Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

LTIQ
Un Calcolo Lineare per Microsoft Q#

Relatore:
Chiar.mo Prof.
Ugo Dal Lago

Presentata da:
Andrea Guerra

Correlatore:
Dott.
Francesco Gavazzo

Sessione
III Anno Accademico 2019/2020

Indice

Sommario

Il teorema di non duplicazione della meccanica quantistica afferma che uno stato quantistico non possa mai essere duplicato o cancellato. Diversi linguaggi di programmazione quantistici, ovvero basati sui principi della meccanica quantistica, garantiscono a tempo di compilazione, grazie ai sistemi di tipi lineari, che i loro programmi non violino mai il teorema di non duplicazione. Non è questo il caso del *linguaggio specifico di dominio* Microsoft Q#: l'esecuzione di un programma Q# può portare ad errori a tempo di esecuzione dovuti a violazioni del teorema di non duplicazione causando uno spreco di risorse quantistiche. Per ovviare a questo problema, in questo lavoro di tesi viene formalizzato il core computazionale di Q# introducendo il calcolo fondazionale *Linearly Typed Idealized Q#* (abbreviato in LTIQ). Dopo aver provvisto LTIQ di una sintassi e una semantica operazionale viene definito un sistema di tipi lineare per esso. Il risultato principale di questo elaborato, assieme alla definizione del sistema di tipi, è la dimostrazione del teorema di sicurezza rispetto ai tipi (*type safety*). Da questo segue che in un programma ben tipato nessun dato tipato linearmente verrà duplicato o cancellato durante l'esecuzione del programma. Avendo i dati quantistici tipo lineare, nessun programma ben tipato violerà quindi il teorema di non duplicazione durante la sua esecuzione.

Capitolo 1

Introduzione

La computazione quantistica (*Quantum Computing*) (Coutinho, 2009; Nielsen & Chuang, 2016), ovvero la teoria della computazione basata sui principi della meccanica quantistica, è oggi una delle aree emergenti dell'informatica, sia applicata che teorica. Infatti, i computer quantistici stanno superando i computer classici in alcuni compiti cruciali, come la fattorizzazione in numeri primi (Shor, 1997) e la ricerca all'interno di strutture dati (Grover, 1996), al punto che è stato recentemente coniato il termine di supremazia quantistica (*quantum supremacy*) (Preskill, 2012) per riferirsi al superamento delle performance dei computer quantistici rispetto a quelli classici.

L'idea fondamentale della computazione quantistica è quello di codificare l'informazione in oggetti fisici governati dalle leggi della meccanica quantistica. Come nella teoria della computazione classica la più piccola unità di informazione è il *bit*, nella teoria della computazione quantistica l'unità d'informazione di base è il *bit quantistico* (*quantum bit*) — detto anche *qubit* o *qbit* — ovvero un qualsiasi sistema meccanico-quantistico a due stati. Il passaggio da bit a bit quantistici comporta radicali cambiamenti circa la natura stessa della manipolazione dell'informazione: infatti, mentre un bit può essere duplicato a piacimento, non è questo il caso per un bit quantistico, cosicchè l'informazione quantistica risulta a tutti gli effetti una risorsa fisica (*data as resources*). Questo è il contenuto del noto teorema di *non duplicazione* (*no cloning*). Nonostante ciò la computazione quantistica porta numerosi vantaggi rispetto alla computazione classica grazie a fenonemi come l'*entanglement* e la *sovrapposizione quantistica* (*quantum superposition*).

L'avvento della computazione quantistica pone delle nuove domande non solo per quanto riguarda la costruzione dei calcolatori quantistici e lo sviluppo di algoritmi quantistici, ma anche nel campo dei linguaggi di programmazione. Infatti la natura quantistica dei nuovi calcolatori rende necessaria l'introduzione di linguaggi che consentano la programmazione di tali calcolatori ad alto livello, così da ottenere maggiori garanzie in termini di correttezza ed efficienza. Per questo motivo, diversi gruppi di ricerca, sia in accademia che in industria, studiano e sviluppano linguaggi e framework

di programmazione quantistici, come ad esempio il linguaggio Q# di Microsoft (Svore et al., 2018), i framework Python QISKit sviluppato da IBM (McKay et al., 2018) e Cirq realizzato da Google (C. & contributors, 2017), e il linguaggio di ricerca Quipper (Green, Lumsdaine, Ross, Selinger, & Valiron, 2013b, 2013a).

L'introduzione di questi linguaggi solleva nuovi e non banali interrogativi: come compilare ed ottimizzare programmi quantistici? Come verificarli e testarli? Come capirne la semantica? Quali proprietà un programma quantistico deve rispettare per avere un comportamento corretto? In questa tesi ci concentriamo soprattutto su quest'ultima domanda, muovendo dall'osservazione che molti dei linguaggi di programmazione quantistici esistenti (tra cui tutti quelli sopra menzionati) non garantiscono il rispetto del teorema di non duplicazione, lasciando quindi al programmatore il compito di assicurarsi che i dati quantistici non vengano duplicati (o cancellati) durante l'esecuzione di un programma. In questa tesi viene sviluppato un sistema di tipi *lineare* (Wadler, 1990; Girard, 1987; Walker, 2004) per un sottoinsieme del linguaggio di programmazione Microsoft Q# che permette di identificare violazioni del teorema di non duplicazione a tempo di compilazione. Il resto di questo capitolo è dedicato alla spiegazione del problema affrontato e della soluzione proposta.

Microsoft Q# e il Teorema di Non Duplicazione Il linguaggio di programmazione Microsoft Q# (soltanto Q# d'ora in poi) è un *linguaggio specifico di dominio* che permette lo sviluppo e l'esecuzione di algoritmi quantistici. È realizzato con lo scopo di fornire al programmatore una visione ad alto livello nello sviluppo di programmi quantistici. Inoltre i programmi sviluppati in Q# possono essere eseguiti su hardware quantistico. Lo stato quantistico di un programma Q# è nascosto al programmatore, il linguaggio infatti descrive come un computer classico interagisce con i qbit e non permette un'introspezione diretta dello stato dei qbit e delle altre proprietà della meccanica quantistica.

I costrutti di Q# permettono la creazione, la manipolazione e la trasformazione di qbit agendo su variabili che fungono da puntatori opachi ai qbit fisici veri e propri. In particolare, una volta creato un qbit, questo può essere passato a funzioni. In un certo senso, è tutto ciò che è possibile fare con un qbit; ogni operazione intrinseca sullo stato di un qbit è effettuata attraverso l'utilizzo di funzioni *built-in* del linguaggio. Queste funzioni agiscono come gate (o operatori) unitari e modificano lo stato dei qbit presi in input. Ad esempio, il gate di Hadamard (*Hadamard gate*) prende in input un qbit e crea lo stato di *sovrapposizione* per esso. Pertanto, in Q#, è possibile agire sui qbit modificandone lo stato interno e quindi provocando dei *side-effect* ogni qualvolta è applicata una funzione *built-in* del linguaggio che opera su qbit. Dato un qbit, Q# offre un costrutto specifico per misurare il qbit: una volta misurato, lo stato quantistico del qbit collassa in uno stato classico con una certa probabilità, rendendo quindi l'esecuzione di un programma Q# probabilistica. Un programma Q#

è strutturato come una sequenza di *namespace*, convenzionalmente divisi su più file. Ogni *namespace* è a sua volta una sequenza di dichiarazioni di *callable* e di tipi definiti dall'utente. Un *callable* può essere in Q# una funzione o un'operazione. La loro distinzione è dovuta al fatto che una funzione è intesa nel senso classico del termine: non è possibile al suo interno creare o manipolare *qbit*, mentre all'interno di un'operazione è possibile lavorare sui qbit.

Un'altra caratteristica importante di Q# è la possibilità di creare delle *variabili mutabili*, ovvero delle variabili il cui contenuto possa essere modificato nel tempo. Questa è una caratteristica tipicamente imperativa dei linguaggi di programmazione che rende il modello di computazione di Q# multi-paradigma. In particolare, attraverso il costrutto sintattico *mutable* è possibile creare un legame mutabile tra variabili e valori. Una volta definito questo legame, in un qualsiasi momento della computazione possiamo modificarlo scegliendo di legare un nuovo valore alle variabili mutabili. Infine Q# permette l'utilizzo di funzioni come dati, fornendo quindi la possibilità di scrivere funzioni all'ordine superiore e di effettuare applicazioni parziali di funzione. Inoltre, Q# tratta tutte le variabili in modo *immutabile* (e non è quindi possibile modificarne lo stato) ad eccezione di quelle che si riferiscono a qbit e di quelle esplicitamente dichiarate mutabili. In questo senso il modello di Q# è multi-paradigma, mescolando caratteristiche dei lingugaggi imperativi e funzionali. Di seguito mostriamo degli esempi di programmi scritti in Q# che utilizzano alcuni dei costrutti sopra descritti.

```
1  namespace quantumGenerator {
2    open Microsoft.Quantum.Canon;
3    open Microsoft.Quantum.Intrinsic;
4    open Microsoft.Quantum.Diagnostics;
5
6    operation sampleBit():Result {
7      using (x = Qubit()) {
8          H(x);
9          return M(x);
10     }
11   }
12
13   @EntryPoint()
14   operation main(): Unit {
15     mutable (a, b) = (Zero, Zero);
16     set (a, b) = (sampleBit(), sampleBit());
17   }
18 }
```

Figura 1.1: Implementazione di un generatore di bit casuali in Q#

In Figura 1.1, definiamo l'operazione *sampleBit* il cui corpo crea un qbit nello stato classico zero, lo mette in uno stato di *sovrapposizione* attraverso l'applicazione del gate Hadamard ed infine lo misura restituendo con la stessa probabilità come valore di ritorno la costante *Zero* (che denota un bit classico dal valore 0) o la costante *One* (che denota un bit classico dal valore 1).

Si noti che l'operazione *main* è preceduta dall'annotazione *@EntryPoint*, la quale indica che la funzione successivamente definita è il punto di partenza dell'intero programma. All'interno del corpo della funzione *main*, è creato un legame mutabile tra le variabili a e b con la costante di linguaggio *Zero*. Il legame per a e b è modificato nella linea 16 attraverso il costrutto *set*: entrambe le variabili assumeranno un nuovo valore risultante dalla chiamata dell'operazione *sampleBit*().

```
1  namespace entanglement {
2    open Microsoft.Quantum.Canon;
3    open Microsoft.Quantum.Intrinsic;
4    open Microsoft.Quantum.Diagnostics;
5
6    operation entangle(x:Qubit, y:Qubit):Unit {
7      H(x);
8      CNOT(x, y);
9    }
10
11
12   @EntryPoint()
13   operation main(): Unit {
14     using (x = Qubit()) {
15         let y = x;
16         entangle(x, y);
17     }
18   }
19 }
```

Figura 1.2: Esempio di violazione del teorema di non duplicazione

In Figura 1.2 mostriamo un esempio di programma Q# la cui esecuzione, come stiamo per vedere, porterà ad un errore: la funzione *entangle* prende in input una coppia di qbit e crea uno stato di *entanglement* per essi. Osservando la funzione *main* ci si accorge però che la variabile y, in realtà, è legata alla variabile x, e di conseguenza la chiamata di operazione $entangle(x, y)$ prova a duplicare un qbit, violando così il teorema di non duplicazione. Questo programma genera un errore a tempo di esecuzione (*run-time*) nel momento in cui viene eseguito lo statement $CNOT(x, y)$, e quindi solamente dopo aver creato il qbit assegnato alla variabile x, risultando così in uno

spreco di risorse quantistiche. Allo stato attuale, il compilatore di Q# non offre nessuna garanzia a fronte di problemi di questo tipo. Come anticipato, il programmatore dovrà tenere traccia di tutti i legami creati per i qbit e assicurarsi che il programma non provi a duplicare nessuno di essi.

È importante notare l'interesse pratico, oltre che teorico, di questo problema: infatti eseguire un programma quantistico è dispendioso in termini di risorse. Nello specifico, ogni operazione di allocazione di qbit su hardware quantistico è un'operazione costosa ed è quindi fondamentale avere un modo per prevenire errori di duplicazione di dati quantistici *prima* che i programmi vengano eseguiti (e quindi *prima* che le risorse vengano allocate). Per far ciò è necessario disporre di un meccanismo che sia in grado di predire la presenza di questi errori in un programma esplorandone solamente la struttura sintattica. In questo modo un programma che violi il teorema di non duplicazione non verrà eseguito, evitando così uno spreco di risorse. In questa tesi andiamo a definire un tale meccanismo sviluppando un sistema di tipi *lineare* per un frammento del linguaggio Q# (sostanzialmente, la restrizione di Q# al prim'ordine).

Sistemi di Tipi Lineari I sistemi di tipi lineare (Wadler, 1990; Walker, 2004) sono sistemi di tipi (Cardelli, 1997; Pierce, 2002) basati (tramite la corrispondenza di Curry-Howard) (Sørensen & Urzyczyn, 2006)) sulla logica lineare (Girard, 1987; Wadler, 1991). L'idea centrale di tali sistemi è quella di trattare i dati come risorse fisiche, impedendone quindi la duplicazione (o la cancellazione) arbitraria: ogni variabile (di un programma) di tipo lineare dovrà essere usata dal programma *esattamente* una volta. Il teorema di non duplicazione afferma quindi che i dati quantistici sono intrinsecamente lineari, ossia non possono essere duplicati (o cancellati). Per questo motivo i sistemi di tipo lineare sono stati utilizzati per prevenire a tempo di compilazione duplicazioni dei dati quantistici: ogni qualvolta un dato quantistico viene duplicato il sistema di tipi segnala un errore del programma prima che quest'ultimo venga eseguito.

A partire dal lavoro di Selinger (Selinger, 2004), vari sistemi di tipo lineari sono stati sviluppati per calcoli fondazionali per linguaggi di programmazione quantistici (Selinger & Valiron, 2006; Green et al., 2013b, 2013a; Rios & Selinger, 2017) portando alla più recente implementazione di linguaggi con sistemi di tipo lineare *built-in* (Paykin, Rand, & Zdancewic, 2017; Bichsel, Baader, Gehr, & Vechev, 2020). Ciò nonostante, come visto nell'esempio 1.2, il sistema di tipi di Q# non è in grado di determinare la linearità dei dati quantistici e quindi di prevenire *staticamente* errori dovuti alla duplicazione di quest'ultimi.

Contributi e Struttura del Lavoro Il contributo principale di questo lavoro è lo sviluppo di un sistema di tipi lineare per il frammento al prim'ordine di Q# con variabili mutabili non quantistiche. Nonostante sia privo di funzioni di ordine superiore e consenta solamente l'assegnamento di dati quantistici a variabili immutabili, tale

frammento è sufficientemente ricco ed espressivo per la scrittura di algoritmi quantistici non triviali (come il lettore avrà modo di vedere negli esempi proposti in questo lavoro). Infatti, il lettore può facilmente verificare che gli esempi visti in questo capitolo sono entrambi scritti in questo frammento. I contributi di questo lavoro sono i seguenti:

- La definizione di *Linearly Typed Idealized Q#* (abbreviato in LTIQ), una formalizzazione del frammento al prim'ordine di Q# con variabili mutabili non quantistiche.
- La definizione di una semantica operazionale per LTIQ.
- Lo sviluppo di un sistema di tipi lineare per LTIQ.
- La dimostrazione del teorema di sicurezza rispetto ai tipi (*type safetey*) (Cardelli, 1997; Pierce, 2002) per LTIQ.

Commentiamo ora i punti sopraelencati.

Al fine di introdurre un sistema di tipi lineare per (il frammento di interesse di) Q# è necessario avere a dispozione una formalizzazione di tale linguaggio e del suo modello di esecuzione dei programmi. Non avendo Q# una tale formalizzazione, il primo contributo di questo lavoro è l'introduzione di LTIQ, un calcolo fondazionale per il frammento al prim'ordine di Q# con variabili mutabili non quantistiche. LTIQ è definito da una sintassi e una semantica operazionale: la prima è usata per formalizzare programmi scritti in Q#, mentre la seconda per formalizzare la loro esecuzione. Una volta definita la sintassi e semantica di LTIQ, viene introdotto il suo sistema di tipi lineare e ne viene dimostrata la correttezza. Il principale teorema dimostrato è il teorema di sicurezza rispetto ai tipi (*type safetey*), il quale asserisce che il tipo di un programma è preservato durante la sua esecuzione. Di conseguenza, poichè i dati di tipo lineare manipolati da un programma non possono essere duplicati da quest'ultimo, il teorema di *type safety* garantisce che questa condizione venga preservata durante tutto il processo di esecuzione e quindi che un programma ben tipato non violi il teorema di non duplicazione.

Il lavoro è strutturato come segue: nel Capitolo 2 è introdotta la sintassi di LTIQ e vengono illustrati alcuni esempi di codifica di programmi Q# in LTIQ. Nel Capitolo 3 viene introdotta la semantica operazionale di LTIQ e la sua applicazione agli esempi introdotti del Capitolo 2. Nel Capitolo 4 è definito il sistema di tipi lineare per LTIQ e viene fatto vedere tramite esempi come il suo utilizzo possa prevenire staticamente la duplicazione di dati quantistici. Infine, nel Capitolo 4 quest'intuizione viene formalizzata dimostrando il teorema di *type safety*.

1.1 Revisione della letteratura

Come già detto, molti linguaggi di programmazione non offrono garanzie sulla violazione del teorema di non duplicazione. Alcuni di questi linguaggi sono QisKit (McKay et al., 2018), un framework di Python sviluppato da IBM; ProjectQ (Steiger, Häner, & Troyer, 2016), un framework di Python sviluppato dall'ETH di Zurigo; Cirq (C. & contributors, 2017), un framework di Python sviluppato da Google; Quipper (Green et al., 2013b, 2013a), un linguaggio embedded in Haskell; e naturalmente Q# (Svore et al., 2018), sviluppato da Miscrosoft. Di seguito, invece, sono elencati alcuni dei lavori presenti in letteratura in cui sono stati sviluppati dei sistemi di tipi lineari.

Il linguaggio QPL (Selinger, 2004) è stato il primo lavoro in cui sia stato utilizzato un sistema di tipi lineare per garantire la non duplicazione dei dati quantistici. In seguito, è stato introdotto da Selinger e Valiron il *quantum λ-calculus* (Selinger & Valiron, 2006), un formalismo fondazionale per programmi quantistici di ordine superiore basato sul λ-calcolo (Barendregt, 1985) con un sistema di tipi *affine* (Walker, 2004) (il sistema assicura la non duplicazione di qbit, ma ne permette la cancellazione). Il *quantum λ-calculus* si basa sul modello di controllo classico (*classical control*), secondo il quale un programma quantistico integra costrutti di controllo classici con operazioni e dati quantistici: l'aspetto classico e quello quantistico interagiscono tramite l'operazione di misurazione di bit quantistici. Questo modello è attualmente quello più utilizzato (tutti i linguaggi sopra menzionati si basano su questo modello). Il *quantum λ-calculus* di Selinger e Valiron è alla base degli studi sulla semantica operazionale dei linguaggi di programmazione quantistici (e.g., (Dal Lago, Masini, & Zorzi, 2009; Zorzi, 2016)). La semantica operazionale sviluppata in questo lavoro si basa anche'essa su quella proposta da Selinger e Valiron per il *quantum λ-calculus*. Per completezza menzioniamo anche il λ-calcolo quantistico di Van Tonder (van Tonder, 2004), un calcolo fondazionale per linguaggi puramente quantistici (ovvero senza costrutti classici).

In aggiunta ai lavori sopra menzionati, la famiglia di calcoli Proto-Quipper (Ross, 2015; Rios & Selinger, 2017) è stata introdotta come formalismo fondazionale per (sottoinsiemi di) Quipper: in particolare, questi calcoli consentono di assegnare tipi lineari ai termini di Quipper basandosi sul *quantum λ-calculus* di Selinger e Valiron. Il linguaggio Qwire (Paykin et al., 2017), un linguaggio di programmazione per definire circuiti quantistici, è un linguaggio minimale ma sicuro rispetto al teorema di non duplicazione grazie al suo sistema di tipi lineare. Infine, un recente linguaggio con un sistema di tipi lineare sviluppato presso il centro di ricerca ETH di Zurigo è il linguaggio Silq (Bichsel et al., 2020). Anche se ormai datata una valida lettura sui linguaggi programmazione quantistici, sopratutto per quelli di natura imperativa, resta il *survey* di Sofge (Sofge, 2008).

Capitolo 2

Sintassi

LTIQ (Linearly Typed Idealized Q#) è un calcolo minimale per il linguaggio di programmazione Microsoft Q#. LTIQ è pensato per catturare il core computazionale del linguaggio Q#, permettendone uno studio formale. In particolare LTIQ è sviluppato per lo studio degli aspetti riguardanti la creazione, la manipolazione e l'utilizzo delle risorse quantistiche messe a disposizione dal linguaggio Q#. Inoltre, LTIQ modella anche la natura imperativa di Q#, includendo definizioni di funzioni al prim'ordine e includendo la nozione di *store* mutabile, fornendo quindi la possibilità di creare e modificare variabili mutabili.

Definizione 2.0.1. L'insieme *Term* dei termini di LTIQ è definito dalla grammatica in Figura 2.1, dove x varia su un insieme numerabile di variabili, c varia su un insieme fissato di costanti (come ad esempio numerali, booleani, etc..) e U^n su un insieme fissato di *gate quantistic* n-ari (ovvero operatori unitari: si veda Appendice A), con $n \in \mathbb{N}$. È assunto che nella dichiarazione di funzione, all'interno del corpo M, non sia possibile scrivere una nuova definizione di funzione per rispecchiare il fatto che Q# non permetta di avere definizioni annidate di operazioni o funzioni.

In Figura 2.1 è presente la categoria sintattica t: essa descrive la forma sintattica delle annotazioni di tipo di LTIQ. Tali annotazioni, che ricalcano quelle presenti in Q#, hanno funzione puramente grammaticale e servono per definire quando un termine è sintatticamente ben formato. È importante distinguere sin da ora le annotazioni di tipo dalla nozione di tipo che introdurremo nel Capitolo 4: mentre la prima ha una funzione grammaticale, la seconda ha una funzione logica e serve a garantire specifiche proprietà del comportamento di un programma.

La sintassi è intenzionalmente resa liberale: termini (logicamente) mal formati, come $CNOT(x, x)$ sono permessi ma, come mostrato in seguito, verranno rigettati dal sistema di tipi. Di seguito è spiegata in dettaglio la sintassi di LTIQ.

- () denota la tupla vuota.

- c appartiene all'insieme *Const* che assumiamo sia definito come l'insieme sintattico di tutte le costanti del linguaggio: $Const = \mathbb{Z} \cup \{true, false\} \cup String \cup ....$
- x appartiene all'insieme infinito di variabili $Var = \{x_0, x_1, x_2, ...\}$.
- $(M_1, ..., M_n)$ denota una tupla di termini.
- $U^n(M_1, ..., M_n)$ è l'applicazione del gate U di arietà n alla tupla $(M_1, ..., M_n)$. Assumiamo che $U^n \in \mathbb{U}$, dove $\mathbb{U}$ è l'insieme dei gate che operano sui qubit. Ad esempio, si ha che $CNOT, H \in \mathbb{U}$ (Appendice A, (Nielsen & Chuang, 2016)).
- $meas(M)$ è l'operazione di misurazione di un qubit applicata al termine M.
- $let\ f = op(x_1 : t_1, ..., x_n : t_n) : t_{n+1}\{M\}\ in\ N$ è il termine che denota la dichiarazione di una funzione f. L'identificatore f appartiene all'insieme *Fun* che assumiamo sia definito come l'insieme delle stringhe alfanumeriche che identificano i nomi di funzioni. La tupla $(x_1 : t_1, ..., x_n : t_n)$ denota i parametri formali della funzione f (possibilmente vuota con $i = 0$); ad ogni x_i è associata un'annotazione di tipo t_i descritta dalla categoria sintattica t. L'annotazione di tipo t_{n+1} rappresenta il tipo di ritorno di f, il termine M è il corpo della funzione e N è la continuazione dell'interno termine.
- $f\,(M_1, ..., M_n)$ è l'applicazione di funzione identificata da f sulla tupla di termini $(M_1, ..., M_n)$.
- $let\ (x_1, ..., x_n) = M\ in\ N$ è il termine utilizzato per creare un binding immutabile tra le variabili $x_1, ..., x_n$ e i valori $v_1, ..., v_n$ risultanti dall'esecuzione di M. Una volta eseguito M ed effettuato il binding, viene eseguito il termine N.
- $using\ (x_1, ..., x_n)\ in\ M$ permette il binding delle variabili $x_1, ..., x_n$ con n nuovi qubit. Questo costrutto permette la creazione di n nuovi qubit allocati nello stato quantistico $|0\rangle$.
- $mut\ (x_1, ..., x_n) = M\ in\ N$ è il termine utilizzato per creare un binding mutabile tra $x_1, ..., x_n$ e i valori $v_1, ..., v_n$ risultanti dall'esecuzione di M. Uno volta eseguito M ed effettuato il binding, viene eseguito il termine N.
- $set\ (x_1, ..., x_n) = M\ in\ N$ permette di cambiare il binding delle variabili $x_1, ..., x_n$ con i nuovi valori $v_1, ..., v_n$. Tale operazione è consentita solamente se i tipi di $v_1, ..., v_n$ sono gli stessi delle variabili $x_1, ..., x_n$.
- $if\ M\ then\ N_1\ else\ N_2$ è il termine che denota il classico costrutto condizionale.

$$
\begin{array}{lr}
M ::= \ () & (\textit{Unit}) \\
\quad | \ c & (\textit{Costante}) \\
\quad | \ x & (\textit{Variabile}) \\
\quad | \ (M_1, ..., M_n) & (\textit{Tupla}) \\
\quad | \ U^n(M_1, ..., M_n) & (\textit{Applicazione Gate}) \\
\quad | \ \mathit{meas}(M) & (\textit{Misurazione}) \\
\quad | \ \mathit{let}\ f = \mathit{op}(x_1 : t_1, ..., x_n : t_n) : t_{n+1}\{M\}\ \mathit{in}\ M & (\textit{Dichiarazione funzione}) \\
\quad | \ f\,(M_1, ..., M_n) & (\textit{Applicazione funzione}) \\
\quad | \ \mathit{let}\,(x_1, ..., x_n) = M\ \mathit{in}\ M & (\textit{Dichiarazione immutabile}) \\
\quad | \ \mathit{using}\,(x_1, ..., x_n)\ \mathit{in}\ M & (\textit{Allocazione qbit}) \\
\quad | \ \mathit{mut}\,(x_1, ..., x_n) = M\ \mathit{in}\ M & (\textit{Dichiarazione mutabile}) \\
\quad | \ \mathit{set}\,(x_1, ..., x_n) = M\ \mathit{in}\ M & (\textit{Assegnamento}) \\
\quad | \ \mathit{if}\ M\ \mathit{then}\ M\ \mathit{else}\ M & (\textit{Condizionale}) \\
 & \\
t := \ \top & (\textit{Unit}) \\
\quad | \ \mathit{qbit} & (\textit{Quantum bit}) \\
\quad | \ \mathit{bit} & (\textit{Classico bit}) \\
\quad | \ \mathit{int} & (\textit{Interi}) \\
\quad | \ \mathit{bool} & (\textit{Booleani}) \\
\quad | \ ... & (...) \\
\quad | \ (t_1 \otimes ... \otimes t_n) & (\textit{Tuple})
\end{array}
$$

Figura 2.1: Sintassi astratta per il linguaggio LTIQ

Il linguaggio definito dai termini *Term* è quindi un sottoinsieme del linguaggio Q#. Per semplificare il modello formale e le dimostrazioni associate non sono stati presi in considerazione alcuni costrutti come ad esempio lo statement *While* o lo statement *Repeat* e sono stati esclusi anche array e tipi definiti dall'utente. Ciò nondimeno è stato incluso il costrutto condizionale *if* e i cicli sono permessi attraverso la ricorsione. Si noti che non vi è alcuna distinzione tra *funzione* ed *operazione* come vi è invece in Qsh. Inoltre sono state effettuate le seguenti scelte per semplificare il modello formale: nonostante ciò riduca il potere espressivo di LTIQ, quest'ultimo si qualifica come un formalismo minimale per l'analisi delle computazioni quantistiche in Q#, come enfatizzato in Sezione 2.1

- Le funzioni considerate sono solamente al prim'ordine;
- Il binding mutabile tra variabili e valori è permesso solamente nel caso in cui i valori siano valori classici e non quantistici.

Convenzione 2.0.2. Introduciamo alcune convenzioni di notazione che semplificano e abbreviano i termini di LTIQ:

- Un insieme di elementi $\{p_1, ..., p_n\}$ è delle volte denotato con $\overline{p}$.
- Una tupla di elementi $(p_1, ..., p_n)$ è invece denotata con $\vec{p}$ o con $\overrightarrow{(p)}$.
- La notazione $(M_{i=0}^{j-1}, M_j, M_{j+1}^n)$ descrive la tupla $(M_0, ..., M_{j-1}, M_j, M_{j+1}, ..., M_n)$.

2.1 LTIQ: Alcuni Esempi

In questa sezione sono confrontate l'espressività di Q# e LTIQ tramite alcuni esempi. Il primo esempio che consideriamo analizza l'implementazione dell'algoritmo di teletrasporto quantistico, mostrando il caso in cui un'entità *Alice* invia un quantum bit q ad un'altra entità *Bob*. Il secondo esempio, analizza invece l'implementazione di un generatore quantistico di bit casuali e il suo utilizzo. Come ultimo esempio è analizzata l'implementazione dell'operazione di *entanglement* sullo stesso quantum bit che, come vedremo nel capitolo successivo, porterà l'esecuzione del programma in uno stato di *errore*.

Esempio 2.1.1. L'algoritmo di teletrasporto quantistico è un protocollo nel quale esistono due entità, chiamate Alice e Bob, in cui Alice vuole inviare un quantum bit q a Bob. Per via del teorema di *non duplicazione*, Alice non può generare una copia dello stato di q e inviarla a Bob. Per risolvere questo problema entrerà in gioco una nuova entità che fornirà ad Alice e a Bob una coppia di qubit entangled. A questo punto, Alice eseguirà alcune operazioni sul proprio qubit e sul qubit q, inviando a Bob i risultati su un canale di comunicazione "classico". Bob, una volta ricevuti tali risultati li utilizza al fine di trasformare il suo stesso qbit nello stato originale del qbit q. In Figura 2.3 è mostrato il circuito per l'algoritmo, mentre in Figura 2.2 è mostrata l'implementazione dell'algoritmo in Q# nel caso in cui Alice vuole mandare il qubit q nello stato $|0\rangle$ a Bob.

La Figura 2.4, d'altra parte, mostra lo stesso algoritmo con la sintassi di LTIQ. La differenza principale tra i due linguaggi è l'utilizzo dei qbit. Si noti come in Q# una variabile di tipo qbit sia un identificatore opaco che punti ad un qbit logico. Utilizzare un identficatore del genere nell'applicazione di un gate significa modificare la natura del qubit puntato in modo implicito come forma di *side-effect*. Il linguaggio LTIQ vuole catturare il cambio di stato per i qbit in modo esplicito già a partire dalla sintassi.

```
namespace teleportation {
  open Microsoft.Quantum.Canon;
  open Microsoft.Quantum.Intrinsic;
  open Microsoft.Quantum.Diagnostics;

  operation bell00(x: Qubit, y: Qubit) : (Qubit, Qubit) {
    H(x);
    CNOT(x, y);
    return (x, y);
  }

  operation alice(a:Qubit, x:Qubit) : (Result, Result) {
    CNOT(a, x);
    H(a);
    return (M(a), M(x));
  }

  operation bob(a:Result, x:Result, y:Qubit):Qubit {
    let y1 = (x == One)? X(y) | ();
    let a1 = (a == One)? Z(y) | ();
    return y;
  }

  @EntryPoint()
  operation teleport(): Unit {
    using((a, x, y) = (Qubit(), Qubit(), Qubit())) {
    let (x1, y1) = bell00(x, y);
    let (ra, rx1) = alice(a, x1);
    let r = bob(ra, rx1, y1);
    }
  }
}
```

Figura 2.2: Implementazione dell'algoritmo di teletrasporto in Q#

Infatti si può notare come l'encoding per l'applicazione del gate $U(x_1, ..., x_n)$, sia il seguente: *let* $(x'_1, ..., x'_n) = U(x_1, ..., x_n)$. Le variabili $(x'_1, ..., x'_n)$ hanno lo scopo di mettere in evidenza il cambio dello stato dei qbit puntati da $(x_1, ..., x_n)$.

Per una migliore leggibilità sono rimosse le parentesi nel caso in cui la tupla di variabili *left-value* per i costrutti di binding abbia un singolo elemento. In Q# un bit è rappresentato dalle costanti di linguaggio *Zero* e *One* che hanno entrambi il tipo *Result*. Per essere fedeli a questa scelta di Q#, l'encoding in LTIQ per tali valori è

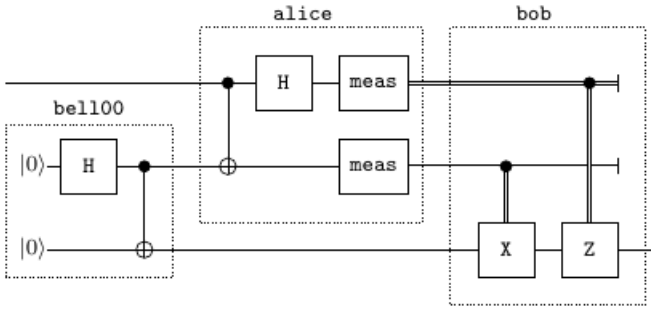

Figura 2.3: Circuito di teletrasporto

```
let bell = op(u:qbit, v:qbit):(qbit ⊗ qbit) {
    CNOT(H(u), v)
} in
let alice = op(x0:qbit, y0:qbit):(qbit ⊗ qbit) {
    let (x1, y1) = CNOT(x0, y0) in
    (meas(H(x1)), meas(y1))
} in
let bob = op(c0:bit, c1:bit, q0: qbit):qbit {
    let z = if (c0 == one) then X(q0) else q0 in
    if (c0 == zero) then Z(z) else z
} in
using (b, a, q) in
    let (a1, b1) = bell(a, b) in
    let (bq1, ba2) = alice(q, a1) in
    bob(bq1, ba2, b1)
```

Figura 2.4: Encoding dell'algoritmo di teletrasporto in LTIQ

dato rispettivamente dalle costanti *zero* e *one* mentre *bit* corrisponde all'encoding dell' annotazione di tipo *Result*. Inoltre, nella nostra sintassi non è presente l'operatore infisso di uguaglianza == ma può essere visto come l'applicazione di una funzione che presi due oggetti restituisca il valore *true* o *false* nel modo più ovvio.

Esempio 2.1.2. In Figura 2.5 viene mostrato come attraverso la superposizione di un qubit possano essere generati dei numeri casuali. Inoltre, si vuole mostrare un utilizzo concreto delle variabili mutabili messe a disposizione dal linguaggio Q#. È implementata la funzione *sampleBit* che, ad ogni sua chiamata, alloca un nuovo qbit, trasforma il suo stato ed infine lo misura in modo tale da ottenere con la stessa probabilità un

bit nello stato 0 o nello stato 1. La Figura 2.6 mostra l'encoding del generatore di bit casuali nella sintassi di LTIQ.

```
namespace quantumGenerator {
  open Microsoft.Quantum.Canon;
  open Microsoft.Quantum.Intrinsic;
  open Microsoft.Quantum.Diagnostics;

  operation sampleBit():Result {
    using (x = Qubit()) {
        H(x);
        return M(x);
    }
  }

  @EntryPoint()
  operation main(): Unit {
    mutable (a, b) = (Zero, Zero);
    set a = sampleBit();
    set b = sampleBit();
  }
}
```

Figura 2.5: Implementazione di un generatore di bit casuali in Q#

```
let sampleBit = op():bit {
    using x in meas(H(x))
} in
mut (a, b) = (zero, zero) in
  set a = sampleBit() in
  set b = sampleBit() in
  (a, b)
```

Figura 2.6: Encoding di un generatore di bit casuali in LTIQ

Esempio 2.1.3. In questo esempio è mostrato il caso in cui si provi in Q# a creare uno stato *entangled* sullo stesso *qbit*. È implementata la funzione *entangle*, la quale appunto

prova a creare uno stato entangled per i quantum bit che sono passati come argomenti della funzione. La funzione *entangle* sarà invocata passandole lo stesso quantum bit per entrambi i parametri. La Figura 2.8 mostra l'encoding in LTIQ per il nostro ultimo esempio. Si noti come quest'ultimo vada a violare il teorema di *non duplicazione*, vedremo nel Capitolo 3 come l'esecuzione di questo programma porti ad un errore e nel Capitolo 4 come prevenire questi tipi di errori a *compile-time* tramite l'adozione di un opportuno sistema di tipi.

```
1  namespace teleportation {
2    open Microsoft.Quantum.Canon;
3    open Microsoft.Quantum.Intrinsic;
4    open Microsoft.Quantum.Diagnostics;
5
6    operation entangle(x:Qubit, y:Qubit):(Qubit, Qubit) {
7        H(x);
8        CNOT(x, y);
9        return (x, y);
10   }
11
12   @EntryPoint
13   operation main():Unit {
14       using (x = Qubit()) {
15         entangle(x, x);
16       }
17   }
18 }
```

Figura 2.7: Implementazione dello stato entagled in Q#

$$
\begin{array}{l}
\textit{let } \textit{entangle} = \textit{op}(x_0{:}\textit{qbit},\, y_0{:}\textit{qbit}){:}(\textit{qbit} \otimes \textit{qbit})\ \{ \\
\quad \textit{let } x_1 = H(x_0)\ \textit{in} \\
\quad \textit{CNOT}(x_1,\, y_0) \\
\}\ \textit{in} \\
\textit{using } q\ \textit{in } \textit{entangle}(q,\, q)
\end{array}
$$

Figura 2.8: Encoding dello stato entangled in LTIQ

Concludiamo questo capitolo introducendo alcune nozioni ausiliare e convenzioni notazionali per LTIQ che saranno utili per parlare dei termini di LTIQ.

2.2 Funzioni Ausiliare

In questa sezione sono definite alcune funzioni ausiliare che permettono di estrarre informazioni dai termini LTIQ. Si assume che sia già definita la funzione Vars : *Term* $\to$ $\mathcal{P}(\mathit{Var})$ che restituisce tutte le occorrenze di variabili presenti all'interno di un generico termine M preso in input.

Definizione 2.2.1. Definisco la funzione FV : *Term* $\to \mathcal{P}(\mathit{Var})$, come la funzione che associa ad ogni termine l'insieme delle sue variabili libere (i.e. variabili che non sono sotto lo scopo di un binder) cattura. La definizione è data per induzione sul termine M:

$$\begin{aligned}
&\mathrm{FV}(()) = \{\}\\
&\mathrm{FV}(c) = \{\}\\
&\mathrm{FV}(x) = \{x\}\\
&\mathrm{FV}(\mathit{meas}(M)) = \mathrm{FV}(M)\\
&\mathrm{FV}((M_1, ..., M_n)) = \mathrm{FV}(M_1) \cup ... \cup \mathrm{FV}(M_n)\\
&\mathrm{FV}(U^n(M_1, ..., M_n)) = \mathrm{FV}(M_1) \cup ... \cup \mathrm{FV}(M_n)\\
&\mathrm{FV}(f\,(M_1, ..., M_n)) = \mathrm{FV}(M_1) \cup ... \cup \mathrm{FV}(M_n)\\
&\mathrm{FV}(\mathit{let}\ f = \mathit{op}(\overrightarrow{x : t}) : t_{n+1}\{M\}\ \mathit{in}\ N) = \mathrm{FV}(M) \setminus \{x_1, ..., x_n\} \cup \mathrm{FV}(N)\\
&\mathrm{FV}(\mathit{let}\ (x_1, ..., x_n) = M\ \mathit{in}\ N) = \mathrm{FV}(M) \cup (\mathrm{FV}(N) \setminus \{x_1, ..., x_n\})\\
&\mathrm{FV}(\mathit{mut}\ (x_1, ..., x_n) = M\ \mathit{in}\ N) = \mathrm{FV}(M) \cup (\mathrm{FV}(N) \setminus \{x_1, ..., x_n\})\\
&\mathrm{FV}(\mathit{set}\ (x_1, ..., x_n) = M\ \mathit{in}\ N) = \mathrm{FV}(M) \cup \mathrm{FV}(N) \cup \{x_1, ..., x_n\}\\
&\mathrm{FV}(\mathit{using}\ (x_1, ..., x_n)\ \mathit{in}\ M) = \mathrm{FV}(M) \setminus \{x_1, ..., x_n\}.
\end{aligned}$$

Definizione 2.2.2. Definisco la funzione BV : *Term* $\to \mathcal{P}(\mathit{Var})$, come la funzione che, dato un generico termine M, restituisce l'insieme delle *variabili legate* presenti in M:

$$\mathrm{BV}(M) = \mathrm{Vars}(M) \setminus \mathrm{FV}(M)$$

dove $\mathrm{Vars}(M)$ è l'insieme delle variabili in M.

Definizione 2.2.3. Assumendo che $\equiv$ sia l'operatore per l'uguaglianza sintattica, la funzione di sostituzione di ogni libera occorrenza della variabile x in N con il termine P, scritto $N[x/P]$, è così definita:

$$\begin{aligned}
&()[x/P] \equiv ()\\
&c[x/P] \equiv c
\end{aligned}$$

$$y[x/P] \equiv \begin{cases} N & \text{if } y \equiv x \\ y & \text{otherwise} \end{cases}$$

$$f\,(M_1, ..., M_n)[x/P] \equiv f\,(M_1[x/P], ..., M_n[x/P])$$

$$(M_1, ..., M_n)[x/P] \equiv (M_1[x/P], ..., M_n[x/P])$$

$$U^n(M_1, ..., M_n)[x/P] \equiv U^n(M_1[x/P], ..., M_n[x/P])$$

$$meas(M)[x/P] \equiv meas(M[x/P])$$

$$let\ f = op(\overrightarrow{y : t}) : t_{n+1}\{M\}\ in\ N[x_j/P] \equiv$$

$$\begin{cases} let\ f = op(\overrightarrow{y : t}) : t_{n+1}\{M\}\ in\ N[x_j/P] & if\ x_j \in \overline{y} \\ let\ f = op(\overrightarrow{y : t}) : t_{n+1}\{M[x_j/P]\}\ in\ N[x_j/P] & otherwise \end{cases}$$

$$let\ (\vec{y}) = M\ in\ N[x_j/P] \equiv \begin{cases} let\ (\vec{y}) = M[x_j/P]\ in\ N & if\ x_j \in \overline{y} \\ let\ (\vec{y}) = M[x_j/P]\ in\ N[x_j/P] & otherwise \end{cases}$$

$$mut\ \vec{y} = M\ in\ N[x_j/P] \equiv \begin{cases} mut\ \vec{y} = M[x_j/P]\ in\ N & if\ x_j \in \overline{y} \\ mut\ \vec{y} = M[x_j/P]\ in\ N[x_j/P] & otherwise \end{cases}$$

$$set\ \vec{y} = M\ in\ N[x_j/P] \equiv \begin{cases} set\ \vec{y} = M[x_j/P]\ in\ N[x_j/P] & if\ x_j \notin \vec{y} \\ set\ \vec{y} = M[x_j/P]\ in\ N[x_j/P] & otherwise \end{cases}$$

$$using\ \vec{y}\ in\ M[x_j/P] \equiv \begin{cases} using\ \vec{y}\ in\ M & if\ x_j \in \vec{y} \\ using\ \vec{y}\ in\ M[x_j/P] & otherwise \end{cases}$$

Convenzione 2.2.4 ((Barendregt, 1985))**.** In ogni contesto matematico assumiamo che l'insieme delle variabili legate sia sempre diverso da quello delle variabili libere.

Si consideri il seguente caso:

$$(let\ (x_1, ...x_n) = M\ in\ x_2)\ [x_2/P]$$

dove x_1 è una variabile libera in P. Ciò è un problema in quanto il termine risultante della sostituzione non è equivalente in termini di legami a quello iniziale. Per prevenire questo tipo di problema, è utilizzata la convenzione sulle variabili.

Per essere vera la convenzione sulle variabili è definita l'operazione di α-conversione.

Definizione 2.2.5. Siano M ed N due termini, M è α-equivalente ad N se ogni occorrenza di variabile legata in M è rimpiazzata da una generica variabile y_j. L'α-conversione è definita come la più piccola congruenza tale che:

$$let\ (x_1, ..., x_n) = M\ in\ N \equiv_\alpha let\ (x_1, ..., y_j, ..., x_n) = M\ in\ N[x_j/y_j]$$

$$let\ f = op(x_1 : t_1, ..., x_n : t_n) : t_{n+1}\{M\}\ in\ N \equiv_\alpha$$

$let\ f = op(x_1 : t_1, ..., y_j : t_j, ...x_n : t_n) : t_{n+1}\{M[x_j/y_j]\}\ in\ N$
$mut\ (x_1, ..., x_n) = M\ in\ N \equiv_\alpha mut\ (x_1, ..., y_j, ..., x_n) = M\ in\ N[x_j/y_j]$
$using\ (x_1, ..., x_n)\ in\ M \equiv_\alpha using\ (x_1, ..., y_j, ...x_n)\ in\ M[x_j/y_j]$.

Avendo introdotto la sintassi di LTIQ, passiamo a definire la sua semantica.

Capitolo 3

Semantica operazionale

In questo capitolo è descritta la semantica operazionale di LTIQ. Fornire tale semantica è necessario per definire un modello di esecuzione per il linguaggio Q#.

Prima di descrivere le regole semantiche, sono introdotte le nozioni di *store* (Winskel, 1993) per funzioni, variabili mutabili e per dati quantistici. Successivamente, sono descritte alcune funzioni che estraggono informazioni dagli store o che modificano il loro stato. Infine è formalizzata la semantica e sono dati alcuni esempi di valutazione.

3.1 Store

Per poter avere una descrizione formale della semantica del linguaggio LTIQ necessitiamo di formalizzare la nozione di *store*. Intuitivamente, un generico *store* non è altro che un'istantanea di ciò che vi è in memoria in un determinato momento dell'esecuzione di un programma. La necessità di avere uno *store* nasce dalla natura multi-paradigma e quantistica del linguaggio Q#. In particolare, avremo tre tipi diversi di *store*:

- Uno store per dati classici, che chiameremo *classic store*, che associa ad una qualsiasi locazione in memoria un dato classico.
- Uno store per dati quantistici, che chiameremo *quantum store*, che associa ad una qualsiasi locazione in memoria un dato quantistico, in particolare un bit quantistico.
- Uno store per funzioni, chiamato *fun store*, il quale associa ad una qualsiasi locazione in memoria una tupla di simboli di variabili e il corpo di una funzione.

Possiamo pensare ad ognuno di questi *store* come un array di *valori*, astraendo sulle rappresentazioni a run-time e sulle dimensioni in byte di essi. Dettagliatamente, definiremo ognuno di questi store come una funzione *parziale* da uno specifico insieme A che rappresenta l'insieme delle locazioni dello *store* in un insieme B che rappresenta

l'insieme dei valori possibili per quel determinato *store*. Iniziamo col definire i codomini per un generico *classical store* e per un generico *quantum store*. Assumiamo di avere un insieme infinito *Qvar* di variabili quantistiche i cui elementi sono denotati da $r_0, r_1, ...$

Definizione 3.1.1. Definiamo l'insieme di valori di uno store classico *ValueST* come:

$$ValueST = Const \cup \{(c_1, ..., c_n) \mid c_i \in Const\}.$$

A questo punto, un'istantanea della memoria classica è catturata completamente da una funzione parziale da variabili a valori di uno *store classico*.

Definizione 3.1.2. Uno *store classico* è una funzione parziale:

$$\sigma : Var \rightharpoonup ValueST.$$

Scriviamo Σ per indicare l'insieme degli *store classici*.

Invece avremo che un'istantanea della memoria quantistica è catturata completamente da una funzione parziale da variabili a variabili quantistiche.

Definizione 3.1.3. Uno *store quantistico* è una funzione parziale:

$$\phi : Var \rightharpoonup QVar.$$

Scriviamo Φ per indicare l'insieme degli *store quantistici*.

Infine, vediamo come è definito formalmente un *fun store*. Ricordiamo che *Fun* denota l'insieme degli identificatori di funzione.

Definizione 3.1.4. Uno *fun store* è una funzione parziale:

$$\theta : Fun \rightharpoonup \mathcal{P}(Var) \times Term.$$

Scriviamo Θ per indicare l'insieme degli *fun store*.

Per poter fornire una descrizione completa ed istantanea di un programma quantistico in un certo momento durante l'esecuzione, è introdotta la nozione di *configurazione*. Essa è comprensiva di un *fun store*, di un *store classico* e di uno *store quantistico* per poter rappresentare la memoria di un programma quantistico, ma comprende anche un registro quantistico Q il cui compito è quello di tener traccia dello stato delle risorse quantistiche.

Definizione 3.1.5. Una *configurazione* è una quintupla che fornisce una completa descrizione dello stato di un programma quantistico, così denotata:

$$\langle \theta, \phi, \sigma \rangle [M, Q]$$

Dove:

- $\theta \in \Theta$; $\phi \in \Phi$; $\sigma \in \Sigma$;
- $M \in \mathit{Term}$;
- Q è un vettore normalizzato[1] di $\otimes_{i=1}^{n} \mathbb{C}^2$ per qualche intero $n \geq 0$. Il vettore Q è chiamato *array quantistico* o *registro quantistico*.

Si noti che lo *store quantistico* ϕ, associa ad ogni variabile x nel suo dominio la variabile quantistica r_i. Come vedremo nella semantica operazionale la variabile r_i non è altro che un'etichetta per indicare il *qbit* i-esimo presente nel vettore normalizzato Q.

Nota Bene 3.1.6. Dimenticandoci dello store classico e del fun store, in letteratura (Selinger & Valiron, 2006) si definisce spesso una configurazione come una struttura $[Q, |x_1, \ldots, x_n\rangle, M]$, dove Q è un registro quantistico in $\otimes_{i=1}^{n} \mathbb{C}^2$, e $|x_1, \ldots, x_n\rangle$ è una lista di variabili (libere) in M indicante che la variabile x_i punta all'i-esimo qbit in Q. Queste definizioni sono moralmente equivalenti a quella data in Definizione 3.1.5. Ad esempio, nella notazione di Definizione 3.1.5 una tupla come $[Q, |xy\rangle, f(x, y)]$, dove f è una funzione data, viene scritta come $\langle \cdot, [x \to r_0, y \to r_1], \cdot \rangle [Q, f(x, y)]$. Per questo motivo, talvolta rappresenteremo un store quantistico ϕ con $\mathrm{dom}(\phi) = \{x_{i_1}, \ldots, x_{i_n}\}$ come $|x_{i_0} \to r_0, \ldots, x_{i_n} \to r_n\rangle$, con $\phi(x_{i_j}) = r_j$.

Al fine di ottenere una maggiore leggibilità, introduciamo la seguente notazione: data una configurazione

$$\langle \theta, [x_{i_0} \to r_0, \ldots, x_{i_{n-1}} \to r_{n-1}], \sigma \rangle [Q, M],$$

con $Q = \sum_i \alpha_i |b_0, \ldots, b_{n-1}\rangle$, dove $b_i \in \{0, 1\}$ (e quindi $\{|b_1, \ldots, b_{n-1}\rangle \mid b_i \in \{0, 1\}\}$ base canonica di $\otimes_{i=0}^{n-1} \mathbb{C}^2$), scriviamo Q come

$$\sum_i \alpha_i |r_0 \to b_0, \ldots, r_{n-1} \to b_{n-1}\rangle$$

[1] $\mathbb{C}^2$ denota lo spazio di Hilbert complesso bidimensionale. Per ulteriori dettagli su spazi di Hilbert, prodototto tensore, etc ... il lettore può consultare l'appendice A. Esposizione più esaustive possono essere trovate in un qualsiasi manuale di computazione quantistica (Nielsen & Chuang, 2016).

così da enfatizzare il ruolo svolto dalle variabili quantistiche. Ad esempio, rappresentiamo la configurazione

$$\langle \cdot, [x \to r_0, y \to r_1], \cdot \rangle [\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), f(x,y)]$$

come:

$$\langle \cdot, [x \to r_0, y \to r_1], \cdot \rangle [\frac{1}{\sqrt{2}}(|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle), f(x,y)].$$

Questa notazione, come vedremo, risulta particolarmente intuitiva e conveniente per definire la semantica operazionale di LTIQ (in particolare riflette la creazione e istanziazione di variabile quantistiche in Q#) e per l'analisi degli esempi introdotti nel precedente capitolo.

Convenzione 3.1.7. Indichiamo con il simbolo $\mathcal{C}$, l'insieme di tutte le possibili *configurazioni*.

3.2 Funzioni Ausiliare per gli Store

In questa sezione sono presentate delle funzioni che modellano delle operazioni che vengono eseguite sulla memoria durante l'esecuzione di un programma. Siccome un generico *store* è modellato attraverso una funzione, l'operazione di lookup in memoria corrisponde alla notazione $S(x)$, dove S è uno *store* e x è una variabile o un identificatore di funzione.

Scriveremo $\mathrm{dom}(f)$ per denotare l'insieme in cui la funzione parziale f è definita. È necessario avere una funzione che permetta di descrivere come un generico *store* venga modificato.

Definizione 3.2.1. Sia S un elemento in $\Sigma \cup \Phi \cup \Theta$, denoto con $S[x/U]$ la funzione che restituisce un nuovo *store* con il valore U aggiornato per x:

$$S[x/U](y) = \begin{cases} U & \text{if } x \equiv y \\ S(x) & \text{otherwise.} \end{cases}$$

Infine per una migliore leggibilità e per rendere le regole più semplici definiamo un'ultima funzione che descrive come un *classical store* venga modificato per una sequenza di valori e variabili.

Definizione 3.2.2. Sia $\sigma \in \Sigma$ un *store classico*, per ogni $n \in \mathbb{N}$ denotiamo con Up la funzione che prende in input σ, una tupla di variabili $\vec{x}$ ed una tupla di valori $\vec{V}$ in

ValueST e restituisce un nuovo *store classico* con i valori $\vec{V}$ aggiornati per le variabili $\vec{x}$:

$$\mathrm{Up}(\sigma, \vec{x}, \vec{V}) = \sigma[x_1/V_1]...[x_n/V_n].$$

3.3 Regole per la Semantica Operazionale

In questa sezione sono elencate le regole formali per la semantica operazionale del linguaggio LTIQ. Innanzitutto, introduciamo informalmente la relazione di transizione che rappresenta un passo di evoluzione nell'esecuzione di un programma quantistico in LTIQ. In particolare, siano $\langle \theta, \phi, \sigma \rangle [Q, M]$ e $\langle \theta', \phi', \sigma' \rangle [Q', M']$ due *configurazioni* e sia $p \in [0, 1]$. La semantica di LTIQ è definita da espressioni della forma $\langle \theta, \phi, \sigma \rangle [Q, M] \rightarrow_p \langle \theta', \phi', \sigma' \rangle [Q', M']$ indicanti che gli *store*, lo stato quantistico Q ed M cambiano con probabilità p a seconda del contenuto del termine M. Si noti quindi che la semantica operazionale è probabilistica: ciò è dovuto alla natura quantistica del calcolo e in particolare all'operazione di misurazione dei qbit. La relazione di transizione è definita induttivamente con le regole e gli assiomi che seguono in questa sezione. Indichiamo con $\rightarrow^*$ la relazione definita induttivamente dalle seguenti clausole, dove C_0, C_1, C_2 appartengono a $\mathcal{C}$.

$$\frac{}{C_0 \rightarrow^* C_0} \qquad \frac{C_0 \rightarrow_p C_1}{C_0 \rightarrow^* C_1} \qquad \frac{C_0 \rightarrow^* C_1 \quad C_1 \rightarrow^* C_2}{C_0 \rightarrow^* C_2}$$

Infine definiamo l'insieme *Value* dei *valori* da cui non è possibile effettuare ulteriori riduzioni. Come vedremo nel Capitolo 4, un *valore* descrive il risultato della computazione di un termine LTIQ.

Definizione 3.3.1. Un *valore* è un termine appartenente all'insieme *Value* definito dalla seguente grammatica:

$$v ::= c \mid x \mid (v_1, ..., v_n).$$

Prima di riportare le regole che costituiscono la base della semantica operazionale di LTIQ si noti che le variabili sono intese come valori solamente nel caso in cui abbiano un bit quantistico associato all'interno di uno *store quantistico*. In tutti gli altri casi le variabili sono intese solamente come "contenitori" per i dati classici associati, di conseguenza ridurranno sempre in una costante o una tupla di costanti.

$$\frac{x \in \mathrm{dom}(\sigma)}{\langle \theta, \phi, \sigma \rangle [Q, x] \to_1 \langle \theta, \phi, \sigma \rangle [Q, \sigma(x)]} \,(\textsc{Var})$$

$$\frac{\mathrm{FV}(\vec{v}) = \emptyset \quad \mathrm{Up}(\sigma, \vec{x}, \vec{v}) = \sigma'}{\langle \theta, \phi, \sigma \rangle [Q, mut\ \vec{x} = \vec{v}\ in\ N] \to_1 \langle \theta, \phi, \sigma' \rangle [Q, N]} \,(\textsc{Mut}_1)$$

$$\frac{\overline{x} \subseteq \mathrm{dom}(\sigma) \quad \mathrm{FV}(\vec{v}) = \emptyset \quad \mathrm{Up}(\sigma, \vec{x}, \vec{v}) = \sigma'}{\langle \theta, \phi, \sigma \rangle [Q, set\ \vec{x} = \vec{v}\ in\ N] \to_1 \langle \theta, \phi', \sigma' \rangle [Q, N]} \,(\textsc{Set}_1)$$

Figura 3.1: Regole per l'utilizzo di variabili mutabili

Variabili Quando una variabile mutabile è dichiarata, essa verrà inserita all'interno di σ: questo comportamento è modellato dalla regola $\textsc{Mut}_1$. La regola $\textsc{Mut}_1$ è derivabile solamente nel caso in cui la tupla $\vec{v}$ non possieda variabili libere, in questo modo vincoliamo σ a possedere nel suo codominio solamente valori classici. La regola Set modella il comportamento di modifica dei valori $v_1, .., v_n$ per le variabili $x_1, .., x_n$ all'interno di σ. Anche in questo caso vogliamo che non siano presenti variabili libere in $\vec{v}$ e ulteriormente vincoliamo le variabili $\vec{x}$ ad essere presenti nel dominio di σ. Questo modella il fatto che non è possibile effettuare un assegnamento per delle variabili che non sono state allocate precedentemente in σ. Per finire, la regola Var modella la lettura di una variabile mutabile presente in σ.

Dati quantistici Per allocare dei nuovi quantum bit il linguaggio mette a disposizione il costrutto *using*. Esso permette la creazione di un numero arbitrario di quantum bit preparati nello stato $|0\rangle$, aggiornando in modo appropriato lo stato quantistico Q. Si noti come lo *store quantistico* venga propriamente modificato aggiungendo il legame appena creato tra le variabili $\vec{x}$ e i quantum bit $\vec{r}$. La regola $\textsc{Gate}_1$ modella il comportamento dell'applicazione del gate U di arietà n sulla tupla di variabili $(x_1, ..., x_n)$. Si noti come la riduzione $\textsc{Gate}_1$ è valida solamente nel momento in cui ogni variabile x_i sia presente nel dominio di ϕ perciò x_i dev'essere un qbit. Cosa ancora più importante è la seconda condizione che impone che i quantum bit dati in pasto al gate U siano tutti diversi tra loro. Ciò è fondamentale per rispettare il teorema di non duplicazione dei quantum bit. Lo stato Q' sarà lo stato quantistico risultante dall'applicazione del gate U sui qubit $(x_1, .., x_n)$ partendo dallo stato Q. Nelle regole $\textsc{Meas}_0$ e $\textsc{Meas}_1$ con la notazione Q_0 e Q_1 denotiamo gli stati normalizzati della forma:

$$|Q_0\rangle = \sum_j \alpha_j \left|\varphi_j^0\right\rangle \otimes |0\rangle \otimes \left|\psi_j^0\right\rangle, \qquad |Q_1\rangle = \sum_j \alpha_j \left|\varphi_j^1\right\rangle \otimes |1\rangle \otimes \left|\psi_j^1\right\rangle$$

$$\frac{\vec{r} \text{ sono } Qvar \text{ fresche}}{\langle \theta, \phi, \sigma \rangle [Q, \mathit{using}\ \vec{x}\ \mathit{in}\ M] \to_1 \langle \theta, \phi[\overrightarrow{x/r}], \sigma \rangle [Q \otimes |\overrightarrow{r \to 0}\rangle, M]} \text{(USE)}$$

$$\frac{\overline{x} \subseteq \mathrm{dom}(\phi) \quad x_i \neq x_j \quad 1 \leq i, j \leq n \quad i \neq j}{\langle \theta, \phi, \sigma \rangle [Q, U^n(x_1, ..., x_n)] \to_1 \langle \theta, \phi, \sigma \rangle [Q', (x_1, ..., x_n)]} \text{(GATE}_1\text{)}$$

$$\frac{\phi(x) = r_i}{\langle \theta, \phi, \sigma \rangle [\alpha|Q_0\rangle + \beta|Q_1\rangle, \mathit{meas}(x)] \to_{|\alpha|^2} \langle \theta, \phi, \sigma \rangle [Q_0, \mathit{zero}]} \text{(MEAS}_0\text{)}$$

$$\frac{\phi(x) = r_i}{\langle \theta, \phi, \sigma \rangle [\alpha|Q_0\rangle + \beta|Q_1\rangle, \mathit{meas}(x)] \to_{|\beta|^2} \langle \theta, \phi, \sigma \rangle [Q_1, \mathit{one}]} \text{(MEAS}_1\text{)}$$

Figura 3.2: Regole per la manipolazione di qbit

dove φ_j^0 e φ_j^1 sono i-qbit (così che il qbit misurato è quello puntato da r_i). Queste regole, che sono standard nella letteratura sulla semantica operazionale dei linguaggi quantistici (Selinger & Valiron, 2006), rendono probabilistica la computazione di LTIQ. Infatti esse modellano la probabilità di ottenere lo stato Q_0 o lo stato Q_1 a fronte della misurazione del qbit x.

$$\frac{}{\langle \theta, \phi, \sigma \rangle [Q, \mathit{let}\ f = \mathit{op}(\overrightarrow{x : t}) : t_{n+1}\{M\}\ \mathit{in}\ N] \to_1 \langle \theta[f/(\overline{x}, M)], \phi, \sigma \rangle [Q, N]} \text{(DEC)}$$

$$\frac{\theta(f) = (\{x_1, ..., x_n\}, N) \quad \mathrm{FV}(\vec{v}) \subseteq \mathrm{dom}(\phi)}{\langle \theta, \phi, \sigma \rangle [Q, f\,(v_1, ..., v_n)] \to_p \langle \theta, \phi, \sigma \rangle [Q, N[x_1/v_1, ..., x_n/v_n]]} \text{(APP}_1\text{)}$$

Figura 3.3: Assiomi per le funzioni

Funzioni Analizzando la regola DEC si vede che nel momento in cui una funzione è dichiarata essa è inserita all'interno dello store θ e sarà associato al suo identificatore una coppia, dove il primo elemento è un'insieme finito di variabili che rappresentano i parametri della funzione e il secondo elemento della coppia è il corpo stesso della funzione. Nel momento in cui la funzione f è invocata allora il passo di riduzione

eseguirà il corpo della funzione in cui i parametri formali sono sostituiti dai valori dei parametri attuali. Tale comportamento è modellato dalla regola $\textsc{App}_1$.

$$\frac{\mathrm{FV}(V_{i=0}^{j-1}) \subseteq \mathrm{dom}(\phi) \quad \langle \theta, \phi, \sigma \rangle [Q, M_j] \to_p \langle \theta', \phi', \sigma' \rangle [Q', M_j']}{\langle \theta, \phi, \sigma \rangle [Q, (V_{i=0}^{j-1}, M_j, M_{j+1}^n)] \to_p \langle \theta', \phi', \sigma' \rangle [Q', (V_{i=0}^{j-1}, M_j', M_{j+1}^n)]} \,(\textsc{Tup})$$

$$\frac{\langle \theta, \phi, \sigma \rangle [Q, M] \to_p \langle \theta', \phi', \sigma' \rangle [Q', M']}{\langle \theta, \phi, \sigma \rangle [Q, let\ \vec{x} = M\ in\ N] \to_p \langle \theta', \phi', \sigma' \rangle [Q', let\ \vec{x} = M'\ in\ N]} \,(\textsc{Let}_2)$$

$$\frac{\mathrm{FV}(\vec{v}) \subseteq \mathrm{dom}(\phi)}{\langle \theta, \phi, \sigma \rangle [Q, let\ \vec{x} = \vec{v}\ in\ N] \to_1 \langle \theta, \phi, \sigma \rangle [Q, N[\vec{x}/\vec{v}]]} \,(\textsc{Let}_1)$$

$$\frac{\langle \theta, \phi, \sigma \rangle [Q, M] \to_p \langle \theta', \phi', \sigma' \rangle [Q', M']}{\langle \theta, \phi, \sigma \rangle [Q, mut\ \vec{x} = M\ in\ N] \to_p \langle \theta', \phi', \sigma' \rangle [Q', mut\ \vec{x} = M'\ in\ N]} \,(\textsc{Mut}_2)$$

$$\frac{\langle \theta, \phi, \sigma \rangle [Q, M] \to_p \langle \theta', \phi', \sigma' \rangle [Q', M']}{\langle \theta, \phi, \sigma \rangle [Q, set\ \vec{x} = M\ in\ N] \to_p \langle \theta', \phi', \sigma' \rangle [Q', set\ \vec{x} = M'\ in\ N]} \,(\textsc{Set}_2)$$

$$\frac{\mathrm{FV}(V_{i=0}^{j-1}) \subseteq \mathrm{dom}(\phi) \quad \langle \theta, \phi, \sigma \rangle [Q, M_j] \to_p \langle \theta', \phi', \sigma' \rangle [Q', M_j']}{\langle \theta, \phi, \sigma \rangle [Q, f\,(V_{i=0}^{j-1}, M_j, M_{j+1}^n)] \to_1 \langle \theta', \phi', \sigma' \rangle [Q', f\,(V_{i=0}^{j-1}, M_j', M_{j+1}^n)]} \,(\textsc{App}_2)$$

Figura 3.4: Regole e assiomi per i costrutti classici

Costrutti classici Infine sono riportate le regole di riduzione per i costrutti classici di LTIQ. È importante notare che ogni qualvolta si abbia una regola con dei valori $\vec{v}$, essa è valida solamente nel momento in cui ogni variabile libera presente in $\vec{v}$ è nel dominio dello store ϕ, il che significa che ogni variabile che non può ridurre punta ad un qbit.

3.4 Semantica Operazionale: Alcuni Esempi

Questa sezione mostra come le regole di riduzione precedentemente definite sono appblicabili. L'esempio 3.4.1 mostra l'esecuzione del programma per il teletrasporto

quantistico. L'esempio 3.4.2 descrive il comportamento del programma di fronte a variabili mutabili. Infine, gli esempi 3.4.3 e 3.4.4 mostrano cosa accade nel momento in cui il teorema di non-duplicazione è violato.

Esempio 3.4.1. Partendo dalla codifica in LTIQ dell'esempio 2.1.1, mostriamo l'esecuzione dell'algoritmo di teletrasporto. Per semplicità assumiamo che le dichiarazioni delle funzioni *alice*, *bob* e *bell* siano state già processate e che quindi resti da eseguire il termine M così definito:

$$\begin{aligned} M \equiv\ & \mathit{using}\ (b,\ a,\ q)\ \mathit{in} \\ & \quad \mathit{let}\ (a_1,\ b_1) = \mathit{bell}(a,\ b)\ \mathit{in} \\ & \quad \mathit{let}\ (bq_1,\ ba_2) = \mathit{alice}(q,\ a_1)\ \mathit{in} \\ & \quad \mathit{bob}(bq_1,\ ba_2,\ b_1) \end{aligned}$$

Per brevità, inoltre definiamo i seguenti termini:

$$\begin{aligned} T_2 &\equiv \mathit{bob}(bq_1, ba_2, b_1) \\ T_1 &\equiv \mathit{let}\ (bq_1, ba_2) = \mathit{alice}(q, a_1)\ \mathit{in}\ T_2 \\ T_0 &\equiv \mathit{let}\ (a_1, b_1) = \mathit{bell}(a, b)\ \mathit{in}\ T_1 \end{aligned}$$

Prima di mostrare l'esecuzione di M, definiamo lo store θ che in questo punto dell'esecuzione è così formato:

$$\theta = [\mathit{bell} \to (\{u, v\}, C_0), \mathit{alice} \to (\{x_0, y_0\}, C_1), \mathit{bob} \to (\{c_0, c_1, q_0\}, C_2)]$$

dove C_0, C_1 e C_2 sono i rispettivi termini che rappresentano i corpi delle funzioni: *bell*, *alice* e *bob*. Inoltre, per brevità scriviamo ϕ per indicare il quantum store così definito:

$$\phi = [b \to r_0, a \to r_1, q \to r_2]$$

A questo punto, mostriamo l'esecuzione di M a partire dalla configurazione $\langle \theta, \cdot, \cdot \rangle [\cdot, M]$, dove il simbolo $\cdot$ sta ad indicare che lo *store*/stato quantistico è vuoto/non inizializzato.

$$\begin{aligned} M &\to_1 \langle \theta, \cdot, \cdot \rangle [\cdot, \mathit{using}(b, a, q)\ \mathit{in}\ T_0] \\ &\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 0, r_2 \to 0\rangle, \mathit{let}\ (a_1, b_1) = \mathit{bell}(a, b)\ \mathit{in}\ T_1] \\ &\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 0, r_2 \to 0\rangle, \mathit{let}\ (a_1, b_1) = (\mathit{CNOT}(H(a), b))\ \mathit{in}\ T_1] \\ &\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0\rangle \otimes \frac{|r_1 \to 0\rangle + |r_1 \to 1\rangle}{\sqrt{2}} \otimes |r_2 \to 0\rangle, \mathit{let}\ (a_1, b_1) = (\mathit{CNOT}(a, b))\ \mathit{in}\ T_1] \\ &\to_1 \langle \theta, \phi, \cdot \rangle [|\frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, \mathit{let}\ (a_1, b_1) = (a, b)\ \mathit{in}\ T_1] \\ &\to_1 \langle \theta, \phi, \cdot \rangle [|\frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, \mathit{let}\ (bq_1, ba_2) = \end{aligned}$$

$$
\begin{aligned}
&\quad alice(q,a)\ in\ T_2[a_1/a, b_1/b] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [| \frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, let\ (bq_1, ba_2) = \\
&\qquad let(x_1, y_1) = CNOT(q,a)\ in\ (meas(H(x_1)), meas(y_1)\ in\ T_2[a_1/a, b_1/b]] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [| \frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, let\ (bq_1, ba_2) = \\
&\qquad let(x_1, y_1) = (q,a)\ in\ (meas(H(x_1)), meas(y_1)\ in\ T_2[a_1/a, b_1/b]] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [| \frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, let\ (bq_1, ba_2) = \\
&\qquad (meas(H(q)), meas(a))\ T_2[a_1/a, b_1/b]] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [\frac{|r_0 \to 0, r_1 \to 0, r_2 \to 0\rangle + |r_0 \to 0, r_1 \to 0, r_2 \to 1\rangle}{2} + \\
&\quad \frac{|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle + |r_0 \to 1, r_1 \to 1, r_2 \to 1\rangle}{2}, let\ (bq_1, ba_2) = \\
&\qquad (meas(q), meas(a))\ in\ T_2[a_1/a, b_1/b]] \\
&\to^* \begin{cases}
\langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 0, r_2 \to 0\rangle, let\ (bq_1, ba_2) = (zero, zero)\ in\ T_2] \\
\langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 0, r_2 \to 1\rangle, let\ (bq_1, ba_2) = (one, zero)\ in\ T_2] \\
\langle \theta, \phi, \cdot \rangle [|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle, let\ (bq_1, ba_2) = (zero, one)\ in\ T_2] \\
\langle \theta, \phi, \cdot \rangle [|r_0 \to 1, r_1 \to 1, r_2 \to 1\rangle, let\ (bq_1, ba_2) = (one, one)\ in\ T_2]
\end{cases}
\end{aligned}
$$

Per semplicità consideriamo solo il caso $|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle$.

$$
\begin{aligned}
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle, let\ (bq_1, ba_2) = (zero, one)\ in\ T_2] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle, bob(zero, one, b)] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle, let\ z = if(one == one)\ X(b)\ else\ b\ in \\
&\quad if(zero == one)\ Z(z)\ else\ z] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle, let\ z = X(b) \\
&\quad if(zero == one)\ Z(z)\ else\ z] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 1, r_2 \to 0\rangle, let\ z = b \\
&\quad if(zero == one)\ Z(z)\ else\ z] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 1, r_2 \to 0\rangle, if(zero == one)\ Z(b)\ else\ b] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 1, r_2 \to 0\rangle, b]
\end{aligned}
$$

Esempio 3.4.2. Il seguente esempio mostra l'esecuzione di un programma di LTIQ che sfrutta il meccanismo delle variabili mutabili. Partendo dalla codifica in LTIQ dell'esempio 2.1.2, mostriamo il comportamento operazionale del nostro generatore quantistico di bit. Per semplicità assumiamo che la funzione *sampleBit* sia stata già

processata e perciò partiremo da un *fun store* così definito:

$$\theta = [sampleBit \to (\emptyset, C)]$$

dove C è il corpo della funzione *sampleBit*. Ora definiamo il termine M:

$$\begin{aligned} M \equiv\ & mut\ (a,b) = (zero, zero)\ in \\ & set\ a = sampleBit()\ in \\ & set\ b = sampleBit()\ in\ (a,b) \end{aligned}$$

Si noti che per questioni di spazio la costante *zero* e la costante *one* sono state denotate con i termini Z e O. Per brevità, sono definiti anche i seguenti termini:

$$\begin{aligned} M_1 &\equiv set\ b = \ sampleBit()\ in (a,b) \\ M_0 &\equiv set\ a = \ sampleBit()\ in\ M_1 \end{aligned}$$

A questo punto, mostriamo l'esecuzione di M a partire dalla configurazione $\langle \theta, \cdot, \cdot \rangle [\cdot, M]$.

$$\begin{aligned} M &\to_1 \langle \theta, \cdot, \cdot \rangle [\cdot, mut\ (a,b) = (zero, zero)\ in\ M_0] \\ &\to_1 \langle \theta, \cdot, [a \to Z, b \to Z] \rangle [\cdot, set\ a = sampleBit()\ in\ M_1] \\ &\to_1 \langle \theta, \cdot, [a \to Z, b \to Z] \rangle [\cdot, set\ a = using\ (x_0)\ in\ meas(H(x_0))\ in\ M_1] \\ &\to_1 \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle, set\ a = meas(H(x_0))\ in\ M_1] \\ &\to_1 \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [\frac{|r_0 \to 0\rangle + |r_0 \to 1\rangle}{\sqrt{2}}, set\ a = meas(x_0)\ in\ M_1] \\ &\to_1 \begin{cases} \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle, set\ a = Z\ in\ M_1] \\ \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [|r_0 \to 1\rangle, set\ a = O\ in\ M_1] \end{cases} \\ &\to_1 \begin{cases} \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle, set\ b = using\ (x_1)\ in\ meas(H(x_1))\ in\ (a,b)] \\ \langle \theta, [x_0 \to r_0], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle, set\ b = using\ (x_1)\ in\ meas(H(x_1))\ in\ (a,b)] \end{cases} \\ &\to_1 \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_1 \to 0\rangle, \\ \quad set\ b = meas(H(x_1))\ in\ (a,b)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_1 \to 1\rangle, \\ \quad set\ b = meas(H(x_1))\ in\ (a,b)] \end{cases} \\ &\to_1 \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes \frac{|r_1 \to 0\rangle + |r_1 \to 1\rangle}{\sqrt{2}}, \\ \quad set\ b = meas(x_1)\ in\ (a,b)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes \frac{|r_1 \to 0\rangle + |r_1 \to 1\rangle}{\sqrt{2}}, \\ \quad set\ b = meas(x_1)\ in\ (a,b)] \end{cases} \end{aligned}$$

$$
\rightarrow_* \begin{cases}
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 0\rangle, \mathit{set}\ b = Z\ \mathit{in}\ (a, b)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 1\rangle, \mathit{set}\ b = O\ \mathit{in}\ (a, b)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 0\rangle, \mathit{set}\ b = Z\ \mathit{in}\ (a, b)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 1\rangle, \mathit{set}\ b = O\ \mathit{in}\ (a, b)]
\end{cases}
$$

$$
\rightarrow_1 \begin{cases}
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 0\rangle, (a, b)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to O] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 1\rangle, (a, b)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 0\rangle, (a, b)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to O] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 1\rangle, (a, b)]
\end{cases}
$$

$$
\rightarrow_1 \begin{cases}
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 0\rangle, (Z, Z)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to O] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 1\rangle, (Z, O)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 0\rangle, (O, Z)] \\
\langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to O] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 1\rangle, (O, O)]
\end{cases}
$$

Esempio 3.4.3. Mostriamo l'esecuzione della codifica in LTIQ dell'esempio 2.1.3. In questo esempio si vuole enfatizzare come la semantica operazionale catturi lo stato di errore che è raggiunto nel momento in cui è violato il teorema di non duplicazione. Riprendendo l'esempio 2.1.3 definiamo il termine M:

let entangle $= op(x_0 : qbit, y_0 : qbit) : (qbit \otimes qbit)$ {
  let $x_1 = H(x_0)$ *in*
  $CNOT(x_1, y_0)$
} *in using* q *in entangle*(q, q)

Dalla semantica operazionale si vede facilmente che raggiungeremo la seguente configurazione:

$$\langle \theta, [x_0 \to r_0], \cdot \rangle [\frac{|r_0 \to 0\rangle + |r_1 \to 1\rangle}{\sqrt{2}}, CNOT(q, q)]$$

Ma la regola di riduzione $\textsc{Gate}_1$ è applicabile solamente nel caso in cui le variabili passate come argomento siano differenti. Per questo motivo non esiste nessuna regola di riduzione valida per la nostra configurazione, perciò questa situazione ci porta in uno *stato di errore*. Ciò cattura esattamente ciò che succede durante l'esecuzione del programma equivalente in Q#: è lanciato un errore a run-time nel momento in cui lo statement $CNOT$ deve essere eseguito sullo stesso *qbit*.

Esempio 3.4.4. Nell'esempio 3.4.3 la violazione del teorema di non-duplicazione può essere immediatamente riconosciuta dalla chiamata di funzione *entangle*(q, q) in cui il dato quantistico q viene utilizzato due volte (e quindi duplicato). In questo esempio

osserviamo come l'utilizzo di binder e renaming di variabili complichino i pattern di violazione del teorema di non duplicazione rendendo così il tracciamento delle variabili quantistiche un arduo compito per il programmatore. Definiamo il termine M in questo modo:

$$\begin{array}{l} using\ x_0\ in \\ let\ x_1 = x_0\ in \\ CNOT(x_0, x_1) \end{array}$$

Si noti come nel termine M non vi sia alcuna duplicazione esplicita dei termini quantistici, che diventa invece tale solo dopo alcuni passi di riduzione: [2]

$$\begin{aligned} M &\to \langle \cdot, \cdot, \cdot \rangle [\cdot, using\ x_0\ in\ let\ x_1 = x_0\ in\ CNOT(x_0, x_1)\ ] \\ &\to \langle \cdot, [x_0 \to r_0], \cdot \rangle [|r_0 \to 0\rangle, let\ x_1 = x_0\ in\ CNOT(x_0, x_1)\ ] \\ &\to \langle \cdot, [x_0 \to r_0], \cdot \rangle [|r_0 \to 0\rangle, CNOT(x_0, x_0)\ ] \end{aligned}$$

Ancora una volta ci ritroviamo in una situazione in cui non è possibile ridurre: questo comportamento modella la violazione del teorema di non duplicazione ed in particolare il lancio di un eccezione a run-time da parte del linguaggio Q#.

Lo scopo delle regole di riduzione è quello di valutare l'esecuzione di una configurazione fino a quando non si raggiunga un risultato. Come visto, questo non succede sempre: ci sono *configurazioni* (che chiameremo *configurazioni di errore*) che non sono dei risultati accettabili, e dai quali nessuna riduzione è possibile. Tali *configurazioni* corrispondono agli errori a run-time o a comportamenti indefiniti di Q#. Come possiamo prevenire questi tipi di errore? Q# mette a disposizione un sistema di tipi che cerca di predire a livello statico se un programma abbia un comportamento corretto. Il sistema di tipi di Q# però non tiene conto del teorema di non duplicazione di uno stato quantistico e non previene perciò l'esecuzione di statement come quelli visti in precedenza. Per porre rimedio a questo problema nel prossimo capitolo è presentato un sistema di tipi per LTIQ basato sulla logica lineare che permette di prevenire gli errori dovuti alla violazione del teorema di non duplicazione. In questo modo non sarà più compito del programmatore tener traccia delle variabili quantistiche ed assicurarsi che non venga copiato il loro stato ma sarà il sistema di tipi ad intervenire e lanciare un errore di tipo.

[2] Nonostante la semplicità del termine M, a partire da quest'ultimo il lettore può facilmente immaginare termini più complessi in cui la violazione del teorema di non duplicazione risulta essere nascosta dall'introduzione di svariati binding per variabili.

Capitolo 4

Sistema di tipi

In questo capitolo descriviamo il sistema di tipi di LTIQ, definendo formalmente prima le nozioni di ambiente e di giudizio e poi mostrando le regole di tipaggio. Infine è mostrata l'applicazione delle regole di tipaggio per gli esempi visti in precedenza.

I sistemi di tipi (Cardelli, 1997; Pierce, 2002) sono realizzati per fornire al programmatore delle garanzie sul comportamento dei programmi. In dettaglio, i sistemi di tipi aiutano a rigettare dei termini grammaticalmente ben formati ma che hanno una sorta di significato "indefinito", come ad esempio il termine $X(4)$: il gate unitario X è definito solamente su un *qbit* per cui non ha senso applicare tale gate su un intero. La caratteristica principale di un sistema di tipi è che esso lavora a livello statico: esplorando la struttura sintattica di un programma cerca di prevenire degli errori che possono avvenire durante l'esecuzione del programma stesso. Può essere visto come un potente meccanismo per l'analisi di programmi che a *compile-time* cerca di ridurre il più possibile gli errori a *run-time*. Il suo scopo è quindi quello di facilitare la scrittura di un codice robusto e corretto. Il sistema di tipi di Q# non offre alcuna garanzia in merito alla violazione del teorema di non duplicazione: ad esempio, la compilazione del programma presentato nel Esempio 2.1.3 va a buon fine ma la sua esecuzione porta ad un errore. Infatti è compito del programmatore tenere traccia dell'utilizzo di dati quantistici e nel caso in cui si provi a duplicarli (o cancellarli) si va incontro ad un errore a run-time. LTIQ risolve questo problema adottando un sistema di tipi *lineare*, in modo tale che ogni tentativo di duplicazione di uno stato quantistico è individuato come un errore a compile-time. I sistemi di tipi *lineari* sono utilizzati in letteratura per riflettere sulle risorse di sistema come memoria, lock, etc., inoltre trattano le variabili come se fossero *risorse* (Wadler, 1990; Fähndrich & DeLine, 2002; Wadler, 1990; Walker, 2004). Questo significa che quando una variabile ha un tipo lineare ed è utilizzata, essa non sarà più accessibile nel resto del programma. L'interpretazione di variabili lineari come risorse è caratterizzata da due regole fondamentali:

- Le risorse lineari non possono essere duplicate;

- Le risorse lineari non possono essere scartate.

Per tale ragione l'utilizzo dei sistemi di tipi lineari per trattare le risorse quantistiche è la soluzione naturale al problema di non duplicazione di uno stato quantistico. Questa restrizione in realtà è problematica nell'utilizzo di dati classici: non è importante vincolare il numero di volte che un dato classico, come ad esempio un intero, possa essere utilizzato. Per questo motivo il sistema di tipi per LTIQ integra i tipi lineari adottati dalle risorse quantistiche con i tipi non-lineari adottati dalle risorse classiche. In questo capitolo parleremo di tipi lineari o non-duplicabili e di tipi non-lineari o duplicabili, informalmente, diremo che A è un tipo lineare, mentre $!A$ è un tipo non lineare seguendo la notazione della logica lineare.

Per presentare il sistema di tipi iniziamo col definire la grammatica che descrive i tipi del linguaggio LTIQ.

Definizione 4.0.1. L'insieme dei *tipi* per LTIQ è dato dalla seguente sintassi astratta.

$$\begin{aligned} t &:= \top \mid \mathit{qbit} \mid \mathit{bit} \mid \mathit{int} \mid \mathit{bool} \mid ... \mid (t_1 \otimes ... \otimes t_n) \\ S &:= t \mid !S \mid (S_1 \otimes ... \otimes S_n) \\ A &:= S \mid !(S_1 \multimap S_2) \end{aligned}$$

La categoria A descrive i tipi logici di LTIQ, in dettaglio avremo che:

- t varia sulle annotazioni di tipo definite dall'ononima categoria sintattica (Capitolo 1, Def. 1.0.1), in più varia sulle tuple di annotazioni di tipo; questa categoria è utilizzata per descrivere la forma sintattica delle annotazioni di tipi presenti nelle dichiarazioni di funzioni, nello specifico per i parametri e per il tipo di ritorno delle funzioni.

- S è la categoria sintattica che permette di decorare le costanti di tipo e le tuple di tipi (compresi gli elementi al loro interno) con l'operatore "!". Quest'operatore indica che il tipo associato è un tipo duplicabile e quindi non lineare.

- A è la categoria sintattica che descrive l'insieme dei tipi di LTIQ. Un tipo A può assumere le forme sintattiche descritte nella categoria S oppure avrà la forma $!(S_1 \multimap S_2)$ che descrive il tipo di una funzione in LTIQ. Quest'ultimo è sempre un tipo duplicabile e rappresenta il tipo di una funzione che prende un argomento di tipo S_1 e restituisce dei risultati di tipo S_2. .

Prima di entrare nel dettaglio, si noti che per rendere più leggero il sistema di tipi e dato che l'espressività di LTIQ non risulta troppo compromessa sono state fatte le seguenti scelte:

- Una funzione il cui corpo utilizzi delle variabili globali che sono state tipate con un tipo lineare non sarà tipabile nel nostro sistema di tipi, perciò sarà rigettato un termine ad esempio così formato:

$$using\ x\ in\ let\ f = op() : qbit\{x\}\ in\ f()$$

- Il sistema di tipi non permetterà di creare delle variabili mutabili il cui contenuto sia di tipo lineare, ad esempio non è ammesso dal sistema di tipi il seguente termine:

$$using\ x\ in\ mut\ y = x\ in\ ...$$

Nota Bene 4.0.2. Il linguaggio Q# non permette la dichiarazione di variabili globali. Un programma Q#, come visto nell'introduzione, infatti è una sequenza di dichiarazioni di funzione. In questo senso, nonostante le restrizioni sopra menzionate, il linguaggio LTIQ è più generale e permette al corpo di una funzione di riferirsi alle variabili presenti nello scope globale.

4.1 Giudizi

In questa sezione è definita la nozione di contesto di tipaggio. Inoltre, è mostrata la definizione di *giudizio* e sono introdotte alcune convenzioni per la presentazione delle regole di tipaggio.

Lo scopo del sistema di tipi è quello di assegnare un tipo ad un termine ben formato, quando possibile, in modo da garantire un buon comportamento nell'esecuzione di quel termine. Dato un termine M in LTIQ , scriviamo $M : A$ per indicare che il termine M è ben tipato con il corrispondente tipo A. In generale, non è sufficiente ragionare su un giudizio della forma $M : A$ per via del fatto che il tipaggio di M dipende da tutti i tipi delle variabili libere che compaiono in esso. Per questo motivo introduciamo la nozione di contesto di tipaggio.

Definizione 4.1.1. Un *contesto di tipaggio* è un insieme finito di coppie:

$$\Delta = \{x_1 : A_1, ..., x_n : A_n, f_1 : A_{n+1}, ..., f_m : A_{n+m}\}$$

dove $x_i \in Var$ e $f_i \in Fun$. Assumiamo che un contesto assegni ad ogni variabile/identificatore di funzione al più un tipo (cosicchè, e.g. non si può avere $x : A$, $x : B$ all'interno dello stesso contesto).

L'operatore $|\Delta|$ è usato per denotare l'insieme: $\{x_1, ..., x_n, f_1, ..., f_n\}$ (ossia l'insieme di tutti gli identificatori presenti in Δ) e si ha che $\Delta(x_i) = A_i$. È scritto $!\Delta$ per indicare il contesto della forma:

$$!\Delta = \{x_1 :!A_1, ..., x_n :!A_n, f_1 :!A_{n+1}, ..., f_m :!A_{n+m}\}$$

ossia il contesto Δ con solamente gli elementi non-lineari al suo interno. Gli elementi di $!\Delta$ quindi potranno essere duplicati e scartati senza portare a nessun tipo di problema. Se $|\Delta|$ e $|\Delta'|$ sono disgiunti è scritto Δ, Δ' per indicare l'unione dei due contesti. A questo punto, possiamo definire il concetto di *giudizio*.

Definizione 4.1.2. Un *giudizio* è un'espressione $\Delta \ \triangleright M : A$, dove Δ è un contesto di tipaggio, M è un termine di LTIQ, e A è un tipo. Una derivazione di tipaggio è chiamata valida se può essere inferita dalle regole mostrate in seguito.

Convenzione 4.1.3. È stato definito il simbolo c appartenente all'insieme *Const* come uno strumento sintattico per rappresentare le costanti del linguaggio. Ad esempio c può denotare il valore $n \in \mathbb{Z}$ o può denotare il valore *true*. Ad ogni costante c quindi assumiamo che venga assegnato il tipo corretto $!A_c$, per esempio se $c = n$ allora $c :!int$, oppure se $c = true$ allora $c :!bool$. Un'altra assunzione è fatta sul tipo dei gate n-ari U^n, difatti si assume che essendo gate unitari, essi prendano in input n qubit e restituiscano n qubit, perciò un generico gate U^n è tipato nel seguente modo: $U^n : \otimes_n qbit \multimap \otimes_n qbit$. In seguito è utilizzata la notazione $!^m A$ con $m \in \{0, 1\}$ per denotare il tipo A quando $m = 0$ ed il tipo $!A$ se $m = 1$.

Definizione 4.1.4. Definiamo la promozione di una qualsiasi annotazione di tipo t nel tipo A, quando possibile, attraverso la seguente funzione:

$$\mathrm{Dup}(t) = \begin{cases} t & \text{if } t = qbit \\ !^m(A_1 \otimes ... \otimes A_n) & \text{if } t = (t_1 \otimes ... \otimes t_n), \\ & \mathrm{Dup}(t_1) =!^m A_1, ..., \mathrm{Dup}(t_n) =!^m A_n \\ !t & \text{otherwise.} \end{cases}$$

4.1.1 Regole di tipaggio

In questa sezione sono elencate le regole di tipaggio per LTIQ.

Assiomi Si noti come dalla regola di tipaggio per la tupla vuota Top e dalla regola di tipaggio per le costanti Const, il contesto di tipaggio contiene esclusivamente oggetti duplicabili. Mentre dall'assioma Var A può essere di tipo lineare, — nel qual caso il contesto lineare sarà dato da $x : A$ — oppure di tipo non-lineare (ovvero della forma $!B$, per qualche tipo B) — nel qual caso caso il contesto lineare sarà vuoto.

Costrutti mutabili Per tipare la creazione di variabili mutabili, la regola Mut, richiede che il termine M abbia il tipo di una tupla di esattamente n elementi, ognuno

$$\frac{}{!\Delta \triangleright () : !\top}\,(\textsc{Top})$$

$$\frac{}{!\Delta \triangleright c : !A_c}\,(\textsc{Const})$$

$$\frac{}{!\Delta, x : A \triangleright x : A}\,(\textsc{Var})$$

Figura 4.1: Assiomi per il sistema di tipi

$$\frac{!\Delta \triangleright M : !\vec{A}_{\otimes} \quad !\Delta, \overline{x : !A}, \Gamma \triangleright N : B}{!\Delta, \Gamma \triangleright mut\,(x_1, ..., x_n) = M\ in\ N : B}\,(\textsc{Mut})$$

$$\frac{!\Delta, \overline{x : !A} \triangleright M : !\vec{A}_{\otimes} \quad !\Delta, \overline{x : !A}, \Gamma \triangleright N : B}{!\Delta, \overline{x : !A}, \Gamma \triangleright set\,(x_1, ..., x_n) = M\ in\ N : B}\,(\textsc{Set})$$

Figura 4.2: Regole di tipaggio per i costrutti mutabili di LTIQ

di tipo duplicabile. A questo punto N sarà tipabile nell'ambiente in cui sono stati aggiunte le variabili $x_1, ...x_n$ ognuna col rispettivo tipo non-lineare.

La regola di tipaggio Set si comporta allo stesso modo tranne per il fatto che richiede che le variabili libere $x_1, ..., x_n$ siano contenute già nel suo contesto di tipaggio e che il loro tipo combaci con quello di M.

In entrambi i casi è importante notare come l'ambiente non-lineare Γ, sia utilizzato solamente una volta per il tipaggio dei termini N.

Costrutti quantistici La regola di tipaggio meas si comporta nel modo più ovvio: si assicura che il termine M sia di tipo *qbit* e assegna al termine *meas*(M) il tipo lineare $!bit$. Il comportamento della regola Gate è analogo. Per finire la regola Using è vera solamente se il termine M sia tipabile nell'ambiente in cui sono state aggiunte le variabili $x_1, ..., x_n$ ciascuna col tipo *qbit*.

Costrutti classici La regola Tuple è valida solo se ogni termine M_i è tipato nel proprio ambiente lineare Γ_i. Questa condizione evidenzia la linearità dei tipi: se la variabile x appartiene all'ambiente Γ_i allora essa verrà consumata solamente nel tipag-

$$\frac{!\Delta, \Gamma \triangleright (M_1, ..., M_n) : (qbit \otimes ... \otimes_n qbit)}{!\Delta, \Gamma \triangleright U^n(M_1, ..., M_n) : (qbit \otimes ... \otimes_n qbit)} \,(\textsc{Gate})$$

$$\frac{!\Delta, \Gamma \triangleright M : qbit}{!\Delta, \Gamma \triangleright meas(M) : !bit} \,(\textsc{Meas})$$

$$\frac{\Delta, x_1 : qbit, ..., x_n : qbit \triangleright M : A}{\Delta \triangleright using\ (x_1, ..., x_n)\ in\ M : A} \,(\textsc{Use})$$

Figura 4.3: Regole di tipaggio per i costrutti quantistici di LTIQ

$$\frac{!\Delta, \Gamma_1 \triangleright M_1 : !^m A_1 \quad ... \quad !\Delta, \Gamma_n \triangleright M_n : !^m A_n}{!\Delta, \Gamma_1, ..., \Gamma_n \triangleright (M_1, ..., M_n) : !^m(A_1 \otimes ... \otimes A_n)} \,(\textsc{Tuple})$$

$$\frac{\Delta, f : !(!^m \vec{A}_{\otimes} \multimap B) \triangleright \vec{M} : !^m \vec{A}_{\otimes}}{\Delta, f : !(!^m \vec{A}_{\otimes} \multimap B) \triangleright f\,(M_1, ..., M_n) : B} \,(\textsc{App})$$

$$\frac{\begin{array}{r}\mathrm{FV}(M) \cap \mathrm{dom}(\Gamma) = \emptyset \\ \mathrm{Dup}(\vec{t}) = !^m \vec{A}_{\otimes}, \mathrm{Dup}(t_{n+1}) = B \\ !\Delta, f : !(!^m \vec{A}_{\otimes} \multimap B), \overline{x : !^m A} \triangleright M : B \quad !\Delta, \Gamma, f : !(!^m \vec{A}_{\otimes} \multimap B) \triangleright N : A\end{array}}{!\Delta, \Gamma \triangleright let\ f = op(\overrightarrow{x : t}) : t_{n+1}\{M\}\ in\ N : A} \,(\textsc{Dec})$$

$$\frac{!\Delta, \Gamma_1 \triangleright M : !^m \vec{A}_{\otimes} \quad !\Delta, \Gamma_2, \overrightarrow{x : !^m A} \triangleright N : B}{!\Delta, \Gamma_1, \Gamma_2 \triangleright let\ (x_1, ..., x_n) = M\ in\ N : B} \,(\textsc{Let})$$

$$\frac{!\Delta, \Gamma_1 \triangleright M : !bool \quad !\Delta, \Gamma_2 \triangleright N_1, N_2 : A}{!\Delta, \Gamma_1, \Gamma_2 \triangleright if\ M\ then\ N_1\ else\ N_2 : A} \,(\textsc{If})$$

Figura 4.4: Regole di tipaggio per i costrutti classici di LTIQ

gio del termine M_i e di conseguenza gli altri termini non potranno avere occorrenze libere di x. L'altra regola importante da notare è la regola FunDec. Una generica funzione f è tipabile solamente nel caso in cui le variabili libere presenti nel corpo di f non hanno tipi lineari. Perciò possiamo tipare con sicurezza f con un tipo duplicabile, questo ci porta ad avere anche funzioni ricorsive ben tipate. È utilizzata la funzione *Dup* sulle notazioni di tipo per gli argomenti e per il tipo di ritorno di f per rendere duplicabili i tipi associati quando possibile.

4.1.2 Sistema di tipi mediante esempi

Questa sezione mostra come le regole di tipaggio precedentemente definite sono applicate. L'esempio 2.1.1 mostra il tipaggio del programma per il teletrasporto quantistico, il secondo esempio invece mostra il tipaggio per un generatore di bit casuali. Infine è mostrato il comportamento del sistema di tipi di fronte ad un tentativo di violazione del teorema di non duplicazione.

Esempio 4.1.5. Partendo dalla codifica in LTIQ dell'esempio 2.1.1, mostriamo il tipaggio dell'algoritmo di teletrasporto. Per semplicità assumiamo che le funzioni *alice*, *bob* e *bell* siano già presenti nel contesto di tipaggio da cui partiremo e che il termine M da tipare sia così definito:

$$
\begin{aligned}
M \equiv\ & \mathit{using}\ (b,\ a,\ q)\ \mathit{in} \\
& \quad \mathit{let}\ (a_1,\ b_1) = \mathit{bell}(a,\ b)\ \mathit{in} \\
& \quad \mathit{let}\ (bq_1,\ ba_2) = \mathit{alice}(q,\ a_1)\ \mathit{in} \\
& \quad \mathit{bob}(bq_1,\ ba_2,\ b_1)
\end{aligned}
$$

Per brevità, inoltre definiamo i seguenti termini:

$$
\begin{aligned}
T_2 &\equiv \mathit{bob}(bq_1, ba_2, b_1) \\
T_1 &\equiv \mathit{let}\ (bq_1, ba_2) = \mathit{alice}(q, a_1)\ \mathit{in}\ T_2 \\
T_0 &\equiv \mathit{let}\ (a_1, b_1) = \mathit{bell}(a, b)\ \mathit{in}\ T_1
\end{aligned}
$$

Prima di mostrare il tipaggio per M, assumiamo che il contesto $!\Delta$ possieda già le seguenti funzioni con il relativo tipo associato:

- $\mathit{bell} : !((\mathit{qbit} \otimes \mathit{qbit}) \multimap (\mathit{qbit} \otimes \mathit{qbit}))$
- $\mathit{alice} : !((\mathit{qbit} \otimes \mathit{qbit}) \multimap !(\mathit{bit} \otimes \mathit{bit}))$
- $\mathit{bob} : !((!\mathit{bit} \otimes !\mathit{bit} \otimes \mathit{qbit}) \multimap \mathit{qbit})$

L'albero di derivazione è mostrato attraverso la notazione (x, y, z, R), che sta per: "la regola R è usata con le ipotesi presenti nelle linee x, y e z".

$$(2, \mathit{use})\ !\Delta \qquad \triangleright\ \mathit{using}\ (b, a, q)\ \mathit{in}\ T_0 : \mathit{qbit} \qquad 1$$

$$
\begin{array}{rll}
(3,7,let)\ !\Delta, b : qbit, a : qbit, q : qbit & \triangleright\ let\ (a_1, b_1) = bell(a,b)\ in\ T_1 : qbit & 2 \\
(4,app)\ !\Delta, a : qbit, b : qbit & \triangleright\ bell(a,b) : (qbit \otimes qbit) & 3 \\
(5,6,tup)\ !\Delta, a : qbit, b : qbit & \triangleright\ (a,b) : (qbit \otimes qbit) & 4 \\
(var)\ !\Delta, a : qbit & \triangleright\ a : qbit & 5 \\
(var)\ !\Delta, b : qbit & \triangleright\ b : qbit & 6 \\
(8,12,let)\ !\Delta, a_1 : qbit, b_1 : qbit, q : qbit & \triangleright\ let\ (bq_1, ba_2) = & \\
 & \quad alice(q, a_1)\ in\ T_2 : qbit & 7 \\
(9,app)\ !\Delta, q : qbit, a_1 : qbit & \triangleright\ alice(q, a_1) : !(bit \otimes bit) & 8 \\
(10,11,tup)\ !\Delta, q : qbit, a_1 : qbit & \triangleright\ (q, a_1) : (qbit \otimes qbit) & 9 \\
(var)\ !\Delta, q : qbit & \triangleright\ q : qbit & 10 \\
(var)\ !\Delta, a_1 : qbit & \triangleright\ a_1 : qbit & 11 \\
(13,app)\ !\Delta, bq_1 :!bit, ba_2 :!bit, b_1 : qbit & \triangleright\ bob(bq_1, ba_2, b_1) : qbit & 12 \\
(*,tup)\ !\Delta, bq_1 :!bit, ba_2 :!bit, b_1 : qbit & \triangleright\ (bq_1, ba_2, b_1) : (!bit \otimes !bit \otimes qbit) & 13 \\
(var)\ !\Delta, bq_1 :!bit, ba_2 :!bit & \triangleright\ bq_1 : !bit & 14 \\
(var)\ !\Delta, bq_1 :!bit, ba_2 :!bit & \triangleright\ ba_2 : !bit & 15 \\
(var)\ !\Delta, bq_1 :!bit, ba_2 :!bit, b_1 : qbit & \triangleright\ b_1 : qbit & 16
\end{array}
$$

Esempio 4.1.6. Il seguente esempio mostra il tipaggio per un termine LTIQ che coinvolge il meccanismo delle variabili mutabili. Partendo dalla codifica in LTIQ dell'esempio 2.1.2, mostriamo l'albero di derivazione per il nostro generatore quantistico di bit. Per semplicità assumiamo che la funzione *sampleBit* di tipo $!(!\top \multimap !bit)$ sia già in $!\Delta$. Ora definiamo il termine M:

$$
\begin{array}{l}
M \equiv mut\ (a,b) = (zero, zero)\ in \\
\qquad set\ a = sampleBit()\ in \\
\qquad set\ b = sampleBit()\ in\ (a,b)
\end{array}
$$

Per brevità, sono definiti anche i seguenti termini:

$$
\begin{array}{l}
M_1 \equiv set\, b = \ sampleBit()\ in (a,b) \\
M_0 \equiv set\, a = \ sampleBit()\ in\ M_1
\end{array}
$$

Ora mostriamo il tipaggio di M seguendo la notazione dell'esempio precedente:

$$
\begin{array}{rll}
(2,5,mut)\ !\Delta & \triangleright\ mut\ (a,b) = (zero, zero)\ in\ M_0 : !(bit \otimes bit) & 1 \\
(3,4,tup)\ !\Delta & \triangleright\ (zero, zero) : !(bit \otimes bit) & 2
\end{array}
$$

$$
\begin{array}{rll}
(const)\ !\Delta & \triangleright\ zero : !bit & 3 \\
(const)\ !\Delta & \triangleright\ zero : !bit & 4 \\
(6,7,set)\ !\Delta, a :!bit, b :!bit & \triangleright\ set\ a = sampleBit()\ in\ M_1 : !(bit \otimes bit) & 5 \\
(tup)\ !\Delta, a :!bit, b :!bit & \triangleright\ () : !\top & 6 \\
(8,9,set)\ !\Delta, a :!bit, b :!bit & \triangleright\ set\ b = sampleBit()\ in\ (a,b) : !(bit \otimes bit) & 7 \\
(tup)\ !\Delta, a :!bit, b :!bit & \triangleright\ () : !\top & 8 \\
(10,11,tup)\ !\Delta, a :!bit, b :!bit & \triangleright\ (a,b) : !(bit \otimes bit) & 9 \\
(var)\ !\Delta, a :!bit, b :!bit & \triangleright\ a : !bit & 10 \\
(var)\ !\Delta, a :!bit, b :!bit & \triangleright\ b : !bit & 11
\end{array}
$$

Esempio 4.1.7. Ora mostriamo l'esecuzione della codifica in LTIQ dell'esempio 2.1.3. In questo esempio si vuole enfatizzare come il sistema di tipi non permetta il tipaggio di un termine che cerca di violare il teorema di non duplicazione. Riprendendo l'esempio 2.1.3 definiamo il termine M:

$$
\begin{array}{l}
let\ entangle = op(x_0 : qbit, y_0 : qbit) : (qbit \otimes qbit)\ \{ \\
\quad let\ x_1\ = H(x_0)\ in \\
\quad CNOT(x_1, y_0) \\
\}\ in\ using\ q\ in\ entangle(q,q)
\end{array}
$$

Per semplicità assumiamo che la funzione *entangle* sia tipata correttamente e sia nel contesto di tipaggio $!\Delta$. È facile vedere che il tipaggio per il termine $entangle(q,q)$ nel contesto $!\Delta, q : qbit$ fallisce:

$$
\dfrac{\dfrac{!\Delta, q : qbit \triangleright q : qbit \quad !\Delta \triangleright q : ?}{!\Delta, q : qbit \triangleright (q,q) : ?}}{!\Delta, q : qbit \triangleright entangle(q,q) : ?}
$$

Ciò accade in quanto la variabile q è già utilizzata una volta nel tipaggio della tupla (q,q). Perciò il nostro sistema di tipi è in grado di catturare l'errore come volevamo.

Esempio 4.1.8. Mostriamo un ulteriore esempio in cui il tipaggio per un termine che cerca di violare il teorema di non duplicazione fallisce. Riprendiamo il termine M dell'esempio 3.4.4 così definito:

$$
\begin{array}{l}
using\ x_0\ in \\
let\ x_1 = x_0\ in \\
CNOT(x_0, x_1)
\end{array}
$$

Proviamo a tipare questo termine nel contesto di tipaggio vuoto:

$$\dfrac{\dfrac{}{x_0 : qbit \triangleright x_0 : qbit} \quad \dfrac{\dfrac{\dfrac{}{x_1 : qbit \triangleright x_1 : qbit} \quad \cdot \triangleright x_0 : ?}{x_1 : qbit \triangleright (x_0, x_1) : ?}}{x_1 : qbit \triangleright CNOT(x_0, x_1) : ?}}{\dfrac{x_0 : qbit \triangleright let\ x_1 = x_0\ in\ CNOT(x_0, x_1) : ?}{\cdot \triangleright using\ x_0\ in\ let\ x_1 = x_0\ in\ CNOT(x_0, x_1) : ?}}$$

Si noti come il nostro sistema di tipi sia in grado di catturare il fatto che x_1 sia in realtà lo stesso *qbit* puntato da x_0.

Esempio 4.1.9. Ora è mostrato un termine di LTIQ che crea un quantum bit senza utilizzarlo. Il nostro sistema di tipi non permetterà il tipaggio di una situazione del genere, infatti la linearità del tipo *qbit* garantisce che la variabile associata non possa essere scartata. Sia M il nostro termine così definito: *using* x *in* 1, per le regole di tipaggio avremo la seguente situazione:

$$\dfrac{\dfrac{}{!\Delta, x : qbit \triangleright 1 : ?}}{!\Delta \triangleright using\ x\ in\ 1 : ?}$$

Infatti dalla regola di tipaggio Const, una costante c è tipabile solamente in un ambiente in cui *tutti* gli identificatori abbiano un tipo duplicabile.

A questo punto resta da dimostrare formalmente quali garanzie il nostro sistema di tipi ci offre.

Capitolo 5

Proprietà del sistema di tipi

In questo capitolo dimostreremo la più importante proprietà del nostro sistema di tipi, ossia la *type safety*. Prima di enunciare i teoremi e le dimostrazioni principali è necessario riportare alcune proprietà base del nostro sistema di tipi.

Innanzitutto, consideriamo nuovamente la definizione di valore, già vista nel capitolo per la semantica operazionale, e definiamo le nozione di *configurazione valore* e di *configurazione errore*.

Definizione 5.0.1. Un *valore* è un termine definito dalle seguente sintassi:

$$V ::= c \mid x \mid (V_1, ..., V_n)$$

Una *configurazione valore* è una configurazione:

$$\langle \theta, \phi, \sigma \rangle [Q, V]$$

tale per cui $\text{Vars}(V) \subseteq \text{dom}(\phi)$.

Informalmente, una *configurazione valore* è una configurazione che non può ridurre in alcun modo e una volta raggiunta indica che la computazione è terminata correttamente producendo un risultato. Si noti che una *configurazione valore* è valida solamente se tutte le variabili libere in V appartengono al dominio di ϕ: una variabile che punti ad un quantum bit non può ridurre ulteriormente.

Definizione 5.0.2. Uno *stato errore* è una configurazione che non è uno *stato valore* ma dalla quale nessuna riduzione è possibile.

La definizione 5.0.2 cattura la nozione di *errore* durante la computazione: la configurazione in questione è *bloccata* in quanto non può ridurre in nessun modo, ma allo stesso tempo non essendo una *configurazione valore* essa indica che il programma non

termina correttamente. Ad esempio, assumendo che x sia in ϕ l'esecuzione del termine $CNOT(x, x)$ porta ad una configurazione errore in quanto non si hanno due qbit distinti.

La più importante garanzia del nostro sistema di tipi è la *type safety*. Intuitivamente, LTIQ è type-safe in quanto se abbiamo una configurazione chiusa (gli identificatori liberi del termine sono contenuti negli store) e ben tipata (non è una configurazione mal formata) allora essa non riduce ad una configurazione errore (ma potrebbe non terminare a causa della ricorsione).

Iniziamo col definire cosa s'intende per *configurazione chiusa* e *ben tipata*.

Definizione 5.0.3. Siano $\theta \in \Theta$, $\phi \in \Phi$, $\sigma \in \Sigma$ rispettivamente un *fun store*, uno *store classico* e uno *store quantistico*. La tripla di *store* $\langle \theta, \phi, \sigma \rangle$ è detta essere *chiusa*, se per ogni $f \in \mathrm{dom}(\theta)$ tale che $\theta(f) = (\overline{x}, M)$ si ha che:

- Ogni variabile $x \in \mathrm{FV}(M)$ è nel dominio di ϕ o nel dominio di σ.
- Ogni occorrenza libera di un identificatore di funzione in M è contenuto in θ.

Definizione 5.0.4. Sia $\langle \theta, \phi, \sigma \rangle$ una tripla di store *chiusa*, e sia $\langle \theta, \phi, \sigma \rangle [Q, M]$ una configurazione. Diremo che $\langle \theta, \phi, \sigma \rangle [Q, M]$ è una *configurazione chiusa* se e solo se

- Ogni $x \in \mathrm{FV}(M)$ è nel dominio di ϕ oppure nel dominio di σ.
- Ogni occorrenza di un identificatore di funzione libero in M è in θ.

Per definire cosa s'intende per configurazione *ben tipata* sono introdotte le successive tre definizioni. In particolare:

- La definizione 5.0.5 mostra qual è la relazione tra un *classical store* σ e un contesto di tipaggio Δ.
- La definizione 5.0.6 pone enfasi su come siano tipate le variabili presenti in un *quantum store*.
- La definizione 5.0.7 mostra la relazione tra un *fun store* e il contesto di tipaggio Δ.

Definizione 5.0.5. Un *classical store* $\sigma \in \Sigma$, è detto *ben tipato* rispetto ad un contesto di tipaggio $!\Delta, \Gamma$, scritto con $!\Delta, \Gamma \triangleright \sigma$, se e solo se: $\mathrm{dom}(\sigma) \subseteq |!\Delta|$ e $\forall x \in \mathrm{dom}(\sigma)$ si ha che: $!\Delta \triangleright \sigma(x) : !\Delta(x)$.

Informalmente, la definizione 5.0.5 ci dice che un contesto di tipaggio tipa uno *store classico* solamente se tutte le variabili contenute nello *store classico* hanno un tipo *non-lineare* $!A$. Il tipo $!A$ deve corrispondere esattamente al tipo del valore a cui la variabile punta nello *store classico*.

Definizione 5.0.6. Un *quantum store* $\phi \in \Phi$, è detto *ben tipato* rispetto ad un contesto di tipaggio $!\Delta, \Gamma$, scritto con $!\Delta, \Gamma \triangleright \phi$, se e solo se l'insieme $Z = \{x \mid \Gamma(x) = qbit\}$ è tale che $Z \in \mathcal{P}(\mathrm{dom}(\phi))$.

La definizione 5.0.6 mette in luce la relazione tra le variabili contenute nello *store quantistico* ϕ e le variabili contenute in Γ per un contesto di tipaggio. In particolare, ogni variabile in ϕ punta ad un qbit nello stato quantistico Q, perciò vorremmo che queste variabili siano di tipo *qbit* nell'ambiente *lineare* Γ che tipa lo store ϕ. È importante notare che un qualsiasi sottoinsieme di variabili presenti nel dominio di ϕ possa essere presente in Γ. Ciò è dovuto alla *linearità* dei *qbit*, in un certo momento della computazione una variabile x può essere in ϕ ma non è detto che sia in Γ, basti pensare al caso in cui x sia stato già misurato o al caso della tupla in cui il contenuto di ϕ è ripartito per diversi ambienti Γ_i.

Definizione 5.0.7. Un *fun store* $\theta \in \Theta$, è detto *ben tipato* rispetto ad un contesto di tipaggio Δ, scritto con $\Delta \triangleright \theta$, se e solo se: $\mathrm{dom}(\theta) \subseteq |!\Delta|$ e per ogni $f \in \mathrm{dom}(\theta)$, con $\theta(f) = (\overline{x}, M)$ se $!\Delta, \overline{x :!^m A} \triangleright M : B$ allora $!\Delta(f) =!(!^m \overrightarrow{A} \multimap B)$.

Si ricordi che il linguaggio Q# non permette la dichiarazione di variabili globali (si veda Nota Bene 4.0.2). Per mantenere un certo livello di semplicità abbiamo visto, nelle regole di tipaggio, che il corpo di una funzione M è tipabile in un contesto $!\Delta, \Gamma$ solamente nel caso in cui tutte le sue variabili libere non hanno un tipo *lineare* associato. Per questo motivo, nella definizione 5.0.7, il corpo di una funzione è sempre tipato nel contesto *non lineare* $!\Delta$ e le uniche variabili che possono avere un tipo *lineare* associato sono quelle per gli argomenti della funzione.

Definizione 5.0.8. Sia $\langle \theta, \phi, \sigma \rangle [Q, M]$ una configurazione, allora è detta essere *ben tipata* se esiste un contesto $!\Delta, \Gamma$ tale che $!\Delta, \Gamma \triangleright \theta$, $!\Delta, \Gamma \triangleright \phi$, $!\Delta, \Gamma \triangleright \sigma$.

Per una migliore leggibilità scriveremo $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ per indicare che:

- $!\Delta, \Gamma \triangleright \theta$,
- $!\Delta, \Gamma \triangleright \phi$,
- $!\Delta, \Gamma \triangleright \sigma$.

Di seguito sono presentate delle proprietà del sistema di tipi che sono fondamentali per le dimostrazioni dei teoremi di 5.0.14 e 5.0.15 come mostrato successivamente.

Lemma 5.0.9. Sia $\theta \in \Theta$, $\phi \in \Phi$ e $\sigma \in \Sigma$. Sia $!\Delta, \Gamma_1, \Gamma_2$ un contesto di tipaggio. Se $!\Delta, \Gamma_1, \Gamma_2 \triangleright \theta, \phi, \sigma$ allora $!\Delta, \Gamma_i \triangleright \theta, \phi, \sigma$ con $1 \leq i \leq 2$.

Dimostrazione. Diretta dalle definizioni di tipaggio per gli store: dall'assunzione abbiamo che $\Delta \triangleright \sigma$, dalla definizione 5.0.5 abbiamo già che $!\Delta, \Gamma_i \triangleright \sigma$ per $1 \leq i \leq 2$. Allo stesso modo $!\Delta, \Gamma_i \triangleright \theta$. Ora, siccome $!\Delta, \Gamma_1, \Gamma_2 \triangleright \phi$ abbiamo che l'insieme $Z_1 = \{x \mid \Gamma_1(x) = qbit\}$ e l'insieme $Z_2 = \{x \mid \Gamma_2(x) = qbit\}$ sono tali che $Z_1, Z_2 \in \mathcal{P}(\mathrm{dom}(\phi))$ e perciò possiamo concludere che per i che assume i valori 1 e 2, $\Delta, \Gamma_i \triangleright \theta, \phi, \sigma$.
□

Lemma 5.0.10. Siano $!\Delta, \Gamma$ e $!\Delta', \Gamma$ due contesti di tipaggio tali che $\theta \subseteq \theta'$ e $!\Delta \subseteq !\Delta$. Se $!\Delta, \Gamma \triangleright \theta$ allora $!\Delta', \Gamma \triangleright \theta$.

Dimostrazione. Diretta dalla definizione 5.0.7. Infatti dall'assunzione sappiamo che ogni funzione in θ è tipata correttamente in $!\Delta$, dato che $!\Delta$ è contenuto in $!\Delta'$ allora ogni funzione in θ continuerà ad essere tipata correttamente in $!\Delta'$. □

Lemma 5.0.11. Siano $!\Delta, \Gamma$ e $!\Delta', \Gamma$ due contesti di tipaggio tali che $!\Delta \subseteq !\Delta'$. Se $!\Delta, \Gamma \triangleright \phi$ allora $!\Delta', \Gamma \triangleright \phi$.

Dimostrazione. Diretta dalla definizione 5.0.6. Infatti dall'assunzione sappiamo che ogni variabile in ϕ è ben tipata in Γ. Ma trivialmente esse continueranno ad essere ben tipate anche nell'ambiente $!\Delta', \Gamma$. □

Lemma 5.0.12. Sia $!\Delta, \Gamma$ un contesto di tipaggio. Sia $!\Delta'$ un altro contesto di tipagggio tale che $!\Delta \subseteq !\Delta'$, se $!\Delta, \Gamma \triangleright M : B$ allora $!\Delta', \Gamma \triangleright M : B$.

Dimostrazione. È evidente immediatamente per induzione sulla derivazione di tipaggio $!\Delta, \Gamma \triangleright M : B$. □

Lemma 5.0.13 (Lemma di Sostituzione)**.** Siano M e V, rispettivamente, un termine ed un valore di *LTIQ*. Sia $!\Delta, \Gamma_1, \Gamma_2$ un contesto di tipaggio. Nel caso in cui:

$$!\Delta, \Gamma_1 \triangleright V : A$$
$$!\Delta, \Gamma_2, x : A \triangleright M : B$$

allora si ha che:

$$!\Delta, \Gamma_1, \Gamma_2 \triangleright M[x/V] : B.$$

Dimostrazione. Il lemma è dimostrato per induzione sulla derivazione di tipaggio per il termine M.

- Var. $M \equiv y$. Per la regola di tipaggio per le variabili:

$$\frac{}{!\Delta, \Gamma_2, x : A \triangleright y : B}$$

  Abbiamo due casi:

– $y \not\equiv x$, perciò per inversione si ha che $\Gamma_2 = \emptyset$ e che A è un tipo duplicabile $A = !C$:

$$\frac{}{!\Delta, x :!C \triangleright y : B}$$

Ora per assunzione:

$$!\Delta, \Gamma_1 \triangleright V : !C$$

con $\Gamma_1 = \emptyset$ per inversione sull'assunzione.
Dato che $y[x/V] \equiv y$ allora $!\Delta \triangleright y : B$.

– $y \equiv x$, per inversione si ha che $\Gamma_2 = \emptyset$ e che:

$$\frac{}{!\Delta, x : A \triangleright x : A}$$

per assunzione:

$$!\Delta, \Gamma_1 \triangleright V : A$$

Dato che $y[x/V] \equiv V$ allora $!\Delta, \Gamma_1 \triangleright V : A$.

- Const. $M \equiv c$. Per la regola di tipaggio per le costanti:

$$\frac{}{!\Delta, \Gamma_2, x : A \triangleright c : !A_c}$$

Ma per inversione si ha che $\Gamma_2 = \emptyset$ e A è un tipo duplicabile $A = !B$. Dall'assunzione avremo che $\Gamma_1 = \emptyset$ e che:

$$!\Delta \triangleright V : !B$$

Dato che $M \equiv c \equiv M[x/v]$, allora $!\Delta \triangleright c : !A_c$

- Tuple1. $M \equiv ()$. La dimostrazione è identica a quella per *Const.*

- Tuple2. $M \equiv (M_1, ..., M_n)$. Per la regola di tipaggio per le tuple:

$$!\Delta, \Gamma_2, x : A \triangleright (M_1, ..., M_n) : !^m \vec{A}_{\otimes}$$

Per inversione abbiamo due casi:

– A è un tipo lineare, perciò:

$$\frac{\begin{array}{c}\vdots\ \pi_j \\ ...\quad !\Delta, \Gamma_j, x : A \triangleright M_j : !^m A_j \quad ...\end{array}}{!\Delta, \Gamma_2, x : A \triangleright (M_1, ..., M_n) : !^m (A_1 \otimes ... \otimes A_n)}$$

con $\Gamma_2 = \Gamma_1, ..., \Gamma_j, ..., \Gamma_n$. Siccome $(M_1, ..., M_n)[x/v] \equiv (M_1[x/v], ..., M_n[x/v])$ ma dall'inversione sappiamo che x compare come variabile libera solamente in M_j altrimenti la tupla $(M_1, ..., M_n)$ non sarebbe tipabile. Per questo motivo, $(M_1, ..., M_n)[x/v] \equiv (M_{i=0}^{j-1}, M_j[x/v], M_{j+1}^{n})$. Ora per assunzione abbiamo che:

$$!\Delta, \Gamma' \triangleright V : A$$

Applicando l'ipotesi induttiva avremo perciò:

$$\frac{\begin{array}{c}\vdots\ \pi'_j \\ ...\quad !\Delta, \Gamma_j, \Gamma' \triangleright M_j[x/V] : !^m A_j \quad ...\end{array}}{!\Delta, \Gamma_2, \Gamma' \triangleright (M_1, ..., M_n)[x/V] : !^m(A_1 \otimes ... \otimes A_n)}$$

– A è un tipo duplicabile $A = !B$, perciò dall'inversione:

$$\frac{\begin{array}{cc}\vdots\ \pi_1 & \vdots\ \pi_n \\ !\Delta, x :!B, \Gamma_1 \triangleright M_1 : !^m A_1 \quad ... & !\Delta, x :!B, \Gamma_n \triangleright M_n : !^m A_n\end{array}}{!\Delta, x :!B, \Gamma_2 \triangleright (M_1, ..., M_n) : !^m(A_1 \otimes ... \otimes A_n)}$$

con $\Gamma_2 = \Gamma_1, ..., \Gamma_n$. Per assunzione abbiamo anche che:

$$!\Delta, x :!B \triangleright V : !B$$

Siccome $(M_1, ..., M_n)[x/v] \equiv (M_1[x/v], ..., M_n[x/v])$, e per l'ipotesi induttiva possiamo concludere con:

$$\frac{\begin{array}{cc}\vdots\ \pi'_1 & \vdots\ \pi'_n \\ !\Delta, \Gamma_1 \triangleright M_1[x/V] : !^m A_1 \quad ... & !\Delta, \Gamma_n \triangleright M_n[x/V] : !^m A_n\end{array}}{!\Delta, \Gamma_2 \triangleright (M_1, ..., M_n)[x/V] : !^m(A_1 \otimes ... \otimes A_n)}$$

- USING. $M \equiv \mathit{using}\ \vec{x}\ \mathit{in}\ N$. Per l'$\alpha$-equivalenza possiamo assumere senza perdita di generalità che $x \notin \vec{x}$. Perciò $M[x/V] \equiv \mathit{using}\ x\ \mathit{in}\ N[x/V]$. Ora per assunzione:

$$!\Delta, \Gamma_2, x : A \triangleright \mathit{using}\ \vec{x}\ \mathit{in}\ N[x/V] : B$$

Per inversione:

$$\frac{\begin{array}{c}\vdots\ \pi_1 \\ !\Delta, x : A, \Gamma_2, x_1 : \mathit{qbit}, ..., x_n : \mathit{qbit} \triangleright N : A\end{array}}{!\Delta, \Gamma_2, x : A \triangleright \mathit{using}\ (x_1, ..., x_n)\ \mathit{in}\ N : A}$$

per assunzione abbiamo anche che:

$$!\Delta, \Gamma_1 \triangleright V : A$$

Dall'ipotesi induttiva possiamo concludere che:

$$\frac{\begin{array}{c}\vdots\ i.i.\\ !\Delta, \Gamma_1, \Gamma_2, x_1 : qbit, ..., x_n : qbit \triangleright N[x/V] : A\end{array}}{!\Delta, \Gamma_2, x : A \triangleright \mathit{using}\ (x_1, ..., x_n)\ \mathit{in}\ N[x/V] : A}$$

- Let. $M \equiv \mathit{let}\ \vec{x} = M\ \mathit{in}\ N$. Per l'$\alpha$-equivalenza possiamo assumere senza perdita di generalità che $x \notin \vec{x}$. Perciò $M[x/V] \equiv \mathit{let}\ x = N[x/V]\ \mathit{in}\ P[x/V]$. Per assunzione si ha che $!\Delta, \Gamma_2, x : A \triangleright \mathit{let}\ \vec{x} = N\ \mathit{in}\ P : B$. Si hanno due casi:
  - A è un tipo lineare, per cui per inversione:

$$\frac{\begin{array}{cc}\vdots\ \pi_1 & \vdots\ \pi_2\\ !\Delta, \Gamma_1' \triangleright N : !^m \vec{A}_{\otimes} & !\Delta, \Gamma_2', \overline{x :!^m A} \triangleright P : B\end{array}}{!\Delta, \Gamma' \triangleright \mathit{let}\ (x_1, ..., x_n) = N\ \mathit{in}\ P : B}$$

    con $\Gamma' = \Gamma_2, x : A$ e $x : A \in \Gamma_1'$ oppure $x : A \in \Gamma_2'$. Per assunzione abbiamo anche che:

$$!\Delta, \Gamma'' \triangleright V : A$$

    - Se $x : A \in \Gamma_1'$ abbiamo che $M[x/V] \equiv \mathit{let}\ x = N[x/V]\ \mathit{in}\ P$, quindi applicando l'ipotesi induttiva possiamo concludere con:

$$\frac{\begin{array}{cc}\vdots\ i.i & \vdots\ \pi_2\\ !\Delta, \Gamma_3 \triangleright N[x/V] : !^m \vec{A}_{\otimes} & !\Delta, \Gamma_2', \overline{x :!^m A} \triangleright P : B\end{array}}{!\Delta, \Gamma' \triangleright \mathit{let}\ (x_1, ..., x_n) = N\ \mathit{in}\ P : B}$$

      con $\Gamma_3 = (\Gamma_1' \setminus \{x : A\}), \Gamma''$.
    - Se $x : A \in \Gamma_2'$ segue la struttura dell'altro caso, con la differenza che $M[x/V] \equiv \mathit{let}\ x = N\ \mathit{in}\ P[x/V]$ e che l'ipotesi induttiva è applicata sull'albero di derivazione per P.
  - A è un tipo duplicabile: $A =!C$. Per inversione:

$$\frac{\begin{array}{cc}\vdots\ \pi_1 & \vdots\ \pi_2\\ !\Delta, x :!C, \Gamma_1' \triangleright N : !^m \vec{A}_{\otimes} & !\Delta, x :!C, \Gamma_2', \overline{x :!^m A} \triangleright P : B\end{array}}{!\Delta, x :!C, \Gamma_2 \triangleright \mathit{let}\ (x_1, ..., x_n) = N\ \mathit{in}\ P : B}$$

con $\Gamma_2 = \Gamma_1', \Gamma_2'$. Per assunzione abbiamo anche che:

$$!\Delta \triangleright V : !B$$

Applicando l'ipotesi induttiva possiamo concludere con:

$$\frac{\begin{array}{cc}\vdots\ i.i. & \vdots\ i.i. \\ !\Delta, \Gamma_1' \triangleright N[x/V] : !^m \overrightarrow{A}_{\otimes} & !\Delta, \Gamma_2', \overline{x :!^m A} \triangleright P[x/V] : B\end{array}}{!\Delta, \Gamma_2 \triangleright (let\ (x_1, ..., x_n) = N\ in\ P)[x/V] : B}$$

□

Il primo passo per dimostrare la *type safety* è mostrare che una configurazione ben tipata preserva il suo tipo dalle regole di riduzione. La proprietà è conosciuta anche con il nome di *Preservation*.

Teorema 5.0.14 (Preservation)**.** *Il teorema di Preservation afferma che se:*

$$!\Delta, \Gamma \triangleright M : A$$
$$\langle \theta, \phi, \sigma \rangle [Q, M] \rightarrow_p \langle \theta', \phi', \sigma' \rangle [Q', M']$$
$$!\Delta, \Gamma \triangleright \theta, \phi, \sigma$$

allora per qualche $!\Delta', \Gamma'$ *si ha che:*

$$!\Delta', \Gamma' \triangleright M' : A$$
$$!\Delta', \Gamma' \triangleright \theta', \phi', \sigma'$$

con:

$$\theta \subseteq \theta', \phi \subseteq \phi', \sigma \subseteq \sigma'$$

Dimostrazione. Dimostro per induzione sulla derivazione di un passo di riduzione:

$$\langle \theta, \phi, \sigma \rangle [Q, M] \rightarrow_p \langle \theta', \phi', \sigma' \rangle [Q', M']$$

Casi base Questi casi corrispondono agli assiomi nelle regole della semantica operazionale.

- Var. Per la regola:

$$\frac{x \in \mathrm{dom}(\sigma)}{\langle \theta, \phi, \sigma \rangle [Q, x] \rightarrow_1 \langle \theta, \phi, \sigma \rangle [Q, \sigma(x)]}\ (\textsc{Var})$$

Per assunzione $\Delta \triangleright x : A$, con $\Delta \triangleright \theta, \phi, \sigma$.
Dato che $x \in \mathrm{dom}(\sigma)$ sappiamo che x non può avere un tipo lineare, perciò $A = !B$. Per inversione quindi:

$$\frac{}{!\Delta', x :!B \triangleright x : !B}$$

con $\Delta = !\Delta = !\Delta', x :!B$.
Dalla definizione 5.0.5 si ha che per ogni y in σ:

$$\frac{}{!\Delta \triangleright \sigma(y) : !\Delta(y)}$$

possiamo concludere che $\Delta \triangleright \sigma(x) : !B$.

- GateApp. Per la regola:

$$\frac{\overline{x} \subseteq \mathrm{dom}(\phi) \quad x_i \neq x_j \quad 1 \leq i, j \leq n \quad i \neq j}{\langle \theta, \phi, \sigma \rangle [Q, U^n(x_1, ..., x_n)] \rightarrow_1 \langle \theta, \phi, \sigma \rangle [Q', (x_1, ..., x_n)]} (\textsc{Gate}_1)$$

Per assunzione $!\Delta, \Gamma \triangleright U^n(x_1, ..., x_n) : A$, con $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$.
Per inversione:

$$\frac{\dfrac{!\Delta, x_1 : qbit \triangleright x_1 : qbit \quad ... \quad !\Delta, x_n : qbit \triangleright x_n : qbit}{!\Delta, \Gamma \triangleright (x_1, ..., x_n) : \overrightarrow{qbit}_{\otimes}}}{!\Delta, \Gamma \triangleright U^n(x_1, ..., x_n) : \overrightarrow{qbit}_{\otimes}}$$

con $\Gamma = \{x_1 : qbit, ..., x_n : qbit\}$ e $A = \overrightarrow{qbit}_{\otimes}$.
Dato che $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$, per la regola di tipaggio delle tuple possiamo concludere che: $!\Delta, \Gamma \triangleright (x_1, ..., x_n) : \overrightarrow{qbit}_{\otimes}$

- Meas. Le regole per *meas* sono:

$$\frac{\phi(x) = r_i}{\langle \theta, \phi, \sigma \rangle [\alpha|Q_0\rangle + \beta|Q_1\rangle, meas(x)] \rightarrow_{|\alpha|^2} \langle \theta, \phi, \sigma \rangle [Q_0, zero]} (\textsc{Meas}_0)$$

$$\frac{\phi(x) = r_i}{\langle \theta, \phi, \sigma \rangle [\alpha|Q_0\rangle + \beta|Q_1\rangle, meas(x)] \rightarrow_{|\beta|^2} \langle \theta, \phi, \sigma \rangle [Q_1, one]} (\textsc{Meas}_1)$$

È considerato solamente il primo caso, il secondo è uguale.
Dall'assunzione $\Delta \triangleright meas(x) : A$, con $\Delta \triangleright \theta, \phi, \sigma$. Per inversione:

$$\frac{\dfrac{}{!\Delta, x : qbit \triangleright x : qbit}}{!\Delta, x : qbit \triangleright meas(x) : !bit}$$

con $\Delta = !\Delta, x : qbit$ e $A = !bit$.
Dato che per il lemma 5.0.9 si ha che $!\Delta \triangleright \theta, \phi, \sigma$, per la regola di tipaggio per le costanti possiamo concludere che:

$$!\Delta \triangleright zero : !bit$$

- FunDec. Per la regola:

$$\frac{}{\langle \theta, \phi, \sigma \rangle [Q, let\ f = op(\overrightarrow{x : t}) : t_{n+1}\{M\}\ in\ N] \to_1 \langle \theta[f/(\overline{x}, M)], \phi, \sigma \rangle [Q, N]}$$

Per assunzione, abbiamo che $!\Delta, \Gamma \triangleright let\ f = op(\overrightarrow{x : t}) : t_{n+1}\{M\}\ in\ N : C$ e che $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$.
Dall'inversione, risulta che:
$\mathrm{FV}(M) \cap \mathrm{dom}(\Gamma) = \emptyset$, $\mathrm{Dup}(\overrightarrow{t}) = !^m \overrightarrow{A}_{\otimes}$, $\mathrm{Dup}(t_{n+1}) = B$.

$$\frac{\begin{array}{cc} \vdots\ \pi_1 & \vdots\ \pi_2 \\ !\Delta', \overline{x : !^m A} \triangleright M : B & !\Delta', \Gamma \triangleright N : C \end{array}}{!\Delta, \Gamma \triangleright let\ f = op(\overrightarrow{x : t}) : t_{n+1}\{M\}\ in\ N : C}$$

con $!\Delta' = !\Delta, f : !(!^m \overrightarrow{A}_{\otimes} \multimap B)$.
Dalla Definizione 5.0.7, si ha che $!\Delta', \Gamma \triangleright \theta[f/(\overline{x}, M)], \phi, \sigma$.
Infatti, dall'assunzione ogni $g \in \theta$ con $g \not\equiv f$, è in $!\Delta$.
Avendo da π_1 che $!\Delta', \overline{x : !^m A} \triangleright M : B$ per la definizione 5.0.7 si può concludere che $!\Delta', \Gamma \triangleright \theta[f/(\overline{x}, M)]$.
Per l'assunzione $!\Delta, \Gamma \triangleright \phi, \sigma$, quindi $!\Delta', \Gamma \triangleright \theta[f/(\overline{x}, M)], \phi, \sigma$.
Concludiamo utilizzando π_2 per mostrare che $!\Delta', \Gamma \triangleright N : C$.

- FunApp. Per la regola:

$$\frac{\theta(f) = (\{x_1, ..., x_n\}, N) \quad \mathrm{FV}(\overrightarrow{v}) \subseteq \mathrm{dom}(\phi)}{\langle \theta, \phi, \sigma \rangle [Q, f(v_1, ..., v_n)] \to_p \langle \theta, \phi, \sigma \rangle [Q, N[x_1/v_1, ..., x_n/v_n]]}\ (\text{App}_1)$$

Per assunzione $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ e $!\Delta, \Gamma \triangleright f(v_1, ..., v_n) : B$.
Dall'inversione, abbiamo $!\Delta = !\Delta', f : !(!^m \overrightarrow{A}_{\otimes} \multimap B)$ e che:

$$\frac{\dfrac{!\Delta, \Gamma_1 \triangleright v_1 : !^m A \quad ... \quad !\Delta, \Gamma_n \triangleright v_n : !^m A}{!\Delta, \Gamma \triangleright \overrightarrow{v} : !^m \overrightarrow{A}_{\otimes}}}{!\Delta, \Gamma \triangleright f(v_1, ..., v_n) : B}$$

con $\Gamma = \Gamma_1, ..., \Gamma_n$. Dalla definizione 5.0.7 sappiamo che $!\Delta, \overline{x : !^m A} \triangleright N : B$.
Per cui, dal substitution lemma 5.0.13 concludiamo con

$$!\Delta, \Gamma \triangleright N[x_1/v_1, ..., x_n/v_n] : B$$

.

- LetVal. Per la regola:

$$\frac{\mathrm{FV}(\vec{v}) \subseteq \mathrm{dom}(\phi)}{\langle \theta, \phi, \sigma \rangle [Q, let\ \vec{x} = \vec{v}\ in\ N] \to_1 \langle \theta, \phi, \sigma \rangle [Q, N[\vec{x}/\vec{v}]]}\ (\textsc{Let}_1)$$

Per assunzione $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ e $!\Delta, \Gamma \triangleright let\ \vec{x} = \vec{v}\ in\ N : B$.
Dall'inversione, risulta che:

$$\dfrac{\dfrac{!\Delta, \Gamma_1' \triangleright v_1 : !^m A \quad ... \quad !\Delta, \Gamma_n' \triangleright v_n : !^m A}{!\Delta, \Gamma_1 \triangleright \vec{v} : !^m \vec{A}_{\otimes}} \qquad \begin{matrix}\vdots\ \pi_2 \\ !\Delta, \Gamma_2, \overline{x :!^m A} \triangleright N : B\end{matrix}}{!\Delta, \Gamma \triangleright let\ \vec{x} = \vec{v}\ in\ N : B}$$

con $\Gamma = \Gamma_1, \Gamma_2$ e $\Gamma_1 = \Gamma_1', ..., \Gamma_n'$.
Perciò per il lemma 5.0.13, concludiamo con $!\Delta, \Gamma \triangleright N[x_1/v_1, ..., x_n/v_n] : B$.

- MutVal. Per la regola:

$$\frac{\mathrm{FV}(\vec{v}) = \emptyset \quad \mathrm{Up}(\sigma, \vec{x}, \vec{v}) = \sigma'}{\langle \theta, \phi, \sigma \rangle [Q, mut\ \vec{x} = \vec{v}\ in\ N] \to_1 \langle \theta, \phi, \sigma' \rangle [Q, N]}$$

Per assunzione, $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ e $!\Delta, \Gamma \triangleright mut\ \vec{x} = \vec{v}\ in\ N : B$.
Per inversione abbiamo il seguente albero di derivazione:

$$\dfrac{\begin{matrix}\vdots\ \pi_1 \\ !\Delta \triangleright \vec{v} : !\vec{A}_{\otimes}\end{matrix} \qquad \begin{matrix}\vdots\ \pi_2 \\ !\Delta, \overline{x :!A}, \Gamma \triangleright N : B\end{matrix}}{!\Delta, \Gamma \triangleright mut\ (x_1, ..., x_n) = \vec{v}\ in\ N : B}$$

Per semplicità definiamo $!\Delta' = !\Delta, \overline{x :!A}$. Ora mostriamo che $!\Delta', \Gamma \triangleright \theta, \phi, \sigma'$.

  - per l'assunzione $!\Delta, \Gamma \triangleright \theta, \phi$, perciò segue direttamente dai lemmi 5.0.10 e 5.0.11 che $!\Delta', \Gamma \triangleright \theta, \phi$;
  - dalla definizione di Up, abbiamo che $\sigma' = \sigma[x_1/v_1], ..., [x_n/v_n]$. Siccome per l'assunzione si ha che $!\Delta, \Gamma \triangleright \sigma$, sappiamo che ogni $x \in \sigma$ è ben tipata in $!\Delta$. Per inversione su π_1 si ha che $!\Delta \triangleright v_i : !A_i$ ma dall'assunzione abbiamo che $\Delta'(x_i) = !A_i$ per ogni $1 \leq i \leq n$. Di conseguenza $!\Delta', \Gamma \triangleright \sigma'$.

Per cui dalla derivazione π_2 concludiamo con $!\Delta, \overline{x :!A}, \Gamma \triangleright N : B$.

- SetVal. Per la regola:

$$\frac{\overline{x} \subseteq \mathrm{dom}(\sigma) \quad \mathrm{FV}(\vec{v}) = \emptyset \quad \mathrm{Up}(\sigma, \vec{x}, \vec{v}) = \sigma'}{\langle \theta, \phi, \sigma \rangle [Q, set\ \vec{x} = \vec{v}\ in\ N] \to_1 \langle \theta, \phi, \sigma' \rangle [Q, N]}$$

Per assunzione, $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ e $!\Delta, \Gamma \triangleright \mathit{set}\ \vec{x} = \vec{v}\ \mathit{in}\ N : B$.
Per inversione abbiamo il seguente albero di derivazione:

$$\frac{\begin{array}{cc}\vdots\ \pi_1 & \vdots\ \pi_2 \\ !\Delta \triangleright \vec{v} : !\vec{A}_{\otimes} & !\Delta, \Gamma \triangleright N : B\end{array}}{!\Delta, \Gamma \triangleright \mathit{set}\ (x_1, ..., x_n) = \vec{v}\ \mathit{in}\ N : B}$$

con $!\Delta = !\Delta', \overline{x : !A}$. Ora mostriamo che $!\Delta, \Gamma \triangleright \theta, \phi, \sigma'$.

- per l'assunzione abbiamo già che $!\Delta, \Gamma \triangleright \theta, \phi$,
- dalla definizione di Up, abbiamo che $\sigma' = \sigma[x_1/v_1], ..., [x_n/v_n]$. Siccome per l'assunzione si ha che $!\Delta, \Gamma \triangleright \sigma$, sappiamo che ogni $x \in \sigma$ è ben tipata in $!\Delta$. Per inversione su π_1 si ha che $!\Delta \triangleright v_i : !A_i$ ma dall'assunzione abbiamo che $\Delta(x_i) = !A_i$ per ogni $1 \leq i \leq n$. Di conseguenza $!\Delta, \Gamma \triangleright \sigma'$.

Per cui dalla derivazione π_2 concludiamo con $!\Delta, \overline{x : !A}, \Gamma \triangleright N : B$.

- Using. Per la regola:

$$\frac{\vec{r} \text{ sono } \mathit{Qvar} \text{ fresche}}{\langle \theta, \phi, \sigma \rangle [Q, \mathit{using}\ \vec{x}\ \mathit{in}\ M] \to_1 \langle \theta, \phi[\overrightarrow{x/r}], \sigma \rangle [Q \otimes |\overrightarrow{r \to 0}\rangle, M]}\ (\textsc{Use})$$

Per l'assunzione $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ e $!\Delta, \Gamma \triangleright \mathit{using}\ \vec{x}\ \mathit{in}\ M : B$.
Per inversione si ha che:

$$\frac{\vdots\ \pi_1}{!\Delta, \Gamma, \overline{x : \mathit{qbit}} \triangleright M : A}{!\Delta, \Gamma \triangleright \mathit{using}\ (x_1, ..., x_n)\ \mathit{in}\ M : A}$$

Per la definizione 5.0.6 e per via di $!\Delta, \Gamma \triangleright \phi$, segue che $!\Delta, \Gamma, \overline{x : \mathit{qbit}} \triangleright \phi[x_1/r_j]...[x_n/r_{j+n}]$.
Per le definizioni 5.0.5, 5.0.7 segue in modo naturale che:
$!\Delta, \Gamma, \overline{x : \mathit{qbit}} \triangleright \theta, \phi[x_1/r_j]...[x_n/r_{j+n}], \sigma$.
Quindi da π_1 il teorema è vero: $!\Delta, \Gamma, \overline{x : \mathit{qbit}} \triangleright M : A$.

- IfVal. Le regole base del costrutto *if* sono le seguenti:

$$\frac{}{\langle \theta, \phi, \sigma \rangle [Q, \mathit{if\ true\ then}\ N_1\ \mathit{else}\ N_2] \to_1 \langle \theta, \phi, \sigma \rangle [Q, N_1]}\ (\textsc{If}_1)$$

$$\frac{}{\langle \theta, \phi, \sigma \rangle [Q, \mathit{if\ false\ then}\ N_1\ \mathit{else}\ N_2] \to_1 \langle \theta, \phi, \sigma \rangle [Q, N_2]}\ (\textsc{If}_0)$$

È considerato solamente il primo caso dato che il secondo segue esattamente la stessa struttura.
Per assunzione $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ e $!\Delta, \Gamma \triangleright \mathit{if\ true\ then}\ N_1\ \mathit{else}\ N_2 : A$.
Per inversione:

$$\frac{!\Delta \triangleright \mathit{true} : \mathit{bool} \quad \begin{matrix}\vdots\ \pi_1, \pi_2 \\ !\Delta, \Gamma \triangleright N_1, N_2 : A\end{matrix}}{!\Delta, \Gamma \triangleright \mathit{if\ true\ then}\ N_1\ \mathit{else}\ N_2 : A}$$

Dato che $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$, allora dalla dimostrazione π_1 si può concludere con $!\Delta, \Gamma \triangleright N_1 : A$.

Casi induttivi.

- LetM. Per la regola:

$$\frac{\langle \theta, \phi, \sigma \rangle [Q, M] \to_p \langle \theta', \phi', \sigma' \rangle [Q', M']}{\langle \theta, \phi, \sigma \rangle [Q, \mathit{let}\ \vec{x} = M\ \mathit{in}\ N] \to_p \langle \theta', \phi', \sigma' \rangle [Q', \mathit{let}\ \vec{x} = M'\ \mathit{in}\ N]}\ (\text{Let}_2)$$

Per assunzione abbiamo che:

$$!\Delta, \Gamma \triangleright \theta, \phi, \sigma$$
$$!\Delta, \Gamma \triangleright \mathit{let}\ \vec{x} = M\ \mathit{in}\ N : B$$

Per inversione:

$$\frac{\begin{matrix}\vdots\ \pi_1 \\ !\Delta, \Gamma_1 \triangleright M : \overrightarrow{!^m A}\end{matrix} \quad \begin{matrix}\vdots\ \pi_2 \\ !\Delta, \Gamma_2, \overrightarrow{x : !^m A} \triangleright N : B\end{matrix}}{!\Delta, \Gamma \triangleright \mathit{let}\ \vec{x} = M\ \mathit{in}\ N : B}$$

con $\Gamma = \Gamma_1, \Gamma_2$. Per ipotesi induttiva, abbiamo che:

$$!\Delta', \Gamma' \triangleright \theta', \phi', \sigma'$$
$$!\Delta', \Gamma' \triangleright M' : !^m \vec{A}$$
$$\sigma \subseteq \sigma', \phi \subseteq \phi', \theta \subseteq \theta'$$

Mostriamo ora che $!\Delta', \Gamma', \Gamma_2 \triangleright \theta', \phi', \sigma'$.
Innanzitutto $|\Gamma'| \cap |\Gamma_2| = \emptyset$, dal fatto che $\Gamma' = \Gamma'_1, \Gamma'_2$ con $\Gamma'_1 \subseteq \Gamma_1$ e $\Gamma'_2 \subseteq \mathrm{BV}(M)$. Inoltre, se $\phi' \neq \phi$ allora ϕ' sarà uguale a ϕ con l'aggiunta delle nuove variabili contenute in $\mathrm{BV}(M)$ dove ognuna di esse avrà associato un *qbit* 'fresco'. Per cui dalla definizione 5.0.6 segue che $!\Delta', \Gamma', \Gamma_2 \triangleright \phi'$. Ora dalle

definizioni 5.0.6, 5.0.5, per l'assunzione e l'ipotesi induttiva trivialmente avremo che $!\Delta', \Gamma', \Gamma_2 \triangleright \sigma', \theta'$. Quindi, dal giudizio di tipaggio per il costrutto *let* possiamo concludere che:

$$\frac{\begin{array}{cc}\vdots\ \pi_1' & \vdots\ \pi_2' \\ !\Delta', \Gamma' \triangleright M' : !^m\overrightarrow{A} & !\Delta', \Gamma_2, \overrightarrow{x :!^m A} \triangleright N : B\end{array}}{!\Delta', \Gamma', \Gamma_2 \triangleright \mathit{let}\ \vec{x} = M'\ \mathit{in}\ N : B}$$

con π_1' che deriva dall'ipotesi induttiva e con π_2' che esiste per il lemma 5.0.12.

- Tuple. Per la regola:

$$\frac{\mathrm{FV}(V_{i=0}^{j-1}) \subseteq \mathrm{dom}(\phi) \quad \langle \theta, \phi, \sigma \rangle [Q, M_j] \rightarrow_p \langle \theta', \phi', \sigma' \rangle [Q', M_j']}{\langle \theta, \phi, \sigma \rangle [Q, (V_{i=0}^{j-1}, M_j, M_{j+1}^n)] \rightarrow_p \langle \theta', \phi', \sigma' \rangle [Q', (V_{i=0}^{j-1}, {M_j}', M_{j+1}^n)]}\ (\textsc{Tup})$$

Per assunzione:

$$!\Delta, \Gamma \triangleright \theta, \phi, \sigma$$
$$!\Delta, \Gamma \triangleright (V_{i=0}^{j-1}, {M_j}', M_{j+1}^n) : !^m \vec{A}_{\otimes}$$

Per inversione:

$$\frac{\begin{array}{ccc}\vdots\ \pi_1 & \vdots\ \pi_j & \vdots\ \pi_n \\ !\Delta, \Gamma_1 \triangleright V_1 : !^m A_1 & \ldots & !\Delta, \Gamma_n \triangleright M_n : !^m A_n\end{array}}{!\Delta, \Gamma \triangleright (V_{i=0}^{j-1}, M_j, M_{j+1}^n) : !^m(A_1 \otimes \ldots \otimes A_n)}$$

con $\Gamma = \Gamma_1, ..., \Gamma_n$. Dall'ipotesi induttiva:

$$!\Delta', \Gamma_j' \triangleright \theta', \phi', \sigma'$$
$$!\Delta', \Gamma_j' \triangleright M_j' : !^m A_j$$
$$\sigma \subseteq \sigma', \phi \subseteq \phi', \theta \subseteq \theta'$$

Ora mostriamo che $!\Delta', \Gamma_1, ..., \Gamma_j', ..., \Gamma_n \triangleright \theta', \phi', \sigma'$. Innanzitutto, siccome $\Gamma_j' = \Gamma_{j1}', \Gamma_{j2}'$ con $\Gamma_{j1}' \subseteq \Gamma_j$ e $\Gamma_{j2}' \subseteq \mathrm{BV}(M_j)$ abbiamo che ogni x presente in Γ_j' non è in nessun altro ambiente Γ_i. Inoltre, se $\phi' \neq \phi$ allora ϕ' sarà uguale a ϕ con l'aggiunta delle nuove variabili presenti in $\mathrm{BV}(M_j)$, dove ognuna di esse avrà associato un *qbit* 'fresco'. Per cui dalla definizione 5.0.6 segue che $!\Delta', \Gamma_1, ..., \Gamma_j', ..., \Gamma_n \triangleright \phi'$. Ora dalle definizioni 5.0.6, 5.0.5, per l'assunzione e l'ipotesi induttiva trivialmente avremo che $!\Delta', \Gamma_1, ..., \Gamma_j', ..., \Gamma_n \triangleright \sigma', \theta'$.

Quindi, dal giudizio di tipaggio per le tuple e dal lemma 5.0.12, possiamo concludere che:

$$\frac{!\Delta', \Gamma_1 \triangleright V_1 : !^m A_1 \quad ... \quad !\Delta', \Gamma_n \triangleright M_n : !^m A_n}{!\Delta', \Gamma_1, ..., \Gamma'_j, ..., \Gamma_n \triangleright (V_{i=0}^{j-1}, M'_j, M_{j+1}^n) : !^m(A_1 \otimes ... \otimes A_n)}$$

- La dimostrazione per gli altri casi induttivi segue la struttura dei due casi precedenti.

□

Il secondo ed ultimo passo che servirà nella dimostrazione della *type safety* consiste nel mostrare che, partendo da una configurazione ben tipata si ha che tale configurazione è una configurazione valore se non può ridurre, altrimenti esiste una nuova configurazione raggiungibile per le regole di riduzione.

Teorema 5.0.15 (Progress)**.** *Sia* $\langle \theta, \phi, \sigma \rangle [Q, M]$ *una configurazione chiusa.*
Il teorema di Progress afferma che se:

$!\Delta, \Gamma \triangleright \theta, \phi, \sigma$
$!\Delta, \Gamma \triangleright M : A$

allora

$\langle \theta, \phi, \sigma \rangle [Q, M]$ *è una configurazione valore oppure*
$\langle \theta, \phi, \sigma \rangle [Q, M] \rightarrow_p \langle \theta', \phi', \sigma' \rangle [Q', M']$

Dimostrazione. La dimostrazione è fatta per induzione sulla derivazione del tipaggio $!\Delta, \Gamma \triangleright M : A$.

- Const. Dalla regola di tipaggio:

$$\frac{}{!\Delta \triangleright c : !A_c} \text{(Const)}$$

Dall'assunzione sia $\langle \theta, \phi, \sigma \rangle$ una tripla di store tali che $!\Delta \triangleright \theta, \phi, \sigma$. Si ha immediatamente che $\langle \theta, \phi, \sigma \rangle [Q, c]$ è una configurazione valore.

- Var. Dalla regola di tipaggio:

$$\frac{}{!\Delta, x : A \triangleright x : A} \text{(Var)}$$

Dall'assunzione sia $\langle \theta, \phi, \sigma \rangle$ una tripla di store tali che $!\Delta \triangleright \theta, \phi, \sigma$. Per la definizione di configurazione chiusa si hanno due casi:

– $x \in \sigma$, perciò per la regola di riduzione per le variabili:

$$\frac{x \in \mathrm{dom}(\sigma)}{\langle \theta, \phi, \sigma \rangle [Q, x] \to_1 \langle \theta, \phi, \sigma \rangle [Q, \sigma(x)]} \, (\textsc{Var})$$

– $x \in \phi$, si ha immediatamente che $\langle \theta, \phi, \sigma \rangle [Q, x]$ è una configurazione valore.

- Tuple. Dalla regola di tipaggio:

$$\frac{!\Delta, \Gamma_1 \triangleright M_1 : !^m A_1 \quad ... \quad !\Delta, \Gamma_n \triangleright M_n : !^m A_n}{!\Delta, \Gamma_1, ..., \Gamma_n \triangleright (M_1, ..., M_n) : !^m (A_1 \otimes ... \otimes A_n)} \, (\textsc{Tuple})$$

Per semplicità definiamo $\Gamma = \Gamma_1, ..., \Gamma_n$. Dall'assunzione sia $\langle \theta, \phi, \sigma \rangle$ una tripla di store tali che $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$. Dal lemma 5.0.9, abbiamo che $!\Delta, \Gamma_i \triangleright \theta, \phi, \sigma$ per ogni i che varia da 1 ad n. Ora dall'ipotesi induttiva abbiamo due casi:

- $\langle \theta, \phi, \sigma \rangle [Q, M_i]$ è una configurazione valore per ogni i compreso tra 1 e n. Per ipotesi induttiva, sappiamo che ogni M sarà della forma V con $\mathrm{FV}(V) \subseteq \mathrm{dom}(\phi)$, per cui $\langle \theta, \phi, \sigma \rangle [Q, (M_1, ..., M_n)]$ è una configurazione valore.

- $\langle \theta, \phi, \sigma \rangle [Q, M_i]$ è una configurazione valore per ogni i compreso tra 1 e $j-1$. Avremo quindi che M_j può effettuare un passo di riduzione $\langle \theta, \phi, \sigma \rangle [Q, M_j] \to_p \langle \theta, \phi, \sigma \rangle [Q, M_j']$. Siccome ogni M_i sarà della forma V_i con $\mathrm{FV}(V_i) \subseteq \mathrm{dom}(\phi)$, dalla regola di riduzione per le tuple possiamo concludere che:

$$\frac{\mathrm{FV}(V_{i=0}^{j-1}) \subseteq \mathrm{dom}(\phi) \quad \langle \theta, \phi, \sigma \rangle [Q, M_j] \to_p \langle \theta', \phi', \sigma' \rangle [Q', M_j']}{\langle \theta, \phi, \sigma \rangle [Q, (V_{i=0}^{j-1}, M_j, M_{j+1}^n)] \to_p \langle \theta', \phi', \sigma' \rangle [Q', (V_{i=0}^{j-1}, M_j', M_{j+1}^n)]} \, (\textsc{Tup})$$

- Gate. Dalla regola di tipaggio:

$$\frac{!\Delta, \Gamma \triangleright (M_1, ..., M_n) : (qbit \otimes ... \otimes_n qbit)}{!\Delta, \Gamma \triangleright U^n (M_1, ..., M_n) : (qbit \otimes ... \otimes_n qbit)} \, (\textsc{Gate})$$

Dall'assunzione sia $\langle \theta, \phi, \sigma \rangle$ una tripla di store tali che $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$. Ora dall'ipotesi induttiva abbiamo due casi:

- $\langle \theta, \phi, \sigma \rangle [Q, (M_1, ..., M_n)]$ è una configurazione valore e dato il suo tipo avremo che $(M_1, ..., M_n) = (x_1, ..., x_n)$ con $\mathrm{FV}((M_1, ..., M_n)) \subseteq \mathrm{dom}(\phi)$. Per la definizione 5.0.6 possiamo concludere dal passo di riduzione per l'applicazione dei gate:

$$\frac{\overline{x} \subseteq \mathrm{dom}(\phi) \quad x_i \neq x_j \quad 1 \leq i, j \leq n \quad i \neq j}{\langle \theta, \phi, \sigma \rangle [Q, U^n (x_1, ..., x_n)] \to_1 \langle \theta, \phi, \sigma \rangle [Q', (x_1, ..., x_n)]} \, (\textsc{Gate}_1)$$

- $\langle \theta, \phi, \sigma \rangle [Q, M_i]$ è una configurazione valore per ogni i compreso tra 1 e $j-1$. Avremo quindi che M_j può effettuare un passo di riduzione $\langle \theta, \phi, \sigma \rangle [Q, M_j] \to_p \langle \theta, \phi, \sigma \rangle [Q, M_j']$. Siccome ogni M_i sarà della forma V_i con $\mathrm{FV}(V_i) \subseteq \mathrm{dom}(\phi)$, dalla regola di riduzione possiamo concludere che:

$$\frac{\mathrm{FV}(V_{i=0}^{j-1}) \subseteq \mathrm{dom}(\phi) \quad \langle \theta, \phi, \sigma \rangle [Q, M_j] \to_p \langle \theta', \phi', \sigma' \rangle [Q', M_j']}{\langle \theta, \phi, \sigma \rangle [Q, U^n(V_{i=0}^{j-1}, M_j, M_{j+1}^n)] \to_p \langle \theta', \phi', \sigma' \rangle [Q', U^n(V_{i=0}^{j-1}, M_j', M_{j+1}^n)]}$$

- Gli altri casi seguono la stessa struttura per la dimostrazione. Si consideri che ogni altro caso ridurrà sempre in un passo in un'altra configurazione.

□

A questo punto, non ci resta che enunciare e dimostrare il teorema di *type safety.*

Teorema 5.0.16 (Type Safetey)**.** *Sia* $\langle \theta, \phi, \sigma \rangle [Q, M]$ *una configurazione chiusa. Sia* $!\Delta, \Gamma$ *un contesto di tipaggio tale che* $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ *e che* $!\Delta, \Gamma \triangleright M : A$*. Se:*

$$\langle \theta, \phi, \sigma \rangle [Q, M] \to^* \langle \theta', \phi', \sigma' \rangle [Q', M'] \quad \textit{ed} \quad \langle \theta', \phi', \sigma' \rangle [Q', M'] \not\to$$

allora $\langle \theta', \phi', \sigma' \rangle [Q', M']$ *è una configurazione valore.*

Dimostrazione. La dimostrazione segue dei teoremi 5.0.14 e 5.0.15. Sia $\langle \theta', \phi', \sigma' \rangle [Q', M']$ la configurazione tale che $\langle \theta, \phi, \sigma \rangle [Q, M] \to^* \langle \theta', \phi', \sigma' \rangle [Q', M']$ e che $\langle \theta', \phi', \sigma' \rangle [Q', M'] \not\to$. Siccome dall'assunzione esiste $!\Delta, \Gamma$ tale che $!\Delta, \Gamma \triangleright \theta, \phi, \sigma$ e che $!\Delta, \Gamma \triangleright M : A$, allora dal teorema 5.0.14 avremo che esiste $!\Delta', \Gamma'$ tali che $!\Delta', \Gamma' \triangleright \theta', \phi', \sigma'$ e che $!\Delta', \Gamma' \triangleright M' : A$. Perciò dal teorema 5.0.15 possiamo concludere che $\langle \theta', \phi', \sigma' \rangle [Q', M']$ è una configurazione valore. □

Il Teorema 5.0.16 afferma che il nostro sistema di tipi offre la garanzia di non avere una violazione del teorema di *non duplicazione*. In particolare, garantisce che un termine chiuso ben tipato di LTIQ non duplicherà (o cancellerà) lo stato di un qbit durante la sua esecuzione. L'utilizzo dell'ambiente lineare Γ è fondamentale per questo risultato e, come visto, esso garantisce che una variabile tipata linearmente una volta utilizzata non possa essere più accessibile in nessun punto del programma. Allo stesso modo, una variabile lineare dev'essere necessariamente utilizzata durante l'esecuzione del termine in questione, altrimenti il sistema di tipi non ne permetterà il tipaggio. In particolare, se una variabile ha tipo *qbit* (nel nostro sistema di tipi), allora sarà necessariamente in Γ e perciò non potrà essere nè duplicata nè cancellata.

Capitolo 6

Conclusioni e Sviluppi Futuri

In questo lavoro di tesi abbiamo definito un sistema di tipi lineare per il frammento di Q# ristretto a funzioni al prim'ordine e binding mutabili solo per valori classici. Tale sistema di tipi è in grado prevenire a tempo di compilazione se un programma scritto nel frammento di Q# sopra menzionato viola il teorema di non duplicazione. Per fare ciò, abbiamo introdotto dapprima un nuovo calcolo fondazionale, chiamato Linearly Typed Idealized Q# (LTIQ), che formalizza il (frammento di interesse del) linguaggio Q# ed il suo modello di esecuzione. LTIQ è introdotto da una sintassi astratta il cui scopo è quello di formalizzare i programmi scritti in Q#. Il modello di esecuzione di Q# è invece formalizzato da una semantica operazionale che descrive in modo non ambiguo il comportamento dell'esecuzione di un termine LTIQ. In seguito abbiamo definito un sistema di tipi lineare per LTIQ: ogni dato quantistico di un programma tipabile in questo sistema avrà necessariamente tipo lineare e quindi non potrà essere né duplicato né scartato dal programma. Quest'ultimo risultato è un'immediata conseguenza del teorema di sicurezza rispetto ai tipi (*type safety*), il principale teorema dimostrato in questo elaborato. Il teorema afferma che ogni qualvolta il sistema di tipi assegni un tipo lineare ad un dato all'interno di un programma, questo non sarà mai duplicato (o scartato) durante l'intera esecuzione del programma.

Gli esempi studiati evidenziano come l'aggiunta di un sistema di tipi lineari a Q# possa prevenire numerosi (e non banali) errori di programmazione che, se non identificati staticamente, possono portare a uno spreco di risorse quantistiche. In quest'ottica, i contributi presentati in questo lavoro possono leggersi come un primo passo verso lo sviluppo di un fondamento formale per il linguaggio Q# e l'estensione del suo sistema di tipi.

Sviluppi Futuri LTIQ è un calcolo fondazionale *minimale* per Q#: permette la definizione di funzioni solo al prim'ordine e non offre la possibilità di effettuare un binding mutabile tra variabili e dati quantistici, come invece avviene in Q#. È comunque un calcolo sufficientemente espressivo da permettere la scrittura di algoritmi quantistici

significativi presenti in letteratura, come visto nei vari esempi trattati in questa tesi. Un naturale sviluppo futuro per questo lavoro è l'estensione di LTIQ e del suo sistema di tipi all'intero linguaggio Q#. In questa direzione, una prima estensione da studiare è quella alle funzioni di ordine superiore. Si noti che in questo contesto anche le funzioni potranno avere un tipo lineare: si consideri, ad esempio, il caso in cui sia applicata parzialmente una funzione con un valore quantistico. Un'ulteriore estensione di LTIQ è l'aggiunta dei binding mutabili tra variabili e dati quantistici: in questo caso lo store quantistico non solo crescerà nel tempo ma modificherà anche il suo stato. Questo comportamento dovrà essere perciò modellato nella semantica operazionale con conseguenti modifiche anche nel sistema di tipi.

Un argomento che non è stato studiato in questo lavoro è dato dagli aspetti algoritmici associati al sistema di tipi di LTIQ. In particolare, non è stato trattato né il problema del *type checking* né quello della *type inference*. Nonostante lo sviluppo di algoritmi di type checking e di type inference sembri non problematico, il loro studio merita un'indagine propria, soprattutto nell'ottica delle già menzionate estensioni di LTIQ.

Appendice A

Preliminari Matematici

In questa appendice richiamiamo alcune definizioni e convenzioni notazionali matematiche di base del quantum computing.

Spazi Vettoriali Complessi Il principlae spazio vettoriale di cui ci occupiamo in questo lavoro è lo spazio dei vettori complessi: denotiamo con $\mathbb{C}^n$ lo spazio vettoriale dei vettori di numeri complessi $\begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}$ di dimensione n. Come è standard in letteratura sulla meccanica quantistica, utilizziamo la notazione di Dirac per rappresentare gli elementi di uno spazio vettoriale complesso. Denotiamo quindi con $|\psi\rangle$ (*ket*) un generico vettore $\begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix}$ in $\mathbb{C}^n$ e con $\langle\psi|$ (*bra*) il suo coniugato complesso $(\alpha_1^*, \ldots, \alpha_n^*)$. Scriviamo inoltre $|i\rangle$ per l'i-esimo elemento della base ortonormale canonica (cosicché, $|1\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$, $|2\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$, e così via). In particolare, ogni vettore $|\psi\rangle$ può essere scritto come una combinazione lineare della form $\sum_i \alpha_i |i\rangle$.

Dato un vettore $|\psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$, la norma di $|\psi\rangle$ è definita come $\sqrt{|\alpha|^2 + |\beta|^2}$, dove $|z|$ denota il modulo di un numero complesso z. Dati $|\psi\rangle = \sum_i \alpha_i |i\rangle$ e $|\varphi\rangle = \sum_i \beta_i |i\rangle$,

denotiamo il prodotto scalare (*inner product*) di $|\psi\rangle$ e $|\varphi\rangle$ come segue:

$$\langle\psi|\varphi\rangle = (\alpha_1^*, \ldots, \alpha_n^*) \begin{pmatrix} \beta_1 \\ \vdots \\ \beta_n \end{pmatrix}.$$

Qbit Lo spazio vettoriale $\mathbb{C}^2$ con il suo prodotto scalare è detto *spazio di Hilbert* a due dimensioni. L'insieme $\{|0\rangle, |1\rangle\}$ è una base di $\mathbb{C}^2$, detta base canonica. Di conseguenze, ogni vettore $|\psi\rangle$ in $\mathbb{C}^2$ può essere scritto come una combinazione lineare della forma $\alpha|0\rangle + \beta|1\rangle$, con α, β coefficienti complessi. La rilevanza dello spazio di Hilber bidimensionale per la computazione quantistica sta nel fatto che un qbit viene definito come un *vettore unitario* in $\mathbb{C}^2$, ovvero come un vettore $\alpha|0\rangle + \beta|1\rangle$ tale che $|\alpha|^2 + |\beta|^2 = 1$. I coefficienti complessi α e β sono detti *ampiezze di probabilità*, motivo per cui viene richiesto che $|\alpha|^2 + |\beta|^2 = 1$. I bit classici 0 e 1 corrispondono ai qbit $|0\rangle$ e $|1\rangle$, mentre un generico qbit $\alpha|0\rangle + \beta|1\rangle$ è detto essere in stato di sovrapposizione, in quanto una volta misurato avrà stato $|0\rangle$ con probabilità $|\alpha|^2$ e stato $|1\rangle$ con probabilità $|\beta|^2$.

Registri Quantistici Dati due spazi vettoriali $\mathbb{C}^n$ e $\mathbb{C}^m$, definiamo il prodotto tensore come la funzione $\otimes : \mathbb{C}^n \otimes \mathbb{C}^m \to \mathbb{C}^{nm}$ data da:

$$|\psi\rangle \otimes |\varphi\rangle = \begin{pmatrix} \alpha_1|\varphi\rangle \\ \vdots \\ \alpha_n|\varphi\rangle \end{pmatrix}$$

dove $|\psi\rangle = \sum_i \alpha_i |i\rangle$. È convenzone scrivere $|\psi\varphi\rangle$ (o anche $|\psi\rangle|\varphi\rangle$) al posto di $|\psi\rangle \otimes |\varphi\rangle$. Ad esempio, i vettori $|00\rangle$, $|01\rangle$, $|10\rangle$, e $|11\rangle$ corrispondono a:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}.$$

Un registro quantistico di n qubits è un elemento dello spazio di Hilbert 2^n-dimensionale $\mathbb{C}^{2^n}$. Si noti che l'insieme $\{|b_0, \ldots, b_{n-1}\rangle \mid b_i \in \{0,1\}\}$ è una base per $\mathbb{C}^{2^n}$. Ad esempio, un registro quantistico a due qbit è una combinazione lineare

$$|\psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$$

normalizzata: $\sum_{ij\in\{0,1\}^2} |\alpha_{ij}|^2 = 1$. Il vettore $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ è un registro quantistico a due qbit (e quindi un vettore in $\mathbb{C}^4$).

Operatori Unitari Un operatore lineare su $\mathbb{C}^n$ è una mappa $L : \mathbb{C}^n \to \mathbb{C}^n$ tale che:

$$L(\sum_i \alpha_i |i\rangle) = \sum_i \alpha L |i\rangle .$$

Al solito, possiamo rappresentare operatori lineari come matrici.

Data una matrice A di dimensione $n \times m$, scriviamo A^T per la trasposta di A (si ricordi che $(A^T)_{ji} = A_{ij}$), A^{-1} per l'inversa di A, ed I per la matrice identità (cosicché $AA^{-1} = I$). La coniugata A^* di A è definita punto-a-punto: $(A^*)_{ij} = (A_{ij})^*$. Infine, la matrice aggiunta $A^\dagger$ di A è definita come $(A^T)^*$.

Diciamo che una matrice A è *unitaria* se $A^\dagger = A^{-1}$ e che un operatore lineare è unitario se la sua rappresentazione matriciale lo è. Operatori/matrici unitari modellano operazioni su qbit. Infatti, vale il seguente risultato:

Teorema A.0.1. *Un operaotre lineare trasforma un qbit in un qubit (ovvero preserva vettori normalizzati) se e solo se è unitario.*

Un operatore/matrice unitario/a è detto anche porta quantistica (*quantum gate*). Ad esempio, l'operatore di Hadamard è definito dalla seguente matrice:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

L'operatore di Hadamard è utilizzato per mettere qbit in sovrapposzione. Per esempio, si ha che $H |0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$. Concludiamo questa appendice enunciando il teorema di non duplicazione.

Teorema A.0.2 (Non Duplicazione)**.** *Non esiste una trasformazione unitaria M tale che $M |\psi\rangle |0\rangle = |\psi\rangle |\psi\rangle$, per ogni stato $|\psi\rangle$.*

Appendice B

Esempi

B.0.1 Errori

```
namespace teleportation {
  open Microsoft.Quantum.Canon;
  open Microsoft.Quantum.Intrinsic;
  open Microsoft.Quantum.Diagnostics;

  operation entangle(x:Qubit, y:Qubit):(Qubit, Qubit) {
      H(x);
      CNOT(x, y);
      return (x, y);
  }

  @EntryPoint
  operation main():Unit {
      using (x = Qubit()) {
        entangle(x, x);
      }
  }
}
```

Figura B.1: Implementazione dello stato entagled in Q#

B.0.2 Generatore di numeri casuali

$$
\begin{array}{l}
let\ entangle = op(x_0{:}qbit,\ y_0{:}qbit){:}(qbit \otimes qbit)\ \{ \\
\quad let\ x_1 = H(x_0)\ in \\
\quad CNOT(x_1,\ y_0) \\
\}\ in \\
using\ q\ in\ entangle(q,\ q)
\end{array}
$$

Figura B.2: Encoding dello stato entangled in LTIQ

$$
\dfrac{\dfrac{}{x_0 : qbit \triangleright x_0 : qbit} \quad \dfrac{\dfrac{}{x_1 : qbit \triangleright x_1 : qbit} \quad \cdot \triangleright x_0 : ?}{x_1 : qbit \triangleright (x_0, x_1) : ?}}{\dfrac{x_0 : qbit \triangleright let\ x_1 = x_0\ in\ CNOT(x_0, x_1) : ?}{\cdot \triangleright using\ x_0\ in\ let\ x_1 = x_0\ in\ CNOT(x_0, x_1) : ?}}
$$

```
namespace quantumGenerator {
  open Microsoft.Quantum.Canon;
  open Microsoft.Quantum.Intrinsic;
  open Microsoft.Quantum.Diagnostics;

  operation sampleBit():Result {
    using (x = Qubit()) {
        H(x);
        return M(x);
    }
  }

  @EntryPoint()
  operation main(): Unit {
    mutable (a, b) = (Zero, Zero);
    set a = sampleBit();
    set b = sampleBit();
  }
}
```

Figura B.3: Implementazione di un generatore di bit casuali in Q#

B.0.3 Procedura di teletrasporto

```
let sampleBit = op():bit {
    using x in meas(H(x))
} in
mut (a, b) = (zero, zero) in
  set a = sampleBit() in
  set b = sampleBit() in
  (a, b)
```

Figura B.4: Encoding di un generatore di bit casuali in LTIQ

$$
\begin{aligned}
M &\to \langle \theta, \cdot, \cdot \rangle [\cdot, mut\ (a,b) = (zero, zero)\ in\ M_0] \\
&\to \langle \theta, \cdot, [a \to Z, b \to Z] \rangle [\cdot, set\ a = sampleBit()\ in\ M_1] \\
&\to \langle \theta, \cdot, [a \to Z, b \to Z] \rangle [\cdot, set\ a = using\ (x_0)\ in\ meas(H(x_0))\ in\ M_1] \\
&\to \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle, set\ a = meas(H(x_0))\ in\ M_1] \\
&\to \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [\frac{|r_0 \to 0\rangle + |r_0 \to 1\rangle}{\sqrt{2}}, set\ a = meas(x_0)\ in\ M_1] \\
&\to \begin{cases} \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle, set\ a = Z\ in\ M_1] \\ \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [|r_0 \to 1\rangle, set\ a = O\ in\ M_1] \end{cases} \\
&\to \begin{cases} \langle \theta, [x_0 \to r_0], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle, set\ b = using\ (x_1)\ in\ meas(H(x_1))\ in\ (a,b)] \\ \langle \theta, [x_0 \to r_0], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle, set\ b = using\ (x_1)\ in\ meas(H(x_1))\ in\ (a,b)] \end{cases} \\
&\to \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_1 \to 0\rangle, \\ set\ b = meas(H(x_1))\ in\ (a,b)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_1 \to 1\rangle, \\ set\ b = meas(H(x_1))\ in\ (a,b)] \end{cases} \\
&\to \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes \frac{|r_1 \to 0\rangle + |r_1 \to 1\rangle}{\sqrt{2}}, \\ set\ b = meas(x_1)\ in\ (a,b)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes \frac{|r_1 \to 0\rangle + |r_1 \to 1\rangle}{\sqrt{2}}, \\ set\ b = meas(x_1)\ in\ (a,b)] \end{cases} \\
&\to \begin{cases} \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 0\rangle, set\ b = Z\ in\ (a,b)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 1\rangle, set\ b = O\ in\ (a,b)] \end{cases} \\ \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 0\rangle, set\ b = Z\ in\ (a,b)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 1\rangle, set\ b = O\ in\ (a,b)] \end{cases} \end{cases} \\
&\to \begin{cases} \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 0\rangle, (a,b)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to O] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 1\rangle, (a,b)] \end{cases} \\ \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 0\rangle, (a,b)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to O] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 1\rangle, (a,b)] \end{cases} \end{cases} \\
&\to \begin{cases} \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to Z] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 0\rangle, (Z,Z)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to Z, b \to O] \rangle [|r_0 \to 0\rangle \otimes |r_0 \to 1\rangle, (Z,O)] \end{cases} \\ \begin{cases} \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to Z] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 0\rangle, (O,Z)] \\ \langle \theta, [x_0 \to r_0, x_1 \to r_1], [a \to O, b \to O] \rangle [|r_0 \to 1\rangle \otimes |r_0 \to 1\rangle, (O,O)] \end{cases} \end{cases}
\end{aligned}
$$

$$
\begin{array}{rll}
(2,5,mut)\ !\Delta & \triangleright\ mut\ (a,b) = (zero, zero)\ in\ M_0 : !(bit \otimes bit) & 1 \\
(3,4,tup)\ !\Delta & \triangleright\ (zero, zero) : !(bit \otimes bit) & 2 \\
(const)\ !\Delta & \triangleright\ zero : !bit & 3 \\
(const)\ !\Delta & \triangleright\ zero : !bit & 4 \\
(6,7,set)\ !\Delta, a :!bit, b :!bit & \triangleright\ set\ a = sampleBit()\ in\ M_1 : !(bit \otimes bit) & 5 \\
(tup)\ !\Delta, a :!bit, b :!bit & \triangleright\ () : !\top & 6 \\
(8,9,set)\ !\Delta, a :!bit, b :!bit & \triangleright\ set\ b = sampleBit()\ in\ (a,b) : !(bit \otimes bit) & 7 \\
(tup)\ !\Delta, a :!bit, b :!bit & \triangleright\ () : !\top & 8 \\
(10,11,tup)\ !\Delta, a :!bit, b :!bit & \triangleright\ (a,b) : !(bit \otimes bit) & 9 \\
(var)\ !\Delta, a :!bit, b :!bit & \triangleright\ a : !bit & 10 \\
(var)\ !\Delta, a :!bit, b :!bit & \triangleright\ b : !bit & 11
\end{array}
$$

Figura B.5: Tipaggio per il generatore di bit casuali in LTIQ

```
namespace teleportation {
  open Microsoft.Quantum.Canon;
  open Microsoft.Quantum.Intrinsic;
  open Microsoft.Quantum.Diagnostics;

  operation bell00(x: Qubit, y: Qubit) : (Qubit, Qubit) {
    H(x);
    CNOT(x, y);
    return (x, y);
  }

  operation alice(a:Qubit, x:Qubit) : (Result, Result) {
    CNOT(a, x);
    H(a);
    return (M(a), M(x));
  }

  operation bob(a:Result, x:Result, y:Qubit):Qubit {
    let y1 = (x == One)? X(y) | ();
    let a1 = (a == One)? Z(y) | ();
    return y;
  }

  @EntryPoint()
  operation teleport(): Unit {
    using((a, x, y) = (Qubit(), Qubit(), Qubit())) {
    let (x1, y1) = bell00(x, y);
    let (ra, rx1) = alice(a, x1);
    let r = bob(ra, rx1, y1);
    }
  }
}
```

Figura B.6: Implementazione dell'algoritmo di teletrasporto in Q#

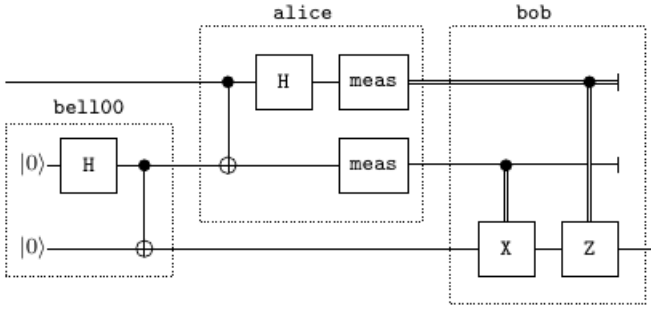

Figura B.7: Circuito di teletrasporto

```
let bell = op(u:qbit, v:qbit):(qbit ⊗ qbit)  {
    CNOT(H(u), v)
}  in
let alice = op(x0:qbit, y0:qbit):(qbit ⊗ qbit) {
    let (x1, y1) = CNOT(x0, y0) in
    (meas(H(x1)), meas(y1))
} in
let bob = op(c0:bit, c1:bit, q0: qbit {
    let z = if (c0 == one) then X(q0) else q0 in
    if (c0 == zero) then Z(z) else z
} in
using (b, a, q) in
    let (a1, b1) = bell(a, b) in
    let (bq1, ba2) = alice(q, a1) in
    bob(bq1, ba2, b1)
```

Figura B.8: Encoding dell'algoritmo di teletrasporto in LTIQ

$$
\begin{aligned}
M &\to_1 \langle \theta, \cdot, \cdot \rangle [\cdot, using(b, a, q)\ in\ T_0] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 0, r_2 \to 0\rangle, let\ (a_1, b_1) = bell(a, b)\ in\ T_1] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 0, r_2 \to 0\rangle, let\ (a_1, b_1) = (CNOT(H(a), b))\ in\ T_1] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|r_0 \to 0\rangle \otimes \frac{|r_1 \to 0\rangle + |r_1 \to 1\rangle}{\sqrt{2}} \otimes |r_2 \to 0\rangle, let\ (a_1, b_1) = (CNOT(a, b))\ in\ T_1] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|\frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, let\ (a_1, b_1) = (a, b)\ in\ T_1] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|\frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, let\ (bq_1, ba_2) = \\
&alice(q, a)\ in\ T_2[a_1/a, b_1/b] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|\frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, let\ (bq_1, ba_2) = \\
&let(x_1, y_1) = CNOT(q, a)\ in\ (meas(H(x_1)), meas(y_1)\ in\ T_2[a_1/a, b_1/b]] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|\frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, let\ (bq_1, ba_2) = \\
&let(x_1, y_1) = (q, a)\ in\ (meas(H(x_1)), meas(y_1)\ in\ T_2[a_1/a, b_1/b]] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [|\frac{|r_0 \to 0, r_1 \to 0\rangle + |r_0 \to 1, r_1 \to 1\rangle}{\sqrt{2}} \otimes r_0 \to 0\rangle, let\ (bq_1, ba_2) = \\
&(meas(H(q)), meas(a))\ T_2[a_1/a, b_1/b]] \\
&\to_1 \langle \theta, \phi, \cdot \rangle [\frac{|r_0 \to 0, r_1 \to 0, r_2 \to 0\rangle + |r_0 \to 0, r_1 \to 0, r_2 \to 1\rangle}{2} + \\
&\frac{|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle + |r_0 \to 1, r_1 \to 1, r_2 \to 1\rangle}{2}, let\ (bq_1, ba_2) = \\
&(meas(q), meas(a))\ in\ T_2[a_1/a, b_1/b]] \\
&\to^* \begin{cases}
\langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 0, r_2 \to 0\rangle, let\ (bq_1, ba_2) = (zero, zero)\ in\ T_2] \\
\langle \theta, \phi, \cdot \rangle [|r_0 \to 0, r_1 \to 0, r_2 \to 1\rangle, let\ (bq_1, ba_2) = (one, zero)\ in\ T_2] \\
\langle \theta, \phi, \cdot \rangle [|r_0 \to 1, r_1 \to 1, r_2 \to 0\rangle, let\ (bq_1, ba_2) = (zero, one)\ in\ T_2] \\
\langle \theta, \phi, \cdot \rangle [|r_0 \to 1, r_1 \to 1, r_2 \to 1\rangle, let\ (bq_1, ba_2) = (one, one)\ in\ T_2]
\end{cases}
\end{aligned}
$$

Figura B.9: Esecuzione dell'algoritmo di teletrasporto per LTIQ

$(2, use)$	$!\Delta$	$\triangleright$ $using\ (b, a, q)\ in\ T_0 : qbit$	1
$(3, 7, let)$	$!\Delta, b : qbit, a : qbit, q : qbit$	$\triangleright$ $let\ (a_1, b_1) = bell(a, b)\ in\ T_1 : qbit$	2
$(4, app)$	$!\Delta, a : qbit, b : qbit$	$\triangleright$ $bell(a, b) : (qbit \otimes qbit)$	3
$(5, 6, tup)$	$!\Delta, a : qbit, b : qbit$	$\triangleright$ $(a, b) : (qbit \otimes qbit)$	4
(var)	$!\Delta, a : qbit$	$\triangleright$ $a : qbit$	5
(var)	$!\Delta, b : qbit$	$\triangleright$ $b : qbit$	6
$(8, 12, let)$	$!\Delta, a_1 : qbit, b_1 : qbit, q : qbit$	$\triangleright$ $let\ (bq_1, ba_2) =$ $alice(q, a_1)\ in\ T_2 : qbit$	7
$(9, app)$	$!\Delta, q : qbit, a_1 : qbit$	$\triangleright$ $alice(q, a_1) : !(bit \otimes bit)$	8
$(10, 11, tup)$	$!\Delta, q : qbit, a_1 : qbit$	$\triangleright$ $(q, a_1) : (qbit \otimes qbit)$	9
(var)	$!\Delta, q : qbit$	$\triangleright$ $q : qbit$	10
(var)	$!\Delta, a_1 : qbit$	$\triangleright$ $a_1 : qbit$	11
$(13, app)$	$!\Delta, bq_1 :!bit, ba_2 :!bit, b_1 : qbit$	$\triangleright$ $bob(bq_1, ba_2, b_1) : qbit$	12
$(*, tup)$	$!\Delta, bq_1 :!bit, ba_2 :!bit, b_1 : qbit$	$\triangleright$ $(bq_1, ba_2, b_1) : (!bit \otimes !bit \otimes qbit)$	13
(var)	$!\Delta, bq_1 :!bit, ba_2 :!bit$	$\triangleright$ $bq_1 : !bit$	14
(var)	$!\Delta, bq_1 :!bit, ba_2 :!bit$	$\triangleright$ $ba_2 : !bit$	15
(var)	$!\Delta, bq_1 :!bit, ba_2 :!bit, b_1 : qbit$	$\triangleright$ $b_1 : qbit$	16

Figura B.10: Tipaggio per l'algoritmo di teletrasporto in LTIQ

Bibliografia

Barendregt, H. P. (1985). *The lambda calculus - its syntax and semantics* (Vol. 103). North-Holland.

Bichsel, B., Baader, M., Gehr, T., & Vechev, M. T. (2020). Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In A. F. Donaldson & E. Torlak (Eds.), *Proceedings of the 41st ACM SIGPLAN international conference on programming language design and implementation, PLDI 2020, london, uk, june 15-20, 2020* (pp. 286–300). ACM.

C., G., & contributors. (2017). *Cirq 2017.* `https://github.com/quantumlib/Cirq`.

Cardelli, L. (1997). Type systems. In A. B. Tucker (Ed.), *The computer science and engineering handbook* (pp. 2208–2236). CRC Press.

Coutinho, S. C. (2009). Quantum computing for computer scientists noson s. yanofsky and mirco a. mannucci, cambridge university press, 2008. *SIGACT News*, *40*(4), 14–17.

Dal Lago, U., Masini, A., & Zorzi, M. (2009). On a measurement-free quantum lambda calculus with classical control. *Math. Struct. Comput. Sci.*, *19*(2), 297–335.

Fähndrich, M., & DeLine, R. (2002). Adoption and focus: Practical linear types for imperative programming. In J. Knoop & L. J. Hendren (Eds.), *Proceedings of the 2002 ACM SIGPLAN conference on programming language design and implementation (pldi), berlin, germany, june 17-19, 2002* (pp. 13–24). ACM.

Girard, J. (1987). Linear logic. *Theor. Comput. Sci.*, *50*, 1–102.

Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., & Valiron, B. (2013a). An introduction to quantum programming in quipper. In G. W. Dueck & D. M. Miller (Eds.), *Reversible computation - 5th international conference, RC 2013, victoria, bc, canada, july 4-5, 2013. proceedings* (Vol. 7948, pp. 110–124). Springer.

Green, A. S., Lumsdaine, P. L., Ross, N. J., Selinger, P., & Valiron, B. (2013b). Quipper: a scalable quantum programming language. In H. Boehm & C. Flanagan (Eds.), *ACM SIGPLAN conference on programming language design and implementation, PLDI '13, seattle, wa, usa, june 16-19, 2013* (pp. 333–342). ACM.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual acm symposium on theory of computing*

(p. 212–219). New York, NY, USA: Association for Computing Machinery.

McKay, D. C., Alexander, T., Bello, L., Biercuk, M. J., Bishop, L., Chen, J., … Gambetta, J. M. (2018). Qiskit backend specifications for openqasm and openpulse experiments. *CoRR*, *abs/1809.03452*. Retrieved from `http://arxiv.org/abs/1809.03452`

Nielsen, M. A., & Chuang, I. L. (2016). *Quantum computation and quantum information (10th anniversary edition)*. Cambridge University Press.

Paykin, J., Rand, R., & Zdancewic, S. (2017). Qwire: a core language for quantum circuits. *ACM SIGPLAN Notices*, *52*(1), 846–858.

Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

Preskill, J. (2012). Quantum computing and the entanglement frontier. *arXiv preprint arXiv:1203.5813*.

Rios, F., & Selinger, P. (2017). A categorical model for a quantum circuit description language. In B. Coecke & A. Kissinger (Eds.), *Proceedings 14th international conference on quantum physics and logic, QPL 2017, nijmegen, the netherlands, 3-7 july 2017* (Vol. 266, pp. 164–178).

Ross, N. J. (2015). *Algebraic and logical methods in quantum computation.* (Unpublished doctoral dissertation). DalhousieUniversity.

Selinger, P. (2004). Towards a quantum programming language. *Math. Struct. Comput. Sci.*, *14*(4), 527–586.

Selinger, P., & Valiron, B. (2006). A lambda calculus for quantum computation with classical control. *Math. Struct. Comput. Sci.*, *16*(3), 527–552.

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, *26*(5), 1484–1509.

Sofge, D. (2008). A survey of quantum programming languages: History, methods, and tools. In *Second international conference on quantum, nano, and micro technologies, ICQNM 2008, february 10-15, 2008, sainte luce, martinique, french caribbean* (pp. 66–71). IEEE Computer Society.

Sørensen, M., & Urzyczyn, P. (2006). *Lectures on the curry-howard isomorphism* (No. v. 10). Elsevier.

Steiger, D. S., Häner, T., & Troyer, M. (2016). Projectq: An open source software framework for quantum computing. *CoRR*, *abs/1612.08091*. Retrieved from `http://arxiv.org/abs/1612.08091`

Svore, K. M., Geller, A., Troyer, M., Azariah, J., Granade, C. E., Heim, B., … Roetteler, M. (2018). Q#: Enabling scalable quantum computing and development with a high-level DSL. In *Proceedings of the real world domain specific languages workshop, rwdsl@cgo 2018, vienna, austria, february 24-24, 2018* (pp. 7:1–7:10). ACM.

van Tonder, A. (2004). A lambda calculus for quantum computation. *SIAM J. Comput.*, *33*(5), 1109–1135.

Wadler, P. (1990). Linear types can change the world! In *Programming concepts and methods: Proceedings of the IFIP working group 2.2, 2.3 working conference on programming concepts and methods, sea of galilee, israel, 2-5 april, 1990* (p. 561).

Wadler, P. (1991). Is there a use for linear logic? In C. Consel & O. Danvy (Eds.), *Proceedings of the symposium on partial evaluation and semantics-based program manipulation, pepm'91, yale university, new haven, connecticut, usa, june 17-19, 1991* (pp. 255–273). ACM.

Walker, D. (2004). Substructural type systems. In *Advanced topics in types and programming languages* (pp. 3–43). The MIT Press.

Winskel, G. (1993). *The formal semantics of programming languages - an introduction.* MIT Press.

Zorzi, M. (2016). On quantum lambda calculi: a foundational perspective. *Math. Struct. Comput. Sci.*, *26*(7), 1107–1195.