \pi_{2^n-1}\} = \{|e_i\rangle \langle e_i| , 0 \leq i < 2^n\}$ devuelve los mismos resultados que medir el estado purificado ϕ con los operadores $\pi' = \{\pi_0 \otimes I^n, \ldots, \pi_{2^n-1} \otimes I^n\}$. Es decir:

$$p(i|\pi\ \rho^n) = p(i|\pi'\ |\phi\rangle)$$

Demostración.

$$\pi_i^\dagger \pi_i = (|e_i\rangle \langle e_i|)^\dagger |e_i\rangle \langle e_i| = |e_i\rangle \langle e_i|e_i\rangle \langle e_i| = |e_i\rangle \langle e_i| \tag{4.2}$$

Siendo $\sum_i^{2^n} \lambda_i |v_i\rangle \langle v_i|$ la descomposición espectral de ρ^n:

$$\begin{aligned} p(i|\pi\ \rho^n) = \mathsf{tr}(\pi_i^\dagger \pi_i \rho^n) &\overset{4.2}{=} \mathsf{tr}\left(|e_i\rangle \langle e_i| \sum_j^{2^n} \lambda_j |v_j\rangle \langle v_j|\right) \\ &= \sum_j^{2^n} \lambda_j\ \mathsf{tr}\left(|e_i\rangle \langle e_i|\ \ |v_j\rangle \langle v_j|\right) \\ &= \sum_j^{2^n} \lambda_j \langle v_j|e_i\rangle \langle e_i|v_j\rangle = \sum_j^{2^n} \lambda_j \langle e_i|v_j\rangle \langle v_j|e_i\rangle \end{aligned} \tag{4.3}$$

$$\pi'_i = \pi_i \otimes I^n = |e_i\rangle \langle e_i| \otimes (\sum_k^{2^n} |e_k\rangle \langle e_k|) = \sum_k^{2^n} |e_i e_k\rangle \langle e_i e_k| \tag{4.4}$$

$$\pi'^\dagger_i \pi'_i \overset{4.4}{=} \left(\sum_k^{2^n} |e_i e_k\rangle \langle e_i e_k|\right) \sum_k^{2^n} |e_i e_k\rangle \langle e_i e_k| = \sum_k^{2^n} |e_i e_k\rangle \langle e_i e_k| \tag{4.5}$$

$$
\begin{aligned}
p(i|\pi' \left|\phi\right\rangle) &= \langle\phi| \, {\pi'}_i^\dagger \pi'_i \left|\phi\right\rangle \overset{4.5}{=} \langle\phi| \left(\sum_k^{2^n} |e_i e_k\rangle \langle e_i e_k| \right) |\phi\rangle \\
&= \langle\phi| \left(\sum_k^{2^n} |e_i e_k\rangle \langle e_i e_k| \right) |\sum_j^{2^n} \sqrt{\lambda_j} \left|v_j e_j\right\rangle\rangle \\
&= \langle\phi| \sum_k^{2^n} \sum_j^{2^n} \sqrt{\lambda_j} \left|e_i e_k\right\rangle \langle e_i e_k|v_j e_j\rangle\rangle \\
&= \langle\phi| \sum_k^{2^n} \sum_j^{2^n} \sqrt{\lambda_j} \left|e_i e_k\right\rangle \langle e_i|v_j\rangle \langle e_k|e_j\rangle\rangle \\
&= \langle\phi| \sum_k^{2^n} \sum_j^{2^n} \sqrt{\lambda_j} \left|e_i e_k\right\rangle \langle e_i|v_j\rangle \, \delta_{k,i}\rangle \\
&= \langle\phi| \sum_j^{2^n} \sqrt{\lambda_j} \left|e_i e_j\right\rangle \langle e_i|v_j\rangle\rangle \\
&= \langle \sum_j^{2^n} \sqrt{\lambda_j} \left|v_j e_j\right\rangle | \sum_j^{2^n} \sqrt{\lambda_j} \left|e_i e_j\right\rangle \langle e_i|v_j\rangle\rangle \\
&= \sum_j^{2^n} \sum_j^{2^n} \sqrt{\lambda_j}^* \sqrt{\lambda_j} \langle e_i|v_j\rangle \langle v_j e_j|e_i e_j\rangle \\
&= \sum_j^{2^n} \sum_j^{2^n} \sqrt{\lambda_j}^* \sqrt{\lambda_j} \langle e_i|v_j\rangle \langle v_j|e_i\rangle \langle e_j|e_j\rangle \\
&= \sum_j^{2^n} \lambda_j \langle e_i|v_j\rangle \langle v_j|e_i\rangle = p(i|\pi \ \rho^n)
\end{aligned} \tag{4.6}
$$

□

Observación. Notar que la purificación no es distributiva respecto al producto tensorial, es decir:

$$\mathrm{pur}(\rho^n \otimes \sigma^m) \neq \mathrm{pur}(\rho^n) \otimes \mathrm{pur}(\sigma^m)$$

Intuitivamente, esto se debe a que, por un lado, $\mathrm{pur}(\rho^n \otimes \sigma^m)$ agrega qubits en los bits menos significativos, mientras que en $\mathrm{pur}(\rho^n) \otimes \mathrm{pur}(\sigma^m)$ esos qubits agregados se encuentran en diferentes posiciones. Existe una forma de solucionar este inconveniente que se expone en la siguiente sección.

4.2.2. Purificación con permutaciones

Como se vio en la observación anterior la purificación clásica no preserva la composición. Existen dos posibles soluciones a esto, una es redefinir la composición y la otra es redefinir la purificación. Se optó por la segunda opción.

Para redefinir la purificación debemos tener en cuenta el invariante que queremos preservar: las mediciones y operadores que se apliquen al estado purificado deben ser sobre los qubits originales y no sobre los agregados por la purificación, aún luego de componer los estados.

La manera en la que se resolvió este problema es **colocando los qubits originales en las posiciones pares** y los adicionales en las posiciones impares. De este modo la composición común va a preservar el invariante de que los qubits "importantes" estén siempre en las posiciones pares. Por otro lado, las compuertas y operadores de medición pueden ser alterados para actuar solo sobre los qubits pares. En lo que resta de esta sección vamos a probar formalmente estas propiedades.

Definición de la purificación con permutaciones

Al purificar n-qubits utilizando la purificación de la Definición 1.28 se agregan otros n-qubits al estado en las posiciones menos significativas. Como se mencionó anteriormente, luego de purificar de la manera clásica deseamos colocar los n qubits más significativos en las posiciones pares. La siguiente permutación reordena $2n$ qubits de la manera deseada:

$$P_n(x) = \begin{cases} n + \frac{x}{2} & \text{si x es par} \\ \frac{x-1}{2} & \text{si x es impar} \end{cases} \tag{4.7}$$

Ejemplo 4.9. Si tenemos el estado $|i_3 i_2 i_1 i_0\rangle = |1100\rangle$. El estado deseado es

$|i_{P_2(3)}\ i_{P_2(2)}\ _{P_2(1)}\ i_{P_2(0)}\rangle = |i_1 i_3 i_0 i_2\rangle = |0101\rangle$.

Según este resultado se debe construir una compuerta SWP_n que aplique dicha permutación. Para encontrarla basta definirla para los vectores de la base canónica. Si tenemos $2n$ qubits, existen 2^{2n} de estos elementos. Sea un elemento de esta base representado en su forma binaria: $|i_0 \dots i_{2n-1}\rangle$, donde cada i es cero o uno. Sabemos que este estado tiene que ser transformado al estado $|i_{P_n(0)} \dots i_{P_n(2n-1)}\rangle$. Por lo tanto, aplicando esta lógica para todos los elementos de la base tenemos:

$$\mathrm{SWP}_n = \sum_{i_0,\dots,i_{2n-1}} |i_{P_n(0)} \dots i_{P_n(2n-1)}\rangle \langle i_0 \dots i_{2n-1}| \tag{4.8}$$

Observación. Así como SWP_n reordena los qubits de la mitad más significativa a las posiciones pares, SWP_n^{-1} realiza la operación inversa. Es decir, coloca los qubits de las posiciones pares en las posiciones más significativas.

De esta forma redefinimos la purificación de la siguiente manera:

Definición 4.10 (Pur)**.**

$$\mathrm{Pur}(\rho^n) = \mathrm{SWP}_n\ \mathrm{pur}(\rho^n) = \mathrm{SWP}_n \left(\sum_i^{2^n} \sqrt{\lambda_i}\, |v_i\rangle \otimes |e_i\rangle \right)$$

Observación. Cabe destacar que esta operación puede ser aplicada en tiempo de compilación.

4.2.3. Propiedades de la purificación con permutaciones

Teorema 4.11 (Compuerta sobre la purificación)**.** Aplicar una compuerta G de dimensión $|G| = m$ sobre los qubits pares equivale a aplicar esa compuerta sobre el estado original. Si $G' = I^p \otimes G \otimes I^{n-m-p}$:

$$(\mathrm{Pur}(\rho^n), r, \textbf{Gate}\ G\ [p, p+2, p+2m]; L) \xrightarrow{qk}_1 (\mathrm{Pur}(G'\ \rho^n\ G'^{\dagger}), r, L)$$

Demostración.

Utilizando la Definición 3.2 donde se define la operación **Gate**, sabemos que el estado resultante debe ser:

$$G''\ \mathrm{Pur}(\rho^n)$$

donde G'' está definida por:

$$G'' = (G' \otimes I^n)\ \mathrm{SWP}_n^{-1}$$

Expandiendo y haciendo uso del Teorema 4.7:

$$\begin{aligned} G''\ \mathrm{Pur}(\rho^n) &= (G' \otimes I^n)\ \mathrm{SWP}_n^{-1}\ \mathrm{SWP}_n\ \mathrm{pur}(\rho^n) \\ &= (G' \otimes I^n)\ \mathrm{pur}(\rho^n) \overset{\text{Teo. } 4.7}{=} \mathrm{pur}(G'\ \rho^n\ G'^{\dagger}) \end{aligned}$$

□

Teorema 4.12 (Medición de la purificación)**.** Sea $\theta = \mathrm{Pur}(\rho^n)$. Medir el qubit $2i$-ésimo del estado purificado (θ) en la base canónica equivale a medir el qubit i-ésimo del estado original (ρ^n). Siendo $\pi = \{|e_i\rangle \langle e_i|, 0 \leq i < 2^n\}$:

$(\theta, r, \textbf{Measure } [0, 2, \ldots, 2n]; L) \xrightarrow{qk}_{q_i} (\theta', i, L)$

con $q_i = \mathsf{tr}(\pi_i^\dagger \ \pi_i \ \rho^n)$.

Demostración.

Primero notar que este lema es muy similar al Teorema 4.8, donde se utilizaban los siguientes operadores de medición:

$$\pi' = \{\pi_i \otimes I^n, 0 \leq i < 2^n\}$$

En este caso, los operadores que miden los qubits en posiciones pares son:

$$\pi'' = \{\pi'_i \ \mathrm{SWP}_n^{-1}, 0 \leq i < 2^n\}$$

Si $\phi = \mathrm{pur}(\rho^n)$, es cierto que:

$$\begin{aligned} \pi''_i \left|\theta\right\rangle &= \pi''_i \ \mathrm{Pur}(\rho^n) = \pi'_i \ \mathrm{SWP}_n^{-1} \ \mathrm{SWP}_n \ \mathrm{pur}(\rho^n) \\ &= \pi'_i \ \mathrm{pur}(\rho^n) = \pi'_i \ \left|\phi\right\rangle \end{aligned}$$

Y análogamente:

$$\left\langle\theta\right| \pi''^\dagger_i = \left\langle\phi\right| \pi'^\dagger_i$$

Finalmente, utilizando la Definición 3.2 donde se define la operación **Measure**:

$$\begin{aligned} q_i &= \langle\theta|\pi''^\dagger_i \pi''_i|\theta\rangle = \langle\phi|\pi'^\dagger_i \pi'_i|\phi\rangle \\ &= p(m|\pi' \left|\phi\right\rangle) \overset{\text{Teo. } 4.8}{=} p(m|\pi \ \rho^n) = \mathsf{tr}(\pi_i^\dagger \pi_i \ \rho^n) \end{aligned}$$

□

Para poder demostrar el teorema de la composición de la purificación necesitamos dos pequeños lemas primero.

Lema 4.13 (Descomposición espectral del producto tensorial)**.** Sean ρ^n y σ^m dos matrices de densidad con descomposición espectral igual a $\sum_i^{2^n} \lambda_i \left|v_i\right\rangle \left\langle v_i\right|$ y $\sum_j^{2^m} \mu_j \left|w_j\right\rangle \left\langle w_j\right|$, respectivamente. La descomposición espectral de $\rho^n \otimes \sigma^m$ es:

$$\sum_i^{2^n} \sum_j^{2^m} \lambda_i \mu_j \left|v_i w_j\right\rangle \left\langle v_i w_j\right|$$

Demostración.

$$\begin{aligned}\rho^n \otimes \sigma^m &= \left(\sum_i^{2^n} \lambda_i \left|v_i\right\rangle \left\langle v_i\right|\right) \otimes \left(\sum_j^{2^m} \mu_j \left|w_j\right\rangle \left\langle w_j\right|\right) \\ &= \sum_i^{2^n}\sum_j^{2^m} \lambda_i\mu_j \left|v_i\right\rangle \left\langle v_i\right| \otimes \left|w_j\right\rangle \left\langle w_j\right| \\ &= \sum_i^{2^n}\sum_j^{2^m} \lambda_i\mu_j \left|v_iw_j\right\rangle \left\langle v_iw_j\right|\end{aligned}$$

Notar que como $\lambda_i \geq 0, \mu_j \geq 0 \implies \lambda_i\mu_j \geq 0$. Basta probar que $\lambda_i\mu_j$ y $\left|v_iw_j\right\rangle$ son los autovalores y autovectores de $\rho^n \otimes \sigma^m$, respectivamente:

$$\begin{aligned}\left(\rho^n \otimes \sigma^m\right)\left|v_iw_j\right\rangle &= \left(\sum_k^{2^n}\sum_l^{2^m} \lambda_k\mu_l \left|v_kw_l\right\rangle \left\langle v_kw_l\right|\right)\left|v_iw_j\right\rangle \\ &= \sum_k^{2^n}\sum_l^{2^m} \lambda_k\mu_l \left|v_kw_l\right\rangle \left\langle v_kw_l|v_iw_j\right\rangle \\ &= \sum_k^{2^n}\sum_l^{2^m} \lambda_k\mu_l \left|v_kw_l\right\rangle \delta_{k,i}\,\delta_{l,j} \\ &= \lambda_i\mu_j \left|v_iw_j\right\rangle\end{aligned}$$

□

Lema 4.14 (SWP)**.** Sea $\left|a\right\rangle, \left|b\right\rangle, \left|c\right\rangle, \left|d\right\rangle$ vectores de estado de n, n, m, m qubits, respectivamente.

$$\left(\mathrm{SWP}_n \otimes \mathrm{SWP}_m\right)\left|abcd\right\rangle = \mathrm{SWP}_{n+m}\left|acbd\right\rangle$$

□

Ejemplo 4.15. Este lema no se demostrará, pero se puede entender intuitivamente. Veamos para $n = m = 3$ que $\mathrm{SWP}_{n+m}^{-1}(\mathrm{SWP}_n \otimes \mathrm{SWP}_m)\left|abcd\right\rangle = \left|acbd\right\rangle$. Los qubits de $\left|abcd\right\rangle$ están ordenados de la siguiente forma:

$$a_2\ a_1\ a_0\ b_2\ b_1\ b_0\ c_2\ c_1\ c_0\ d_2\ d_1\ d_0$$

SWP_n a la primera parte va a mover los qubits de $\left|a\right\rangle$ a las posiciones pares:

$$b_2\ a_2\ b_1\ a_1\ b_0\ a_0\ c_2\ c_1\ c_0\ d_2\ d_1\ d_0$$

SWP_m a la segunda parte va a mover los qubits de $\left|c\right\rangle$ a las posiciones pares:

$$b_2\ a_2\ b_1\ a_1\ b_0\ a_0\ d_2\ c_2\ d_1\ c_1\ d_0\ c_0$$

Observar que en las posiciones pares tenemos $|a\rangle \otimes |c\rangle$ y en las impares $|b\rangle \otimes |d\rangle$. Por lo tanto, al aplicar SWP_{n+m}^{-1}:

$$a_2\ a_1\ a_0\ c_2\ c_1\ c_0\ b_2\ b_1\ b_0\ d_2\ d_1\ d_0$$

Que es exactamente $|acbd\rangle$.

Teorema 4.16 (Composición de la purificación)**.** Componer dos estados purificados equivale a la purificación del estado compuesto:

$$\mathrm{Pur}(\rho^n \otimes \sigma^m) = \mathrm{Pur}(\rho^n) \otimes \mathrm{Pur}(\sigma^m)$$

Demostración. Siendo $\mathrm{pur}(\rho^n) = \sum_i^{2^n} \sqrt{\lambda_i}\,|v_i\rangle \otimes |e_i\rangle$ y $\mathrm{pur}(\sigma^n) = \sum_j^{2^m} \sqrt{\mu_j}\,|w_j\rangle \otimes |e_j\rangle$.

$$\begin{aligned}
&\mathrm{Pur}(\rho^n) \otimes \mathrm{Pur}(\sigma^m) \\
&\quad = \mathrm{SWP}_n \left(\sum_i^{2^n} \sqrt{\lambda_i}\,|v_i\rangle \otimes |e_i\rangle \right) \otimes \mathrm{SWP}_m \left(\sum_j^{2^m} \sqrt{\mu_j}\,|w_j\rangle \otimes |e_j\rangle \right) \\
&\quad = (\mathrm{SWP}_n \otimes \mathrm{SWP}_m) \left(\sum_i^{2^n} \sqrt{\lambda_i}\,|v_i\rangle \otimes |e_i\rangle \right) \otimes \left(\sum_j^{2^m} \sqrt{\mu_j}\,|w_j\rangle \otimes |e_j\rangle \right) \\
&\quad = (\mathrm{SWP}_n \otimes \mathrm{SWP}_m) \sum_i^{2^n} \sum_j^{2^m} \sqrt{\lambda_i \mu_j}\,|v_i e_i w_j e_j\rangle \\
&\quad = \sum_i^{2^n} \sum_j^{2^m} \sqrt{\lambda_i \mu_j}\,(\mathrm{SWP}_n \otimes \mathrm{SWP}_m)\,|v_i e_i w_j e_j\rangle \\
&\quad \overset{\text{Lema 4.14}}{=} \sum_i^{2^n} \sum_j^{2^m} \sqrt{\lambda_i \mu_j}\,\mathrm{SWP}_{n+m}\,|v_i w_j e_i e_j\rangle \\
&\quad = \mathrm{SWP}_{n+m} \sum_i^{2^n} \sum_j^{2^m} \sqrt{\lambda_i \mu_j}\,|v_i w_j e_i e_j\rangle \\
&\quad \overset{\text{Lema 4.13}}{=} \mathrm{SWP}_{n+m}\ \mathrm{pur}(\rho^n \otimes \sigma^m) = \mathrm{Pur}(\rho^n \otimes \sigma^m)
\end{aligned}$$

□

4.3. Definición de la traducción

Antes de definir la traducción resulta necesario agregar una pequeña función auxiliar a Python. Está función construye un estado puro basado en un entero.

Definición 4.17.

$$\texttt{t} ::= \texttt{from\_int(t, t)} \mid \ldots$$

$$\texttt{from\_int(i, n)} \xrightarrow{py}_1 \texttt{Circuit(Pur}(\pi_i^n)\texttt{, [])} \qquad (\texttt{from\_int}_{py})$$

donde $n \in \mathbb{N}, i \in \mathbb{N}_0, 0 \leq i < 2^n$ y $\{\pi_i^n\}$ son las matrices de medición de la base canónica de n qubits.

Ejemplo 4.18. Vamos a reducir `from_int(6, 4)`. El valor once en binario es 110. Como se trata de cuatro qubits el vector de estado sería $|0110\rangle$. Luego, $\pi_i^n = |0110\rangle\langle 0110|$. Su purificación es $\text{Pur}(|0110\rangle\langle 0110|) = \text{SWP}_4\ \text{pur}(|0110\rangle\langle 0110|) = \text{SWP}_4\,|01100110\rangle = |00111100\rangle$. De esta forma:

$$\texttt{from\_int(6, 4)} \xrightarrow{py}_1 \texttt{Circuit(}\,|00111100\rangle\texttt{ , [])}$$

Utilizando esta función auxiliar y el método de purificación presentado (Pur), la Definición 4.19 presenta formalmente la traducción de λ_ρ^* a Python. Seguidamente, se procede a probar propiedades sobre ella.

Definición 4.19 (Traducción de λ_ρ^* a Python)**.**

Se define la función $(\!|\cdot|\!) : \Lambda_\rho^* \to \Lambda_{py}$:

$$\begin{aligned}
(\!|x|\!) &= \texttt{x} \\
(\!|\lambda x.t|\!) &= \texttt{lambda x:}(\!|t|\!) \\
(\!|t_1\ t_2|\!) &= (\!|t_1|\!)((\!|t_2|\!)) \\
(\!|\rho^n|\!) &= \texttt{Circuit(Pur}(\rho^n)\texttt{, [])} \\
(\!|G_p\ t|\!) &= (\!|t|\!)\texttt{.gate(}G\texttt{, [}p\texttt{, }p+2\texttt{, ..., }p+2|G|\texttt{])} \\
(\!|t_1 \otimes t_2|\!) &= (\!|t_1|\!)\texttt{.compose(}(\!|t_2|\!)\texttt{)} \\
(\!|(b^m, \rho^n)|\!) &= \texttt{((b, m), }(\!|\rho^n|\!)\texttt{)} \\
(\!|\textsf{letcase } x = r \text{ in } \{t_0, \ldots, t_{2^m-1}\}|\!) &= \texttt{letcase(}(\!|r|\!)\texttt{, [}(\!|t_0|\!)\texttt{, ..., }(\!|t_{2^m-1}|\!)\texttt{])}
\end{aligned}$$

$$(\!|\pi^m\ t|\!) = \texttt{(lambda rho: (lambda vi: build\_pair)(meas))(}(\!|t|\!)\texttt{)}$$

donde

- `vn = rho.size()`.
- `meas = rho.measure(0, 2, ..., vn)`.
- `build_pair = ((vi/(2**(vn/2-m)), m), from_int(vi, vn/2))`.

- $b \in \mathbb{N}_0, m, n \in \mathbb{N}$.
- $G \in \{U^{\theta,\phi,\lambda}, UC^{\theta,\phi,\lambda}, \text{SWAP}, \text{CSWAP}\}$, con $\theta, \phi, \lambda \in \mathbb{R}$.

4.4. Correctitud

Cuando hablamos de correctitud nos referimos a que si una expresión en el lenguaje original reduce a cierta expresión, entonces la expresión traducida también debe reducir a una expresión equivalente.

Como se verá en las pruebas en algunos casos los circuitos generados por la traducción pueden no ser exactamente los mismos a los originales, pero lo que nos interesa es si los mismos producen los mismos vectores de estado. En la Definición 4.20 se formaliza este concepto.

Definición 4.20 (Expresiones equivalentes)**.** Se define la relación de equivalencia $\approx$: $\Lambda_{py} \times \Lambda_{py}$ que indica cuando dos expresiones difieren únicamente en la manera de construir el circuito.

$$\frac{(|0\rangle^n, r, \textbf{Init } \texttt{I; L}) \xrightarrow{qk}{}^*_1 (|\phi\rangle, r, [\,]) \quad (|0\rangle^n, r, \textbf{Init } \texttt{J; L'}) \xrightarrow{qk}{}^*_1 (|\phi\rangle, r, [\,])}{\texttt{Circuit(I, L)} \approx \texttt{Circuit(J, L')}}$$

$$\frac{}{\texttt{x} \approx \texttt{x}} \qquad \frac{\texttt{t} \xrightarrow{py}_1 \texttt{r}}{\texttt{t} \approx \texttt{r}}$$

$$\frac{\texttt{t} \approx \texttt{r} \quad \texttt{s} \approx \texttt{u}}{\texttt{s(t)} \approx \texttt{u(r)}} \qquad \frac{\texttt{t} \approx \texttt{r}}{\texttt{t.gate(G, A)} \approx \texttt{r.gate(G, A)}}$$

$$\frac{\texttt{t} \approx \texttt{r} \quad \texttt{s} \approx \texttt{u}}{\texttt{t.compose(s)} \approx \texttt{r.compose(u)}}$$

$$\frac{\texttt{t} \approx \texttt{r} \quad \texttt{c}_1 \approx \texttt{d}_1 \quad \ldots \quad \texttt{c}_{2^n} \approx \texttt{d}_{2^n}}{\texttt{letcase(t, [c}_1\texttt{, ..., c}_{2^n}\texttt{])} \approx \texttt{letcase(r, [d}_1\texttt{, ..., d}_{2^n}\texttt{])}}$$

Antes de probar la correctitud resultan útil los siguientes lemas.

Lema 4.21 (Sustitución)**.** Dado $t, r \in \Lambda^*_\rho$, $(\!|[r/x]t|\!) = [(\!|r|\!)/x](\!|t|\!)$

Demostración. Este lema fue demostrado por Agustín B. para λ_ρ en [Bor19] (Lema

3.1.8), utilizando inducción estructural en t. La adaptación de la prueba a λ^*_ρ es trivial. □

Lema 4.22 (Preservación de Valores)**.** $v \in \Lambda^*_\rho$ es un valor si y solo si $(\!|v|\!)$ es un valor en Λ_{py}.

Demostración. Se procede por inducción estructural en v. Es decir vamos a probar el lema para las posibles formas de v, asumiendo que cualquier sub-expresión w de v es un valor si y solo si $(\!|w|\!)$ es un valor. De esta manera se puede concluir que el lema vale para todas las expresiones de Λ^*_ρ.

Caso $v = x$

$(\!|x|\!) =$ `x`. Ambos x y `x` son trivialmente valores.

Caso $v = (b^m, \rho^n)$

$(\!|(b^m, \rho^n)|\!) =$ `((b, m), Circuit(`Pur(ρ^n)`, [])`.

Ambos (b^m, ρ^n) y `((b, m), Circuit(`Pur(ρ^n)`, [])` son valores porque no existe ninguna regla de reducción aplicable para ninguno de los dos.

Caso $v = \rho^n$

$(\!|\rho^n|\!) =$ `Circuit(`Pur(ρ^n)`, [])`. Ambos ρ^n y `Circuit(`Pur(ρ^n)`, [])` son trivialmente valores.

Caso $v = \lambda x.t$

$\Longrightarrow$) $(\!|\lambda x.t|\!) =$ `lambda x:`$(\!|t|\!)$. No existe ninguna regla en la Definición 3.4 que tenga `lambda x` como operación más externa, por lo que es un valor.

$\Longleftarrow$) No existe ninguna regla en la Definición 4.5 que tenga λx como operación más externa, por lo que es un valor.

Caso $v = t_1\ t_2$

$(\!|t_1\ t_2|\!) = (\!|t_1|\!)((\!|t_2|\!))$

Debido a la forma de $t_1\ t_2$ solo se pueden aplicar las reglas Lr_ρ, Rr_ρ y λ_ρ. Debido a la forma de $(\!|t_1\ t_2|\!)$ solo se pueden aplicar las reglas Lr_{py}, Rr_{py} y λ_{py}. Veamos que si asumimos que una es no aplicable la otra tampoco lo es, respectivamente.

- La regla Lr_ρ no se puede aplicar sobre $t_1\ t_2 \Leftrightarrow t_1$ es un valor $\overset{\text{HI}}{\Leftrightarrow}$ $(\!|t_1|\!)$ es un valor $\Leftrightarrow$ La regla Lr_{py} no se puede aplicar sobre $(\!|t_1\ t_2|\!)$.

- La regla Rr_ρ no se puede aplicar sobre $t_1\ t_2 \Leftrightarrow t_1$ no es un valor o bien t_2 es un valor $\overset{\text{HI}}{\Leftrightarrow} (\!|t_1|\!)$ no es un valor o bien $(\!|t_2|\!)$ es un valor $\Leftrightarrow$ La regla Rr_{py} no se puede aplicar sobre $(\!|t_1\ t_2|\!)$.
- La regla λ_ρ no se puede aplicar sobre $t_1\ t_2 \Leftrightarrow t_1$ no es de la forma $\lambda x.r$ o bien t_2 no es un valor $\overset{\text{HI}}{\Leftrightarrow} (\!|t_1|\!)$ no es de la forma `lambda x:r` o bien $(\!|t_2|\!)$ no es un valor $\Leftrightarrow$ La regla λ_{py} no se puede aplicar sobre $(\!|t_1\ t_2|\!)$.

Por lo tanto, $t_1\ t_2$ es un valor $\Leftrightarrow (\!|t_1\ t_2|\!)$ es un valor.

Caso $v = G\ t$

$(\!|G\ t|\!) = (\!|t|\!)$`.gate(`G`, [`p`, `$p+2$`, ..., `$p+2|G|$`])`

Debido a la forma de $G\ t$ solo se pueden aplicar las reglas Gr_ρ y G_ρ. Debido a la forma de $(\!|G\ t|\!)$ solo se pueden aplicar las reglas Gr_{py} y G_{py}. Veamos que si asumimos que una es no aplicable la otra tampoco lo es, respectivamente.

- La regla Gr_ρ no se puede aplicar sobre $G\ t \Leftrightarrow t$ debe ser un valor $\overset{\text{HI}}{\Leftrightarrow} (\!|t|\!)$ es un valor $\Leftrightarrow$ La regla Gr_{py} no se puede aplicar sobre $(\!|G\ t|\!)$.
- La regla G_ρ no se puede aplicar sobre $G\ t \Leftrightarrow t$ no es de la forma $\rho^n \Leftrightarrow (\!|t|\!)$ no es de la forma `Circuit(I, L)` $\Leftrightarrow$ La regla G_{py} no se puede aplicar sobre $(\!|G\ t|\!)$.

Por lo tanto, $G\ t$ es un valor $\Leftrightarrow (\!|G\ t|\!)$ es un valor.

Caso $v = t_1 \otimes t_2$

$(\!|t_1 \otimes t_2|\!) = (\!|t_1|\!)$`.compose(`$(\!|t_2|\!)$`)`

Debido a la forma de $t_1 \otimes t_2$ solo se pueden aplicar las reglas $L\otimes_\rho$, $R\otimes_\rho$ y $\otimes_\rho$. Debido a la forma de $(\!|t_1 \otimes t_2|\!)$ solo se pueden aplicar las reglas $L\otimes_{py}$, $R\otimes_{py}$ y $\otimes_{py}$. Veamos que si asumimos que una es no aplicable la otra tampoco lo es, respectivamente.

- La regla $L\otimes_\rho$ no se puede aplicar sobre $t_1 \otimes t_2 \Leftrightarrow t_1$ es un valor $\overset{\text{HI}}{\Leftrightarrow} (\!|t_2|\!)$ es un valor $\Leftrightarrow$ La regla $L\otimes_{py}$ no se puede aplicar sobre $(\!|t_1 \otimes t_2|\!)$.
- La regla $R\otimes_\rho$ no se puede aplicar sobre $t_1 \otimes t_2 \Leftrightarrow t_1$ no es un valor o bien t_2 es un valor $\overset{\text{HI}}{\Leftrightarrow} (\!|t_1|\!)$ no es un valor o bien $(\!|t_2|\!)$ es un valor $\Leftrightarrow$ La regla $R\otimes_{py}$ no se puede aplicar sobre $(\!|t_1 \otimes t_2|\!)$.
- La regla $\otimes_\rho$ no se puede aplicar sobre $t_1 \otimes t_2 \Leftrightarrow t_1$ ó t_2 no es de la forma $\rho^n \Leftrightarrow (\!|t_1|\!)$ ó $(\!|t_1|\!)$ no es de la forma `Circuit(I, L)` $\Leftrightarrow$ La regla $\otimes_{py}$ no se puede aplicar sobre $(\!|t_1 \otimes t_2|\!)$.

Por lo tanto, $t_1 \otimes t_2$ es un valor $\Leftrightarrow (\!|t_1 \otimes t_2|\!)$ es un valor.

Caso $v = \mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\}$

$(\!|\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_0, \ldots, t_{2^m-1}\}|\!) =$ `letcase((|r|), [(|t0|), ..., (|t2^m-1|)])`

Debido a la forma de $\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_1, \ldots, t_m\}$ solo se pueden aplicar las reglas letr_ρ y let_ρ. Debido a la forma de $(\!|\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_1, \ldots, t_m\}|\!)$ solo se pueden aplicar las reglas letr_{py} y let_{py}. Veamos que si asumimos que una es no aplicable la otra tampoco lo es, respectivamente.

- La regla letr_ρ no se puede aplicar sobre $\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_1, \ldots, t_m\} \Leftrightarrow r$ debe ser un valor $\overset{\text{HI}}{\Leftrightarrow} (\!|r|\!)$ es un valor $\Leftrightarrow$ La regla letr_{py} no se puede aplicar sobre $(\!|\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_1, \ldots, t_m\}|\!)$.
- La regla let_ρ no se puede aplicar sobre $\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_1, \ldots, t_m\} \Leftrightarrow r$ no es de la forma $(b^m, \rho^n) \Leftrightarrow (\!|r|\!)$ no es de la forma `((b, m), rho)` $\Leftrightarrow$ La regla let_{py} no se puede aplicar sobre $(\!|\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_1, \ldots, t_m\}|\!)$.

Por lo tanto,

$\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_1, \ldots, t_m\}$ es un valor $\Leftrightarrow$ $(\!|\mathsf{letcase}\ x = r\ \mathsf{in}\ \{t_1, \ldots, t_m\}|\!)$ es un valor.

Caso $v = \pi^m\ t$

$(\!|\pi^m\ t|\!) =$ `(lambda rho: (lambda vi: build_pair)(meas))((|t|))`

Debido a la forma de $\pi^m\ t$ solo se pueden aplicar las reglas πr_ρ y π_ρ. Debido a la forma de $(\!|\pi^m\ t|\!)$ solo se pueden aplicar las reglas Rr_{py} y λ_{py}. Veamos que si asumimos que una es no aplicable la otra tampoco lo es, respectivamente.

- La regla πr_ρ no se puede aplicar sobre $\pi^m\ t \Leftrightarrow t$ debe ser un valor $\overset{\text{HI}}{\Leftrightarrow} (\!|t|\!)$ es un valor $\Leftrightarrow$ La regla Rr_{py} no se puede aplicar sobre $(\!|\pi^m\ t|\!)$.
- Como la expresión $\pi^m\ t$ tiene un tipo y la regla de tipado π es la única aplicable, debe ser que $t : n$, para algún $n \geq m \implies$ si t es un valor, el único valor posible es de la forma ρ^n.
- La regla π_ρ no se puede aplicar sobre $\pi^m\ t \Leftrightarrow t$ no es de la forma $\rho^n \Leftrightarrow t$ no es un valor $\Leftrightarrow (\!|t|\!)$ no es un valor $\Leftrightarrow$ La regla λ_{py} no se puede aplicar sobre $(\!|\pi^m\ t|\!)$.

Por lo tanto, $\pi^m\ t$ es un valor $\Leftrightarrow (\!|\pi^m\ t|\!)$ es un valor.

□

Por último agregamos dos lemas necesarios para demostrar la correctitud sobre la regla π_ρ.

Lema 4.23. La purificación de una matriz de densidad de n qubits posee $2n$ qubits, es decir:

$$(\!|\rho^n|\!)\texttt{.size()} \xrightarrow{py}_1 2n$$

Demostración.

$$\begin{aligned}(\!|\rho^n|\!)\texttt{.size()} &= \texttt{Circuit(}\mathrm{Pur}(\rho^n)\texttt{, [])}\texttt{.size()} \\ &\xrightarrow{py}_1 \log_2 |\mathrm{Pur}(\rho^n)| \overset{\text{Def. 4.10}}{=} \log_2(2^{2n}) = 2n\end{aligned}$$

□

Lema 4.24.

$$(\!|\rho^n|\!)\texttt{.measure([0, 2, ..., 2n])} \xrightarrow{py}_{p_i} \texttt{i}$$

donde $p_i = \mathsf{tr}(\pi^{n\dagger}_i \pi^n{}_i \rho^n)$

Demostración.

$$\begin{aligned}&(\!|\rho^n|\!)\texttt{.measure([0, 2, ..., 2n])} \\ &\quad = \texttt{Circuit(}\mathrm{Pur}(\rho^n)\texttt{, [])}\texttt{.measure([0, 2, ..., 2n])} \\ &\quad \xrightarrow{py}_{p_i} \texttt{i}\end{aligned}$$

Donde la última reducción es válida porque:

$$\begin{aligned}&(|0\rangle^{2n}, 0, \mathbf{Init}\ \mathrm{Pur}(\rho^n); \mathbf{Measure}\ [0, 2, \ldots, 2n]) \\ &\qquad \xrightarrow{qk}_1 (\mathrm{Pur}(\rho^n), 0, \mathbf{Measure}\ [0, 2, \ldots, 2n]) \\ &\qquad \xrightarrow{qk}_{p_i} (\phi, i, [\,])\end{aligned}$$

Donde la última reducción es válida por el Teorema 4.12. □

Lema 4.25. Sea ρ^n una matriz de densidad, $\{\pi_i\} = \{|e_i\rangle \langle e_i|, 0 \leq i < 2^n\}$ (las matrices de medición de la base canónica). El estado del sistema al aplicar la medición $\{\pi_i\}$ luego obtener el resultado i es:

$$\pi_i$$

Demostración. Por la Ecuación 1.15, la probabilidad de obtener el resultado i al medir con $\{\pi\}$ es:

$$p_i = \mathsf{tr}(\pi_i^\dagger \pi_i \rho^n).$$

Notar que como π_i es simétrico, $\pi_i^\dagger = \pi_i$. Por la Ecuación 1.16, el estado luego de obtener el resultado i es:

$$\rho_i^n = \frac{\pi_i \rho^n \pi_i^\dagger}{p_i}$$

Vamos a expandir el denominador y el numerador. Sea $\sum_j \lambda_j |v_j\rangle \langle v_j|$ la descomposición espectral de ρ^n.

$$\begin{aligned} p_i &= \mathsf{tr}(\pi_i^\dagger \pi_i \rho^n) = \mathsf{tr}(\pi_i \pi_i \rho^n) = \mathsf{tr}\left(\pi_i \pi_i \sum_j \lambda_j |v_j\rangle \langle v_j|\right) \\ &= \sum_j \lambda_j \mathsf{tr}\left(\pi_i \pi_i |v_j\rangle \langle v_j|\right) = \sum_j \lambda_j \langle v_j| \pi_i \pi_i |v_j\rangle \\ &= \sum_j \lambda_j \langle v_j|e_i\rangle \langle e_i|e_i\rangle \langle e_i|v_j\rangle = \sum_j \lambda_j \langle v_j|e_i\rangle \langle e_i|v_j\rangle \\ &= \sum_j \lambda_j \langle e_i|v_j\rangle \langle v_j|e_i\rangle \end{aligned} \tag{4.9}$$

$$\begin{aligned} \pi_i \rho^n \pi_i^\dagger = \pi_i \rho^n \pi_i &= \pi_i \left(\sum_j \lambda_j |v_j\rangle \langle v_j|\right) \pi_i = \sum_j \lambda_j \pi_i |v_j\rangle \langle v_j| \pi_i \\ &= \sum_j \lambda_j |e_i\rangle \langle e_i|v_j\rangle \langle v_j|e_i\rangle \langle e_i| = \sum_j \lambda_j \pi_i \langle e_i|v_j\rangle \langle v_j|e_i\rangle \\ &= \pi_i \sum_j \lambda_j \langle e_i|v_j\rangle \langle v_j|e_i\rangle \end{aligned} \tag{4.10}$$

Finalmente, utilizando los resultados 4.9 y 4.10:

$$\rho_i^n = \frac{\pi_i \rho^n \pi_i^\dagger}{p_i} = \frac{\pi_i \sum_j \lambda_j \langle e_i|v_j\rangle \langle v_j|e_i\rangle}{p_i} = \frac{\pi_i \sum_j \lambda_j \langle e_i|v_j\rangle \langle v_j|e_i\rangle}{\sum_j \lambda_j \langle e_i|v_j\rangle \langle v_j|e_i\rangle} = \pi_i$$

□

Teorema 4.26 (Correctitud)**.** Sea $t, r \in \Lambda_\rho^*$.

Si $t \xrightarrow{\lambda_\rho}_p r$, entonces $\exists r'$ tal que $(\!|t|\!) \xrightarrow{py}_p^* r' \approx (\!|r|\!)$.

Demostración. Se procede por inducción en la definición de la relación $\xrightarrow{\lambda_\rho}_p$.

Caso $(\lambda x.t)\ v \xrightarrow{\lambda_\rho}_1 [v/x]t$, donde v es un valor

$$(\!|(\lambda x.t)\ v|\!) = \texttt{(lambda x:}(\!|t|\!)\texttt{)}((\!|v|\!)) \overset{\text{Lema 4.22}}{\xrightarrow{py}_1} [(\!|v|\!)/x](\!|t|\!) \overset{\text{Lema 4.21}}{=} (\!|[v/x]t|\!)$$

Caso $G_p\ \rho^n \xrightarrow{\lambda_\rho}_1 \rho'^n \qquad$ con $\rho'^n = \overline{G_p}\ \rho^n\ \overline{G_p}^\dagger$

$(\!|G_p\ \rho^n|\!) =$
$\texttt{Circuit}(\text{Pur}(\rho^n),\ \texttt{[]}).\texttt{gate}(G,\ \texttt{[}p,\ p+2,\ \ldots,\ p+2|G|\texttt{]})$
$\xrightarrow{py}_1 \texttt{Circuit}(\text{Pur}(\rho^n),\ \texttt{[Gate G [}p,\ p+2,\ \ldots,\ p+2|G|\texttt{]})$

$(\!|\rho'^n|\!) = \texttt{Circuit}(\text{Pur}(\overline{G}\ \rho^n\ \overline{G}^\dagger),\ \texttt{[]})$

Se debe probar que los circuitos generados por

$$L = \texttt{[Init Pur}(\rho^n)\texttt{, Gate G [}p,\ p+2,\ \ldots,\ p+2|G|\texttt{]}$$
$$L' = \texttt{[Init Pur}(\overline{G}\ \rho^n\ \overline{G}^\dagger)\texttt{]}$$

son equivalentes. Por la Definición 3.2:

$$(|0\rangle^n, r, L') \xrightarrow{qk}_1 (\text{pur}(\overline{G}\ \rho^n\ \overline{G}^\dagger), r, [\,])$$

Usando la Definición 3.2 y el Teorema 4.11:

$$\begin{aligned}(|0\rangle^n, r, L) &\xrightarrow{qk}_1 (\text{Pur}(\rho^n), r, \mathbf{Gate}\ G\ [p, p+2, \ldots, p+2|G|)\\ &\xrightarrow{qk}_1 (\text{Pur}(\overline{G_p}\ \rho^n\ \overline{G_p}^\dagger), r, [\,])\end{aligned}$$

De esta forma:

$$\texttt{Circuit}(\text{Pur}(\rho^n),\ \texttt{[Gate G [}p,\ p+2,\ \ldots,\ p+2|G|\texttt{]})$$
$$\approx$$
$$\texttt{Circuit}(\text{Pur}(\overline{G}\ \rho^n\ \overline{G}^\dagger),\ \texttt{[]}) = (\!|\rho'^n|\!)$$

Caso $\rho \otimes \rho' \xrightarrow{\lambda_\rho}_1 \rho'' \qquad$ con $\rho'' = \rho \otimes \rho'$

Siendo ρ y ρ' de n y m qubits respectivamente:

$$\begin{aligned}(\!|\rho \otimes \rho'|\!) &= \texttt{Circuit}(\text{Pur}(\rho),\ \texttt{[]}).\texttt{compose}(\texttt{Circuit}(\text{Pur}(\rho'),\ \texttt{[]})\\ &\xrightarrow{py^*}_1 \texttt{Circuit}(\text{Pur}(\rho) \otimes \text{Pur}(\rho'),\ \texttt{[]})\\ &\overset{\text{Teo. 4.16}}{=} \texttt{Circuit}(\text{Pur}(\rho \otimes \rho'),\ \texttt{[]}) = (\!|\rho''|\!)\end{aligned}$$

Caso $\mathsf{letcase}\ x = (b^m, \rho^n)\ \text{in}\ \{t_0, \ldots, t_{2^m-1}\} \xrightarrow{\lambda_\rho}_1 [\rho^n/x]t_{b^m}$

$$\begin{aligned}(\!|\mathsf{letcase}\ x = (b^m, \rho^n)\ \text{in}\ \{t_0, \ldots, t_{2^m-1}\}|\!) &=\\ &\texttt{letcase(((b, m)},\ (\!|\rho^n|\!)\texttt{)},\ \texttt{[}(\!|t_0|\!)\texttt{,}\ \ldots\texttt{,}\ (\!|t_{2^m-1}|\!)\texttt{]})\\ &\xrightarrow{py}_1 (\!|t_\texttt{b}|\!)((\!|\rho^n|\!))\\ &\overset{\text{Lema 4.22}}{\xrightarrow{py}_1} [(\!|\rho^n|\!)/x](\!|t_\texttt{b}|\!)\\ &\overset{\text{Lema 4.21}}{=} (\!|[\rho^n/x]t_{b^m}|\!)\end{aligned}$$

Caso $t\ s \xrightarrow{\lambda_\rho}_p r\ s$, con $t \xrightarrow{\lambda_\rho}_p r$

Por hipótesis de inducción, $\exists \mathtt{u} \approx (\!|r|\!)/(\!|t|\!) \xrightarrow{py}{}^*_p \mathtt{u}$.

$(\!|t\ s|\!) = (\!|t|\!)\mathtt{(}(\!|s|\!)\mathtt{)} \xrightarrow{py}{}^*_p \mathtt{u(}(\!|s|\!)\mathtt{)} \approx (\!|r|\!)\mathtt{(}(\!|s|\!)\mathtt{)} = (\!|r\ s|\!)$

Caso $v\ t \xrightarrow{\lambda_\rho}_p v\ r$, con $t \xrightarrow{\lambda_\rho}_p r$, donde v es un valor

Por hipótesis de inducción, $\exists \mathtt{u} \approx (\!|r|\!)/(\!|t|\!) \xrightarrow{py}{}^*_p \mathtt{u}$.

$(\!|v\ t|\!) = (\!|v|\!)\mathtt{(}(\!|t|\!)\mathtt{)} \overset{\text{Lema 4.22}}{\xrightarrow{py}{}^*_p} (\!|v|\!)\mathtt{(u)} \approx (\!|v|\!)\mathtt{(}(\!|r|\!)\mathtt{)} = (\!|v\ r|\!)$

Caso $v \otimes t \xrightarrow{\lambda_\rho}_p v \otimes r$, con $t \xrightarrow{\lambda_\rho}_p r$, donde v es un valor

Por hipótesis de inducción, $\exists \mathtt{u} \approx (\!|r|\!)/(\!|t|\!) \xrightarrow{py}{}^*_p \mathtt{u}$.

$(\!|v \otimes t|\!) = (\!|v|\!)\mathtt{.compose(}(\!|t|\!)\mathtt{)} \overset{\text{Lema 4.22}}{\xrightarrow{py}{}^*_p} (\!|v|\!)\mathtt{.compose(u)} \approx (\!|v|\!)\mathtt{.compose(}(\!|r|\!)\mathtt{)} = (\!|v \otimes r|\!)$

Caso $t \otimes s \xrightarrow{\lambda_\rho}_p r \otimes s$, con $t \xrightarrow{\lambda_\rho}_p r$

Por hipótesis de inducción, $\exists \mathtt{u} \approx (\!|r|\!)/(\!|t|\!) \xrightarrow{py}{}^*_p \mathtt{u}$.

$(\!|t \otimes s|\!) = (\!|t|\!)\mathtt{.compose(}(\!|s|\!)\mathtt{)} \xrightarrow{py}{}^*_p \mathtt{u.compose(}(\!|s|\!)\mathtt{)} \approx (\!|r|\!)\mathtt{.compose(}(\!|s|\!)\mathtt{)} = (\!|r \otimes s|\!)$

Caso $\mathsf{letcase}\ x = t\ \mathrm{in}\ \{s_0, \ldots, s_n\} \xrightarrow{\lambda_\rho}_p \mathsf{letcase}\ x = r\ \mathrm{in}\ \{s_0, \ldots, s_n\}$, con $t \xrightarrow{\lambda_\rho}_p r$

Por hipótesis de inducción, $\exists \mathtt{u} \approx (\!|r|\!)/(\!|t|\!) \xrightarrow{py}{}^*_p \mathtt{u}$.

$(\!|\mathsf{letcase}\ x = t\ \mathrm{in}\ \{s_0, \ldots, s_n\}|\!) = \mathtt{letcase(}(\!|t|\!)\mathtt{,\ [}(\!|s_0|\!)\mathtt{,...,}(\!|s_n|\!)\mathtt{])}$

$\xrightarrow{py}{}^*_p \mathtt{letcase(u,\ [}(\!|s_0|\!)\mathtt{,...,}(\!|s_n|\!)\mathtt{])} \approx \mathtt{letcase(}(\!|r|\!)\mathtt{,\ [}(\!|s_0|\!)\mathtt{,...,}(\!|s_n|\!)\mathtt{])}$

$= (\!|\mathsf{letcase}\ x = r\ \mathrm{in}\ \{s_0, \ldots, s_n\}|\!)$

Caso $\pi^m\ t \xrightarrow{\lambda_\rho}_p \pi^m\ r$, con $t \xrightarrow{\lambda_\rho}_p r$

Por hipótesis de inducción, $\exists \mathtt{u} \approx (\!|r|\!)/(\!|t|\!) \xrightarrow{py}{}^*_p \mathtt{u}$.

$$
\begin{aligned}
(\!|\pi^m\ t|\!) &= \mathtt{(lambda\ rho:\ (lambda\ i:\ build\_pair)(meas))(}(\!|t|\!)\mathtt{)} \\
&\xrightarrow{py}{}^*_p \mathtt{(lambda\ rho:\ (lambda\ i:\ build\_pair)(meas))(u)} \\
&\approx \mathtt{(lambda\ rho:\ (lambda\ i:\ build\_pair)(meas))(}(\!|r|\!)\mathtt{)} \\
&= (\!|\pi^m\ r|\!)
\end{aligned}
$$

Caso $\pi^m\ \rho^n \xrightarrow{\lambda_\rho}_{p_i} (\lceil i/2^{n-m} \rceil, \rho_i^n)$ con $\begin{cases} p_i = \mathsf{tr}(\pi^{n\dagger}_i\ \pi_i^n\ \rho^n) \\ \rho_i^n = (\pi_i^n\ \rho^n\ \pi^{n\dagger}_i)/p_i \end{cases}$

Recordando las definiciones auxiliares:

- `vn = rho.size()`
- `meas = rho.measure(0, 2, ..., vn)`.
- `build_pair = ((vi/(2**(vn/2-m)), m), from_int(vi, vn/2))`.

Tenemos que:

$(\!|\pi^m\ \rho^n|\!) =$ `(lambda rho: (lambda vi: build_pair)(meas))(`$(\!|\rho^n|\!)$`)`

Lema 4.22

$\xrightarrow{py}_1$ `(lambda vi: [`$(\!|\rho^n|\!)$`/rho]build_pair)`
`(`$(\!|\rho^n|\!)$`.measure(0, 2, ..., `$(\!|\rho^n|\!)$`.size()))`

Lema 4.23

$\xrightarrow{py}_1$ `(lambda vi: [`$(\!|\rho^n|\!)$`/rho]build_pair)`
`(`$(\!|\rho^n|\!)$`.measure(0, 2, ..., `$2n$`))`

$=$ `(lambda vi: [`$(\!|\rho^n|\!)$`/rho]build_pair)`
`(`$(\!|\rho^n|\!)$`.measure(0, 2, ..., `$2n$`))`

Lema 4.24

$\xrightarrow{py}_{p_i}$ `(lambda vi: [`$(\!|\rho^n|\!)$`/rho]build_pair)(i)`

$\xrightarrow{py}_1$ `[i/vi][`$(\!|\rho^n|\!)$`/rho]build_pair)`

$=$ `((i/(2**(`$(\!|\rho^n|\!)$`.size()/2-m)), m), from_int(i, `$(\!|\rho^n|\!)$`.size()/2))`

Lema 4.23

$\xrightarrow{py}_1$ `((i/(2**(`$2n$`/2-m)), m), from_int(i, `$(\!|\rho^n|\!)$`.size()/2))`

$\xrightarrow{py^*}_1$ `((`$\lceil i/2^{n-m} \rceil$`, m), from_int(i, `$(\!|\rho^n|\!)$`.size()/2))`

Lema 4.23

$\xrightarrow{py}_1$ `((`$\lceil i/2^{n-m} \rceil$`, m), from_int(i, `$2n$`/2))`

$\xrightarrow{py}_1$ `((`$\lceil i/2^{n-m} \rceil$`, m), from_int(i, `n`))`

$\xrightarrow{py}_1$ `((`$\lceil i/2^{n-m} \rceil$`, m), `$(\!|\pi_i^n|\!)$`)`

Lema 4.25

$\xrightarrow{py}_1$ `((`$\lceil i/2^{n-m} \rceil$`, m), `$(\!|\rho_i^n|\!)$`)`

$=$ $(\!|(\lceil i/2^{n-m} \rceil, \rho_i^n)|\!)$

□

4.5. Retracción de Python a λ^*_ρ

Si bien la formalización dada de Python es lo suficientemente poderosa como para poder definir una traducción inversa $f : \lambda_{py} \to \Lambda^*_\rho$, no es posible definirla de manera tal que la composición con la traducción dada previamente de la función identidad. Así como la traducción $\Lambda^*_\rho \to \lambda_{py}$ reduce a circuitos equivalentes debido a la purificación y construcción del estado, componer f con la traducción no genera la identidad, sino que genera la versión "purificada" del programa original.

Sin embargo, nos interesaría definir una inversa por izquierda (también conocida como retracción) para la traducción definida en la Definición 4.19, para probar que $(\!|\cdot|\!)$ no pierde información de los términos originales.

Teorema 4.27 (Inversa de la purificación)**.** La inversa de la función de purificación (Pur) está dada por:

$$\mathrm{Pur}^{-1}(|\phi\rangle) = \mathrm{pur}^{-1}(\mathrm{SWP}^{-1}\ |\phi\rangle)$$

Demostración. Sea ρ^n una matriz de densidad.

$$\begin{aligned}\mathrm{Pur}^{-1}(\mathrm{Pur}(\rho^n)) &= \mathrm{Pur}^{-1}(\mathrm{SWP}\ \mathrm{pur}(\rho^n)) = \mathrm{pur}^{-1}(\mathrm{SWP}^{-1}\ \mathrm{SWP}\ \mathrm{pur}(\rho^n)) \\ &= \mathrm{pur}^{-1}(\mathrm{pur}(\rho^n))\end{aligned}$$

Donde la última expresión equivale a ρ^n por el Teorema 4.6. □

Definición 4.28 (Inversa por izquierda de la traducción)**.** Se define inductivamente la inversa por izquierda de la traducción:

$$(\!|\cdot|\!)^{-1} : Im((\!|\cdot|\!)) \to \Lambda^*_\rho$$

$$\begin{aligned}(\!|\texttt{x}|\!)^{-1} &= x \\ (\!|\texttt{lambda x:t}|\!)^{-1} &= \lambda x.(\!|t|\!)^{-1} \\ (\!|\texttt{t}_1\texttt{(t}_2\texttt{)}|\!)^{-1} &= (\!|\texttt{t}_1|\!)^{-1}\ (\!|\texttt{t}_2|\!)^{-1} \\ (\!|\texttt{Circuit(I, [])}|\!)^{-1} &= \mathrm{Pur}^{-1}(\texttt{I}) \\ (\!|\texttt{t.gate(}G\texttt{, [}p\texttt{, }p+2\texttt{, ..., }p+2|G|\texttt{])}|\!)^{-1} &= G_p\ (\!|t|\!)^{-1} \\ (\!|\texttt{t}_1\texttt{.compose(t}_2\texttt{)}|\!)^{-1} &= (\!|\texttt{t}_1|\!)^{-1} \otimes (\!|\texttt{t}_2|\!)^{-1} \\ (\!|\texttt{((b, m), Circuit(I, [])}|\!)^{-1} &= (b^m, \mathrm{Pur}^{-1}(I)) \\ (\!|\texttt{letcase(r, [t}_0\texttt{, ..., t}_{2^m-1}\texttt{])}|\!)^{-1} &= \\ &\quad \mathsf{letcase}\ x = (\!|\texttt{r}|\!)^{-1}\ \mathrm{in}\ \{(\!|\texttt{t}_0|\!)^{-1}, \ldots, (\!|\texttt{t}_{2^m-1}|\!)^{-1}\}\end{aligned}$$

$$(\!|\texttt{(lambda rho: (lambda vi: build\_pair)(meas))(t)}|\!)^{-1} = \pi^m\ (\!|\texttt{t}|\!)^{-1}$$

donde

- `vn = rho.size()`
- `meas = rho.measure(0, 2, ..., vn)`.
- `build_pair = ((vi/(2**(vn/2-m)), m), from_int(vi, vn/2))`.
- $b \in \mathbb{N}_0, m, n \in \mathbb{N}$
- $G \in \{U^{\theta,\phi,\lambda}, UC^{\theta,\phi,\lambda}, \text{SWAP}, \text{CSWAP}\}$, con $\theta, \phi, \lambda \in \mathbb{R}$.

Teorema 4.29. Sea $t \in \Lambda^*_\rho$, entonces $(\!|(\!|t|\!)|\!)^{-1} = t$.

Demostración. Se procede por inducción en t.

Caso x

$(\!|(\!|x|\!)|\!)^{-1} = (\!|\texttt{x}|\!)^{-1} = x$

Caso $\lambda x.t$

$(\!|(\!|\lambda x.t|\!)|\!)^{-1} = (\!|\texttt{lambda x:}(\!|t|\!)|\!)^{-1} = \lambda x.(\!|(\!|t|\!)|\!)^{-1} \overset{\text{HI}}{=} \lambda x.t$

Caso $t_1\ t_2$

$(\!|(\!|t_1\ t_2|\!)|\!)^{-1} = (\!|(\!|t_1|\!)((\!|t_2|\!))|\!)^{-1} = (\!|(\!|t_1|\!)|\!)^{-1}\ (\!|(\!|t_2|\!)|\!)^{-1} \overset{\text{HI}}{=} t_1\ t_2$

Caso ρ^n

$(\!|(\!|\rho^n|\!)|\!)^{-1} = (\!|\texttt{Circuit(}\text{Pur}(\rho^n)\texttt{, [])}|\!)^{-1} = \text{Pur}^{-1}(\text{Pur}(\rho^n))$

Que por el Teorema 4.27 es igual a ρ^n.

Caso $G_p\ t$

$(\!|(\!|G_p\ t|\!)|\!)^{-1} = (\!|(\!|t|\!)\texttt{.gate(}G\texttt{, [}p\texttt{, }p+2\texttt{, ..., }p+2|G|\texttt{])}|\!)^{-1} = G_p\ (\!|(\!|t|\!)|\!)^{-1} \overset{\text{HI}}{=} G_p\ t$

Caso $t_1 \otimes t_2$

$(\!|(\!|t_1 \otimes t_2|\!)|\!)^{-1} = (\!|(\!|t_1|\!)\texttt{.compose(}(\!|t_2|\!)\texttt{)}|\!)^{-1} = (\!|(\!|t_1|\!)|\!)^{-1} \otimes (\!|(\!|t_2|\!)|\!)^{-1} \overset{\text{HI}}{=} t_1 \otimes t_2$

Caso (b^m, ρ^n)

$$\begin{aligned}(\!|(\!|(b^m, \rho^n)|\!)|\!)^{-1} &= (\!|\texttt{((b, m), Circuit(}\text{Pur}(\rho^n)\texttt{, []))}|\!)^{-1} \\ &= (b^m, \text{Pur}^{-1}(\text{Pur}(\rho^n)))\end{aligned}$$

Que por el Teorema 4.27 es igual a (b^m, ρ^n).

Caso $\mathsf{letcase}\ x = r\ \text{in}\ \{t_0, \ldots, t_{2^m-1}\}$

$$\begin{aligned}(\!(\!(\mathsf{letcase}\ x = r\ \text{in}\ \{t_0, \ldots, t_{2^m-1}\})\!)\!)^{-1} &= (\!(\texttt{letcase(}(\!(r)\!)\texttt{, [}(\!(t_0)\!)\texttt{, ..., }(\!(t_{2^m-1})\!)\texttt{])}\!)\!)^{-1} \\ &= \mathsf{letcase}\ x = (\!(\!(r)\!)\!)^{-1}\ \text{in}\ \{(\!(\!(t_0)\!)\!)^{-1}, \ldots, (\!(\!(t_{2^m-1})\!)\!)^{-1}\} \\ &\overset{\text{HI}}{=} \mathsf{letcase}\ x = r\ \text{in}\ \{t_0, \ldots, t_{2^m-1}\}\end{aligned}$$

Caso $\pi^m\ t$

$$\begin{aligned}(\!(\!(\pi^m\ t)\!)\!)^{-1} &= (\!(\texttt{(lambda rho: (lambda vi: build\_pair)(meas))(}(\!(t)\!)\texttt{)}\!)\!)^{-1} \\ &= \pi^m\ (\!(\!(t)\!)\!)^{-1} \overset{\text{HI}}{=} \pi^m\ t\end{aligned}$$

□

Capítulo 5

Implementación

La implementación del parsing, tipado y traducción se elaboró enteramente en *Haskell* bajo la herramienta *Stack* para gestionar el proyecto. En este capítulo comentamos sobre el papel de las mónadas, como se testeó el código, bibliotecas empleadas y estructuración de los módulos.

El código se encuentra disponible en `https://github.com/vmartinv/quantum-lambda-calculus-with-density-matrix`, bajo la licencia de código abierto "Licencia Pública General de GNU" (GNU GPL).

5.1. Uso de mónadas

La mónada de error (Except) aparece extensamente en casi todos los módulos para devolver errores en el tipado o parsing, entre otros casos.

En el algoritmo de Hindley se utilizaron las siguientes mónadas:

- La mónada de error (Except) para representar errores durante el tipado.
- Una mónada de estado (State), cuyo estado consiste en un entero y sirve para crear variables frescas.
- Una mónada de Reader que almacena el entorno de tipado (un *map* de variables a tipos).

Las mismas fueron combinadas utilizando los respectivos transformadores de mónadas incluidos en el lenguaje de Haskell estándar, resultando en los siguientes tipos:

```
newtype TypeEnv = TypeEnv (M.Map T.Text QType)
type ExceptInfer = Except TypeError
type TypeState = Int
```

```
type HindleyM a = ReaderT
                      TypeEnv -- Entorno de tipado
                      (StateT
                          TypeState  -- Estado del tipado
                          ExceptInfer) -- Errores
                      a -- Valor de retorno de la función
```

5.2. Testing

Haciendo uso de las bibliotecas disponibles (listadas en la Sección 5.3) se escribieron una diversa variedad de tests para corroborar la implementación. Estos pueden dividirse en cuatro tipos:

Tests unitarios Verifican módulos individuales del proyecto mediante casos de entrada individuales.

Tests automáticos Dada una propiedad sobre una función a testear se derivan muchos casos de entrada.

Ejemplo 5.1. Para testar la función $\log_2$, una propiedad que podemos probar es: $\log_2(2^x) = x$. La biblioteca QuickCheck recibe esta propiedad y testea x para muchos valores posibles (intentando incluir casos bordes como 0, 1, 0.5, etc.).

Tests de integración Comprueban la validez del resultado ejecutando varias etapas del programa en cada test, por ejemplo un test puede tomar una expresión de Λ^*_ρ, parsearla, tiparla, traducirla y verificar que dio el resultado esperado.

Tests de extremo a extremo Estos tests abarcan todo el proceso de la traducción y su correcta ejecución en Python. El texto con la expresión de Λ^*_ρ pasa por el lexer, el parser, el algoritmo de tipado, la traducción y finalmente la ejecución en Python. El resultado de Python es procesado y validado.

5.2.1. Cobertura de pruebas

Si bien es difícil de medir objetivamente cuan bueno es un conjunto de pruebas, una cosa es segura: tal conjunto debe ejecutar toda la lógica del código. Si este no es el caso si hay fallas en lugares nunca ejecutados resultaría imposible detectarlos con tal conjunto. Así es como para guiar la creación de pruebas, resulta muy útil saber que partes del código son ejecutadas durante la ejecución de todos los tests. Afortunadamente, Haskell posee una poderosa herramienta para medir esto llamada *Haskell Program Coverage*(HPC), que fue definida en [GR07].

El enfoque tradicional para calcular la cobertura es inadecuado para los lenguajes

funcionales *lazy* como Haskell, donde cada expresión es evaluada solo cuando es necesario. A diferencia de cualquier otra herramienta de cobertura conocida, HPC detecta cualquier expresión, sin importar cuan pequeña sea, que nunca es evaluada en la ejecución de un programa.

Ejemplo 5.2. Si en el código tenemos una expresión similar a esta:

```
if (condicion1 && condicion2 && condicion3)
    then valor1
    else valor2
```

HPC puede detectar si cada una de las tres condiciones fue evaluada. Por ejemplo puede informar si `condicion1` es siempre verdadera para todos los casos, lo que causa que `condicion2` y `condicion3` nunca se evalúen. En estas condiciones el programador puede agregar tests que hagan la `condicion1` falsa para poder testear las otras condiciones.

Gracias a la extensa cantidad de tests se logró una cobertura del 100 % de todas las expresiones del código (quitando el módulo autogenerado del parser y el lexer).

5.3. Bibliotecas y programas auxiliares

Se listan a continuación las distintas herramientas empleadas para elaborar el compilador para verificar la correctitud de la implementación.

GHC El Glasgow Haskell Compiler es un compilador nativo de código libre para Haskell.

Stack Programa multiplataforma para desarrollar proyectos en Haskell similar a 'cmake' para C.

Happy Sistema generador de parsers para Haskell, similar a la herramienta 'yacc' para C.

Alex Herramienta para generar analizadores léxicos en Haskell. Es similar a la herramienta 'lex' o 'flex' para C/C++.

matrix, hmatrix, hmatrix-glpk Diferentes bibliotecas de álgebra lineal que implementan vectores, matrices, descomposición espectral y Simplex, entre otras funcionalidades.

Prettyprinter Esta biblioteca provee un DSL embebido para dar formato a texto de una manera flexible y conveniente.

Repline Un wrapper de Haskeline para crear interfaces similares a GHCi (la consola interactiva de GHC).

optparse-applicative Biblioteca que provee un parser para recibir y procesar los argumentos de la línea de comandos.

regex-tdfa Un motor de expresiones regulares que resulta útil para comprobar los resultados en los tests.

Tasty, QuickCheck, SmallCheck Bibliotecas utilizadas para realizar testing automático y manual.

aeson Biblioteca que provee un parser de JSON, aparece en los tests para verificar el output de los programas de Python.

5.4. Estructuración del código

Como cualquier proyecto de Haskell la aplicación se organiza en módulos. Se agruparon los módulos en diversas carpetas:

Parsing se compone de un módulo para definir el tipo que representan las expresiones de Λ^*_ρ, y módulos para el lexer y el parser que juntos pueden procesar el texto de entrada.

Python se compone de un módulo para definir el tipo que representa las expresiones de Python, y un módulo para convertir dichas expresiones a texto legible.

Translation contiene el módulo que implementa la traducción, las subpartes más complejas de la misma (como purificación) se implementan en módulos separados.

Typing contiene el conjunto de módulos que declaran el sistema de tipos junto con el algoritmo de tipado, incluye la estructura de datos de los tipos de λ^*_ρ, el algoritmo de Robinson y el de Hindley, declaración de errores de tipado, etc.

El archivo `Compiler.hs` utiliza todas estas subpartes para generar la función que dado un texto de entrada, parsea, realiza el tipado y devuelve la traducción. Finalmente, con esta función `REPL.hs` implementa una consola interactiva.

Todos los archivos relevantes del código se encuentran listados a continuación.

`app`

- `Main.hs` Punto de entrada del ejecutable.
- `REPL.hs` Implementa una consola interactiva.

src/Parsing

- **LamRhoLexer.x** Lexer de Alex para las expresiones de λ_ρ^*.
- **LamRhoParser.y** Parser de Happy para las expresiones de λ_ρ^*.
- **LamRhoExp.hs** Define el tipo de las expresiones de λ_ρ^*.

src/Python

- **PyExp.hs** Define el tipo de las expresiones de Python.
- **PyRender.hs** Contiene la lógica para convertir expresiones de Python a texto legible.

src/Translation

- **DensMat.hs** Operaciones sobre matrices de densidad.
- **Purification.hs** Implementa el algoritmo de purificación.
- **Translation.hs** Contiene la traducción de λ_ρ^* a Qiskit en sí.

src/Typing

- **GateChecker.hs** Módulo que verifica que las compuertas son válidas.
- **Hindley.hs** Implementa el algoritmo de Hindley.
- **MatrixChecker.hs** Módulo que verifica que las matrices de densidad son válidas.
- **QType.hs** Especifica la estructura usada para los tipos de λ_ρ^*.
- **Robinson.hs** Implementa el algoritmo modificado de Robinson.
- **Subst.hs** Especifica una sustitución y como aplicarla a los distintos tipos usados durante el tipado.
- **TypeChecker.hs** El módulo principal que hace el tipado (utilizando los otros módulos).
- **TypeEq.hs** Define la estructura usada para las ecuaciones de tipo.

src/Compiler.hs Define el compilador combinando el parser, con el tipado y la traducción.

src/CompilerError.hs Define los errores posibles durante el parseado, tipado y traducción.

src/Utils.hs Contiene la función $\log_2$.

test Contiene todos los tests siguiendo la misma estructura que **src**.

- **Parsing** Tests para el parser.
- **Translation** Tests de la traducción.
- **Typing** Tests para el tipado.
- **CompilerTests.hs** Tests de integración que combinan el parser, tipado y traducción.
- **PythonTests.hs** Tests de extremo a extremo que ejecutan la traducción utilizando Python y verifican el resultado.
- **TestSpec.hs** Importa y lista todos los tests que son ejecutados con 'stack test'.
- **TestUtils.hs** Contiene funciones auxiliares usadas por los tests.

package.yaml Archivo de Stack que especifica las dependencias, metadata y como se compone el proyecto.

Capítulo 6

Conclusiones

En el transcurso de esta tesina, hemos explorado la intersección entre la computación cuántica y el cálculo lambda. Nuestro objetivo principal fue traducir una variante del cálculo lambda (λ_ρ) a Python, lo que permite por medio de la biblioteca Qiskit la ejecución de expresiones de λ_ρ en hardware cuántico real.

6.1. Desafíos técnicos resueltos

En primera instancia se presentó un algoritmo de tipado para λ_ρ en dos partes basado en el sistema de inferencia de tipos de Hindley y el algoritmo de unificación de Robinson. Se adaptaron ambas partes para poder resolver inecuaciones del tipo $A \leq B$, que surgen debido a que λ_ρ posee enteros en su tipado. Luego se probó la correctitud y completitud de la primera etapa del tipado. Este resultado permitió demostrar que el tipado pertenece a la clase de complejidad NP-completo, mostrando que es equivalente al problema de cubrimiento por vértices mínimo.

Ya teniendo la expresión tipada, para poder dar la traducción primero tuvimos que especificar los lenguajes involucrados y adaptarlos a este trabajo. Presentamos una simplificación del lenguaje de Python y modificamos ligeramente λ_ρ para definir λ_ρ^*, que posee un conjunto limitado de compuertas, restricciones en la medición y una estrategia de reducción. Se necesitó codificar estados cuánticos mixtos en estados cuánticos puros utilizando el método de la purificación. A su vez se aplicó una novedosa técnica de permutación de qubits para poder traducir la composición de manera eficiente y concisa. Esta técnica se basa en colocar los qubits adicionales en las posiciones pares al realizar la purificación.

Con la traducción definida, se probó formalmente que esta traducción es correcta, es decir que preserva las reglas de reescritura y a su vez que tiene una retracción por izquierda.

Seguidamente, se implementaron estas ideas en el lenguaje de programación funcional *Haskell*, incluyendo un extenso conjunto de tests unitarios, automáticos, de integración y de extremo a extremo para corroborar su correctitud.

6.2. Implicaciones prácticas y teóricas

En el ámbito práctico, nuestros hallazgos sugieren que es posible ejecutar en hardware existente un lenguaje cuántico y funcional como lo es λ_ρ con pequeñas modificaciones. Desde una perspectiva teórica, este trabajo contribuye a la comprensión de cómo las abstracciones de la programación funcional se pueden utilizar en el contexto cuántico.

También descubrimos que agregar enteros a un sistema de tipos puede aumentar significativamente la complejidad de llevar a cabo inferencia de tipos sobre las expresiones del lenguaje.

En resumen, al demostrar la viabilidad de ejecutar un lenguaje que inicialmente era puramente teórico en hardware cuántico real, hemos abierto nuevas vías de investigación y desarrollo en estas áreas. Esto marca un paso en la exploración de cómo la programación funcional puede influir en la forma en que abordamos la computación cuántica. Al ser un campo sumamente novedoso, la intersección entre la computación cuántica y la programación funcional promete desafíos y descubrimientos emocionantes para el futuro.

6.3. Trabajo a futuro

A lo largo de este trabajo identificamos desafíos importantes que pueden abordarse en investigaciones futuras. Más allá de estos desafíos también reconocemos que existe un vasto espacio para explorar dado la corta edad de este campo.

6.3.1. Condicionales clásicos en QASM 3.0

En la Sección 4.1 se presentó la opción de ejecutar lógica clásica dentro de la computadora cuántica. Esto solo es posible si la arquitectura lo permite. Si tuviéramos estas instrucciones condicionales o de salto sería posible ejecutar programas de λ_ρ completamente en estas máquinas, sin necesidad de incluir la lógica clásica en Python. A pesar de que algunas máquinas cuánticas tienen esta capacidad, por el momento no es posible hacerlo en Qiskit debido a que internamente Qiskit compila los circuitos al lenguaje ensamblador cuántico estandarizado OpenQASM en su versión 2.0 [McK+18] y el mismo carece de tales instrucciones.

El lenguaje ensamblador cuántico abierto (OpenQASM 2) se propuso como un lenguaje de programación imperativo para circuitos cuánticos. En principio, cualquier cálculo cuántico podría describirse utilizando OpenQASM 2. Sin embargo, no puede describir cualquier cálculo clásico.

Recientemente en septiembre del 2022, se ha publicado la versión 3 de QASM [Cro+22]. Tan esperadamente, esta nueva iteración agrega soporte para control de flujo arbitrario, así como también llamadas a funciones clásicas externas, entre otras funcionalidades. Esto permitiría tener control clásico definido en la propia computadora cuántica. El soporte de Qiskit para esta nueva versión de OpenQASM está mejorando día a día.

Es con este nuevo avance de la tecnología que resultaría plausible reformular la traducción de manera que se incluya toda la lógica en la computadora cuántica y así solucionar las limitaciones especificadas en la Sección 4.1.

6.3.2. Correctitud del algoritmo de Robinson para λ_ρ

En la Sección 2.3 se presentó un algoritmo para resolver las ecuaciones del tipado, sin embargo, no se demostró su correctitud o completitud. Para probar su correctitud es necesario demostrar que:

- El algoritmo termina.
- Si el algoritmo falla el sistema no tiene solución (por lo que la expresión siendo tipada es inválida).
- La solución satisface las ecuaciones.

Por otro lado, para probar completitud se debería demostrar que si existe una solución a las ecuaciones, entonces el algoritmo de Robinson la encuentra. Estas pruebas no son fáciles, pero a simple vista parecen posibles.

6.3.3. Profundización en la resolución de las ecuaciones de Robinson

La versión modificada de Robinson provista en la Sección 2.3 no especifica que algoritmo utilizar para resolver el sistema de ecuaciones sobre enteros generados. En este trabajo se optó por aplicar Simplex porque la minimización de los tamaños de los tipos resulta deseable y las bibliotecas que lo implementan son muy accesibles. Sin embargo, existen dos cuestiones que no fueron completamente investigadas:

- Realizar una minimización no es necesaria: un algoritmo que encuentre al menos un tipo es satisfactorio para tipar una expresión. Encontrar y enunciar tal algoritmo resulta interesante desde el punto de vista de reducir la complejidad

del problema. Debido a la naturaleza de los sistemas generados (es un sistema sobre los naturales y las matrices contienen solo ceros y unos), se estipula que una solución podría encontrarse en tiempo polinomial.

- No se implementó el algoritmo de Ramificación y Poda necesario para resolver los casos cuando Simplex no devuelve una solución entera al sistema de ecuaciones. En tal caso el tipado falla, lo cual sería correcto si efectivamente no hay una solución entera, pero no lo es en el caso de que simplemente Simplex haya encontrado una solución no entera más óptima que la entera.

6.3.4. Construcción y demostración de la inversa

En la Sección 4.5 se dio una inversa por izquierda, pero también se mencionó que es posible definir una traducción inversa general para todas las expresiones de Python, es decir una función $f : \lambda_{py} \to \Lambda_{\rho}^{*}$. Esto es definitivamente posible, no obstante cabe resaltar que la composición con la traducción dada en este trabajo no da la identidad, sino la versión "purificada" del programa original. Resultaría entonces factible plantear esta inversa y probar tal propiedad.

Bibliografía

[AD05] Pablo Arrighi y Gilles Dowek. «Lineal: A linear-algebraic Lambda-calculus». En: *Logical Methods in Computer Science* Volume 13, Issue 1 (feb. de 2005). DOI: 10.23638/LMCS-13(1:8)2017.

[ADV17] Pablo Arrighi, Alejandro Díaz-Caro y Benoît Valiron. «The vectorial λ-calculus». En: *Information and Computation* 254 (2017), págs. 105-139. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2017.04.001. URL: https://www.sciencedirect.com/science/article/pii/S0890540117300482.

[AG05] T. Altenkirch y J. Grattage. «A Functional Quantum Programming Language». En: *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. IEEE, 2005. DOI: 10.1109/lics.2005.1. URL: https://doi.org/10.1109%2Flics.2005.1.

[Asf+21] Abraham Asfaw et al. *Learn Quantum Computation Using Qiskit*. Accedido: 01/10/2023. 2021. URL: https://qiskit.org/textbook/preface.html.

[Bor19] Agustín Borgna. «Simulación del lambda cálculo de matrices de densidad en el lambda cálculo cuántico de Selinger y Valiron». Director: Alejandro Díaz-Caro. Tesis de Licenciado en Ciencias de la Computación. Buenos Aireas: Universidad de Buenos Aires Facultad de Ciencias Exactas y Naturales Departamento de Computación, 2019.

[Cro+22] Andrew Cross et al. «OpenQASM 3: A Broader and Deeper Quantum Assembly Language». En: *ACM Transactions on Quantum Computing* 3.3 (sep. de 2022), págs. 1-50. DOI: 10.1145/3505636. URL: https://doi.org/10.1145%2F3505636.

[DD17] Alejandro Díaz-Caro y Gilles Dowek. «Typing quantum superpositions and projective measurements». En: *Lecture Notes in Computer Science* (sep. de 2017).

[Dev21] Qiskit Developers. *Qiskit: An Open-source Framework for Quantum Computing*. Accedido: 01/10/2023. 2021. URL: https://qiskit.org/.

[Día17] Alejandro Díaz-Caro. «A lambda calculus for density matrices with classical and probabilistic controls». En: *Programming Languages and Systems (APLAS 2017)*. Ed. por Bor-Yuh Evan Chang. Vol. 10695. Lecture Notes in Computer Science. Springer, Cham, 2017, págs. 448-467.

[DP06] Ellie D'Hondt y Prakash Panangaden. «Quantum weakest preconditions». En: *Mathematical Structures in Computer Science* 16.3 (2006), págs. 429-451. DOI: 10.1017/S0960129506005251.

[FDY11] Yuan Feng, Runyao Duan y Mingsheng Ying. «Bisimulation for Quantum Processes». En: *SIGPLAN Not.* 46.1 (ene. de 2011), págs. 523-534. ISSN: 0362-1340. DOI: 10.1145/1925844.1926446. URL: https://doi.org/10.1145/1925844.1926446.

[Fou21] Python Software Foundation. *Python Language Reference, version 3.-11.5*. Accedido: 01/10/2023. 2021. URL: https://docs.python.org/3/.

[FYY13] Yuan Feng, Nengkun Yu y Mingsheng Ying. «Model checking quantum Markov chains». En: *Journal of Computer and System Sciences* 79.7 (2013), págs. 1181-1198. ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2013.04.002. URL: https://www.sciencedirect.com/science/article/pii/S0022000013000780.

[Gil11] Jean-Jacques Lévy Gilles Dowek. *Introduction to the Theory of Programming Languages*. Springer London, 2011. DOI: 10.1007/978-0-85729-076-2.

[GR07] Andy Gill y Colin Runciman. «Haskell program coverage». En: sep. de 2007, págs. 1-12. DOI: 10.1145/1291201.1291203.

[Gre+13] Alexander S. Green et al. «Quipper». En: *ACM SIGPLAN Notices* 48.6 (jun. de 2013), págs. 333-342. DOI: 10.1145/2499370.2462177. URL: https://doi.org/10.1145%2F2499370.2462177.

[HMU02] John E. Hopcroft, Rajeev Motwani y Jeffrey D. Ullman. *Introducción a la teoría de autómatas, lenguajes y computación / John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman*. spa. Madrid: Addison Wesley, 2002. ISBN: 8478290567.

[IBM21] IBM. *IBM Quantum*. Accedido: 01/10/2023. 2021. URL: https://quantum-computing.ibm.com/.

[Kar72] Richard M. Karp. «Reducibility among Combinatorial Problems». En: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathema-*

tical Sciences Department. Ed. por Raymond E. Miller, James W. Thatcher y Jean D. Bohlinger. Boston, MA: Springer US, 1972, págs. 85-103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[KC91] David R. Kincaid y E. Ward Cheney. *Numerical Analysis: Mathematics of Scientific Computing*. Pacific Grove, Calif.: Brooks/Cole, 1991. ISBN: 9780534130145.

[Kni96] E. Knill. *Conventions for Quantum Pseudocode*. Accedido: 01/10/2023. University of North Texas Libraries, UNT Digital Library, jun. de 1996. DOI: 10.2172/366453. URL: https://digital.library.unt.edu/ark:/67531/metadc687305/.

[Kuh11] D. Kuhlman. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. Platypus Global Media, 2011. ISBN: 9780984221233. URL: https://books.google.co.uk/books?id=1FL-ygAACAAJ.

[McK+18] David C. McKay et al. «Qiskit Backend Specifications for OpenQASM and OpenPulse Experiments». En: *IBM Research* (2018). DOI: 10.48550/ARXIV.1809.03452. URL: https://arxiv.org/abs/1809.03452.

[NC10] Michael A. Nielsen e Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. DOI: 10.1017/CBO9780511976667.

[New03] Jeff Newbern. *All about monads*. Accedido: 01/10/2023. 2003. URL: https://wiki.haskell.org/All_About_Monads.

[PSV14] Michele Pagani, Peter Selinger y Benoît Valiron. «Applying Quantitative Semantics to Higher-Order Quantum Computing». En: *SIGPLAN Not.* 49.1 (ene. de 2014), págs. 647-658. ISSN: 0362-1340. DOI: 10.1145/2578855.2535879. URL: https://doi.org/10.1145/2578855.2535879.

[Sel04] Peter Selinger. «Towards a quantum programming language». En: *Mathematical Structures in Computer Science* 14.4 (2004), págs. 527-586.

[SV05] Peter Selinger y Benoît Valiron. «A Lambda Calculus for Quantum Computation with Classical Control». En: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2005, págs. 354-368. DOI: 10.1007/11417170_26. URL: https://doi.org/10.1007%2F11417170_26.

[Ton04] André van Tonder. «A Lambda Calculus for Quantum Computation». En: *SIAM Journal on Computing* 33.5 (ene. de 2004), págs. 1109-1135. DOI: 10.1137/s0097539703432165. URL: https://doi.org/10.1137%2Fs0097539703432165.

[Vaz08] Vijay V. Vazirani. «Approximation Algorithms». En: *Combinatorial Optimization: Theory and Algorithms*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, págs. 393-437. ISBN: 978-3-540-71844-4. DOI: 10.1007/978-3-540-71844-4_16. URL: https://doi.org/10.1007/978-3-540-71844-4_16.

[Win22] W.L. Winston. *Operations Research: Applications and Algorithms*. Cengage Learning, 2022. ISBN: 9780357907818. URL: https://books.google.co.uk/books?id=Y9NYEAAAQBAJ.

[Yin16] Mingsheng Ying. *Foundations of Quantum Programming*. Retrieved October 1, 2023. Morgan Kaufmann an imprint of Elsevier, 2016. URL: http://site.ebrary.com/id/11187870.

[YM08] Noson S. Yanofsky y Mirco A. Mannucci. *Quantum Computing for Computer Scientists*. Cambridge University Press, 2008. DOI: 10.1017/CBO9780511813887.

[YYW17] Mingsheng Ying, Shenggang Ying y Xiaodi Wu. «Invariants of Quantum Programs: Characterisations and Generation». En: *SIGPLAN Not.* 52.1 (ene. de 2017), págs. 818-832. ISSN: 0362-1340. DOI: 10.1145/3093333.3009840. URL: https://doi.org/10.1145/3093333.3009840.

[Zor16] Margherita Zorzi. «On quantum lambda calculi: a foundational perspective». En: *Mathematical Structures in Computer Science* 26.7 (oct. de 2016), págs. 1107-1195. DOI: 10.1017/S0960129514000425.