UNIVERSITY OF CALGARY

An investigation of the underpinnings of quantum and reversible computing

subtitle

by

Brett Gordon Giles

A DISSERTATION

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

August, 2013

Abstract

Acknowledgements

Table of Contents

List of Tables

List of Figures and Illustrations

List of Symbols, Abbreviations and Nomenclature

Symbol	Definition
U of C	University of Calgary
$\mathbb{N}$	The set of natural numbers, i.e., $\{0, 1, 2, \ldots\}$
$\mathbb{Z}$	The ring of integers numbers, i.e., $\{0, \pm 1, \pm 2, \ldots\}$
$\mathbb{C}$	The field of complex numbers

Chapter 1

Category theory

1.1 Restriction categories

Restriction categories were originally introduced in [5] as an algebraic way to formulate partial map categories:

Definition 1.1.1. A *restriction category* is a category $\mathbb{X}$ together with a *restriction operator* on maps:

$$\frac{f : A \to B}{\bar{f} : A \to A}$$

where f is an map of $\mathbb{X}$ and A, B are objects of $\mathbb{X}$, such that the following four *restriction identities* hold, whenever the compositions[1] are defined.

$$[\textbf{\textit{R.1}}] \ \bar{f}f = f \qquad\qquad\qquad [\textbf{\textit{R.2}}] \ \bar{g}\bar{f} = \bar{f}\bar{g}$$

$$[\textbf{\textit{R.3}}] \ \overline{\bar{f}g} = \bar{f}\bar{g} \qquad\qquad\qquad [\textbf{\textit{R.4}}] \ \bar{f}\bar{g} = \overline{f\bar{g}}f$$

Definition 1.1.2. A *restriction functor* is a functor which preserves the restriction. That is, given a functor $F : \mathbb{X} \to \mathbb{Y}$ with $\mathbb{X}$ and $\mathbb{Y}$ restriction categories, F is a restriction functor if:

$$F(\bar{f}) = \overline{F(f)}.$$

Any map such that $r = \bar{r}$ is an idempotent, as $\overline{rr} = \overline{\bar{r}r} = \bar{r}$, and is called a *restriction idempotent*. All maps $\bar{f}$ are restriction idempotents as $\bar{f} = \overline{\bar{f}}$. Below, we record some basic facts for restriction categories shown in [5] pp 4-5:

Lemma 1.1.3. *In a restriction category* $\mathbb{X}$,

---

[1]Note that composition is written in diagrammatic order throughout this paper.

(i) $\overline{f}$ *is idempotent;*

(ii) $\overline{fg} = \overline{fg}\,\overline{f};$

(iii) $\overline{fg} = \overline{f\overline{g}}\,;$

(iv) $\overline{\overline{f}} = \overline{f};$

(v) $\overline{f}\,\overline{g} = \overline{\overline{f}\,\overline{g}};$

(vi) f *monic implies* $\overline{f} = 1;$

(vii) $f = \overline{g}f \implies \overline{g}\,\overline{f} = \overline{f}.$

A map $f : A \rightarrow B$ in a restriction category is said to be *total* when $\overline{f} = 1_A$. The total maps in a restriction category form a subcategory $Total(\mathbb{X}) \subseteq \mathbb{X}$.

An example of a restriction category is PAR, the category with objects sets and arrows the partial functions between sets. In PAR, the restriction of $f : A \rightarrow B$ is:

$$\overline{f}(x) = \begin{cases} x & \text{if } f(x) \text{ is defined,} \\ \uparrow & \text{if } f(x) \text{ is } \uparrow. \end{cases}$$

(The symbol $\uparrow$ means that the function is undefined at that element). In PAR, the total maps correspond precisely to the functions that are defined on all elements of the domain.

1.1.1 Enrichment and meets

In any restriction category, there is a partialorder on each hom-set, given by $f \leq g$ iff $\overline{f}g = f$, where $f, g : A \rightarrow B$.

Lemma 1.1.4. *In a restriction category* $\mathbb{X}$*:*

(i) $\leq$ *as defined above is a partial order on each hom-set;*

(ii) $f \leq g \implies \overline{f} \leq \overline{g};$

(iii) $f \leq g \implies hf \leq hg;$

(iv) $f \leq g \implies fh \leq gh$;

(v) $f \leq 1 \iff f = \overline{f}$.

Proof.

(i) With f, g, h parallel maps in $\mathbb{X}$, each of the requirements for a partial order is shown below:

Reflexivity: $\overline{f}f = f$ and therefore, $f \leq f$.

Anti-Symmetry: Given $\overline{f}g = f$ and $\overline{g}f = g$, it follows:

$$f = \overline{f}f = \overline{\overline{f}g}f = \overline{f}\,\overline{g}f = \overline{g}\overline{f}f = \overline{g}f = g.$$

Transitivity: Given $f \leq g$ and $g \leq h$,

$$\overline{f}h = \overline{\overline{f}g}h = \overline{f}\,\overline{g}h = \overline{f}g = f$$

showing that $f \leq h$.

(ii) The premise is that $\overline{f}g = f$. From this, $\overline{f}\,\overline{g} = \overline{\overline{f}g} = \overline{f}$, showing $\overline{f} \leq \overline{g}$.

(iii) $\overline{hf}hg = h\overline{f}g = hf$ and therefore $hf \leq hg$.

(iv) $\overline{f}g = f$, this shows $\overline{fh}gh = \overline{\overline{f}gh}gh = \overline{f}\,\overline{gh}gh = \overline{f}gh = fh$ and therefore $fh \leq gh$.

(v) As $f \leq 1$ means precisely $\overline{f}1 = f$.

$\square$

Lemma 1.1.4 shows that restriction categories are enriched in partial orders.

Definition 1.1.5. A restriction category has *meets* if there is an operation $\cap$ on parallel maps:

$$\frac{A \overset{f}{\underset{g}{\rightrightarrows}} B}{A \xrightarrow{f \cap g} B}$$

3

such that $f \cap g \leq f, f \cap g \leq g, f \cap f = f, h(f \cap g) = hf \cap hg$.

Meets were introduced in [2]. The following are basic results on meets:

Lemma 1.1.6. *In a restriction category $\mathbb{X}$ with meets, where f, g, h are maps in $\mathbb{X}$, the following are true:*

(i) $f \leq g$ *and* $f \leq h \iff f \leq g \cap h$;

(ii) $f \cap g = g \cap f$;

(iii) $\overline{f \cap 1} = f \cap 1$;

(iv) $(f \cap g) \cap h = f \cap (g \cap h)$;

(v) $r(f \cap g) = rf \cap g$ *where* $r = \overline{r}$ *is a restriction idempotent;*

(vi) $(f \cap g)r = fr \cap g$ *where* $r = \overline{r}$ *is a restriction idempotent;*

(vii) $\overline{f \cap g} \leq \overline{f}$ *(and therefore* $\overline{f \cap g} \leq \overline{g}$*);*

(viii) $(f \cap 1)f = f \cap 1$;

(ix) $e(e \cap 1) = e$ *where e is idempotent.*

Proof.

(i) $f \leq g$ and $f \leq h$ means precisely $f = \overline{f}g$ and $f = \overline{f}h$. Therefore,

$$\overline{f}(g \cap h) = \overline{f}g \cap \overline{f}h = f \cap f = f$$

and so $f \leq g \cap h$. Conversely, given $f \leq g \cap h$, we have $f = \overline{f}(g \cap h) = \overline{f}g \cap \overline{f}h \leq \overline{f}g$. But $f \leq \overline{f}g$ means $f = \overline{f}\,\overline{f}g = \overline{f}g$ and therefore $f \leq g$. Similarly, $f \leq h$.

(ii) From (i), as by definition, $f \cap g \leq g$ and $f \cap g \leq f$.

(iii) $f \cap 1 = \overline{f \cap 1}(f \cap 1) = (\overline{f \cap 1}f) \cap (\overline{f \cap 1}) \leq \overline{f \cap 1}$ from which the result follows.

4

(iv) By definition and transitivity, $(f \cap g) \cap h \leq f, g, h$ therefore by (i) $(f \cap g) \cap h \leq f \cap (g \cap h)$. Similarly, $f \cap (g \cap h) \leq (f \cap g) \cap h$ giving the equality.

(v) Given $rf \cap g \leq rf$, calculate:

$$rf \cap g = \overline{rf \cap g}\, rf = \overline{r(rf \cap g)}f = \overline{rrf \cap rg}f = \overline{r(f \cap g)}f = r\overline{f \cap g}f = r(f \cap g).$$

(vi) Using the previous point with the restriction idempotent $\overline{fr}$,

$$fr \cap g = f\overline{r} \cap g = \overline{fr}f \cap g = \overline{fr}(f \cap g) = \overline{fr}\,\overline{f \cap g}f$$

$$= \overline{f \cap g}\,\overline{fr}f = \overline{f \cap g}f\overline{r} = (f \cap g)r.$$

(vii) For the first claim,

$$\overline{f \cap g}\,\overline{f} = \overline{\overline{f}(f \cap g)} = \overline{(\overline{f}f) \cap g} = \overline{f \cap g}.$$

The second claim then follows by (ii).

(viii) Given $f \cap 1 \leqslant f$:

$$f \cap 1 \leqslant f \iff \overline{f \cap 1}f = f \cap 1 \iff (f \cap 1)f = f \cap 1$$

where the last step is by item (iii) of this lemma.

(ix) As e is idempotent, $e(e \cap 1) = (ee \cap e) = e$.

$\square$

1.1.2 Partial monics, sections and isomorphisms

Partial isomorphisms play a central role in this paper and below we develop some their basic properties.

Definition 1.1.7. A map f in a restriction category $\mathbb{X}$ is said:

- To be a *partial isomorphism* when there is a *partial inverse*, written $f^{(-1)}$ with $ff^{(-1)} = \overline{f}$ and $f^{(-1)}f = \overline{f^{(-1)}}$;

- To be a *partial monic* if $hf = kf \implies h\overline{f} = k\overline{f}$;

- To be a *partial section* if there exists an h such that $fh = \overline{f}$;

- To be a *restriction monic* if it is a section s with a retraction r such that $rs = \overline{rs}$.

Lemma 1.1.8. *In a restriction category:*

(i) $f,\ g$ *partial monic implies* fg *is partial monic;*

(ii) f *a partial section implies* f *is partial monic;*

(iii) $f,\ g$ *partial sections implies* fg *is a partial section;*

(iv) *The partial inverse of* f*, when it exists, is unique;*

(v) *If* $f,\ g$ *have partial inverses and* $f\,g$ *exists, then* $f\,g$ *has a partial inverse;*

(vi) *A restriction monic* s *is a partial isomorphism.*

Proof.

(i) Suppose $hfg = kfg$. As g is partial monic, $hf\overline{g} = kf\overline{g}$. Therefore:

$$h\overline{fg}f = k\overline{fg}f \qquad\qquad\qquad [\boldsymbol{R.4}]$$

$$h\overline{fg}\,\overline{f} = k\overline{fg}\,\overline{f} \qquad\qquad\qquad f\,\text{partial monic}$$

$$h\overline{fg} = k\overline{fg} \qquad\qquad\qquad \text{Lemma 1.1.3, (ii)}$$

(ii) Suppose $gf = kf$. Then, $g\overline{f} = gfh = kfh = k\overline{f}$.

6

(iii) We have $fh = \bar{f}$ and $gh' = \bar{g}$. Therefore,

$$fgh'h = f\bar{g}h \qquad\qquad g \text{ partial section}$$

$$= \overline{fg}fh \qquad\qquad\qquad [\textbf{\textit{R.4}}]$$

$$= \overline{fg}\,\bar{f} \qquad\qquad f \text{ partial section}$$

$$= \bar{f}\,\overline{fg} \qquad\qquad\qquad [\textbf{\textit{R.2}}]$$

$$= \overline{\bar{f}fg} \qquad\qquad\qquad [\textbf{\textit{R.3}}]$$

$$= \overline{fg} \qquad\qquad\qquad [\textbf{\textit{R.1}}]$$

(iv) Suppose both $f^{(-1)}$ and f^* are partial inverses of f. Then,

$$f^{(-1)} = \overline{f^{(-1)}}f^{(-1)} = f^{(-1)}ff^{(-1)} = f^{(-1)}\bar{f} = f^{(-1)}ff^* = f^{(-1)}f\overline{f^*}f^*$$

$$= \overline{f^{(-1)}}\overline{f^*}f^* = \overline{f^*}\overline{f^{(-1)}}f^* = f^*f\overline{f^{(-1)}}f^* = f^*ff^{(-1)}ff^* = f^*ff^* = f^*$$

(v) For $f : A \to B$, $g : B \to C$ with partial inverses $f^{(-1)}$ and $g^{(-1)}$ respectively, the partial inverse of fg is $g^{(-1)}f^{(-1)}$. Calculating $fgg^{(-1)}f^{(-1)}$ using all the restriction identities:

$$fgg^{(-1)}f^{(-1)} = f\bar{g}f^{(-1)} = \overline{fg}ff^{(-1)} = \overline{fg}\,\bar{f} = \bar{f}\,\overline{fg} = \overline{\bar{f}fg} = \overline{fg}.$$

The calculation of $g^{(-1)}f^{(-1)}fg = \overline{g^{(-1)}f^{(-1)}}$ is similar.

(vi) The partial inverse of s is $\overline{rs}\,r$. First, note that $\overline{\overline{rs}\,r} = \overline{rs}\,\bar{r} = \bar{r}\,\overline{rs} = \overline{\bar{r}\,rs} = \overline{rs}$. Then, it follows that $(\overline{rs}\,r)s = rs = \overline{rs} = \overline{rs}r$ and $s(\overline{rs}\,r) = sr\bar{s} = \bar{s}$.

$\square$

A restriction category in which every map is a partial isomorphism is called an *inverse category*.

An interesting property of inverse categories:

Lemma 1.1.9. *In an inverse category, all idempotents are restriction idempotents.*

Proof. Given an idempotent e,

$$\bar{e} = ee^{(-1)} = eee^{(-1)} = e\bar{e} = \overline{\bar{e}e} = \bar{e}e = e.$$

$\square$

1.1.3   Split restriction categories

The split restriction category, $K_E(\mathbb{X})$ is defined as:

Objects: (A, e), where A is an object of $\mathbb{X}$, $e : A \to A$ and $e \in E$.

Maps: $f : (A, d) \to (B, e)$ is given by $f : A \to B$ in $\mathbb{X}$, where $f = dfe$.

Identity: The map e for (A, e).

Composition: inherited from $\mathbb{X}$.

This is the standard idempotent splitting construction, also known as the Karoubi envelope.

Note that for $f : (A, d) \to (B, e)$, by definition, in $\mathbb{X}$ we have $f = dfe$, giving

$$df = d(dfe) = ddfe = dfe = f \ \text{ and } \ fe = (dfe)e = dfee = dfe = f.$$

When $\mathbb{X}$ is a restriction category, there is an immediate candidate for a restriction in $K_E(\mathbb{X})$. If $f \in K_E(\mathbb{X})$ is $e_1 f e_2$ in $\mathbb{X}$, then define $\overline{f}$ as given by $e_1 \overline{f}$ in $\mathbb{X}$. Note that for $f : (A, d) \to (B, e)$, in $\mathbb{X}$ we have:

$$d\overline{f} = \overline{d} \overline{f} d = \overline{f} d.$$

Proposition 1.1.10. *If $\mathbb{X}$ is a restriction category and E is a set of idempotents, then the restriction as defined above makes $K_E(\mathbb{X})$ a restriction category.*

8

Proof. The restriction takes $f : (A, e_1) \to (B, e_2)$ to an endomorphism of (A, e_1). The restriction is in $\mathrm{K}_E(\mathbb{X})$ as

$$e_1(e_1\overline{f})e_1 = e_1\overline{f}e_1 = \overline{e_1 f}e_1 e_1 = \overline{e_1 f}e_1 = e_1\overline{f}.$$

Checking the 4 restriction axioms:

[**R.1**] $[\![\overline{f}f]\!] = e_1\overline{f}f = e_1 f = [\![f]\!]$

[**R.2**] $[\![\overline{g}\overline{f}]\!] = e_1\overline{g}e_1\overline{f} = e_1 e_1\overline{g}\overline{f} = e_1 e_1\overline{f}\overline{g} = e_1\overline{f}e_1\overline{g} = [\![\overline{f}\overline{g}]\!]$

[**R.3**] $[\![\overline{\overline{f}g}]\!] \equiv e_1\overline{e_1\overline{f}g} = \overline{e_1 e_1\overline{f}g}e_1 = \overline{e_1\overline{f}g}e_1 = e_1\overline{\overline{f}g} = e_1\overline{f}\overline{g} = e_1 e_1\overline{f}\overline{g} = e_1\overline{f}e_1\overline{g} = [\![\overline{f}\,\overline{g}]\!]$

[**R.4**] $[\![\overline{f}\overline{g}]\!] = e_1 f e_2\overline{g} = \overline{e_1 f e_2 g}e_1 f e_2 = \overline{e_1 e_1 f e_2 g}e_1 e_1 f e_2$

$$= e_1\overline{e_1 f e_2 g}e_1 f e_2 = e_1\overline{f g}e_1 f e_2 = [\![\overline{f g}f]\!]$$

$\square$

Given this, provided all identity maps are in E, $\mathrm{K}_E(\mathbb{X})$ is a restriction category with $\mathbb{X}$ as a full sub-restriction category, via the embedding defined by taking an object A in $\mathbb{X}$ to the object $(A, 1)$ in $\mathrm{K}_E(\mathbb{X})$. Furthermore, the property of being an inverse category is preserved by splitting.

Lemma 1.1.11. *When $\mathbb{X}$ is an inverse category, $K_E(X)$ is an inverse category.*

Proof. The inverse of $f : (A, e_1) \to (B, e_2)$ in $\mathrm{K}_E(\mathbb{X})$ is $e_2 f^{(-1)} e_1$ as

$$[\![f f^{(-1)}]\!] = e_1 f e_2 e_2 f^{(-1)} e_1 = e_1 e_1 f e_2 f^{(-1)} e_1 = e_1 f f^{(-1)} e_1 = e_1 e_1\overline{f}e_1 = e_1\overline{f} = [\![\overline{f}]\!]$$

and

$$[\![f^{(-1)}f]\!] = e_2 f^{(-1)} e_1 e_1 f e_2 = e_2 f^{(-1)} e_1 f e_2 e_2 = e_2 f^{(-1)} f e_2$$

$$= e_2 e_2\overline{f^{(-1)}}e_2 = e_2\overline{f^{(-1)}} = [\![\overline{f^{(-1)}}]\!]$$

$\square$

Proposition 1.1.12. *In a restriction category* $\mathbb{X}$, *with meets, let* R *be the set of restriction idempotents. Then,* $K(\mathbb{X}) \cong K_R(\mathbb{X})$ *(where* $K(\mathbb{X})$ *is the split of* $\mathbb{X}$ *over all idempotents). Furthermore,* $K_R(\mathbb{X})$ *has meets.*

Proof. The proof below first shows the equivalence of the two categories, then addresses the claim that $K_R(\mathbb{X})$ has meets.

For equivalence, we require two functors,

$$U : K_R(\mathbb{X}) \to K(\mathbb{X}) \text{ and } V : K(\mathbb{X}) \to K_R(\mathbb{X}),$$

with:

$$UV \cong I_{K_R(\mathbb{X})} \tag{1.1}$$

$$VU \cong I_{K(\mathbb{X})}. \tag{1.2}$$

U is the standard inclusion functor. V will take the object (A, e) to $(A, e \cap 1)$ and the map $f : (A, e_1) \to (B, e_2)$ to $(e_1 \cap 1)f$.

V is a functor as:

Well Defined: If $f : (A, e_1) \to (B, e_2)$, then $(e_1 \cap 1)f$ is a map in $\mathbb{X}$ from A to B and
$$(e_1 \cap 1)(e_1 \cap 1)f(e_2 \cap 1) = (e_1 \cap 1)(fe_2 \cap f) = (e_1 \cap 1)(f \cap f) = (e_1 \cap 1)f,$$
therefore, $V(f) : V((A, e_1)) \to V((B, e_2))$.

Identities: $V(e) = (e \cap 1)e = e \cap 1$ by lemma 1.1.6.

Composition: $V(f)V(g) = (e_1 \cap 1)f(e_2 \cap 1)g = (e_1 \cap 1)fe_2(e_2 \cap 1)g = (e_1 \cap 1)f(e_2 \cap e_2)g = (e_1 \cap 1)fe_2g = (e_1 \cap 1)fg = V(fg)$.

Recalling from Lemma 1.1.6, $(e \cap 1)$ is a restriction idempotent. Using this fact, the commutativity of restriction idempotents and the general idempotent identities from 1.1.6 on page 4, the composite functor UV is the identity on $K_r(\mathbb{X})$ as when e is a restriction idempotent, $e = e(e \cap 1) = (e \cap 1)e = (e \cap 1)$.

For the other direction, note that for a particular idempotent $e : A \to A$, this gives the maps $e : (A, e) \to (A, e \cap 1)$ and $e \cap 1 : (A, e \cap 1) \to (A, e)$, again by 1.1.6 on page 4. These maps give the natural isomorphism between I and VU as

$$(A, e) \xrightarrow{\;e\;} (A, e \cap 1) \qquad \text{and} \qquad (A, e \cap 1) \xrightarrow{\;e \cap 1\;} (A, e)$$

both commute. Therefore, $UV = I$ and $VU \cong I$, giving an equivalence of the categories.

For the rest of this proof, the bolded functions, e.g., $\mathbf{f}$ are in $\mathrm{K}_R(\mathbb{X})$. Italic functions, e.g., f are in $\mathbb{X}$.

To show that $\mathrm{K}_R(\mathbb{X})$ has meets, designate the meet in $\mathrm{K}_R(\mathbb{X})$ as $\cap_{\mathrm{K}}$ and define $\mathbf{f} \cap_{\mathrm{K}} \mathbf{g}$ as the map given by the $\mathbb{X}$ map $f \cap g$, where $\mathbf{f}, \mathbf{g} : (A, d) \to (B, e)$ in $\mathrm{K}_R(\mathbb{X})$ and $f, g : A \to B$ in $\mathbb{X}$. This is a map in $\mathrm{K}_R(\mathbb{X})$ as $d(f \cap g)e = (df \cap dg)e = (f \cap g)e = (fe \cap g) = f \cap g$ where the penultimate equality is by 1.1.6 on page 4. By definition $\overline{\mathbf{f} \cap_{\mathrm{K}} \mathbf{g}}$ is $\overline{df \cap g}$.

It is necessary to show $\cap_{\mathrm{K}}$ satisfies the four meet properties.

- $\mathbf{f} \cap_{\mathrm{K}} \mathbf{g} \leq \mathbf{f}$: We need to show $\overline{\mathbf{f} \cap_{\mathrm{K}} \mathbf{g}}\,\mathbf{f} = \mathbf{f} \cap_{\mathrm{K}} \mathbf{g}$. Calculating now in $\mathbb{X}$:

$$\overline{df \cap g}\,f = \overline{d(f \cap g)}\,df$$

$$= \overline{df \cap dg}\,df$$

$$= \overline{f \cap g}\,f$$

$$= f \cap g$$

which is the definition of $\mathbf{f} \cap_{\mathrm{K}} \mathbf{g}$.

- $\mathbf{f} \cap_{\mathrm{K}} \mathbf{g} \leq \mathbf{g}$: Similarly and once again calculating in $\mathbb{X}$,

$$\overline{df \cap g}\,g = \overline{d(f \cap g)}\,dg$$

$$= \overline{df \cap dg}\,dg$$

$$= \overline{f \cap g}\,g$$

$$= f \cap g$$

11

which is the definition of $\mathbf{f} \cap_K \mathbf{g}$.

- $\mathbf{f} \cap_K \mathbf{f} = \mathbf{f}$: From the definition, this is $f \cap f = f$ which is just $\mathbf{f}$.

- $\mathbf{h}(\mathbf{f} \cap_K \mathbf{g}) = \mathbf{hf} \cap_K \mathbf{hg}$: From the definition, this is given in $\mathbb{X}$ by $h(f \cap g) = hf \cap hg$ which in $K_R(\mathbb{X})$ is $\mathbf{hf} \cap_K \mathbf{hg}$.

$\square$

1.1.4 Partial Map Categories

In [5], it is shown that split restriction categories are equivalent to *partial map categories*. The main definitions and results related to partial map categories are given below.

Definition 1.1.13. A collection $\mathcal{M}$ of monics is *a stable system of monics* when it includes all isomorphisms, is closed under composition and is pullback stable.

Stable in this definition means that if $m : A \to B$ is in $\mathcal{M}$, then for arbitrary b with codomain B, the pullback

$$
\begin{array}{ccc}
A' & \xrightarrow{a} & A \\
{\scriptstyle m'}\downarrow & & \downarrow{\scriptstyle m} \\
B' & \xrightarrow{b} & B
\end{array}
$$

exists and $m' \in \mathcal{M}$. A category that has a stable system of monics is referred to as an $\mathcal{M}$-category.

Lemma 1.1.14. *If $nm \in \mathcal{M}$, a stable system of monics, and m is monic, then $n \in \mathcal{M}$.*

Proof. The commutative square

$$
\begin{array}{ccc}
A & \xrightarrow{1} & A \\
{\scriptstyle n}\downarrow & & \downarrow{\scriptstyle nm} \\
A' & \xrightarrow{m} & B
\end{array}
$$

is a pullback. $\square$

Given a category $\mathbb{C}$ and a stable system of monics, the *partial map category*, $\mathrm{Par}(\mathbb{C}, \mathcal{M})$ is:

Objects: $A \in \mathbb{C}$

Equivalence Classes of Maps: $(m, f) : A \to B$ with $m : A' \to A$ is in $\mathcal{M}$ and $f : A' \to B$

is a map in $\mathbb{C}$. i.e.,

$$\begin{array}{ccc} & A' & \\ \overset{m}{\swarrow} & & \overset{f}{\searrow} \\ A & & B \end{array} \quad .$$

Identity: $1_A, 1_A : A \to A$

Composition: via a pullback, $(m, f)(m', g) = (m''m, f'g)$ where

$$\begin{array}{ccccccc} & & A'' & & & & \\ & \overset{m''}{\swarrow} & & \overset{f'}{\searrow} & & & \\ & A' & & \text{(pb)} & B' & & \\ \overset{m}{\swarrow} & & \overset{f}{\searrow} & \overset{m'}{\swarrow} & & \overset{g}{\searrow} & \\ A & & B & & & C \end{array}$$
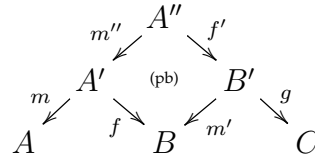
Restriction: $\overline{(m, f)} = (m, m)$

For the maps, $(m, f) \sim (m', f')$ when there is an isomorphism $\gamma : A'' \to A'$ such that $\gamma m' = m$ and $\gamma f' = f$.

In [6], it is shown that:

Theorem 1.1.15 (Cockett-Lack). *Every restriction category is a full subcategory of a partial map category.*

1.1.5 Restriction products and Cartesian restriction categories

Restriction categories have analogues of products and terminal objects.

Definition 1.1.16. In a restriction category $\mathbb{X}$ a *restriction product* of two objects X, Y is an object $X \times Y$ equipped with *total* projections $\pi_0 : X \times Y \to X, \pi_1 : X \times Y \to Y$ where:

$\forall f : Z \to X, g : Z \to Y, \quad \exists$ a unique $\langle f, g \rangle : Z \to X \times Y$ such that

- $\langle f, g \rangle \pi_0 \leq f$,

- $\langle f, g \rangle \pi_1 \leq g$ and

- $\overline{\langle f, g \rangle} = \overline{f}\,\overline{g}(= \overline{g}\,\overline{f}).$

Definition 1.1.17. In a restriction category $\mathbb{X}$ a *restriction terminal object* is an object $\top$ such that $\forall X$, there is a unique total map $!_X : X \to \top$ and the diagram

$$
\begin{array}{ccc}
X & \xrightarrow{\ \overline{f}\ } & X \xrightarrow{\ !_X\ } \top \\
\downarrow{\scriptstyle f} & \nearrow{\scriptstyle !_Y} & \\
Y & &
\end{array}
$$

commutes. That is, $f\,!_Y = \overline{f}\,!_X$. Note this implies that a restriction terminal object is unique up to a unique isomorphism.

Definition 1.1.18. A restriction category $\mathbb{X}$ is *Cartesian* if it has all restriction products and a restriction terminal object.

Definition 1.1.19. An object A in a Cartesian restriction category is *discrete* when the diagonal map,

$$\Delta : A \to A \times A$$

is a partial isomorphism.

A Cartesian restriction category is *discrete* when every object is discrete.

Theorem 1.1.20. *A Cartesian restriction category $\mathbb{X}$ is discrete if and only if it has meets.*

Proof. If $\mathbb{X}$ has meets, then

$$\Delta(\pi_0 \cap \pi_1) = \Delta\pi_0 \cap \Delta\pi_1 = 1 \cap 1 = 1$$

and as $\langle \pi_0, \pi_1 \rangle$ is identity,

$$
\begin{aligned}
\overline{\pi_0 \cap \pi_1} &= \overline{\pi_0 \cap \pi_1}\langle \pi_0, \pi_1 \rangle \\
&= \langle \overline{\pi_0 \cap \pi_1}\pi_0, \overline{\pi_0 \cap \pi_1}\pi_1 \rangle \\
&= \langle \pi_0 \cap \pi_1, \pi_0 \cap \pi_1 \rangle \\
&= (\pi_0 \cap \pi_1)\Delta
\end{aligned}
$$

and therefore, $\pi_0 \cap \pi_1$ is $\Delta^{(-1)}$.

For the other direction, set $f \cap g = \langle f, g \rangle \Delta^{(-1)}$. By the definition of the restriction product:

$$f \cap g = \langle f, g \rangle \Delta^{(-1)} = \langle f, g \rangle \Delta^{(-1)} \Delta \pi_0 = \langle f, g \rangle \overline{\Delta^{(-1)}} \pi_0 \leq \langle f, g \rangle \pi_0 \leq f$$

Similarly, substituting π_1 for π_0 above, this gives $f \cap g \leq g$. For the left distributive law,

$$h(f \cap g) = h \langle f, g \rangle \Delta^{(-1)} = \langle hf, hg \rangle \Delta^{(-1)} = hf \cap hg$$

and finally an intersection of a map with itself is

$$f \cap f = \langle f, f \rangle \Delta^{(-1)} = (f\Delta) \Delta^{(-1)} = f\overline{\Delta} = f$$

as Δ is total. This shows that $\cap$ as defined above is a meet for the Cartesian restriction category $\mathbb{X}$.

$\square$

We shall refer to a Cartesian restriction category in which every object is discrete as simply a discrete restriction category.

1.1.6 Graphic Categories

In a Cartesian restriction category, a map $A \xrightarrow{f} B$ is called *graphic* when the maps

$$A \xrightarrow{\langle f, 1 \rangle} B \times A \qquad \text{and} \qquad A \xrightarrow{\langle \overline{f}, 1 \rangle} A \times A$$

have partial inverses. A Cartesian restriction category is *graphic* when all of its maps are graphic.

Lemma 1.1.21. *In a Cartesian restriction category:*

(i) Graphic maps are closed under composition;

(ii) Graphic maps are closed under the restriction;

(iii) An object is discrete if and only if its identity map is graphic.

Proof.

(i) To show closure, it is necessary to show that $\langle fg, 1 \rangle$ has a partial inverse. By Lemma 1.1.8, the uniqueness of the partial inverse gives

$$(\langle f, 1 \rangle; \langle g, 1 \rangle \times 1)^{(-1)} = \langle g, 1 \rangle^{(-1)} \times 1; \langle f, 1 \rangle^{(-1)}.$$

By the definition of the restriction product, $\overline{\langle fg, 1 \rangle} = \overline{fg}$. Additionally, a straightforward calculation shows that $\overline{\langle f, 1 \rangle; \langle g, 1 \rangle \times 1} = \overline{\langle f\langle g, 1 \rangle, 1 \rangle} = \overline{f; \langle g, 1 \rangle} = \overline{\langle f; g, f \rangle} = \overline{fg}\,\overline{f} = \overline{fg}$ where the last equality is by [**R.2**], [**R.3**] and finally [**R.1**].

Consider the diagram

$$A \xrightarrow{\langle f,1 \rangle} B \times A \xrightarrow{\langle g,1 \rangle \times 1} C \times B \times A$$

with $\langle fg,1 \rangle : A \to C \times A$ and $1 \times \langle f,1 \rangle : C \times A \to C \times B \times A$.

From this:

$$\langle fg, 1 \rangle (1 \times \langle f, 1 \rangle)(\langle g, 1 \rangle^{(-1)} \times 1)\langle f, 1 \rangle^{(-1)} = \langle f, 1 \rangle(\langle g, 1 \rangle \times 1)(\langle g, 1 \rangle^{(-1)} \times 1)\langle f, 1 \rangle^{(-1)}$$

$$= \langle f, 1 \rangle(\overline{g \times 1})\langle f, 1 \rangle^{(-1)}$$

$$= \overline{\langle f, 1 \rangle(g \times 1)}\langle f, 1 \rangle\langle f, 1 \rangle^{(-1)}$$

$$= \overline{\langle f, 1 \rangle(g \times 1)}\overline{\langle f, 1 \rangle}$$

$$= \overline{\langle f, 1 \rangle\langle f, 1 \rangle(g \times 1)}$$

$$= \overline{\langle f, 1 \rangle(g \times 1)}$$

$$= \overline{\langle fg, 1 \rangle}(= \overline{fg})$$

showing that $1 \times \langle f, 1 \rangle (\langle g, 1 \rangle^{(-1)} \times 1) \langle f, 1 \rangle^{(-1)}$ is a right inverse for $\langle fg, 1 \rangle$.

For the other direction, note that in general $hk^{(-1)} = k^{(-1)}h^{(-1)}$ and that we have $\langle fg, 1 \rangle = \langle f, 1 \rangle (\langle g, 1 \rangle \times 1)(1 \times \langle f, 1 \rangle^{(-1)})$, thus $(1 \times \langle f, 1 \rangle)(\langle g, 1 \rangle^{(-1)} \times 1)\langle f, 1 \rangle^{(-1)}$ will also be a left inverse and $\langle fg, 1 \rangle$ is a restriction isomorphism.

(ii) This follows from the definition of graphic and that $\overline{\langle f, 1 \rangle} = \overline{f} = \overline{\overline{f}}$.

(iii) Given a discrete object A, 1_A is graphic as $\langle 1_A, 1 \rangle = \Delta$ and therefore $\langle 1, 1 \rangle^{(-1)} = \Delta^{(-1)}$. Conversely, if $\langle 1_A, 1 \rangle$ has an inverse, then $\Delta = \langle 1_A, 1 \rangle$ has that same inverse and therefore the object is discrete.

$\square$

Lemma 1.1.22. *A discrete restriction category is precisely a graphic Cartesian restriction category.*

Proof. The requirement is that $\langle f, 1 \rangle$ (and $\langle \overline{f}, 1 \rangle$) each have partial inverses. For $\langle f, 1 \rangle$, the inverse is $\overline{(1 \times f)\Delta^{(-1)}}\pi_1$.

To show this, calculate the two compositions. First,

$$\langle f, 1 \rangle \overline{1 \times f \Delta^{(-1)}}\pi_1 = \overline{\langle f, f \rangle \Delta^{(-1)}}\langle f, 1 \rangle \pi_1 = \overline{f \Delta \Delta^{(-1)}}\langle f, 1 \rangle \pi_1 = \overline{f}\langle f, 1 \rangle \pi_1 = \overline{f}.$$

The other direction is:

$$\overline{(1 \times f)\Delta^{(-1)}}\pi_1 \langle f, 1 \rangle = \langle \overline{(1 \times f)\Delta^{(-1)}}\pi_1 f, \overline{(1 \times f)\Delta^{(-1)}}\pi_1 \rangle$$

$$= \langle \overline{(1 \times f)\Delta^{(-1)}}(1 \times f)\pi_1, \overline{(1 \times f)\Delta^{(-1)}}\pi_1 \rangle$$

$$= \langle (1 \times f)\overline{\Delta^{(-1)}}\pi_1, \overline{(1 \times f)\Delta^{(-1)}}\pi_1 \rangle$$

$$= \langle (1 \times f)\overline{\Delta^{(-1)}}\pi_0, \overline{(1 \times f)\Delta^{(-1)}}\pi_1 \rangle$$

$$= \langle \overline{(1 \times f)\Delta^{(-1)}}(1 \times f)\pi_0, \overline{(1 \times f)\Delta^{(-1)}}\pi_1 \rangle$$

$$= \langle \overline{(1 \times f)\Delta^{(-1)}}\pi_0, \overline{(1 \times f)\Delta^{(-1)}}\pi_1 \rangle$$

$$= \overline{(1 \times f)\Delta^{(-1)}}\langle \pi_0, \pi_1 \rangle$$

$$= \overline{(1 \times f)\Delta^{(-1)}}$$

The one tricky step is to realize

$$\overline{\Delta^{(-1)}}\pi_1 = \Delta^{(-1)}\Delta\pi_1$$

$$= \Delta^{(-1)}$$

$$= \Delta^{(-1)}\Delta\pi_0$$

$$= \overline{\Delta^{(-1)}}\pi_0$$

For $\langle \overline{f}, 1 \rangle$, the inverse is $\overline{(1 \times \overline{f})\Delta^{(-1)}}\pi_1$. Similarly to above,

$$\langle \overline{f}, 1 \rangle \overline{1 \times \overline{f}\Delta^{(-1)}}\pi_1 = \overline{\langle \overline{f}, \overline{f} \rangle \Delta^{(-1)}}\langle \overline{f}, 1 \rangle \pi_1 = \overline{\overline{f}\Delta\Delta^{(-1)}}\langle \overline{f}, 1 \rangle \pi_1 = \overline{\overline{f}}\langle \overline{f}, 1 \rangle \pi_1 = \overline{f}.$$

The other direction follows the same pattern as for $\langle f, 1 \rangle$. $\qquad\square$

Chapter 2

Reversible computation

Bennet, in [1], showed that it was possible to emulate a standard Turing machine via a reversible Turing machine and vice-versa. This showed the equivalence of standard and reversible Turing machines. We reproduce the essence of this proof below.

2.1   Reversible Turing machines

Turing machines consist of a tape, a read-write head positioned over the tape, a machine state and a set of instructions. The set of instructions may be given as a set of transitions determining the movement of the read-write head, what it writes and the resulting state of the machine.

Definition 2.1.1. Given an alphabet A which does not contain a space, a tape is in *standard format* when:

1. The tape head is positioned directly over a blank space;

2. The spaces to the left (the $+1$ direction) contain only elements of A.

3. All other spaces of the tape are blank.

Definition 2.1.2. A *turing quintuple* is a quintuple $(s, \alpha, \alpha', \delta, s')$ where:

1. $s, s' \in S$, where S is a predefined set of states;

2. $\alpha, \alpha' \in A$ is predefined set of glyphs;

3. $\delta \in \{-1, 0, 1\}$.

Definition 2.1.3. A *standard turing quintuple set Q* consists of a set of turing quintuples such that:

1. If $q_1 = (s_1, \alpha_1, \alpha_1', \delta_1, s_1')$ and $q_2 = (s_2, \alpha_2, \alpha_2', \delta_2, s_2')$ are in Q, then either $s_1 \neq s_2$ or $\alpha_1 \neq \alpha_2$ or both are not equal.

2. There are two special quintuples contained in Q:

   (a) $(s_1, \_, \_, +1, s_2)$[1], the *start quintuple*;

   (b) $(s_{t-1}, \_, \_, 0, s_t)$, the *end quintuple* where t is the number of states and is the final state of the machine.

Definition 2.1.4. A *standard Turing machine* is given by

- a standard turing quintuple set;

- a tape that starts in standard format;

- and the condition that and if the machine halts, it will halt in state s_t, the final state of the end quintuple and the output will be in standard format.

The turing quintuples may also be regarded as giving the data for a partial function in SETS: $\tau : S \times A \to A \times \{-1, 0, 1\} \times S$.

Remark 2.1.5. A multi-tape Turing machine with n tapes and read-write heads can be described by modifying definition 2.1.4 such that α is an n-tuple of the set of glyphs for the Turing machine and δ is an n-tuple of movement directions.

Example 2.1.6. Suppose $S = \{start, run, reset, done\}$, $A = \{0, 1, \_\}$ and the Turing machine program is given by the quintuples

$$(start, \_, \_, +1, run),$$

$$(run, 0, 1, +1, run), (run, 1, 0, +1, run),$$

$$(run, \_, \_, -1, reset),$$

$$(reset, 0, 0, -1, reset), (reset, 1, 1, -1, reset),$$

$$(reset, \_, \_, 0, done).$$

---

[1] Here, $\_$ is used to signify a blank.

This program will perform a "bit-flip" of all 0s and 1s on the tape until it reads a space, reposition the read head to the standard format and then it will halt.

As can be seen from example 2.1.6, it is *possible* that a Turing machine program is reversible. If we had chosen the second quintuple to be $(run, 0, 0, +1, run)$ instead, the program would not have been reversible.

The essential property that a Turing machine program needs to be reversible is that the function τ defined from the quintuples is injective. In order to simplify the discovery the function being injective, we reformulate the turing quintuples as quadruples.

Definition 2.1.7. A *turing quadruple* is given by a quadruple

$$(s, [b_1, b_2, \ldots, b_n], [b'_1, b'_2, \ldots, b'_n], s')$$

such that:

- $s, s' \in S$, some set of states;

- $b_j \in A \cup \{\phi\}$ where A is some alphabet;

- $b'_j \in A + \{-1, 0, 1\}$;

- $b'_j \in \{-1, 0, 1\}$ if and only if $b_j = \phi$.

In this definition, $b_j = \phi$ means that the value of tape j is ignored.

A turing quadruple explicitly splits the read/write action of the Turing machine away from the movement. In a particular step for tape k, the turing machine will either read and write an item or it will not read and then move.

Remark 2.1.8. Any turing quintuple may be split into two turing quadruples by the addition of a new state a'' in A, where the first quadruple will consist of all the read-write operations and leave the Turing machine in state a''. The second quadruple will start in state a'' and all the b_j will be ϕ, with b'_j being movements on each of the n tapes.

Definition 2.1.9. A set of turing quadruples Q is called *reversible set of turing quadruples* when given $q_1, q_2 \in Q$, with $q_1 = (a, [b_j], [b'_j], a')$ and $q_2 = (c, [d_j], [d'_j], c')$:

- if $a = c$, then there is a k where $b_k, d_k \in A$ and $b_k \neq d_k$;

- if $a' = c'$, then there is a j with $b'_j, d'_j \in A$ and $b'_j \neq d'_j$.

Similarly to turing quintuples, turing quadruples may be taken as the data for a function in SETS:

$$\rho : S \times (A \cup \{\phi\}) \rightarrow (A + \{-1, 0, 1\}) \times S.$$

We can see by inspection that ρ is a reversible partial function when the set of turing quadruples that give ρ is a reversible set of turing quadruples.

Definition 2.1.10. A *reversible Turing machine* is one that is described by a set of reversible turing quadruples.

We will show that a reversible Turing machine with three tapes can emulate a Turing machine.

Theorem 2.1.11 (Bennet[1]). *Given a standard Turing Machine M, it may be emulated by a three tape reversible Turing machine R. In this case, emulated means:*

- *M halts on standard input I if and only if R halts on standard input $(I, \_, \_)$.*

- *M halts on standard input I producint standard output O, if and only if R halts on input $(I, \_, \_)$ producing standard output $(I, \_, O)$.*

Proof. (Sketch only).

The essence of the proof is to convert the quintuples of M to the quadruples of R as noted in remark 2.1.8. Explicitly,

$$(s, [a_j], [a'_j], [\delta_j], s') \mapsto \left((s, [a_j], [a'_j], s''), (s'', [\phi, \phi, \ldots, \phi], [\delta_j], s')\right). \tag{2.1}$$

In equation (equation (2.1)), s'' is a fresh state, not in the states of M. $\qquad\square$

Chapter 3

Quantum computation and circuits

3.1 Linear algebra

Quantum computation requires familiarity with the basics of linear algebra. This section will give definitions of the terms used throughout this thesis.

3.1.1 Basic definitions

The first definition needed is that of a *vector space*.

Definition 3.1.1 (Vector Space)**.** Given a field F, whose elements will be referred to as scalars, a *vector space* over F is a non-empty set V with two operations, *vector addition* and *scalar multiplication*. *Vector addition* is defined as $+ : V \times V \to V$ and denoted as $\mathbf{v} + \mathbf{w}$ where $\mathbf{v}, \mathbf{w} \in V$. The set V must be an abelian group under $+$. *Scalar multiplication* is defined as $: F \times V \to V$ and denoted as $c\mathbf{v}$ where $c \in F, \mathbf{v} \in V$. Scalar multiplication distributes over both vector addition and scalar addition and is associative. F's multiplicative identity is an identity for scalar multiplication.

The specific algebraic requirements are:

1. $\forall \mathbf{u}, \mathbf{v}, \mathbf{w} \in V,\ (\mathbf{u} + \mathbf{v}) + \mathbf{w} = \mathbf{u} + (\mathbf{v} + \mathbf{w})$;

2. $\forall \mathbf{u}, \mathbf{v} \in V,\ \mathbf{u} + \mathbf{v} = \mathbf{v} + \mathbf{u}$;

3. $\exists \mathbf{0} \in V$ such that $\forall \mathbf{v} \in V, \mathbf{0} + \mathbf{v} = \mathbf{v}$;

4. $\forall \mathbf{u} \in V, \exists \mathbf{v} \in V$ such that $\mathbf{u} + \mathbf{v} = \mathbf{0}$;

5. $\forall \mathbf{u}, \mathbf{v} \in V, c \in F,\ c(\mathbf{u} + \mathbf{v}) = c\mathbf{u} + c\mathbf{v}$;

6. $\forall \mathbf{u} \in V, c, d \in F, \ (c + d)\mathbf{u} = c\mathbf{u} + d\mathbf{u}$;

7. $\forall \mathbf{u} \in V, c, d \in F, \ (cd)\mathbf{u} = c(d\mathbf{u})$;

8. $\forall \mathbf{u} \in V, \ 1\mathbf{u} = \mathbf{u}$.

Examples of vector spaces over F are: $F^{n \times m}$ – the set of $n \times m$ matrices over F; and F^n – the $n-$fold Cartesian product of F. $F^{n \times 1}$, the set of $n \times 1$ matrices over F is also called the space of column vectors, while $F^{1 \times n}$, the set of row vectors. Often, F^n is identified with $F^{n \times 1}$.

This thesis shall identify F^n with the column vector space over F.

Definition 3.1.2 (Linearly independent). A subset of vectors $\{\mathbf{v}_i\}$ of the vector space V is said to be *linearly independent* when no finite linear combination of them, $\sum a_j \mathbf{v}_j$ equals $\mathbf{0}$ unless all the a_j are zero.

Definition 3.1.3 (Basis). A *basis* of a vector space V is a linearly independent subset of V that generates V. That is, any vector $u \in V$ is a linear combination of the basis vectors.

3.1.2 Matrices

As mentioned above, the set of $n \times m$ matrices over a field is a vector space. Additionally, matrices compose and the tensor product of matrices is defined.

Matrix composition is defined as usual. That is, for $A = [a_{ij}] \in F^{m \times n}, B = [b_{jk}] \in F^{n \times p}$:

$$A B = \left[\left(\sum_j a_{ij} b_{jk} \right)_{ik} \right] \in F^{m \times p}.$$

Definition 3.1.4 (Diagonal matrix). A *diagonal matrix* is a matrix where the only non-zero entries are those where the column index equals the row index.

The diagonal matrix $n \times n$ with only 1's on the diagonal is the identity for matrix multiplication, and is designated by I_n.

Definition 3.1.5 (Transpose). The *transpose* of an $n \times m$ matrix $A = [a_{ij}]$ is an $m \times n$ matrix A^t with the i, j entry being a_{ji}.

When the base field of a matrix is $\mathbb{C}$, the complex numbers, the *conjugate transpose* (also called the *adjoint*) of an $n \times m$ matrix $A = [a_{ij}]$ is defined as the $m \times n$ matrix A^* with the i, j entry being $\bar{a}_{ji}$, where $\bar{a}$ is the complex conjugate of $a \in \mathbb{C}$.

When working with column vectors over $\mathbb{C}$, note that $\mathbf{u} \in \mathbb{C}^n \implies \mathbf{u}^* \in \mathbb{C}^{1 \times n}$ and that $\mathbf{u}^* \times \mathbf{u} \in \mathbb{C}^{1 \times 1}$. This thesis will use the usual identification of $\mathbb{C}$ with $\mathbb{C}^{1 \times 1}$. A column vector $\mathbf{u}$ is called a *unit vector* when $\mathbf{u}^* \times \mathbf{u} = 1$.

Definition 3.1.6 (Trace). The *trace*, $Tr(A)$ of a square matrix $A = [a_{ij}]$ is $\sum a_{ii}$.

Tensor Product

The tensor product of two matrices is the usual Kronecker product:

$$U \otimes V = \begin{bmatrix} u_{11}V & u_{12}V & \cdots & u_{1m}V \\ u_{21}V & u_{22}V & \cdots & u_{2m}V \\ \vdots & \vdots & \ddots & \\ u_{n1}V & u_{n2}V & \cdots & u_{nm}V \end{bmatrix} = \begin{bmatrix} u_{11}v_{11} & \cdots & u_{12}v_{11} & \cdots & u_{1m}v_{1q} \\ u_{11}v_{21} & \cdots & u_{12}v_{21} & \cdots & u_{1m}v_{2q} \\ \vdots & \vdots & \vdots & \ddots \\ u_{n1}v_{p1} & \cdots & u_{n2}v_{p1} & \cdots & u_{nm}v_{pq} \end{bmatrix}$$

Special matrices

When working with quantum values certain types of matrices over the complex numbers are of special interest. These are:

Unitary Matrix : Any $n \times n$ matrix A with $AA^* = I \, (= A^*A)$.

Hermitian Matrix : Any $n \times n$ matrix A with $A = A^*$.

Positive Matrix : Any Hermitian matrix A in $\mathbb{C}^{n \times n}$ where $\mathbf{u}^*A\mathbf{u} \geq 0$ for all vectors $\mathbf{u} \in \mathbb{C}^n$.
 Note that for any Hermitian matrix A and vector u, $\mathbf{u}^*A\mathbf{u}$ is real.

Completely Positive Matrix : Any positive matrix A in $\mathbb{C}^{n \times n}$ where $I_m \otimes A$ is positive.

The matrix

$$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

is an example of a matrix that is *unitary*, *Hermitian*, *positive* and *completely positive*.

Superoperators

A *Superoperator* S is a matrix over $\mathbb{C}$ with the following restrictions:

1. S is *completely positive*. This implies that S is positive as well.

2. For all positive matrices A, $Tr(S\,A) \leqslant Tr(A)$.

3.2 Basic quantum computation

3.2.1 Quantum bits

Quantum computation deals with operations on **qubit**s. A **qubit** is typically represented in the literature on quantum computation as a complex linear combination of $|0\rangle$ and $|1\rangle$, respectively identified with (1,0) and (0,1) in $\mathbb{C}^2$. Because of the identification of the basis vectors, any **qubit** can be identified with a non-zero vector in $\mathbb{C}^2$. In standard quantum computation, the important piece of information in a **qubit** is its direction rather than amplitude. In other words, given $q = \alpha\,|0\rangle + \beta\,|1\rangle$ and $q' = \alpha'\,|0\rangle + \beta'\,|1\rangle$ where $\alpha = \gamma\alpha'$ and $\beta = \gamma\beta'$, then q and q' represent the same quantum state.

A **qubit** that has either α or β zero is said to be in a *classical state*. Any other combination of values is said to be a *superposition*.

Section 3.3 on page 30 will introduce quantum circuits which act on **qubit**s. This section will have some forward references to circuits to illustrate points introduced here.

3.2.2 Quantum entanglement

Consider what happens when working with a pair of **qubits**, p and q. This can be considered as the a vector in $\mathbb{C}^4$ and written as

$$\alpha_{00} \left|00\right\rangle + \alpha_{01} \left|01\right\rangle + \alpha_{10} \left|10\right\rangle + \alpha_{11} \left|11\right\rangle . \tag{3.1}$$

In the case where p and q are two independent **qubits**, with $p = \alpha \left|0\right\rangle + \beta \left|1\right\rangle$ and $q = \gamma \left|0\right\rangle + \delta \left|1\right\rangle$,

$$p \otimes q = \alpha\gamma \left|00\right\rangle + \alpha\delta \left|01\right\rangle + \beta\gamma \left|10\right\rangle + \beta\delta \left|11\right\rangle \tag{3.2}$$

where $p \otimes q$ is the standard tensor product of p and q regarded as vectors. There are states of two **qubits** that cannot be written as a tensor product. As an example, the state

$$\frac{1}{\sqrt{2}} \left|00\right\rangle + \frac{1}{\sqrt{2}} \left|11\right\rangle \tag{3.3}$$

is not a tensor product of two distinct **qubits**. In this case the two **qubits** are said to be *entangled*.

3.2.3 Quantum gates

Quantum gates operate on **qubits**. These gates are conceptually similar to logic gates in the classical world. In the classical world the only non-trivial single **bit** gate is the Not gate which sends 0 to 1 and 1 to 0. However, there are infinitely many non-trivial quantum gates.

An $n-$**qubit** quantum gate is represented by a $2^n \times 2^n$ matrix. A necessary and sufficient condition for such a matrix to be a quantum gate is that it is *unitary*.

The entanglement of two **qubits**, p and q, is accomplished by applying a Hadamard transformation to p followed by a Not applied to q controlled by p. The circuit in figure 3.2 on page 31 shows how to entangle two **qubits** that start with an initial state of $\left|00\right\rangle$. See for how this can be done in L-QPL.

A list of some common gates, together with their usual quantum circuit representation is given in the next section in table 3.1 on page 32.

3.2.4 Measurement

The other allowed operation on a **qubit** or group of **qubits** is measurement. When a **qubit** is measured it assumes only one of two possible values, either $|0\rangle$ or $|1\rangle$. Given

$$q = \alpha \, |0\rangle + \beta \, |1\rangle \tag{3.4}$$

where $|\alpha|^2 + |\beta|^2 = 1$, then measuring q will result in $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$. Once a **qubit** is measured, re-measuring will always produce the same value.

In multi-**qubit** systems the order of measurement does not matter. If p and q are as in equation (3.1) on page 27, let us suppose measuring p gives $|0\rangle$. The measure will result in that value with probability $|\alpha_{00}|^2 + |\alpha_{01}|^2$, after which the system collapses to the state:

$$\alpha_{00} \, |00\rangle + \alpha_{01} \, |01\rangle \tag{3.5}$$

Measuring the second **qubit**, q, will give $|0\rangle$ with probability $|\alpha_{00}|^2$ or $|1\rangle$ with probability $|\alpha_{01}|^2$.

Conversely, if q was measured first and gave us $|0\rangle$ (with a probability of $|\alpha_{00}|^2 + |\alpha_{10}|^2$) and then p was measured, p will give us $|0\rangle$ with probability $|\alpha_{00}|^2$ or $|1\rangle$ with probability $|\alpha_{10}|^2$.

Thus, when measuring both p and q, the probability of getting $|0\rangle$ from both measures is $|\alpha_{00}|^2$, regardless of which **qubit** is measured first.

Considering states such as in equation (3.3), measuring either **qubit** would actually force the other **qubit** to the same value. This type of entanglement is used in many quantum algorithms such as quantum teleportation.

3.2.5 Mixed states

The notion of *mixed states* refers to an outside observer's knowledge of the state of a quantum system. Consider a 1 **qubit** system

$$\nu = \alpha \left|0\right\rangle + \beta \left|1\right\rangle . \tag{3.6}$$

If ν is measured but the results of the measurement are not examined, the state of the system is either $\left|0\right\rangle$ or $\left|1\right\rangle$ and is no longer in a superposition. This type of state is written as:

$$\nu = |\alpha|^2\{\left|0\right\rangle\} + |\beta|^2\{\left|1\right\rangle\}. \tag{3.7}$$

An external (to the state) observer knows that the state of ν is as expressed in equation (3.7) on page 29. Since the results of the measurement were not examined, the exact state (0 or 1) is unknown. Instead, a probability is assigned as expressed in the equation. Thus, if the **qubit** ν is measured and the results are not examined, ν can be treated as a probabilistic **bit** rather than a **qubit**.

3.2.6 Density matrix notation

The state of any quantum system of **qubits** may be represented via a *density matrix*. In this notation, given a **qubit** ν, the coefficients of $\left|0\right\rangle$ and $\left|1\right\rangle$ form a column vector u. Then the density matrix corresponding to ν is uu^*. If $\nu = \alpha \left|0\right\rangle + \beta \left|1\right\rangle$,

$$\nu = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \begin{pmatrix} \overline{\alpha} & \overline{\beta} \end{pmatrix} = \begin{pmatrix} \alpha\overline{\alpha} & \alpha\overline{\beta} \\ \beta\overline{\alpha} & \beta\overline{\beta} \end{pmatrix} . \tag{3.8}$$

When working with mixed states the density matrix of each component of the mixed state is added. For example, the mixed state shown in equation (3.7) on page 29 would be represented by the density matrix

$$|\alpha|^2 \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + |\beta|^2 \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} |\alpha|^2 & 0 \\ 0 & |\beta|^2 \end{pmatrix} . \tag{3.9}$$

Note that since the density matrix of mixed states is a linear combination of other density matrices, it is possible to have two different mixed states represented by the same density matrix.

The advantage of this notation is that it becomes much more compact for mixed state systems. Additionally, scaling issues are handled by insisting the density matrix has a trace = 1. During a general quantum computation, as we shall see, the trace can actually fall below 1 indicating that the computation is not everywhere total.

3.2.7 Gates and density matrices

When considering a **qubit** q as a column vector and a unitary transform T as a matrix, the result of applying the transform T to q is the new vector Tq. The density matrix of the original **qubit** is given by qq^*, while the density matrix of the transformed **qubit** is $(Tq)(Tq)^*$, which equals $T(qq^*)T^*$. Thus, when a **qubit** q is represented by a density matrix A, the formula for applying the transform T to q is TAT^*.

3.3 Quantum circuits

3.3.1 Contents of quantum circuits

Currently a majority of quantum algorithms are defined and documented using *quantum circuits*. These are wire-type diagrams with a series of **qubits** input on the left of the diagram and output on the right. Various graphical elements are used to describe quantum gates, measurement, control and classical **bits**.

Gates and **qubits**

The simplest circuit is a single wire with no action:

$$d \; d = [d \; \text{[Helefetek, finfabffh}_x ch$$

30

The next simplest circuit is one **qubit** and one gate. The **qubit** is represented by a single wire, while the gate is represented by a box with its name, G, inside it. This is shown in the circuit in figure 3.1. In general, the name of the wire which is input to the gate G may be different from the name of G's output wire. Circuit diagrams may also contain constant components as input to gates as in the circuit in figure 3.3 on page 31.

$$d = [let @ \text{-} \ldots \text{-} \ldots h_y$$

Figure 3.1: Simple single gate circuit

$$d = [let @ \text{-} \ldots fnch\ _{x'}$$
$$d = [let @ \text{-} \ldots fnch\ _{y'}$$

Figure 3.2: Entangling two **qubits**.

$$|1\rangle \;—\!\oplus\!—\; |0\rangle$$
$$|1\rangle \;—\!\bullet\!—\; |1\rangle$$

Figure 3.3: Controlled-Not of $|1\rangle$ and $|1\rangle$

Future diagrams will drop the wire labels except when they are important to the concept under discussion.

Controlled gates, where the gate action depends upon another **qubit**, are shown by attaching a wire between the wire of the control **qubit** and the controlled gate. The circuit in figure 3.2 on page 31 shows two **qubits**, where a Hadamard is applied to the top **qubit**, followed by a controlled-Not applied to the second **qubit**. In circuits, the control **qubit** is on the vertical wire with the solid dot. This is then connected via a horizontal wire to the gate being controlled.

A list of common gates, their circuits and corresponding matrices is given in table 3.1 on page 32.

Gate	Circuit	Matrix
Not (X)	$d = $ [let@token ei fnch	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Z	$d = $ [let Z@token ei fnch	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard	$d = $ [let H@token ei fnch	$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
Swap	$d = $ [let X@token ei fnch $d = $ [let X@token ei fnch	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Controlled-Not	$d = $ [let@token ei fnch $d = $ [let@token ei fnch	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
Toffoli	$d = $ [let@token ei fnch $d = $ [let@token ei fnch $d = $ [let@token ei fnch	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Table 3.1: Gates, circuit notation and matrices

Measurement

Measurement is used to transform the quantum data to classical data so that it may be then used in classical computing (e.g. for output). The act of measurement is placed at the last part of the quantum algorithm in many circuit diagrams and is sometimes just implicitly considered to be there.

While there are multiple notations used for measurement in quantum circuit diagrams, this thesis will standardize on the *D-box* style of measurement as shown in figure 3.4.

$$d = \boxed{} \quad \text{D-box}$$

Figure 3.4: Measure notation in quantum circuits

A measurement may have a double line leaving it, signifying a **bit**, or nothing, signifying a destructive measurement.

Operations affecting multiple **qubit**s at the same time are shown by extending the gate or measure box to encompass all desired wires. In the circuit in figure 3.5 on the previous page, the gate U applies to all of the first three **qubit**s and the measurement applies to the first two **qubit**s.
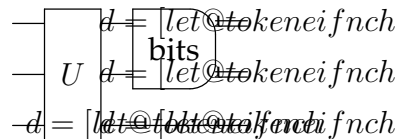
Figure 3.5: Examples of multi-**qubit** gates and measures

0-control and control by **bits**

The examples above have only shown control based upon a **qubit** being $|1\rangle$. Circuits also allow control on a **qubit** being $|0\rangle$ and upon classical values. The circuit in figure 3.6 with four **qubit**s $(r_1, r_2, p$ and $q)$ illustrates all these forms of control.

At g_1, a Hadamard is $1-$controlled by r_2 and is applied to each of r_1 and p. This is followed in column g_2 with the Not transform applied to r_2 being $0-$controlled by r_1 .

In the same column, a Z gate is $0-$controlled by q and applied to p. p and q are then measured in column g_3 and their corresponding classical values are used for control in g_4. In g_4, the U_R gate is applied to both r_1 and r_2, but only when the measure result of p is 0 and the measure result of q is 1.

$$d = [let@token\cdots]$$
$$g_1 \quad g_2 \quad g_3 \quad g_4$$
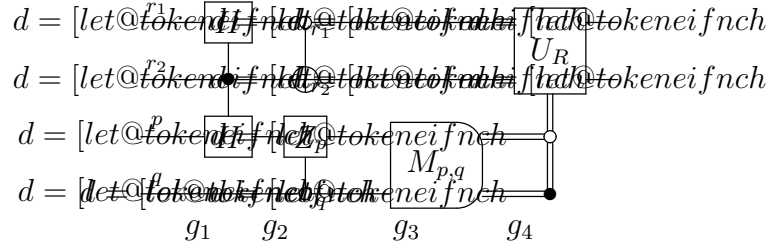
Figure 3.6: Other forms of control for gates
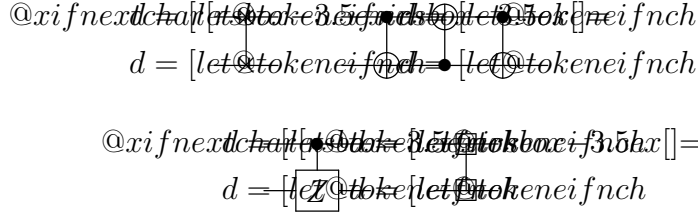
Multi-**qubit** lines

It is common to represent multiple **qubits** on one line. A gate applied to a multi-**qubit** line must be a tensor product of gates of the correct dimensions. The circuit in figure 3.7 on the preceding page shows n **qubits** on one line with the Hadamard gate (tensored with itself n times) applied to all of them. That is followed by a unary gate U_R tensored with $I^{\otimes(n-2)}$ and tensored with itself again. This will have the effect of applying an U_R gate to the first and last **qubits** on the line.
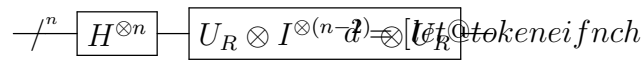
Figure 3.7: n **qubits** on one line

Other common circuit symbols

Two other symbols that are regularly used are the swap and controlled-$\mathbb{Z}$, shown in the circuit in figure 3.8 on page 35. Note that swap is just shorthand for a series of three controlled-Not gates with the control **qubit** changing. This can also be seen directly by multiplying the matrices for the controlled-Not gates as shown in equation (3.10) on page 35.

34

$@xifnext$dchadletStoke3.5fridslQet2Jok$neifnch$

$d = [let$&$tokeneifnch$ $= [le$ $tokeneifnch$

$@xifnext$dchadletStoke3.5Etokboxif3u5hx$[] =

$d = [le$Z$tokelletQtokeneifnch$

Figure 3.8: Swap and controlled-Z

$$
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} . \tag{3.10}
$$

3.3.2 Syntax of quantum circuits

Quantum circuits were originally introduced by David Deutsch in [3]. He extended the idea of standard classical based gate diagrams to encompass the quantum cases. In his paper, he introduced the concepts of quantum gates, sources of **bit**s, sinks and universal gates. One interesting point of the original definition is that it *does* allow loops. Currently, the general practice is not to allow loops of **qubit**s. The commonly used elements of a circuit are summarized in table 3.2 on page 36.

A valid quantum circuit must follow certain restrictions. As physics requires **qubit**s must not be duplicated, circuits must enforce this rule. Therefore, three restrictions in circuits are the *no fan-out*, *no fan-in* and *no loops* rules. These conditions are a way to express the *linearity* of quantum algorithms. Variables (wires) may not be duplicated, may not be destroyed without a specific operation and may not be amalgamated.

3.3.3 Examples of quantum circuits

This section will present three quantum algorithms and the associated circuits. Each of these circuits presented may be found in [4].

Desired element	Element in a quantum circuit diagram.	Example
qubit	A single horizontal line.	$d = [let@tokeneifnch$
Classical **bit**	A double horizontal line.	$d = [let@tokeneifnch$
Single-**qubit** gates	A box with the gate name (G) inside it, one wire attached on its left and one wire attached on the right.	$d = [G tokeneifnch$
Multi-**qubit** gates	A box with the gate name (R) inside it, n wires on the left side and the same number of wires on the right.	$d = [let@tokeneifnch$ R $d = [let@tokeneifnch$
Controlled **qubit** gates	A box with the gate name (H, W) inside, with a solid (1-control) or open (0-control) dot on the control wire with a vertical wire between the dot and the second gate.	$d = [W tokeneifnch$ $H = [let@tokeneifnch$
Controlled-Not gates	A *target* $\oplus$, with a solid (1-control) or open (0-control) dot on the control wire with a vertical wire between the dot and the gate.	$d = [let@tokeneifnch$ $d = [let@tokeneifnch$
Measure-ment	A *D-box* shaped node with optional names or comments inside. One to n single wires are attached on the left (**qubits** coming in) and 0 to n classical **bit** wires on the left. Classical **bits** may be dropped as desired.	$d = [let@tokeneifnch$ q, r H
Classical control	Control bullets attached to horizontal classical wires, with vertical classical wires attached to the controlled gate.	r $d = let@tokeneifXtokeneifnch$
Multiple **qubits**	Annotate the line with the number of **qubits** and use tensors on gates.	$d = [let@tokeneifnch$

Table 3.2: Syntactic elements of quantum circuit diagrams

First, *quantum teleportation*, an algorithm which sends a quantum bit across a distance via the exchange of two classical bits. This is followed by the *Deutsch-Jozsa algorithm*, which provides information about the global nature of a function with less work than a classical deterministic algorithm can. The third example is circuits for the *quantum Fourier transformation* and its inverse.

Quantum teleportation

The standard presentation of this algorithm involves two participants A and B. (Henceforth known as Alice and Bob). Alice and Bob first initialize two **qubit**s to $|00\rangle$, then place them into what is known as an *EPR* (for Einstein, Podolsky and Rosen) state. This is accomplished by first applying the Hadamard gate to Alice's **qubit**, followed by a controlled-Not to the pair of **qubit**s controlled by Alice's **qubit**.

Then, Bob travels somewhere distant from Alice, taking his **qubit** with him[1].

At this point, Alice receives a **qubit**, ν, in an unknown state and has to pass ν on to Bob. She then uses ν as the control and applies a controlled-Not transform to this new pair. Alice then applies a Hadamard transform applied to ν.

Alice now measures the two **qubit**s and sends the resulting two **bit**s of classical information to Bob.

Bob then examines the two **bit**s that he receives from Alice. If the **bit** resulting from measuring Alice's original bit is 1, he applies the Not (also referred to as X) gate (= $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$) to his **qubit**. If the measurement result of ν is one, he applies the Z gate (= $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$). Bob's **qubit** is now in the same state as the **qubit** Alice wanted to send. The circuit for this is shown in figure 3.9 on page 38.

For comparison , see showing how this would be implemented in L-QPL.

---

[1] Notice that all other physical constraints are ignored in this algorithm. There is no concern about how one separates the **qubit**s, transports the **qubit** or potential decoherence of the **qubit**s.
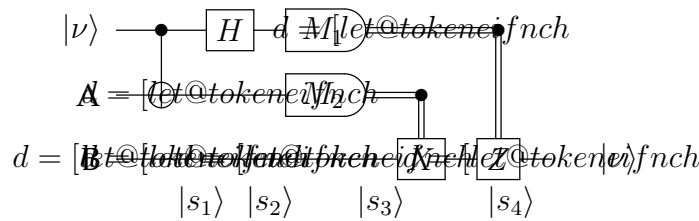
Figure 3.9: Quantum teleportation

Deutsch-Jozsa algorithm

The Deutsch-Jozsa algorithm describes a way of determining whether a function f^2 is *constant* (i.e. always 0 or 1) or *balanced* (i.e. produces an equal number of 0 or 1 results) based on applying it to one quantum bit. The function takes n **bits** as input and produces a single **bit**.

f is assumed to be an expensive function, therefore, a desired effect is to evaluate f as few times as possible before determining if f is balanced or constant. The worst case scenario when evaluating f classically is that determining the result requires $2^{n-1} + 1$ invocations of the function. The best possible case is 2 invocations, which occurs when f is balanced and the first two inputs chosen produce different results.

The quantum circuit requires only one application of the function to $n+1$ **qubits** which have been suitably prepared to make the decision.

The algorithm relies on being able to construct an $n + 1$ order unitary operator based upon f. In general, a unitary operator like this may be constructed by mapping the state $|a, b\rangle$ to $|a, b \oplus f(a)\rangle$ where $\oplus$ is the exclusive-or operator and a is n **bit** values. If we name this operator U_f, the circuit in figure 3.10 on page 38 will solve the problem with just one application. See the appendix, for how this would be done in L-QPL.
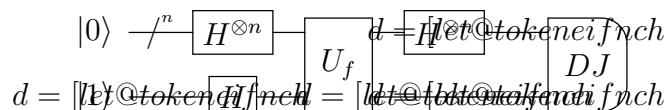
Figure 3.10: Circuit for the Deutsch-Jozsa algorithm

---

[2]The obvious pre-condition for the Deutsch-Jozsa algorithm is that the function f is *either* balanced or constant and not some general function. The results are not well-defined if f does not fit into one of the two possible categories.

The idea of quantum parallelism is what makes this and many other quantum algorithms work. The initial state of the system is set to $|0^{\otimes n} \otimes 1\rangle$ after which the Hadamard gate is applied to all of the **qubit**s. This places the input **qubit**s into a superposition of all possible input values and the answer **qubit** is a superposition of 0 and 1. At this point, the unitary transformation U_f is applied to the **qubit**s. Then the Hadamard transform is applied again to the input **qubit**s.

To complete the algorithm, measure *all* the **qubit**s. It can be shown that if f is constant, the input **qubit**s will all measure to 0, while if it is balanced, at least one of those **qubit**s will be 1.

Quantum Fourier transform

The circuits for the quantum Fourier transformation and its inverse are in figure 3.11 and figure 3.12 respectively. These transforms are used extensively in many quantum algorithms, including Shor's factoring algorithm.

The quantum Fourier transform is definable on an arbitrary number of **qubit**s. This is typically presented by eliding the 3rd to the $n - 3^{\text{rd}}$ lines and interior processing. The L-QPL code for the quantum Fourier transform is in the appendix, In this circuit, the parametrized transform R_n is the rotation transform, given by:

$$R_n = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^n}} \end{bmatrix}$$
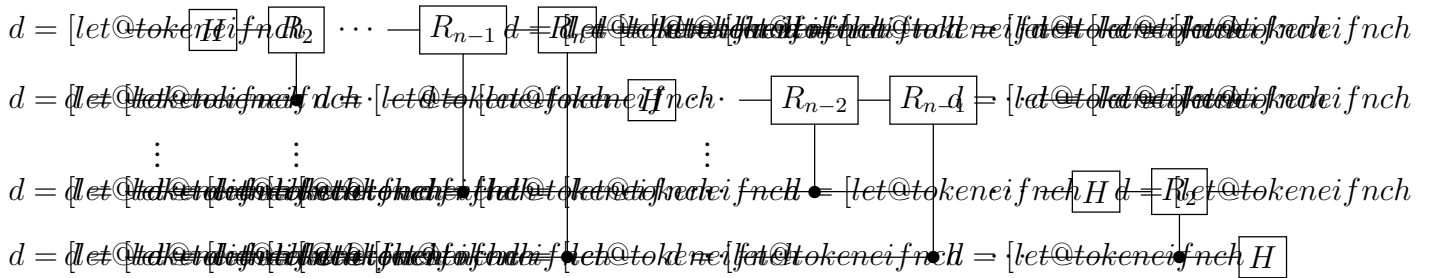
Figure 3.11: Circuit for the quantum Fourier transform

The inverse of a circuit is determined by traversing the circuit from right to left. This process changes the original quantum Fourier circuit to its inverse as shown in figure 3.12 on the preceding page. The L-QPL code for the inverse quantum Fourier transform is in the appendix,
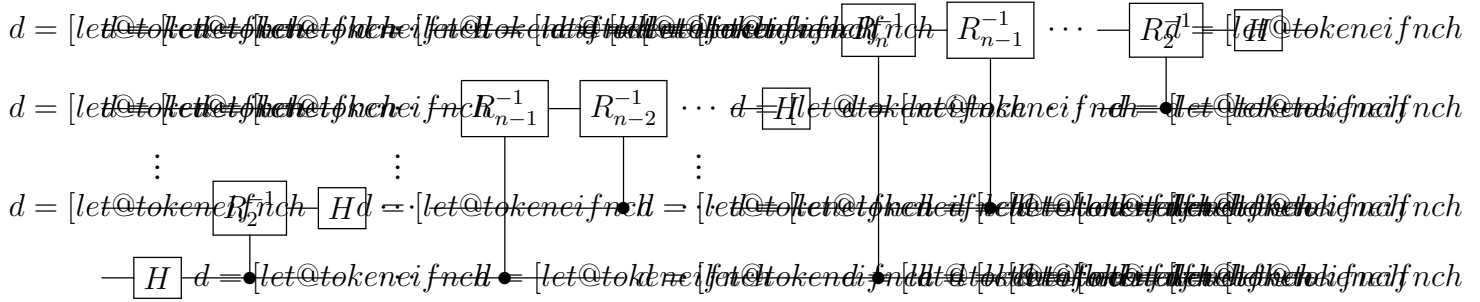
Figure 3.12: Circuit for the inverse quantum Fourier transform

3.4 Extensions to quantum circuits

To facilitate the transition to the programming language L-QPL, this section introduces three extensions to quantum circuits. The extensions are *renaming*, *wire bending and crossing*, and *scoped control*. Each extension adds expressive power to quantum circuits but does not change the semantic power. For each of the extensions, examples of how to re-write the extension in standard quantum circuit terminology will be provided.

3.4.1 Renaming

Quantum circuits currently allow renaming to be an implicit part of any gate. The circuit in figure 3.13 on page 40 gives an operation to explicitly do this and its rewriting in standard circuit notation.
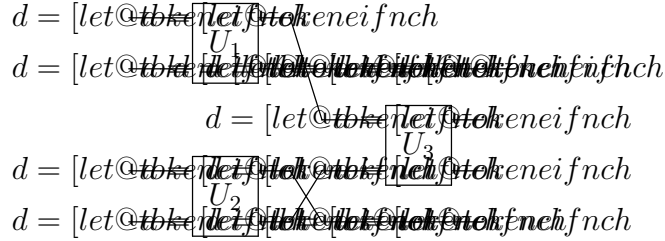

Figure 3.13: Renaming of a **qubit** and its equivalent diagram
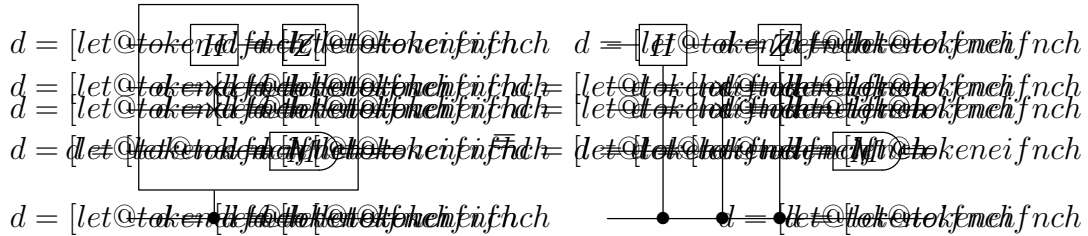
3.4.2 Wire crossing

Crossing and bending of wires in a circuit diagram is added to allow a simpler presentation of algorithms. The circuit in figure 3.14 on the preceding page illustrates the concept of re-organizing and bending of wires.

$$d = [let@token letf token eif nch$$
$$d = [let@token let@token let@token let@token nch f nch ch$$
$$d = [let@token letf token eif nch$$
$$d = [let@token let@token let@token nch eif nch$$
$$d = [let@token let@token let@token let nch f nch$$

Figure 3.14: Bending
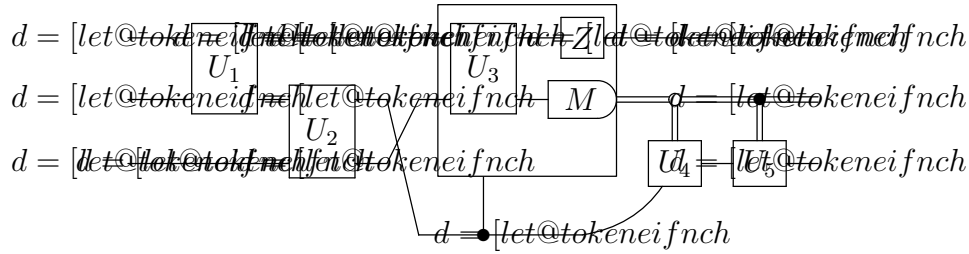
3.4.3 Scoped control

This extension allows us to group different operations in a circuit and show that all of them are controlled by a particular **qubit**. This is the same as attaching separate control wires to each of the gates in the grouped operations. Measurements are not affected by control. Figure 3.15 on the previous page shows a scoped control box on the left which includes a measure. The right hand side of the same figure shows the circuit translated back to standard circuit notation, with the measure not being affected by the control.

$$d = [let@token let@token keinf nch ch \qquad d = [let@token let@token let@token tokf nch f nch$$
$$d = [let@token let@token let@token nch f nch ch = [let@token let@token let@token let tokf nch f nch$$
$$d = [let@token let@token let@token nch f nch ch = [let@token let@token let@token let tokf nch f nch$$
$$d = [let@token let@token let@token keinf nch ch = [let@token let@token let@token token keif nch$$

$$d = [let@token let@token let@token nch f nch ch \qquad d = [let@token let@token tokf nch f nch$$

Figure 3.15: Scope of control

Scoping boxes correspond to procedures and blocks in L-QPL.

Naturally, both scoping and bending may be combined as in figure 3.16.

41

$$d = [let@tokeneifnch \quad U_1 \quad U_3 \quad Z]let@tokeneifnch$$

$$d = [let@tokeneifnch]let@tokeneifnch \quad M \quad d = [let@tokeneifnch$$

$$d = [let@tokeneifnch]let@tokeneifnch \quad U_4 \quad U_5]let@tokeneifnch$$

$$U_2$$

$$d = [let@tokeneifnch$$

Figure 3.16: Extensions sample

However, note that exchanging wires is not the same as swap. Exchanging a pair of wires is not affected by control, but a swap is affected by control, as shown in figure 3.17.
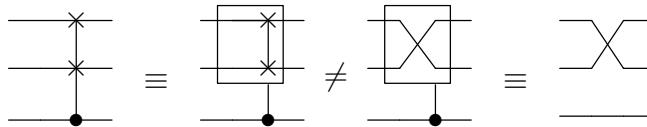

Figure 3.17: Swap in control vs. exchange in control

3.4.4 Circuit identities

Circuits allow the writing of the same algorithm in multiple ways. This sub-section will list some of the circuit identities that hold with the extended notation.

First, note that although a measure may appear inside a control box, it is not affected by the control, as in figure 3.18 on the previous page. Conversely, a measurement commutes with control of a circuit as in figure 3.18 on the preceding page.
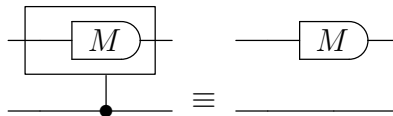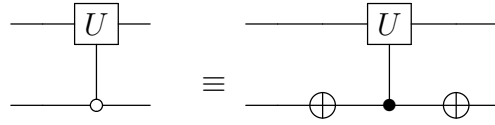

Figure 3.18: Measure is not affected by control

One of the notations introduced earlier was that of *0-control*. This type of control is the same as applying a *Not* transform before and after a *1-control*, as shown in figure 3.20 on page 43.
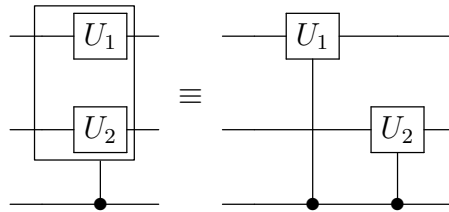
42
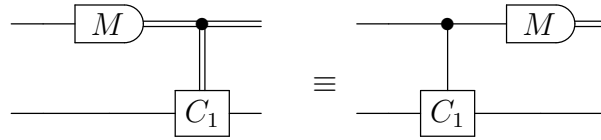
Figure 3.19: Control is not affected by measure

Figure 3.20: Zero control is syntactic sugar

Figure 3.21 on page 43 shows that scoped control of multiple transforms is the same as controlling those transforms individually. Figure 3.22 similarly shows that scoped con-

Figure 3.21: Scoped control is parallel control

trol of multiple transforms of the same **qubit** is the same as controlling those transforms serially.

Multiple control commutes with scoping as shown in figure 3.23 to figure 3.24 on page 44.

3.5 An alternate description of quantum circuits

In order to explore transforms of quantum circuits, it is helpful to have a algebraic description which will allow manipulation of the circuits. The goals of the algebraic description are:

- Represent **qubits** and **bits**;

- represent gates;
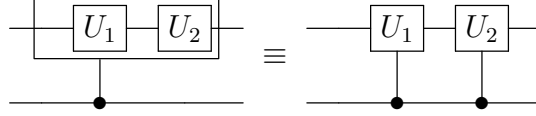
Figure 3.22: Scoped control is serial control

Figure 3.23: Multiple control

- allow algorithmic manipulations of the circuit;

- allow proving that correctness of manipulations.

Note that measurement is not included in this representation.
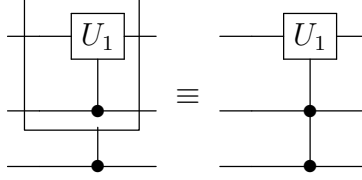
3.5.1 Base types

The base types **bit** and **qubit** are needed for any circuit. These will be taken as elemental and correspond to the classical notions of "bit" (i.e., 0 or 1) and "qubit" (i.e., $\alpha \left| 0 \right\rangle + \beta \left| 1 \right\rangle$).

Definition 3.5.1. The type **endpoint** is either **bit** or **qubit**.

Definition 3.5.2. The type **Arity** is a partial map from $W \subset \mathbb{N}$ to **Endpoint**.

Definition 3.5.3. A *wire* is an element of $W \subset \mathbb{N}$. A *typed wire* is a wire together with a specified **Arity**.

Definition 3.5.4. A *control wire* is a pair (w, b) where w is a typed wire and $b :: $ **Bool**.
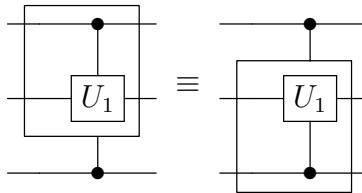
Figure 3.24: Control scopes commute

44

A quantum program uses *typed wire*s as its data.

Definition 3.5.5. A *gate* is a function from W_1, a set of typed wires, to W_2, another set of typed wires. The gate function must be a superoperator as defined in [7].

3.5.2 Types and Shapes

Although circuits, gates and low level subroutines are defined at the level of wires, programmatically, we would like to refer to groupings of wires. For example, a list of **qubit** or a register (tuple) of **qubit**. These groupings may have a required leaf type (e.g., **qubit** or **bit**) or it may be considered polymorphic. Additionally, the leaf type may be mixed or homogenous.

Definition 3.5.6. The singleton type **B** is defined as having the instance $B_{()}$ and the singleton type **Q** is defined as having the singleton instance $Q_{()}$.

Definition 3.5.7. The type family **QCData** consists of algebraic data types built up from **bit** or **qubit**.

Examples of **QCData** include $(\mathbf{bit}, \mathbf{qubit}, [\mathbf{qubit}])$ and **bit**.

Definition 3.5.8. Given an instance I of a type **T** in **QCData**, the *shape* of I is the instance I_s obtained from I by replacing all terms of type **bit** by $B_{()}$ and all terms of type **qubit** by $Q_{()}$.

For example, a 2 element list of **qubits** has shape $[Q_{()}, Q_{()}]$ while a pair of a **qubit** and **bit** has shape $(Q_{()}, B_{()})$

The concepts of shape and the type **QCData** allow us to group the wires of a quantum circuit into higher order types.

Chapter 4

Transformations of Quantum Programs

4.1 Subroutines

In the following, we will assume *gate* as above is a given and that $W \subset \mathbb{N}$ is fixed and finite. We will use typing notation to show membership in W — $w \in W$ is equivalent to $w :: W$.

4.1.1 Definition of a Subroutine

The concept of *subroutine* as defined below is intended to capture the essence of a describable computation in a quantum language. The low level subroutine is considered in isolation, meaning that it contains no information regarding how it fits into a larger circuit.

Definition 4.1.1. A *bare subroutine* is defined as a list of gates, written as $[g_0, g_1, \ldots, g_n]$. The list of gates must satisfy the following:

- range $g_i = \operatorname{dom} g_{i+1}$ for $i \in \{0, 1, \ldots, n-1\}$.

A bare subroutine B can then be viewed as a function from $\operatorname{dom} g_0$ to $\operatorname{range} g_n$ by applying each gate in order to $\operatorname{dom} g_0$.

Definition 4.1.2. A *low level subroutine* is a bare subroutine with a triple (C, I, O) where each of C, I, O are of type **Arity** and

$$\operatorname{dom} C \cap \operatorname{dom} I = \phi \tag{4.1}$$

$$\operatorname{dom} C \cap \operatorname{dom} O = \phi \tag{4.2}$$

In the definition of the low level subroutine, the triple (C, I, O) describes the inputs and outputs of the subroutine. C describes the control wires, which are inputs and outputs without change, I the input wires and O the output wires.

The above data together with three additional flags provides everything we need to know regarding a subroutine:

Definition 4.1.3. A *subroutine* is a low level subroutine together with a tripartite flag c with values in $\{N, B, Q\}$, and two boolean flags, r and n.

The three flags describe the ways this subroutine may be used. Each of these flags provide information that the calling quantum program uses to determine the ways the subroutine may be called:

- *Controllable*: When c is B, a calling program may make this subroutine the target of one or more *control wire*s with type **bit**. When c is Q, the control wires my be of type **bit** or **qubit**. When c is N, no control wires may be used. Note this is separate from the C wires of the subroutine, which may be used for internally controlling portions of the subroutine.

- *Reversible*: A calling program may specify the subroutine run normally or reversed.

- *No-controllable*: In the case where this subroutine is part of a preparation / unpreparation in a (prep, transform, unprep) sequence and that sequence is controlled, then the control wires may be ignored for this subroutine.

Noting that the domains of C and I and the domains of C and O do not overlap, we can also provide an ordering of the inputs and outputs for a low level subroutine. This ordering is used for display purposes and has no additional semantic content.

Definition 4.1.4. An *ordering* of a low level subroutine is a pair of bijections, (i, o) such that:

$$i : \operatorname{dom} C \cup \operatorname{dom} I \leftrightarrow \{0, 1, \ldots, n-1\} \tag{4.3}$$

$$o : \operatorname{dom} C \cup \operatorname{dom} O \leftrightarrow \{0, 1, \ldots, m-1\} \tag{4.4}$$

where $|\operatorname{dom} C \cup \operatorname{dom} I| = n$ and $|\operatorname{dom} C \cup \operatorname{dom} O| = m$.

4.1.2 Subroutine Calls

In this section, we describe the permissible bindings given two sets of wires, where the first set will be considered as control wires and the second as either input or output wires.

Definition 4.1.5. Given C and K are **Arity** functions over the same set of typed wires V, then f is a *permissible binding* to a set of typed wires W with **Arity** T_w when:

- $f : \operatorname{dom} C + \operatorname{dom} K \to W$,

- $\forall x, y \in \operatorname{dom} f, f(x) = f(y) \implies x = y \vee x, y \in \operatorname{dom} C$,

- $x \in \operatorname{dom} C \wedge C(x) = \mathbf{qubit} \implies T_w(f(x)) = \mathbf{bit} \vee T_w(f(x)) = \mathbf{qubit}$,

- $x \in \operatorname{dom} C \wedge C(x) = \mathbf{bit} \implies T_w(f(x)) = \mathbf{bit}$,

- $x \in \operatorname{dom} K \implies T_w(f(x)) = K(x)$.

We denote the permissible bindings to W of C and K by $F(C, K, W)$.

Definition 4.1.6. In a context of typed wires W_1, a *subroutine call*, resulting in the typed wires W_2, of the subroutine $([gates], C, I, O, r, c, n)$ is a tuple $(f, g, h, i, ncf, ctrls)$ consisting of three functions, two boolean flags and a list of control wires. The functions f, g, h must satisfy:

- $f : \operatorname{dom} C \to W_1 \cap W_2$

- $g : \operatorname{dom} I \to W_1$

48

- $h : \operatorname{dom} O \to W_2$

- $f + g \in F(C, I, W_1)$

- $f + h \in F(C, O, W_2)$.

The two flags must satisfy:

- $i \implies r$

- $ncf \implies n$.

The control list must satisfy:

- $\forall w_c \in ctrls, \operatorname{fst}(w_c) \in W_1 \cap W_2$,

- $N = c \implies \operatorname{length}(ctrls) = 0$,

- $B = c \implies \forall w_c \in ctrls, T_1(\operatorname{fst}(w_c)) = \mathbf{bit}$.

4.1.3 High Level Structure

Let s, t be of type of the family **QCData** and a be of shape s, b be of shape t. Further, let $A = \{qt | qt :: a, qt \text{ has shape } s\}$ and $B = \{qt | qt :: b, qt \text{ has shape } t\}$, that is, A and B are the sets of quantum terms of shape s (respectively t) and type a (respectively b).

Definition 4.1.7. A *high level structure* for a call to the subroutine $([gates], C, I, O, r, c, n)$ starting in context W_1 and ending in context W_2 is a pair of maps (i_s, o_s) with $i_s : A \to F(C, I, W_1)$ and $o_s : F(C, O, W_2) \to B$.

Definition 4.1.8. Given the data for a subroutine call as above in definition 4.1.6, a *structured subroutine call* is a high level structure as in definition 4.1.7 and a pair of quantum terms (a, b) such that:

- $i_s(a) = f + g$ and

- $o_s(f + h) = b$.

4.2 Subroutine Calls and Transformers

We are interested in two transformed calls of subroutines. Iteration and folding. We provide the necessary information for creating either a transformed call or first transforming the subroutine and then doing a standard call as in subsection sub-section 4.1.2.

4.2.1 Iteration

Iteration of subroutines means to call the same subroutine within a quantum circuit some positive number of times. Discussion points:

- Can we handle the case of zero iterations? Would this just mean doing a direct mapping of the I to the O in numerical sequence?

- Can we handle the case of negative iterations? This could mean calling the inverse of the subroutine in the case where it is reversible and then iterating.

- Does iteration affect the no-control or other flags?

- The analysis below assumes that "non-linear safety" is an important property to preserve during iteration. If we remove that requirement, iteration becomes more flexible, e.g., the bijections c_b and io_b could be replaced with a single bijection $cio_b : C \cup O \leftrightarrow C \cup I$. This would allow a **qubit** that was affected by the subroutine on one iteration to be used as the control on the next iteration. (Or is this the simple case that has already been handled?)

Definition 4.2.1. Given a subroutine as in subsection sub-section 4.1.2 on page 48, and a subroutine call $(f, g, h, i, ncf, ctrls)$ an *iterated call* of the subroutine $S = ([gates], C, I, O, r, c, n)$ consists of all elements of a subroutine call excepting f plus another tuple of five elements $(f_{in}, f_{out}, c_b, io_b, i_{count})$ where:

- $f_{in} : \operatorname{dom} C \to W_1 \cap W_2$

- $f_{out} : \operatorname{dom} C \to W_1 \cap W_2$

- i_{count} is a positive integer,

- c_b is a bijection (permutation) of C to C,

- io_b is a bijection between I and O.

- $f_{in} + g \in F(C, I, W_1)$

- $f_{out} + h \in F(C, O, W_2)$

- The relation $f_{out} \circ c_b^{i_{count}} \circ f_{in}^{-1}$ is a function.

Note these requirements mean that $|I| = |O|$.

Definition 4.2.2. Given the data for an iterated call, a *structured iterated call* is a high level structure as in definition 4.1.7 and a pair of quantum terms (a, b) such that:

- $i_s(a) = f_{in} + g$ and

- $o_s(f_{out} + h) = b$.

From the definition, note the C wires may be permuted as desired, but the combination of the f_{in}^{-1} and the permutation must leave the wires in a state where f_{out} is well-defined. See below for an example. Note the disposition of the wires due to calling the iterated subroutine is given by:

- Control mapping: $f_{out} \circ c_b^{i_{count}} \circ f_{in}^{-1}$,

- In-out mapping: $h \circ io_b^{i_{count}} \circ g^{-1}$.

Example 4.2.3 (Single call)**.**

For this example, we will elide the details relating to high level structure, invertability, control wires and the no-control flag.

Suppose $C = [c_1, c_2, c_3]$, $I = [i_1, i_2]$ and $O = [o_1, o_2]$. For the first part of the example, assume we are calling S from a context of $W_1 = [w_1, w_2, w_3, w_4]$, resulting in the same context (i.e., $W_1 = W_2$). In this case, the call of S can be given by:

- $f = \{c_1 \mapsto w_1, c_2 \mapsto w_1, c_3 \mapsto w_4\}$,

- $g = \{i_1 \mapsto w_2, i_2 \mapsto w_3\}$

- $h = \{o_1 \mapsto w_3, o_2 \mapsto w_2\}$

Hence we are calling S which will use w_1 as its first two control inputs and w_4 as the third. The inputs will use w_2 and w_3, while the output will "switch" those to w_3 and w_2.

Example 4.2.4 (Iterated call).

Assume S is as above and we are using the call as above. To aid in distinguishing input and output wires of the calling circuit, we will use w' for naming the output wires, so we may write $W_1 = [w_1, w_2, w_3, w_4]$ and $W_2 = [w'_1, w'_2, w'_3, w'_4]$, noting that $w_i = w'_i$ for $i \in \{1, 2, 3, 4\}$. A call iterated 5 times of S is $(g, h, i, ncf, ctrls)$ plus the tuple $(f_{in}, f_{out}, c_b, io_b, 5)$ where

- $f_{in} = \{c_1 \mapsto w_1, c_2 \mapsto w_1, c_3 \mapsto w_4\}$,

- $f_{out} = \{c_1 \mapsto w'_4, c_2 \mapsto w'_1, c_3 \mapsto w'_1\}$,

- $c_b = (c_1, c_2, c_3)$,

- $io_b = \{o_1 \mapsto i_2, o_2 \mapsto i_1\}$

If we calculate the movement of the wires for this iterated call, we see

$$(w_1, w_4) \xrightarrow{f_{in}^{-1}} C = (w_1, w_1, w_4) \xrightarrow{(1,2,3)^5} C = (w_4, w_1, w_1) \xrightarrow{f_{out}} (w'_1(= w_1), w'_4(= w_4)) \quad (4.5)$$

$$(w_2, w_3) \xrightarrow{g^{-1}} I = (w_2, w_3) \xrightarrow{io_b^5 \circ S} O = (w_3, w_2) \xrightarrow{h} (w'_3(= w_3), w'_3(= w_3)) \quad (4.6)$$

In this above example, the choice of f_{in} and f_{out} worked correctly with c_b such that the mappings were well defined. Consider however, if f_{out} were defined as $\{c_1 \mapsto w'_1, c_2 \mapsto w'_1, c_3 \mapsto w'_4\}$. We would then be expected to map / combine w_1 and w_4 into w'_1 and duplicate w_1 into both w'_1 and w'_4.

4.2.2 Iteration transformation of a subroutine

The data required to transform a subroutine is similar to that of an iterated subroutine call.

Definition 4.2.5. The *iteration transform* of a subroutine is a function S_i with parameters (n, c_p, io_b) which takes the subroutine $(gates, C, I, O, r, c, n)$ to the subroutine $(gates', C', I', O', r, c, n)$. The parameters have the following types:

$$n :: \textbf{Int}, n > 0;$$

$$c_p :: \textbf{Perm}_\textbf{C}, \text{Permutations of the elements of } C$$

$$io_b :: \textbf{bijection}(\textbf{I} \leftrightarrow \textbf{O})$$

The function S_i performs the following actions:

- Type checking:

  - Ensure c_p is a permutation of the appropriate number of wires.

  - Ensure n is a positive number greater than 0.

  - Ensure io_b is a bijection between the O and I wires.

- Create n copies of $gates$.

- Connect the O wires or copy $i - 1$ to the I wires of copy i, using the bijection io_b.

- Apply the permutation c_p to the C out wires of copy $i - 1$ and connect to the input C wires of copy I.

- Apply the permutation c_p^k to the C out wires of copy n where $c_p^{n-1+k} = Id$. Connect those wires to the C' output.

- Connect the C' input wires to the C input wires of copy 1.

- Connect the I' wires to the I wires of copy 1.

- Connect the O wires of copy n to the O' wires.

The intended effect of this transform may be seen in figure figure 4.1
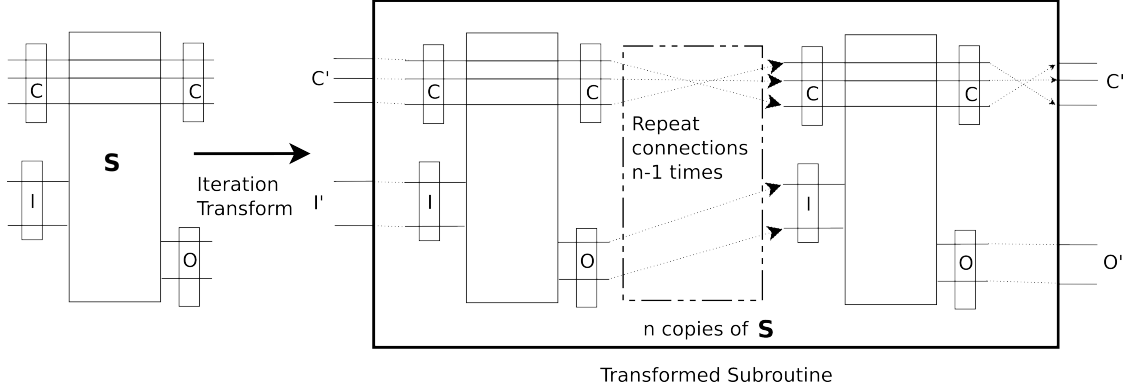
Figure 4.1: Transforming a subroutine to an iterated subroutine

4.2.3 Folding subroutines

In addition to the concept of iteration the ability to fold subroutines over data structures could also be useful. A folded call of a subroutine is similar to an iteration, in that controls and possibly some of the inputs and outputs are iterated. The difference occurs in that we expect to take some of the inputs from a data structure and return some of the outputs to a data structure. An example of this would be folding over a list of **qubit**s where each qubit is taken as a input for each iteration.

First, we will examine the requirements for a non-linear safe fold, that is, where no input duplication on the control wires is allow

Definition 4.2.6. Given a subroutine $S = ([gate], C, I, O, c, r, n)$, a starting context of typed wires W_1 and a data structure on wires $D \subset W_1$, a *linear-only folded call* of S over D resulting in the context of typed wires W_2 and the data structure $E \subset W_2$ consists of the tuple $(CI_f, CO_f, g, h, ciof_b, s_{in}, s_{out})$ where

- $CI_f :: \textbf{Arity}, \operatorname{dom} CI_f \subset \operatorname{dom} C \cup \operatorname{dom} I$,

- $CO_f :: \mathbf{Arity}, \operatorname{dom} CO_f \subset \operatorname{dom} C \cup \operatorname{dom} O$,

- $g : \operatorname{dom} C \cup \operatorname{dom} I / \operatorname{dom} CI_f \to W_1$ and g is injective,

- $h : \operatorname{dom} C \cup \operatorname{dom} O / \operatorname{dom} CO_f \to W_2$ and h is injective

- $ciof_b$ is a bijection between $(\operatorname{dom} C \cup \operatorname{dom} O) / \operatorname{dom} CO_f$ and $(\operatorname{dom} C \cup \operatorname{dom} I) / \operatorname{dom} CI_f$,

- $s_{in} : \operatorname{dom} CI_f \to W_1$ (pulls from D),

- $s_{out} : \operatorname{dom} CO_f \to W_2$ (pushes to E),

- $g + s_{in} \in F(\phi, I, W_1)$,

- $ciof_b^{-1} + s_{in} \in F(\phi, I, W_1)$,

- $ciof_b + s_{out} \in F(\phi, O, W_2)$,

- $h + s_{out} \in F(\phi, O, W_2)$,

- $|\operatorname{dom} CI_f| = leafsize(D)$,

- $leafsize(E) = |\operatorname{dom} CO_f|$.

Definition 4.2.7. Given the data for an linear-only folded call, a *structured linear-only folded call* is a pair of high level structures (i_s, o_s) and (i_{fs}, o_{fs}) and a pair of quantum terms (a, b) such that:

- $i_s(a) = g + s_{in}$,

- $o_s(h + s_{out}) = b$,

- $i_{fs}(a) = s_{in}$, and

- $o_{fs}(s_{out}) = b$.

Discussion points:

- Does the structured call get rid of the last two items regarding leafsize? Same issue with non-linear safe call.

55

- For the structured call, it appears to me that there is only one pair of terms a, b with two different (de)structuring morphisms.
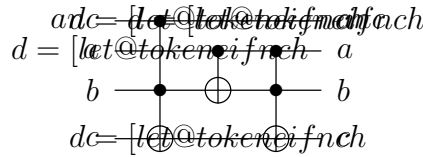
The action on the wires of the calling program will be given by this relation:

$$s_{out} + h \circ (s_{out} + cio_f + s_{in})^{len(D)} \circ (g + s_{in})^{-1}.$$

Example 4.2.8 (Fold over Carry).

For this example, we use the carry portion of the addition algorithm found in [8]. The carry circuit is shown below:

The intent of this circuit is to compute the final carry when adding the quregisters A and B. The input prepares the anc in state $|0\rangle$ and an auxiliary register R, the same size as A and B as $|00\dots0\rangle$. Assume that $A = [w_1, w_2, w_3], B = [w_4, w_5, w_6], anc = w_7$ and $R = [w_8, w_9, w_{10}]$. (A, B, R), a tuple of three registers forms our D — the input to the fold. Then, we may perform a folded call of carry as follows:

- $\mathrm{dom}\, C = \{anc, a\},\ \mathrm{dom}\, I = \{b, c\} = \mathrm{dom}\, O$;

- $\mathrm{dom}\, CI_f = \{a, b, c\}$ and $\mathrm{dom}\, CO_f = \{anc, a, b\}$;

- $g = \{anc \mapsto w_6\},\ h = \{c \mapsto w_{10}\}$

- $cio f_b = \{c \mapsto anc\}$

- $s_{in} = \{a \mapsto D[0], b \mapsto D[1], c \mapsto D[2]\}$

- $s_{out} = \{anc \mapsto E[2], a \mapsto E[0], b \mapsto E[1]\}$.

From the mapping s_{out}, we set $E = (A, B, R')$ where $R' = [w_7, w_8, w_9]$.

Discussion points:

- Is there a non-linear safe use case? Carry seemed quite simple, but it required linear safe inputs when folded.

- Each fold iteration input might be multiple **qubits**, combined into a single data structure, as in example 4.2.8 on page 56.

- The number of inputs and outputs no longer need to agree as they did in iteration. An example would be a subroutine that applied a two **qubit** gate and then discarded one of the **qubits**. The fold would be expected to convert a list of pairs of **qubits** to a list of **qubits**. (Note this subroutine would not be reversible or controllable).

- The shape of the foldable in and out as well as the number of **qubits** at each leaf would need to be known.

- The $F(C, K, W)$ permissible functions are not quite right for linear-only folds — we do not want to allow duplication of any of the inputs, so $F(C, K, W)$ becomes $F(\phi, C + K, W)$.

Definition 4.2.9. Given a subroutine $S = ([gate], C, I, O, c, r, n)$, a starting context of typed wires W_1 and a data structure on wires $D \subset W_1$, a *folded call* of S over D resulting in the context of typed wires W_2 and the data structure $E \subset W_2$ consists of the tuple $(I_f, O_f, f_{in}, f_{out}, g, h, c_b, iof_b, s_{in}, s_{out})$ where

- $I_f :: \textbf{Arity}, \operatorname{dom} I_f \subset \operatorname{dom} I$,

- $O_f :: \textbf{Arity}, \operatorname{dom} O_f \subset \operatorname{dom} O$,

- $f_{in} : \operatorname{dom} C \to W_1 \cup W_2$

- $f_{out} : \operatorname{dom} C \to W_1 \cup W_2$

- $g : \operatorname{dom} I / \operatorname{dom} I_f \to W_1$

- $h : \operatorname{dom} O / \operatorname{dom} O_f \to W_2$

- c_b is a bijection (permutation) of C,

- iof_b is a bijection between $\operatorname{dom} O / \operatorname{dom} O_f$ and $\operatorname{dom} I / \operatorname{dom} I_f$,

- $s_{in} : \operatorname{dom} I_f \to W_1$ (pulls from D)

- $s_{out} : \operatorname{dom} O_f \to W_2$ (pushes to E)

- $f_{in} + g + s_{in} \in F(C, I, W_1)$,

- $f_{out} + h + s_{out} \in F(C, O, W_2)$,

- $iof_b + s_{in} \in F(\phi, I, W_1)$,

- $iof_b + s_{out} \in F(\phi, O, W_2)$,

- $|\operatorname{dom} I_f| = leafsize(D)$

- $leafsize(E) = |\operatorname{dom} O_f|$

Definition 4.2.10. Given the data for an folded call, a *structured folded call* is a pair of high level structures (i_s, o_s) and (i_{fs}, o_{fs}) and a pair of quantum terms (a, b) such that:

- $i_s(a) = f_{in} + g + s_{in}$,

- $o_s(f_{out} + h + s_{out}) = b$,

- $i_{fs}(a) = s_{in}$, and

- $o_{fs}(s_{out}) = b$.

4.2.4   Subroutine to folded subroutine transform

In order to transform a given subroutine, we require the following data:

- A bijection from some subset of the control and output wires to some subset of the control and input wires;

- a count of the number of iterations.

This allows us to derive a new C, I, O for the fold subroutine. We proceed with the formal definition.

Definition 4.2.11. The *fold transform* of a subroutine is a function S_f with parameters (n, b_{cio}) which takes the subroutine $(gates, C, I, O, r, c, n)$ to the subroutine $(gates', C', I', O', r, c, n)$. The parameters have the following types:

$$n :: \textbf{Int}, n > 0;$$

$$b_{cio} :: \textbf{bijection}(\textbf{CI} \leftrightarrow \textbf{CO})$$

$$\text{where } CI \subseteq C \cup I \text{ and } \subseteq C \cup O.$$

Note that b_{cio} may be defined as a set of ordered pairs

$$\{(f, t) | f \in C \cup I, t \in C \cup O, \text{ and } f, t \text{ appear at most once}\}. \tag{4.7}$$

The data we need to create for the end result are the set of control wires, the input and output wires and the new gates sequence. We proceed with presenting the algorithm for the control wires.

When considering which inputs (and hence outputs) are control wires in the transformed subroutine, we must follow the path of a control input. A control input to the base subroutine will remain a control input to the transformed subroutine only if its full folded path contains only control wires before exiting. Depending upon the structure of b_{cio} it is possible all, none or some finite subset of specific control wires become controls for the transformed routine.

To determine if a wire is a control, we will calculate a characteristic of the wire and show that it requires at most k iterations to calculate, where k is the number of control wires of the original subroutine.

First, define:

$$\Omega :: C \to C \cup \{*, @\}$$

$$\Omega(c) = \begin{cases} c' & \text{if } b_{cio}(c) = c' \text{ and } c' \in C, \\ * & \text{if } c \text{ is not the first element of any pair in } b_{cio}, \\ @ & \text{if } b_{cio}(c) = j \text{ and } j \in I. \end{cases}$$

Then, noting that $k = |C|$, define:

$$\Gamma :: C \to \mathbb{N} \cup \{\infty\}$$

$$\Gamma(c) = \begin{cases} \infty & \Omega(c) = *, \\ 0 & \Omega(c) = @, \\ 1 + \Gamma(\Omega(c)) & \Gamma(\Omega(c)) < k, \\ \infty & \Gamma(\Omega(c)) \geq k. \end{cases}$$

Dually, we may define functions $\Theta(c)$ and $\Delta(c)$:

$$\Theta :: C \to C \cup \{*, @\}$$

$$\Theta(c) = \begin{cases} c' & \text{if } b_{cio}(c') = c \text{ and } c' \in C, \\ * & \text{if } c \text{ is not the second element of any pair in } b_{cio}, \\ @ & \text{if } b_{cio}(o) = c \text{ and } o \in O. \end{cases}$$

$$\Delta :: C \to \mathbb{N} \cup \{\infty\}$$

$$\Delta(c) = \begin{cases} \infty & \Theta(c) = *, \\ 0 & \Theta(c) = @, \\ 1 + \Delta(\Theta(c)) & \Delta(\Theta(c)) < k, \\ \infty & \Delta(\Theta(c)) \geq k. \end{cases}$$

Note that in the case of cycles among control wires, the cycle *must* be of size k or less. As such, at most k iterations of Γ are required before confirming a value for $\Gamma(c)$. The same argument applies to computing Θ.

Assuming that C is ordered, the data for b_{cio} may be stored such that computing $b_{cio}(c)$ and therefore $\Omega(c)$ is of $\mathcal{O}(\log k)$. Therefore, Γ is of complexity $\mathcal{O}(k \log k)$ and computing Γ for all of C will then be of $\mathcal{O}(k^2 \log k)$. Computing Δ will have the same complexity.

We may now describe the algorithm for determining control wires, C' input wires, I' and output wires, O' of the transformed subroutine. In the algorithm, n is the number of iterations, k is the cardinality of C. Additionally, we compute a *rank* of each c in C'. The *rank* of c is the number of iterations that c will go through.

1. Add all members of I to I', subscripting with 1.

2. For all $i \in I$ where $i \notin \text{range } b_{cio}$, add $i_2, i_3, \ldots, i_n$ to I'.

3. Add all members of O to O', subscripting with n.

4. For all $o \in O$ where $o \notin \text{dom } b_{cio}$, add $o_1, o_2, \ldots, o_{n-1}$ to O'.

5. Partition C into four subsets:

$$C_* = \{c | c \notin \text{dom } b_{cio} \text{ and } c \notin \text{range } b_{cio}\},$$

$$C_d = \{c | c \in \text{dom } b_{cio} \text{ and } c \notin \text{range } b_{cio}\},$$

$$C_r = \{c | c \notin \text{dom } b_{cio} \text{ and } c \in \text{range } b_{cio}\},$$

$$C_{dr} = \{c | c \in \text{dom } b_{cio} \text{ and } c \in \text{range } b_{cio}\}.$$

6. For each $c \in C_*$, add $c_1, c_2, \ldots, c_n$ to C'. Set the rank of each of these to 0.

7. For each $c \in C_d$, compute $j = \Gamma(c)$. If $j \geq n$, add $c_1, c_2, \ldots, c_n$ to C'. If not, then:

   - Add $c_{n-j}, c_{n-j+1}, \ldots, c_n$ to C', setting the rank of c_ℓ to $n - \ell$.

- Add $c_1, c_2, \ldots, c_{n-j-1}$ to I'.

8. For each $c \in C_r$, compute $m = \Delta(c)$. If $m \geq n$, add $c_1, c_2, \ldots, c_n$ to C', setting each item to rank 0. If not, then:

    - Add $c_1, c_2, \ldots, c_{j+1}$ to C', setting each rank to 0.

    - Add $c_{j+2}, c_{j+3}, \ldots, c_n$ to I'.

9. For each $c \in C_{dr}$, compute $j = \Gamma(c), m = \Delta(c)$. Then if $j > n - 1$, add c_1 to C', setting its rank to $n - 1$, otherwise add it to I'. Conversely, if $m > n - 1$, add c_n to C', setting its rank to 0, otherwise add it to O'. In the case where both c_1 and c_n are added to C', we have actually added one too many items to C' as there will be a duplication. This is addressed in the final step, where the in-out names of the control wires are reconciled.

10. Reconcile the C' names: For each $c_h \in C'$, compute $x = b_{cio}^{Rank(c)}(c)$. In C', replace x_n with c_h. After completing this computation, remove any duplicates.

4.2.5 Examples of folding

Example 4.2.12. Consider $S = (\_, \{a, b\}, \{c\}, \{d\}, \_, \_, \_)$ and $b_{cio} = \{(a, c), (b, a)\}$ and an iteration count of 3. See figure figure 4.2.

From the data, we compute:

$$\Omega(a) = @ \text{ and } \Omega(b) = a,$$

$$\Gamma(a) = 0 \text{ and } \Gamma(b) = 1,$$

$$\Theta(b) = * \text{ and } \Delta(b) = \infty,$$

$$\Theta(a) = b \text{ and } \Delta(a) = \infty.$$

Now, following the steps of the algorithm,

$d = [let@token...let@token...let@token...let@token...let@tokeneifnch$

$d = [let@token...let@token...let@tokeneifnch$

$a_1 \quad d = [let@tokeneifnch \quad a_2 \quad [let@tokeneifnch \quad a_3 \quad [let@tokeneifnch$

$b_1 \quad d = [let@tokeneifnch \quad b_2 \quad [let@tokeneifnch \quad b_3 \quad [let@tokeneifnch$

$S_1 \qquad\qquad S_2 \qquad\qquad S_3$

$c_1 \quad d = [let@tokeneifnch_{d_1} \quad c_2 \quad [let@tokeneifnch_{d_2} \quad c_3 \quad [let@tokeneifnch_{d_3}$

$d = [let@token...let@token...let@tokeneifnch$

$d = [let@token...let@token...let@token...let@token...let@tokeneifnch$

Figure 4.2: Fold with extra in/out

1. $I' \mapsto \{c_1\}$.

2. No change to I'.

3. $O' \mapsto \{d_3\}$.

4. $O' \mapsto \{d_1, d_2, d_3\}$.

5. $C_* = \phi, C_d = \{b\}, C_r = \phi, C_{dr} = \{a\}$.

6. For C_d, As $\Gamma(b) = 1$, we have $C' \mapsto \{b_2, b_3\}$ and $I' \mapsto \{c_1, b_1\}$. The rank of b_2 is 1 and the rank of b_3 is 0.

7. For $C_{dr} = \{a\}$, $\Gamma(a) = 0$ and $\Delta(a) = \infty$, therefore we add a_1 to I' and a_3 to C'. The rank of a_3 is zero. At this stage, we now have $I' = \{c_1, b_1, a_1\}, O' = \{d_1, d_2, d_3\}$ and $C' = \{b_2, b_3, a_3\}$.

8. Reconcile C': $b_{cio}(b) = a$ and $b_{cio}(a) = c$, therefore $b^1_{cio}(b_2) = a_3$, $b^0_{cio}(b_3) = b_3$, and $b^0_{cio}(a_3) = a_3$. Following our replacement scheme, $C' = \{b_2, b_3\}$.
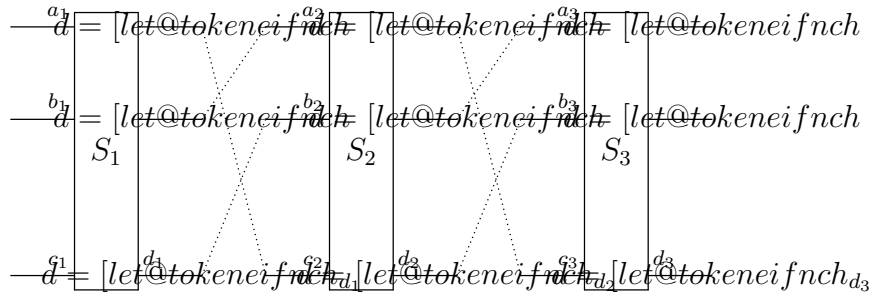
Hence our final result is:

$$I' = \{c_1, b_1, a_1\},$$

$$O' = \{d_1, d_2, d_3\} \text{ and}$$

$$C' = \{b_2, b_3\}.$$

Example 4.2.13. Consider $S = (\_, \{a, b\}, \{c\}, \{d\}, \_, \_, \_)$ as in Example 4.2.12 and $b_{cio} = \{(a, c), (b, a), (d, b)\}$ and an iteration count of 3. See figure figure 4.3.

Figure 4.3: Fold with three iterations

From the data, we have:

$$\Omega(a) = @ \text{ and} \qquad\qquad \Gamma(a) = 0,$$

$$\Omega(b) = a \text{ and} \qquad\qquad \Gamma(b) = 1,$$

$$\Theta(b) = @ \text{ and} \qquad\qquad \Delta(b) = 0,$$

$$\Theta(a) = b \text{ and} \qquad\qquad \Delta(a) = 1.$$

Now, following the steps of the algorithm,

1. $I' \mapsto \{c_1\}$.

2. No change to I'.

3. $O' \mapsto \{d_3\}$.

4. All o are in the range of b_{cio}, therefore no further change to O'.

5. $C_* = C_d = C_r = \phi, C_{dr} = \{a, b\}$.

6. As $n - 1 = 2 > \Gamma(a), \Gamma(b)$, we have $I' \mapsto \{c_1, a_1, b_1\}$. Similarly, since $n - 1 = 2 > \Delta(a), \Delta(b)$, $O' \mapsto \{a_3, b_3, d_3\}$
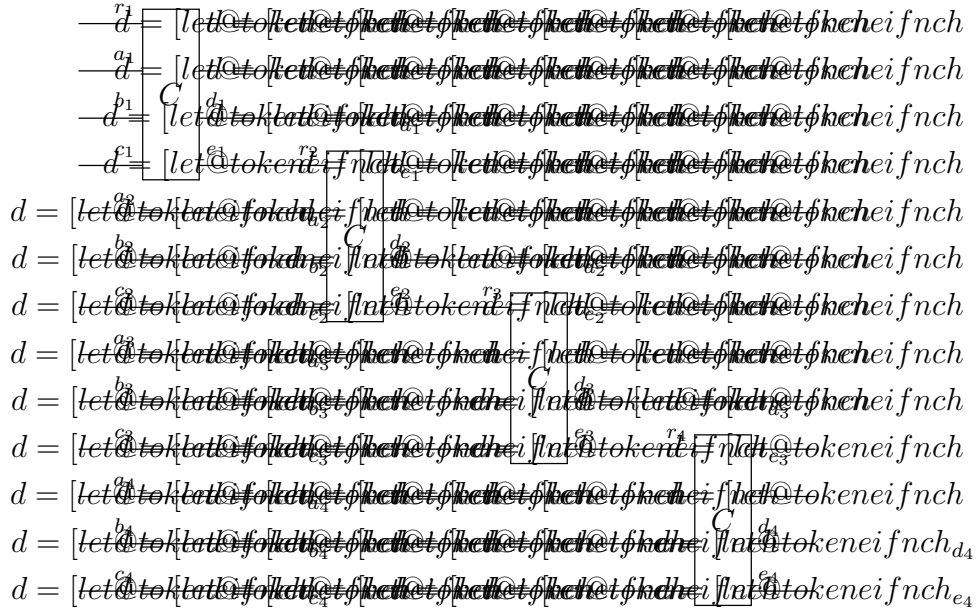
7. With C' empty, there are no further steps.

Hence our final result is:

$$I' = \{c_1, b_1, a_1\},$$

$$O' = \{a_3, b_3, d_3\} \text{ and}$$

$$C' = \phi.$$

Example 4.2.14. Consider $C = (\_, \{r, a\}, \{b, c\}, \{d, e\}, \_, \_, \_)$, $b_{cio} = \{(e, r)\}$ and an iteration count of 4. See figure figure 4.4.

Figure 4.4: Fold of Carry

65

From the data, we have:

$$\Omega(r) = * \text{ and} \qquad\qquad \Omega(a) = *,$$

$$\Gamma(r) = \infty \text{ and} \qquad\qquad \Gamma(a) = \infty,$$

$$\Theta(r) = @ \text{ and} \qquad\qquad \Delta(r) = 0,$$

$$\Theta(a) = * \text{ and } \Delta(a) = \infty.$$

Now, following the steps of the algorithm,

1. $I' \mapsto \{b_1, c_1\}$.

2. No element of I appears in b_{cio}, therefore we now have $I' \mapsto \{b_1, c_1, b_2, c_2, b_3, c_3, b_4, c_4\}$

3. $O' \mapsto \{d_4, e_4\}$.

4. Only e is in the range of b_{cio}, therefore we add d_1, d_2, d_3 to O', giving us $\{d_1, d_2, d_3, d_4, e_4\}$.

5. $C_* = \{a\}, C_d = \phi, C_r = \{r\}, C_{dr} = \phi$.

6. Considering C_*, we add $\{a_1, a_2, a_3, a_4\}$ to C', each with rank 0.

7. Next considering C_r, as $\Delta(r) = 0$, we add r_1 to C' with rank 0 and then add r_2, r_3, r_4 to O'.

8. As each element of C' is of rank 0, there is no changes to the names.

Hence our final result is:

$$I' = \{b_1, c_1, b_2, c_2, b_3, c_3, b_4, c_4\},$$

$$O' = \{d_1, d_2, d_3, d_4, e_4, r_2, r_3, r_4\} \text{ and}$$

$$C' = \{r_1, a_1, a_2, a_3, a_4\}.$$

4.3 Alternate Algorithm for Fold Transformation

In this section, we present an alternate algorithm for calculating the type and names of control, input and output wires for a folded subroutine. The starting point is Definition 4.2.11 and the ordered pairs of b_{cio}: A set of ordered pairs $\{(f,t)|f \in C \cup I, t \in C \cup O\}$.

To implement the algorithm of calculating the folded subroutines C, I and O, we need the following:

- $I \cap O = \phi$, which may be accomplished by renaming if needed.

- We create the sets C_j, I_j, O_j for $j = 1 \ldots n$ where the members of X_j are the elements of X together with the subscript j where X is one of C, I, O.

The algorithm proceeds by creating a set of "type" constraints for each of the elements of the new sets, based upon b_{cio} and membership in a C, I or O set.

The algorithm steps are:

1. Create a set $\mathcal{C}$ of pairs (a_j, b_{j+1}) for j ranging from 0 to $n-1$ based upon the bijection b_{cio}.

2. For each $j = 0 \ldots n - 1$,

    (a) For all $c \in C_j$, if $(c, \_) \notin \mathcal{C}$, add the pair (c, ϕ)

    (b) For all $c \in C_j$, if $(\_, c) \notin \mathcal{C}$, add the pair (ϕ, c)

    (c) For all $o \in O_j$, if $(o, \_) \notin \mathcal{C}$, add the pair (o, ϕ)

    (d) For all $i \in I_j$, if $(\_, i) \notin \mathcal{C}$, add the pair (ϕ, i)

3. Convert the pairs in $\mathcal{C}$ to equations and constraints as follows for each pair (a, b):

    (a) $a = \phi \implies \begin{cases} b :: (b, C) & \text{when } b \in C_j \text{for some } j, \\ b :: (b, I) & \text{when } b \in I_j \text{for some } j. \end{cases}$

$$\text{(b) } a \in C_{j-1} \implies \begin{cases} b = a & \text{when } b \in C_j, \\ b = a, b :: I & \text{when } b \in I_j, \\ a :: (C, a) & \text{when } b = \phi. \end{cases}$$

$$\text{(c) } a \in O_{j-1} \implies \begin{cases} b = a, b :: O & \text{when } b \in C_j, \\ \text{no equation} & \text{when } b \in I_j, \\ a :: (O, a) & \text{when } b = \phi. \end{cases}$$

4. Solve the equations with the constraints, with the assumption that
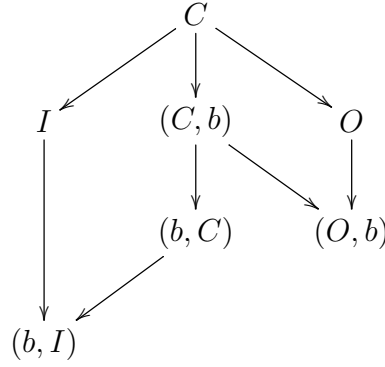
For example,

- $a :: O, a :: C$ is solvable with $a :: O$;

- $a = b, b :: (b, C), a :: I$ is solved by $b, a :: (b, I)$;

- $a = b, b = d, a :: (a, C), d :: (C, d)$ is solved by $a, b, d :: (a, C)$.

5. For the folded subroutine, X will be all items of "type" X, where X is any of C, I, O, the names of each entry will be the companion name with the "type".

> **(Internal Note):** The algorithm above works correctly, but likely has too high of a complexity, depending on the number of iterations. Need to revise in the next version to see if we can make the complexity proportional to the number of wires in/out.

4.3.1 Examples of folding with Alternate Algorithm

Example 4.3.1. We repeat Example 4.2.12, that is, $S = (\_, \{a, b\}, \{c\}, \{d\}, \_, \_, \_)$ and $b_{cio} = \{(a, c), (b, a), (d, b)\}$ and an iteration count of 3. See figure figure 4.3.

There are two control wires a, b, an input wire d and one output wire d. We will do three iterations. Our set $\mathcal{C}$ of pairs becomes:

$$\{(\phi, a), \ (\phi, b), \ (\phi, c),$$
$$(b, a_1), \ (d, b_1), \ (a, c_1),$$
$$(b_1, a_2), \ (d_1, b_2), \ (a_1, c_2),$$
$$(a_2, \phi), \ (b_2, \phi), \ (c_2, \phi)\}.$$

Translating each of these to equations and constraints, we get:

$$a :: (a, C), b :: (b, C), c :: (c, I)$$

$$b = a_1, b_1 = d \ b_1 :: O, a = c_1 \ c_1 :: I$$

$$b_1 = a_2, b_2 = d_1 \ b_2 :: O, a_1 = c_2 \ c_2 :: I$$

$$a_2 :: (C, a_2), b_2 :: (C, b_2), d_2 :: (O, d_2)$$

As we have three wires in and three wires out, we expect to see six separate groupings in the solution to the above equations. We will proceed by starting from the top left.

start: a $a = c_1, :: I, :: (a, C)$ $\implies a :: (a, I)$

start: b $b = a_1, a_1 = c_2, :: I, :: (b, C)$ $\implies b :: (b, I)$

start: c $:: (c, I)$ $\implies c :: (c, I)$

start: b_1 $b_1 = d, b_1 = a_2, :: O, :: (C, a_2)$ $\implies a_2 :: (O, a_2)$

start: b_2 $b_2 = d_1, :: O, :: (C, b_2)$ $\implies b_2 :: (O, b_2)$

start: d_2 $:: (O, d_2)$ $\implies d_2 :: (O, d_2)$

Bibliography

[1] Charles H. Bennet. Logical reversibility of computation. *IBM Journal of Research and Development*, 6:525–532, 1973.

[2] J.R.B. Cockett, Xiuzhan Guo, and Pieter Hofstra. Range categories ii: Towards regularity. Submitted for Publication, June 2012.

[3] David Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London Ser. A*, A425:73–90, 1989.

[4] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 2000. ISBN 0 521 63235 8.

[5] Robin Cockett and Stephen Lack. Restriction categories I: categories of partial maps. *Theoretical Computer Science*, 270:223–259, 2002.

[6] Robin Cockett and Stephen Lack. Restriction categories II: Partial map classification. *Theoretical Computer Science*, 294:61–102, 2003.

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

[8] Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54:147, 1995.