UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

# Programming languages characterizing quantum efficiency

## THÈSE

présentée et soutenue publiquement le 7 juillet 2025

pour l'obtention du

**Doctorat de l'Université de Lorraine**

**(mention informatique)**

par

Mário Alberto Machado da Silva

**Composition du jury**

| | | |
|---|---|---|
| *Président :* | Benoît Valiron | LMF, Université Paris-Saclay |
| *Rapporteurs :* | Ugo Dal Lago | Università di Bologna |
| | Benoît Valiron | LMF, Université Paris-Saclay |
| *Examinateurs :* | Gilles Barthe | MPI-SS, IMDEA Software Institute |
| | Cristina Sernadas | IST, Universidade de Lisboa |
| *Directeurs :* | Emmanuel Hainry | Inria, Loria, Université de Lorraine |
| | Romain Péchoux | Inria, Loria, Université de Lorraine |

**Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503**

# Acknowledgments

There is a quote by the American-British writer Henry James, in a letter to his nephew who wanted advice on becoming a better writer, which I have kept close to me ever since I read it:

> *There is only one recipe – to care a great deal for the cookery.*

I interpreted this sentence as an *if and only if* statement. A recipe exists, and it is always the same – to care about what we are doing, in the moment we are doing it. In caring about our work we improve it, in caring about our friendships we become better friends to those around us. Not automatically, but eventually. For this reason, I am grateful to those around me who have shown me what it looks like to care about being a researcher, a teacher, and a friend.

This work has been my initiation into becoming a real computer scientist, and nearly all that I have learned so far I owe to the help of my team at Loria and my advisors, Romain and Emmanuel. Their patience and eagerness in helping me develop my skills are the reason why this work was possible. To that extent, Alejandro Díaz-Caro also played a guiding role and for that I am very thankful to him.

I also had the privilege of being a teaching assistant at École des Mines de Nancy for two years. I would like to thank Xavier Goaoc in particular for entrusting me this responsibility and for his guidance throughout the years.

Moving to another country, where I did not (yet) speak the language, was in itself quite the challenge. I had a lot of help from a lot of people, who have become some of my closest friends. These people are too many to list but will know who they are.

Alexandre Guernut deserves a special thank you not only for the camaraderie that he brought to our shared office but also for his interest and help in developing the prototype compiler provided in the last chapter of this thesis.

Finally, I would like to thank Laura, who was been by my side since before the thesis started and who helped me navigate the ups and downs of a life spent doing research.

*À Laura*

# Introduction (en français)

## 1 Motivation

### 1.1 L'informatique quantique

À l'échelle atomique et subatomique, la matière et la lumière se comportent de manière très inattendue. Par exemple, au début du 20e siècle, les physiciens ont observé qu'ils pouvaient créer des systèmes de particules dont l'état ne pouvait pas être décrit simplement en décrivant l'état de chaque particule séparément. En outre, il semble qu'aucune description d'un état ne permette de prédire tous les résultats de ses mesures, car la mesure d'une propriété physique (par exemple, le spin d'un électron dans une direction donnée) peut affecter le résultat de la mesure d'une autre propriété (par exemple, le spin d'un *autre* électron, dans une *autre* direction, à une vitesse supérieure à celle de la lumière) [GS22].

Le phénomène selon lequel un état quantique ne peut être décrit par des descriptions séparées de ses sous-parties est appelé *intrication quantique*, et le fait que les systèmes ne peuvent être définis par un état unique pour toutes les mesures possibles est un produit de la *superposition quantique*.

Le *calcul quantique* est le domaine qui vise à exploiter les caractéristiques uniques de la mécanique quantique, telles que l'intrication et la superposition, afin d'obtenir un avantage computationnel par rapport au calcul classique. Deux exemples notables sont l'algorithme de Shor [Sho94], qui permet une accélération exponentielle dans le problème de factorisation, et l'algorithme de Grover [Gro96], qui permet une accélération quadratique dans le problème de recherche non structurée. Il convient de noter que l'accélération apportée par l'algorithme de Shor n'est que relative par rapport au meilleur algorithme classique (c'est-à-dire non quantique) actuellement connu, car, à l'heure où nous écrivons ces lignes, il n'existe aucun résultat de séparation entre les programmes quantiques efficaces et leurs équivalents classiques.

Une autre application notable de l'informatique quantique est la simulation des systèmes quantiques eux-mêmes [Fey82], avec des applications en sciences des matériaux et en chimie [DBK+22].

Le concept d'*avantage informatique quantique* désigne le nombre inférieur de ressources nécessaires pour résoudre un problème ou calculer une fonction avec un *ordinateur quantique* par rapport à un ordinateur classique. Pour tenir compte de la quantité de ressources utilisées dans un calcul, il faut envisager différents *modèles de calcul*.

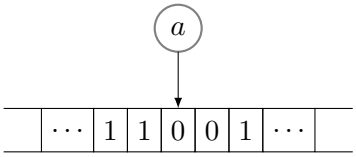
### 1.2 Modèles de calcul quantique

Les modèles de calcul quantique abondent - pour n'en citer que quelques-uns : circuits quantiques [Yao93], machines de Turing quantiques [BV97], ordinateurs quantiques topologiques [FKLW03], calcul quantique optique linéaire [KMN+07], calcul quantique basé sur la mesure

[BBD+09], et calcul quantique adiabatique [AL18].

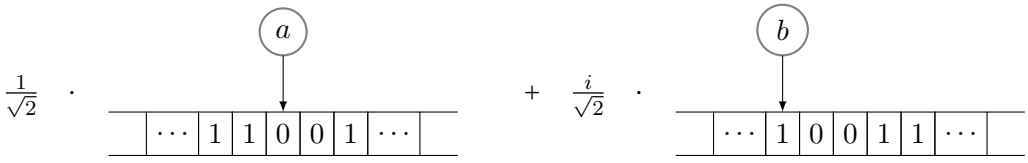
Dans le cadre de ce travail, nous nous concentrerons sur deux modèles, à savoir les machines de Turing quantiques et les circuits quantiques. Une discussion de ces modèles est également utile pour illustrer comment les modèles de bas niveau sont limités dans leurs outils de raisonnement sur l'informatique de manière plus générale.

Une machine de Turing *quantique* peut être définie en considérant les superpositions quantiques des machines de Turing classiques (TM). Un exemple simple de machine de Turing classique consiste en un ruban de longueur infinie divisée en cellules, avec une tête de lecture mobile qui lit ses symboles, et un état interne de la machine. En fonction du symbole lu par la tête de lecture et de l'état interne actuel, la machine réécrit le symbole sur la cellule, fait évoluer son état et déplace éventuellement la tête de lecture vers la gauche ou vers la droite. Par exemple, un état possible d'une machine de Turing peut être représenté comme suit :

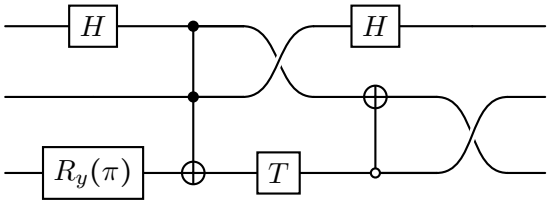



La *fonction de transition* de la machine est son ensemble de règles qui déterminent les mouvements de la tête de lecture, les réécritures de cellules et les changements d'état interne.

Une configuration de machine de Turing quantique (QTM, *quantum Turing machine*) est donnée par une superposition de configurations de machines de Turing, chacune évoluant simultanément selon la fonction de transition de la machine. Par exemple, une configuration d'une QTM possible peut être une combinaison linéaire de deux configurations de machines de Turing classiques :




Comme les QTM, les circuits quantiques peuvent être considérés comme une extension de leurs homologues classiques, à savoir les circuits booléens. Dans les circuits quantiques, les portes sont décrites par des *opérateurs unitaires* qui font évoluer l'état quantique des fils d'entrée. Les circuits se lisent généralement de gauche à droite et sont beaucoup plus faciles à interpréter que la définition d'une QTM, ce qui est l'une des raisons pour lesquelles ils sont devenus le modèle de bas niveau par défaut de l'informatique quantique. Voici un exemple de circuit quantique.




Les circuits constituent un modèle de calcul *non uniforme*, ce qui les distingue des QTM : alors qu'une machine de Turing possède une description unique qui fonctionne pour toute entrée initiale inscrite sur son ruban, un circuit a une taille d'entrée fixe. Dans le contexte des fonctions

dont la taille d'entrée varie, il faut définir une *famille de circuits*, de telle sorte qu'à chaque taille d'entrée corresponde un circuit qui calcule la fonction.

Les propriétés de ces modèles de bas niveau – l'impénétrabilité de la fonction de transition d'une QTM et la non-uniformité des circuits quantiques – contrarient les outils de raisonnement de haut niveau du programmeur quantique, à savoir les abstractions telles que les boucles, les types de données, les procédures récursives ou le flux de contrôle [Sel04]. C'est pourquoi il est nécessaire de disposer de langages de programmation quantique de haut niveau qui permettent au programmeur quantique d'écrire des programmes et de les analyser.

## 1.3 Langages de programmation quantique

Le programmeur quantique dispose aujourd'hui d'un large choix de langages de programmation aux propriétés variées [BBGV20, Dev24, GLR+13, JATK+24, SBZ+24, WS14]. Ces langages abordent des difficultés propres à l'informatique quantique, telles que les exigences de non-clonage et de non-effacement des données quantiques, qui sont des propriétés des systèmes physiques qui finiront par mettre en œuvre ces programmes.

Lorsque nous écrivons des programmes quantiques, nous souhaitons garantir leur réalisabilité physique. Un programme quantique est réalisable physiquement s'il peut être exécuté dans un modèle de calcul quantique. Par exemple, les transformations de qubits doivent correspondre à la description d'un *opérateur unitaire* pour être physiquement réalisables, ce qui empêche le clonage des variables de qubits, ou bien leur effacement. Cela implique un contrôle plus minutieux de la mémoire quantique, car il ne peut y avoir de duplication implicite des variables quantiques dans les instructions du programme.

Le développement d'un langage permettant de raisonner sur la réalisabilité physique des programmes est le premier objectif de cette thèse.

> **Objectif (i).** Développer un langage de programmation quantique de haut niveau permettant de raisonner sur la réalisabilité physique de ses programmes, et assurer une mise en œuvre correcte dans un modèle de bas niveau, tel qu'une QTM ou une famille de circuits quantiques.

En outre, et c'est tout aussi important, nous souhaitons garantir la *faisabilité* des programmes, c'est-à-dire le fait qu'ils puissent être mis en œuvre dans des modèles de bas niveau qui utilisent une quantité *raisonnable* de ressources et qui n'évoluent pas trop rapidement en fonction de la taille des données d'entrée. Naturellement, cela nécessitera que le langage soit restreint afin de garantir que les programmes terminent et qu'ils le fassent avec des limites de ressources raisonnables.

Pour s'assurer que le langage restreint n'est pas trivial, nous veillerons également à ce que les restrictions ne soient pas trop fortes, ce qui signifie que l'ensemble des programmes qui satisfont aux restrictions sont capables d'exprimer *n'importe quel* programme efficace. Ceci constitue notre deuxième objectif.

> **Objectif (ii).** Fournir des restrictions qui rendent le langage de programmation *cohérent* et *complet* pour les programmes quantiques efficaces.

Pour comprendre non seulement comment on peut formellement analyser l'utilisation des ressources d'un modèle de calcul, mais aussi comment on peut démontrer qu'un langage de programmation est cohérent et complet pour les programmes efficaces, nous nous tournons maintenant vers la théorie de la *complexité informatique*.

## 1.4 Complexité informatique

La quantité de ressources nécessaires pour effectuer un calcul, c'est-à-dire sa *complexité*, est généralement définie conformément à un modèle choisi. Les mesures de complexité peuvent généralement être séparées entre l'utilisation du temps et de l'espace, où la *complexité temporelle* consiste en la quantité d'étapes nécessaires pour effectuer un calcul, et la *complexité spatiale* est la quantité de mémoire requise.

La complexité temporelle d'une QTM correspond au nombre de fois où la fonction de transition est appliquée. Sa complexité spatiale est définie par le nombre de cellules du ruban infini qui sont effectivement utilisées (c'est-à-dire qui sont visitées par la tête de lecture). Dans les deux cas, on considère l'utilisation maximale des ressources sur toutes les branches du calcul.

Dans le cas des circuits quantiques, les ressources sont données par la taille et la profondeur du circuit. Sa *taille* est simplement le nombre total de portes, et sa *profondeur* est donnée par le nombre d'étapes nécessaires pour appliquer toutes les portes du circuit, étant donné que les portes peuvent être appliquées en parallèle lorsqu'elles agissent sur des ensembles de fils disjoints.

Après avoir examiné la façon dont l'utilisation des ressources est définie dans les modèles de QTM et de circuits quantiques, nous examinons maintenant les effets de la limitation de la quantité de ressources que les modèles peuvent utiliser pour résoudre un problème donné.

Dans le cas des programmes *efficaces* ou *faisables*, l'approche standard consiste à limiter la complexité temporelle de la machine de manière polynomiale. Dans le cas quantique, cela correspond à la classe FBQP, qui est définie comme l'ensemble des fonctions qui peuvent être approximées par une QTM dont le temps d'exécution est borné de façon polynomiale sur la taille de l'entrée [BV97]. On peut également imposer une restriction encore plus forte de *temps polylogarithmique*, et obtenir la classe FBQPOLYLOG [Yam22].

Récemment, il y a eu des caractérisations de ces classes (ou de leur équivalent pour les problèmes de décision) qui n'utilisent pas de modèle de machine sous-jacent [DMZ10, Yam20, Yam22]. C'est le domaine de la *complexité computationnelle implicite*, qui traite de l'analyse des classes de complexité non pas du point de vue d'un modèle de machine avec des ressources limitées, mais plutôt en plaçant des restrictions sur la construction de programmes [DL12]. L'objectif (ii) peut alors être satisfait en montrant que nos langages restreints sont des *caractérisations implicites* de FBQP et FBQPOLYLOG.

Ces caractérisations sont non seulement intéressantes d'un point de vue théorique, mais elles peuvent également fournir des outils utiles pour l'analyse des programmes. Cependant, dans la mesure où les restrictions placées sur les caractérisations sont trop fortes ou généralement peu pratiques, elles réduisent l'expressivité du langage et deviennent moins intéressantes pour le programmeur moyen. C'est la raison pour laquelle nous énonçons également l'objectif suivant.

> **Objectif (iii).** S'assurer que le langage est capable d'exprimer facilement des programmes quantiques efficaces standard.

Parmi les exemples de programmes standard figurent la transformée de Fourier quantique (utilisée comme sous-programme de l'algorithme de Shor [Sho94]), les algorithmes de recherche binaire quantique [YC22], et d'autres applications telles que les fonctions arithmétiques quantiques [Dra00, RPGE17].

Bien qu'il soit important que le langage soit expressif, sa faisabilité devrait également impliquer qu'il puisse être *compilé efficacement* dans des circuits quantiques afin d'être mis en œuvre dans de véritables ordinateurs quantiques. Il a été démontré que la classe de fonctions FBQP peut être définie de manière équivalente comme l'ensemble des fonctions qui peuvent être approximées

par des familles uniformes de circuits dont les tailles sont bornées par un polynôme en la taille de l'entrée [Yao93].

Une *famille de circuits* signifie que nous considérons un circuit par taille d'entrée, et *uniformité* est une condition sur la façon dont les circuits peuvent être construits [Ruz81]. Il s'agit de tenir compte du fait que des problèmes très complexes (et même intractables) peuvent être résolus par de très petits circuits, et une condition est donc imposée pour que ces circuits ne soient pas trop complexes à trouver, ce qui signifie qu'ils peuvent être obtenus en un temps polynomial.

Les langages qui sont *cohérents* pour le temps polynômial quantique ou le temps polylog peuvent également *garantir l'existence* d'une famille uniforme de circuits efficaces qui mettent en œuvre leurs programmes. Cependant, il n'est pas clair comment on peut obtenir de tels circuits directement à partir du langage. Ceci motive notre quatrième objectif.

> **Objectif (iv).** Garantir la faisabilité des programmes via une stratégie de compilation directe et efficace des programmes en familles uniformes de circuits quantiques.

La difficulté de fournir une telle stratégie de compilation peut être révélée par le problème du *contrôle quantique* qui est utilisé dans certains langages et caractérisations implicites, car il résulte généralement en un écart entre l'*analyse de complexité de haut niveau* d'un programme et la complexité de son circuit.

L'exemple le plus général de contrôle quantique est l'instruction de commutation quantique, de la forme

$$\textbf{qcase } \mathrm{q}_1, \ldots, \mathrm{q}_k \textbf{ of } \{i \to \mathrm{S}_i\}$$

où, selon l'état $i$ des qubits $\mathrm{q}_1, \ldots, \mathrm{q}_k$ (c'est-à-dire, $2^k$ états possibles), l'instruction $\mathrm{S}_i$ est exécutée. Cette instruction ne mesure aucun des qubits, mais exécute les instructions respectives en superposition.

Contrairement au cas du **if** classique, toutes les instructions $\mathrm{S}_i$ doivent être compilées afin d'effectuer cette transformation, et les stratégies de compilation actuelles aboutissent à des circuits dont la complexité correspond à la somme des complexités de chaque $\mathrm{S}_i$ et non au maximum [LS01, STY+23, ZLY22]. C'est pourquoi l'utilisation récursive du cas du commutateur quantique peut conduire à une explosion exponentielle de la complexité du circuit en fonction de la manière dont le programmeur structure le code, un problème qui a été nommé *branch sequentialization* (séquentialisation des branches) dans la littérature [YC22].

Non seulement nous aimerions éviter l'explosion exponentielle qui peut résulter de l'utilisation de la déclaration **qcase**, mais nous aimerions également ne pas altérer la complexité du programme du point de vue du programmeur. En d'autres termes, la stratégie de compilation doit garantir que la complexité de la mise en œuvre de **qcase** est le maximum entre les branches et non la somme.

> **Objectif (v).** Fournir une stratégie de compilation où la complexité du circuit correspond à la complexité de la sémantique du programme.

Enfin, nous aimerions mettre en œuvre tous les objectifs susmentionnés dans un compilateur réel, et nous arrivons ainsi à notre objectif final.

> **Objectif (vi).** Créer un compilateur qui met en œuvre les objectifs de cette thèse.

Nous décrivons maintenant nos contributions à ces objectifs.

## 2 Contributions

Cette thèse contient les contributions suivantes.

- Nous présentons FOQ, un langage de programmation quantique de premier ordre avec des conditionnelles classiques, des branchements quantiques et des appels de procédure. Nous montrons que les programmes FOQ qui terminent peuvent être simulés par des machines de Turing quantiques. Ceci vise l'objectif (i).

- Nous introduisons deux restrictions sur les programmes FOQ, l'une de *bonne fondation*, qui assure la terminaison, l'autre de *largeur bornée*, qui empêche la duplication récursive des appels de procédure. Ces deux restrictions définissent le fragment PFOQ, dont nous démontrons qu'il caractérise implicitement FBQP. Nous montrons que PFOQ satisfait certaines propriétés souhaitables : l'inverse d'un programme peut être obtenu statiquement, l'inverse constitue également un programme PFOQ, et des programmes PFOQ plus grands peuvent être construits à partir de programmes plus petits de manière intuitive. Ceci résout partiellement l'objectif (ii).

- Nous introduisons une restriction supplémentaire, à savoir une restriction qui impose une réduction exponentielle de la quantité de qubits disponibles, en définissant des programmes *recursivement-halving*, et nous obtenons le fragment LFOQ. Nous montrons que LFOQ est cohérent et complet pour le temps polylogarithmique quantique, et qu'il caractérise implicitement la classe FBQPOLYLOG, et nous montrons également qu'il satisfait à la clôture sur l'inverse et à la facilité de construction de programmes plus grands. Ceci complète notre traitement de l'objectif (ii).

- Nous fournissons des exemples pertinents de programmes PFOQ et LFOQ, qui témoignent de l'expressivité de ces fragments. Par exemple, nous montrons des programmes PFOQ pour la transformée de Fourier quantique et son inverse, l'addition et la multiplication quantiques, la détection de langages réguliers, et le calcul du poids de Hamming. Dans le cas de LFOQ, nous donnons des programmes pour la recherche binaire et le comptage trié. Cela répond à notre objectif (iii).

- Nous montrons que les programmes PFOQ peuvent être systématiquement compilés en familles de circuits de taille polynomiale, en développant une nouvelle technique de compilation appelée *ancrage et fusion*, où les appels de procédures orthogonales sont simplifiés pour réduire la complexité asymptotique de la compilation ; un résultat similaire est obtenu pour LFOQ, où les circuits obtenus ont une profondeur polylogarithmique. Ce travail vise l'objectif (iv).

- Nous présentons une technique de compilation pour un fragment de PFOQ, noté $\text{PFOQ}^{\text{UNIF}}$ et qui est encore complet pour le temps polynômial quantique, avec laquelle la complexité du branchement quantique est le maximum des complexités des branches et non leur somme, comme dans les techniques existantes de compilation. Ceci achève notre objectif (v).

- Nous fournissons une implémentation des algorithmes décrits dans cette thèse sous la forme d'un compilateur vers Qiskit. Cette implémentation est écrite en Python et nous permet de vérifier empiriquement les résultats de cette thèse. Ceci remplit l'objectif (vi).

## 3 Plan de thèse

La thèse est organisée de la manière suivante. La partie I introduit les préliminaires nécessaires au travail présenté dans les parties suivantes de la thèse.

- Le Chapitre 1 présente les postulats de la mécanique quantique, ainsi que la notation et certaines notions de base nécessaires pour raisonner sur l'informatique quantique.
- Le Chapitre 2 examine trois modèles différents de calcul quantique : les machines de Turing quantiques (y compris l'accès aléatoire quantique), les circuits quantiques et les langages de programmation quantiques. Nous abordons leurs différences et leur utilité relative.
- Le Chapitre 3 décrit certaines des classes de complexité qui peuvent être définies à l'aide des trois modèles ; plus particulièrement, les définitions de FBQP et FBQPOLYLOG sont données.

La Partie II présente deux nouvelles caractérisations implicites des classes de complexité quantique en tant que restrictions différentes appliquées au même langage de programmation.

- Le Chapitre 4 présente un langage de programmation quantique de premier ordre (FOQ), composé d'opérations unitaires, de séquences, de conditionnelles classiques et quantiques, et d'appels de procédures. Nous définissons la syntaxe et la sémantique de FOQ, et donnons quelques exemples de programmes FOQ simulant l'additionneur quantique ripple-carry (QRCA), l'algorithme de recherche quantique de Grover, ainsi que la transformée de Fourier quantique (QFT) et son inverse.
- Le Chapitre 5 présente un fragment de FOQ caractérisant le temps polynomial quantique. Ce fragment, noté PFOQ (Polytime FOQ), est défini à l'aide de deux restrictions syntaxiquement vérifiables, l'une qui impose la réduction du nombre de qubits accessibles pour chaque appel récursif (la condition *well-foundedness*) et une restriction qui limite le nombre d'appels récursifs effectués en séquence (la condition *bounded width*). L'exhaustivité de la caractérisation est donnée par une simulation d'une algèbre de fonctions caractérisant FBQP et sa solidité est démontrée par la construction d'une simulation QTM polytemporelle de n'importe quel programme PFOQ. Nous démontrons l'expressivité du fragment en montrant que le QRCA et la QFT, qui sont des programmes polytemporels, sont capturés par le fragment.
- Le Chapitre 6 considère un sous-ensemble plus restrictif de programmes FOQ – ceux qui s'exécutent en temps *polylogarithmique*. Ce fragment, noté LFOQ, est obtenu à partir de FOQ en combinant la condition de bien-fondé avec une restriction imposant que le nombre de qubits accessibles est *halved* pour chaque appel récursif. Nous donnons comme exemples un programme de recherche binaire et un programme de comptage du nombre d'instances d'une entrée dans une liste triée. La solidité et la complétude de la caractérisation sont obtenues par des simulations (et l'utilisation) de machines de Turing quantiques à accès aléatoire fonctionnant en temps polylogarithmique.

La Partie III traite de la connexion de ces fragments avec le modèle du circuit quantique. Nous remarquons que le raisonnement basé sur le modèle QTM qui nous a permis de conclure à la complexité des programmes n'est pas directement applicable au modèle de circuit, en raison de l'utilisation de l'instruction **qcase**. Lorsque des instructions de contrôle quantique sont utilisées

de manière récursive, cela peut conduire à une augmentation exponentielle de la complexité du circuit lors de l'utilisation de techniques de compilation simples.

- Dans le Chapitre 7, nous présentons une stratégie de compilation, **compile**, utilisant une technique appelée *ancrage et fusion* qui simplifie les appels de procédure récursifs effectués sur des branches **qcase** orthogonales. Nous montrons que, pour tous les programmes PFOQ, cette stratégie de compilation est efficace et aboutit à des familles de circuits dont la taille croît de façon polynomiale avec le nombre de qubits d'entrée, évitant ainsi l'explosion exponentielle d'autres stratégies de compilation. Nous obtenons un résultat similaire pour les programmes LFOQ, où la *profondeur* des circuits obtenus est limitée de manière polylogarithmique.

- Le Chapitre 8 se concentre sur l'amélioration de l'algorithme **compile** du Chapitre 7, en étendant la technique d'ancrage et de fusion aux procédures de différents niveaux récursifs, nous montrons que ce nouvel algorithme de compilation, dénommé **compile**$^+$, peut fournir une accélération polynomiale arbitrairement grande par rapport à **compile**. Nous identifions également un fragment de programmes PFOQ, noté PFOQ$^{\text{UNIF}}$, où l'ancrage et la fusion peuvent être effectués en temps constant, et nous montrons que ce fragment est également cohérent et complet pour le temps polynomial quantique. Nous montrons que PFOQ$^{\text{UNIF}}$ est le premier langage de programmation pour lequel il existe une stratégie de compilation, à savoir **optimize**$^+$, où la complexité du circuit de l'instruction **qcase** est le *maximum* des complexités des branches.

- Le Chapitre 9 présente une implémentation des idées de la Partie III sous la forme d'un compilateur Python vers Qiskit [JATK$^+$24]. Un certain nombre d'exemples sont testés avec le compilateur, et certaines des résultats de la thèse sont ainsi vérifiées empiriquement.

## 4 Publications

Une partie des travaux présentés dans cette thèse, moyennant quelques modifications et améliorations mineures, a été publiée dans les articles suivants :

[HPS23] Emmanuel Hainry, Romain Péchoux, and Mário Silva. **A programming language characterizing quantum polynomial time**. In Orna Kupferman and Pawel Sobocinski, editors, *Foundations of Software Science and Computation Structures*, pages 156–175, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-30829-1_8. Correspond aux Chapitres 4, 5 and 7.

[HPS25] Emmanuel Hainry, Romain Péchoux, and Mário Silva. **Branch sequentialization in quantum polytime**. In Maribel Fernández, editor, *Formal Structures for Computation and Deduction*, pages 22:1–22:22, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2025.22. Correspond au Chapitre 8.

[FHPS25] Florent Ferrari, Emmanuel Hainry, Romain Péchoux, and Mário Silva. **Quantum programming in polylogarithmic time**. In Paweł Gawrychowski, Filip Mazowiecki, and Michał Skrzypczak, editors, *Mathematical Foundations of Computer Science*, pages 47:1 – 47:17, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2025.47. Correspond aux Chapitres 6 and 7.

Au cours de mon doctorat, j'ai également participé aux travaux suivants, qui ne relèvent pas du sujet de cette thèse :

[SFMZC23] Mário Silva, Ricardo Faleiro, Paulo Mateus, and Emmanuel Zambrini Cruzeiro. **A coherence-witnessing game and applications to semi-device-independent quantum key distribution**. Quantum, 7:1090, August 2023. doi:10.22331/q-2023-08-22-1090.

# Introduction

## 1 Motivation

### 1.1 Quantum computing

At the atomic and subatomic scales, matter and light behave in very counterintuitive ways. For example, during the beginning of the 20th century, physicists observed that they could craft systems of particules where the state of the entire system could not be described by simply describing the state of each particle separately. Furthermore, it seemed that no description of a state could help predict all its measurement outcomes, as performing a measurement of one physical property (for instance, the *spin* of one electron in a given direction) could affect the outcome of measuring another property (for instance, the spin of *another* electron, in *another* direction, faster than the speed of light) [GS22].

The phenomenon where a quantum state cannot be described via separate descriptions of its subparts is called *quantum entanglement*, and the fact that systems cannot be defined by a single state for all possible measurements is a product of *quantum superposition*.

*Quantum computing* is the field that aims at leveraging the unique characteristics of quantum mechanics, such as entanglement and superposition, in order to obtain a computational advantage over classical computation. Two notable examples are Shor's algorithm [Sho94], which provides an exponential speedup in the problem of factoring, and Grover's algorithm [Gro96], allowing for a quadratic speedup in the problem of unstructured search. One should note that the speedup given by Shor's algorithm is only relative to the best currently known classical (i.e. non-quantum) algorithm as, at the time of writing, there is no separation result between quantum efficient programs and their classical counterparts.

Another application of quantum computing of note is the simulation of quantum systems themselves [Fey82], with applications in material sciences and chemistry [DBK+22].

The concept of *quantum computational advantage* denotes the lower number of resources necessary to solve a problem or compute a function with a *quantum computer* as opposed to a classical one. In order to account for the number of resources used in a computation, one must consider different *models of computation*.

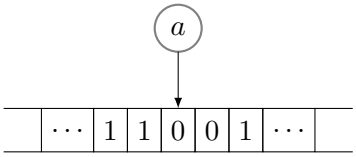
### 1.2 Models of quantum computation

Models of quantum computation abound – to name a few: quantum circuits [Yao93], quantum Turing machines [BV97], topological quantum computers [FKLW03], linear optical quantum computing [KMN+07], measurement-based quantum computing [BBD+09], and adiabatic quantum computing [AL18].

For the purposes of this work, we will be focusing on two models, namely quantum Turing machines and quantum circuits. A discussion of these models is also useful to illustrate how low-

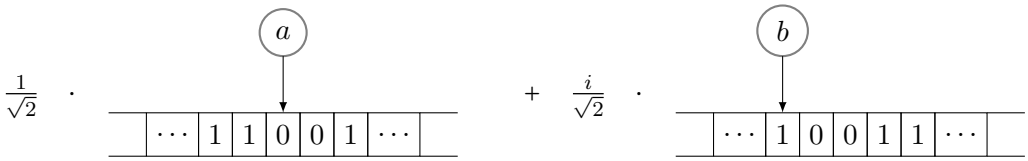
level approaches to quantum computing are limited in their tools for reasoning about computation more generally.

A *quantum* Turing machine can be defined by considering quantum superpositions of classical Turing machines (TMs). A simple example of a classical TM is an infinitely-long tape divided into cells, with a moving tape head that reads its symbols, and an inner machine state. According to the symbol read by the tape head and the current inner state, the machine rewrites the symbol on the cell, evolves its state, and potentially moves its tape head left or right. For instance, a possible state of a Turing machine can be represented as:

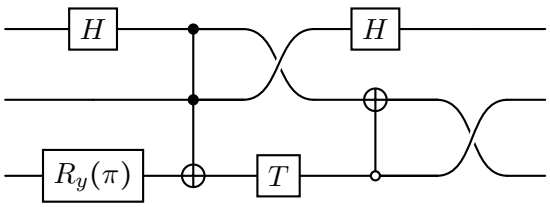



The machine's *transition function* is its set of rules that determine the tape head movements, cell rewrites, and inner state changes.

A quantum Turing machine (QTM) configuration is given by a superposition of Turing machine configurations, each evolving simultaneously according the the machine's transition function. For instance, a possible QTM configuration can be a linear combination of two classical Turing machine configurations:




Unlike QTMs, quantum circuits are *not* defined as superpositions of classical circuits (i.e. Boolean circuits), but rather as circuits consisting of *quantum gates*. These gates correspond to *unitary operators* that evolve the quantum state of the input wires. Circuits are usually read from left to right, and they are considerably easier to interpret, compared to a QTM's transition function, which is one of the reasons why they have become the default low-level model in quantum computing. An example of a quantum circuit is the following:




Circuits are a *non-uniform* model of computation, which further distinguishes them from QTMs: whereas a Turing machine has a single description that works for any initial input written on its tape, a circuit has a fixed input size. In the context of functions with varying input size, one must define a *family of circuits*, such that each input size has a corresponding circuit that computes the function.

The properties of these low-level models – the inscrutability of a QTM's transition function, and the non-uniformity of quantum circuits – do not encourage the high-level reasoning tools of the quantum programmer, namely abstractions such as loops, data types, recursive procedures, or control flow [Sel04]. For this reason, there is a need for high-level quantum programming languages that allow for the quantum programmer write programs and to reason about them.

## 1.3 Quantum programming languages

The quantum programmer has at their disposal a large choice of programming languages with a variety of properties [BBGV20, Dev24, GLR+13, JATK+24, SBZ+24, WS14]. These languages tackle difficulties that are unique to quantum computing, such as the no-cloning and no-erasure requirements of quantum data, which are properties of the physical systems that will eventually implement these programs.

When writing quantum programs, we are interested in guaranteeing their physical realizability. A quantum program is *physically realizable* if it can be performed in a quantum computational model. For instance, qubit transformations must fit the description of a *unitary operator* in order for them to be physically realizable, which prevents cloning of qubit variables, or even their erasure. This entails a more careful control of quantum memory, as there can be no implicit duplication of quantum variables in the instructions of the program.

Developing a language that allows for reasoning about physical realizability of programs is the first objective of this thesis.

> **Objective (i).** Develop a high-level quantum programming language allowing for reasoning about the physical realizability of its programs, and ensure correct implementation in a low-level model, such as a QTM or a family of quantum circuits.

Furthermore, and equally important, we will be interested in ensuring the *feasibility* of programs, that is to say, the fact that they can be implemented into low-level models which make use of a *reasonable* amount of resources, that does not *scale too rapidly* with the size of the input. Naturally, this will require that the language be restricted so as to ensure that programs terminate and that they do so with reasonable resource bounds.

To ensure that the restricted language is not trivial, we will also be concerned with ensuring that the restrictions are not too strong, meaning that the set of programs that satisfy the restrictions are capable of expressing *any* efficient program. This constitutes our second objective.

> **Objective (ii).** Provide restrictions that make the programming language *sound* and *complete* for efficient quantum programs.

To understand not only how one can formally analyze the resource usage of a computational model, but also how a programming language can be shown to be sound and complete for efficient programs, we now turn to the theory of *computational complexity*.

## 1.4 Computational complexity

The amount of resources necessary to perform a computation, that is to say, its *complexity*, is generally defined in accordance with a chosen model. Complexity measures can usually be separated between use of time and space, where *time complexity* consists in the amount of steps necessary to perform a computation, and *space complexity* is the required amount of memory.

A QTM's time complexity corresponds to how many times the transition function is applied. Its space complexity is defined by the number of cells in the infinite tape are actually used (meaning that they are visited by the tape head). In both these cases, one considers the *maximum* use of resources over all branches of computation.

In the case of quantum circuits, the resources are given by the size and depth of the circuit. Its *size* is simply the total number of gates, and its *depth* is given by the number of steps required to apply all the gates in the circuit, given that gates can be applied in parallel when they act on disjoint sets of wires.

Having considered how resource usage is defined in the QTM and quantum circuit models, we now consider the effects of bounding the amount of resources that the models can use in solving a given problem.

In the case of *efficient* or *feasible* programs, the standard approach is to bound the time complexity of the machine polynomially. In the quantum case, this corresponds to the class FBQP, which is defined as the set of functions that can be approximated by a QTM whose runtime is bounded polynomially on the size of the input [BV97]. One can also place an even stronger restriction of *polylogarithmic time*, and obtain the class FBQPOLYLOG [Yam22].

More recently, there have been characterizations of these classes (or their decision-problem equivalent) that do not make use of an underlying machine model [DMZ10, Yam20, Yam22]. This is the field of *implicit computational complexity*, which deals with the analysis of complexity classes not from the point-of-view of a machine model with bounded resources but rather by the placing of restrictions on the construction of programs [DL12]. Objective (ii) can then be satisfied by showing that our restricted languages are *implicit characterizations* of FBQP and FBQPOLYLOG.

These characterizations are not only interesting from a theoretical perspective, but also have the potential to provide useful tools for program analysis. However, insofar as the restrictions placed on the characterizations are too strong or generally impractical, they reduce the expressivity of the language and become less interesting to the typical programmer. For that reason, we also state the following objective.

> **Objective (iii).** Ensure that the language is capable of easily expressing standard efficient quantum programs.

Examples of standard programs include the quantum Fourier transform (used as a subroutine of Shor's algorithm [Sho94]), quantum binary search algorithms [YC22], and other applications such as quantum arithmetic functions [Dra00, RPGE17].

While it is important that the language be expressive, its feasibility should also entail that it can be *efficiently compiled* into quantum circuits so it can be implemented in real quantum computers. The class of functions FBQP has been shown to be equivalently defined as the set of functions that can be approximated by uniform families of circuits whose sizes are bounded polynomially on the input size [Yao93].

A *family of circuits* means that we consider one circuit per input size, and *uniformity* is a condition on how the circuits may be built [Ruz81]. This is to deal with the fact that very complex (and even intractable) problems can be solved by very small circuits, and so a condition is imposed that these circuits must not be too complex to find, meaning that they can be obtained in polynomial time.

Languages that are *sound* for quantum polytime or polylogtime can also *guarantee the existence* of uniform family of efficient circuits that implement their programs. However, it is not clear how one can obtain such cirucits directly from the language. This motivates our fourth objective.

> **Objective (iv).** Guarantee feasibility of programs via a direct and efficient compilation strategy from programs into uniform families of quantum circuits.

The difficuly of providing such a compilation strategy can be found in the problem of *quantum control* which is used in certain languages and implicit characterizations, as it usually results in a gap between the *high-level complexity analysis* of a program and its circuit complexity.

The most general example of quantum control is the quantum switch statement, of the form

$$\textbf{qcase } \mathrm{q}_1, \ldots, \mathrm{q}_k \textbf{ of } \{i \to \mathrm{S}_i\}$$

where, according to the state $i$ of qubits $\mathrm{q}_1, \ldots, \mathrm{q}_k$ (i.e., $2^k$ possible states) the statement $\mathrm{S}_i$ is executed. This instruction does not measure any of the qubits, but rather performs the respective statements in superposition.

Unlike the case of the classical **if**, all statements $\mathrm{S}_i$ must be compiled in order to perform this transformation, and current compilation strategies result in circuits whose complexity scales as the sum of the complexities of each $\mathrm{S}_i$ and not the maximum [LS01, STY+23, ZLY22]. For this reason, the recursive use of the quantum switch case can lead to an *exponential blow-up* in circuit complexity depending on how the programmer structures the code, a problem which has been coined as *branch sequentialization* in the literature [YC22].

Not only would we like to avoid the exponential blow-up that can occur from the use of the **qcase** statement, but we would also like to not alter the complexity of the program from the point-of-view of the programmer. Put differently, the compilation strategy should ensure that the complexity of implementing the **qcase** is the maximum between branches instead of the sum.

> **Objective (v).** Provide a compilation strategy where the circuit complexity matches the complexity of the program's semantics.

Finally, we would like to implement all the aforementioned goals in an actual compiler, and so we arrive at our final objective.

> **Objective (vi).** Create a compiler that implements the goals of this thesis.

We now describe our contributions towards these objectives.

## 2 Contributions

This thesis contains the following contributions.

- We present FOQ, a first-order quantum programming language with classical conditionals, quantum branching, and procedure calls. We show that terminating FOQ programs can be simulated by quantum Turing machines. This targets objective (i).
- We introduce two restrictions on FOQ programs, one for *well-foundedness*, that ensures termination, and one for *bounded width*, that prevents recursive duplication of procedure calls. These two restrictions define the fragment PFOQ, which we demonstrate implicitly characterizes FBQP. We show that PFOQ satisfies certain desirable properties: a program's inverse can be statically obtained, the inverse also constitutes a PFOQ program, and larger PFOQ programs can be constructed from smaller ones in intuitive ways. This partially solves objective (ii).
- We introduce a further restriction, namely one that enforces an exponential reduction in the amount of available qubits, defining *recursively-halving* programs, and obtain the LFOQ fragment. We show that LFOQ is sound and complete for quantum polylogarithmic time, and that it implicitly characterizes the class FBQPOLYLOG, and we also show that it satisfies closure over the inverse and ease of construction of larger programs. This completes our treatment of objective (ii).

- We provide relevant examples of PFOQ and LFOQ programs, that testify to the expressivity of these fragments. For instance, we show PFOQ programs for the quantum Fourier transform and its inverse, quantum addition and multiplication, detection of regular languages, and Hamming weight checking. For the case of LFOQ, we give programs for binary search and sorted counting. This targets our objective (iii).

- We show that PFOQ programs can be systematically compiled into families of circuits of polynomial size, by developing a new compilation technique denoted *anchoring and merging*, where orthogonal procedure calls are simplified to reduce the asymptotic complexity of the compilation; a similar result is obtained for LFOQ, where the obtained circuits have polylogarithmic depth. This work targets objective (iv).

- We present a compilation technique and a fragment of PFOQ, denoted $\text{PFOQ}^{\text{UNIF}}$, that is still complete for quantum polynomial time, where the complexity of quantum branching is the maximum of the branches, instead of their sum, as in existing compilation techniques. This completes our objective (v).

- We provide a code implementation of the algorithms described in this thesis in a compiler into Qiskit, written in Python, which allows for empirically verifying the results of this work. This targets objective (vi).

## 3 Thesis plan

The thesis is organized in the following way. Part I introduces the necessary preliminaries for the work presented in the subsequent parts of the thesis.

- Chapter 1 introduces the postulates of quantum mechanics, as well as the notation and some basic notions required to reason about quantum computation.

- Chapter 2 discusses three different models of quantum computation: quantum Turing machines (including quantum random access), quantum circuits and quantum programming languages. We address their differences and relative usefulness.

- Chapter 3 describes some of the complexity classes that can be defined using the three models; more notably, definitions of FBQP and FBQPOLYLOG are given.

Part II presents two new implicit characterizations of quantum complexity classes as different restrictions applied to the same programming language.

- Chapter 4 introduces a First-Order Quantum (FOQ) programming language, consisting of unitary operations, sequences, classical and quantum conditionals, and procedures calls. We define FOQ's syntax and semantics, and give some examples of FOQ programs implementing the quantum ripple-carry adder (QRCA), Grover's quantum search algorithm, as well as the quantum Fourier transform (QFT) and its inverse.

- Chapter 5 presents a fragment of FOQ characterizing quantum polynomial time. This fragment, denoted PFOQ (Polytime FOQ), is defined using two restrictions, one that enforces the reduction on the number of accessible qubits for every recursive call (the *well-foundedness* condition) and a restriction on the number of recursive calls performed in sequence (the *bounded width* condition). The *completeness* of the characterization is given by a simulation

of a function algebra characterizing FBQP and its *soundness* is shown via a construction of a polytime-bounded QTM simulation of any PFOQ program. We demonstrate the fragment's expressivity by showing that the QRCA and the QFT, which are polytime programs, are captured by the fragment.

- Chapter 6 considers a more restrictive subset of FOQ programs – those that run in *polylogarithmic* time. This fragment, denoted LFOQ, is obtained from FOQ by combining the well-foundedness condition with a restriction imposing that the number of accessible qubits is *halved* for every recursive call. We provide as examples a program for binary search and a program for counting the number of instances of an entry in a sorted list. The soundness and completeness of the characterization are obtained by simulations of (and using) quantum random-access Turing machines running in polylogarithmic time.

Part III addresses the connection of these fragments with the quantum circuit model of computation. We notice that the reasoning based on the QTM model that allowed us to infer conclusions about the complexity of the programs is not directly applicable to the circuit model, due to the use of the **qcase** statement. When quantum control statements are used recursively, this may lead to an *exponential blow-up* in circuit complexity when using straightforward compilation techniques.

- In Chapter 7, we introduce a compilation strategy, **compile**, using a technique called *anchoring and merging* that simplifies recursive procedure calls performed on orthogonal **qcase** branches. We show that, for all PFOQ programs, this compilation strategy is efficient and results in circuit families with size growing polynomially in the number of input qubits, thereby avoiding the exponential blow-up of other compilation strategies. We obtain a similar result for LFOQ programs, where the *depth* of the obtained circuits is bounded polylogarithmically.

- Chapter 8 focuses on improving the **compile** algorithm of Chapter 7, by extending the anchoring-and-merging technique to procedures of different recursive level, we show that this new compilation algorithm, denoted **compile**$^+$, can provide an arbitrarily-large polynomial speedup over **compile**. We also identify a fragment of PFOQ programs, denoted PFOQ$^{\text{UNIF}}$, where anchoring and merging can be performed in constant time, and show that this fragment is also sound and complete for quantum polytime. We show that PFOQ$^{\text{UNIF}}$ is the first programming language for which there is a compilation strategy, namely **optimize**$^+$, where the circuit complexity of the **qcase** statement is the *maximum* of the complexities of the branches.

- Chapter 9 presents an implementation of the ideas of Part III in the form of a Python compiler into Qiskit. A number of examples are tested with the compiler, and some of the results in the thesis are empirically verified.

## 4 Publications

Some of the work presented in this thesis, up to some slight modifications and improvements, has been published, appearing in the following articles:

[HPS23] Emmanuel Hainry, Romain Péchoux, and Mário Silva. **A programming language characterizing quantum polynomial time**. In Orna Kupferman and Pawel Sobocinski, editors, *Foundations of Software Science and Computation Structures*, pages 156–175, Cham,

2023. Springer Nature Switzerland. doi:10.1007/978-3-031-30829-1_8. Corresponds to Chapters 4, 5 and 7.

[HPS25] Emmanuel Hainry, Romain Péchoux, and Mário Silva. **Branch sequentialization in quantum polytime**. In Maribel Fernández, editor, *Formal Structures for Computation and Deduction*, pages 22:1–22:22, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2025.22. Corresponds to Chapter 8.

[FHPS25] Florent Ferrari, Emmanuel Hainry, Romain Péchoux, and Mário Silva. **Quantum programming in polylogarithmic time**. In Paweł Gawrychowski, Filip Mazowiecki, and Michał Skrzypczak, editors, *Mathematical Foundations of Computer Science*, pages 47:1 – 47:17, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2025.47. Corresponds to Chapters 6 and 7.

During the course of the PhD, I was also involved in the following work, which falls outside of the scope of this thesis:

[SFMZC23] Mário Silva, Ricardo Faleiro, Paulo Mateus, and Emmanuel Zambrini Cruzeiro. **A coherence-witnessing game and applications to semi-device-independent quantum key distribution**. Quantum, 7:1090, August 2023. doi:10.22331/q-2023-08-22-1090.

# Contents

# List of Figures

# Part I

# Preliminaries

# 1

# Quantum Mechanics

> Computers are ways of instantiating abstract objects and their relationships in physical objects and their motion.
>
> – David Deutsch

In this chapter, we introduce the basic concepts used for describing quantum systems. We start by discussion standard notions and definitions required to clearly define quantum mechanics, and then move on to detailing its postulates.

## 1.1 Notation and Basic Properties

### 1.1.1 Quantum states

In classical (digital) computation, the smallest unit of information is a *bit*, typically represented as a binary value 0 or 1, or as boolean value **true** or **false**. In quantum computation, the most basic unit of information is a *qubit* (i.e. a quantum bit), which is described as a *superposition* (that is to say, a linear combination) of classical states 0 and 1.

To give a precise definition, let $|0\rangle$ and $|1\rangle$ (*ket zero*, *ket one*) denote the two possible states of a classical bit. A *qubit state* $|\psi\rangle$ (pronounced *ket psi*) is defined as a linear combination

$$\alpha \cdot |0\rangle + \beta \cdot |1\rangle,$$

with $\alpha, \beta \in \mathbb{C}$ being complex numbers, typically called *probability amplitudes*. The state can be denoted in vector form as

$$|0\rangle \triangleq \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \qquad |1\rangle \triangleq \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \qquad |\psi\rangle \triangleq \alpha \cdot |0\rangle + \beta \cdot |1\rangle = \alpha \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}, \qquad \mathbf{0} \triangleq \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

A qubit can then be seen as an element in the vector space $\mathbb{C}^{2\times 1}$, which we will write as $\mathbb{C}^2$ for simplicity.

This description of a quantum state is given to us by the first postulate of quantum mechanics.

> **Postulate 1.** The state of an isolated physical system at a fixed time is defined by a *state vector* $|\psi\rangle$ belonging to a Hilbert space $\mathcal{H}$ called the *state space*. The system is completely described by its state vector, which is a unit vector in the state space.

This postulate states that a system's *entire description* is given by $|\psi\rangle$. Of note, is the fact that this a fundamentally different perspective from that of probabilistic classical systems. In those systems, the *observer* may not have access to enough information to know the system's state, but the system must necessarily be in one of the states. Postulate 1 states that the superposition is the *complete description* of the state, which is a remarkably different description.

The postulate makes a reference to the *Hilbert* space in which the qubit is defined, which consists of the space $\mathbb{C}^2$ with a metric induced by an inner product, which we will define once we introduce unitary operators.

Systems comprising multiple qubits are obtained via the *tensor product*, defined over matrices $A$ and $B$ as follows:

$$A \otimes B \triangleq \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}, \quad \text{where } A = (a_{ij})_{1\le i\le m, 1\le j\le n}.$$

The use of the tensor product in defining larger states is described by the second postulate.

> **Postulate 2.** The state space of a composite physical system is the tensor product of the state spaces of the component physical systems. Moreover, if we have systems numbered 1 through $n$, and system number $i$ is prepared in the state $|\psi_i\rangle$, then the joint state of the total system is $|\psi_1\rangle \otimes |\psi_2\rangle \otimes \ldots \otimes |\psi_n\rangle$.

The tensor product satisfies the properties of *associativity* and *bilinearity*:

$$(A \otimes B) \otimes C = A \otimes (B \otimes C),$$
$$(\lambda A) \otimes B = A \otimes (\lambda B) = \lambda(A \otimes B),$$

meaning that we may write $\lambda \cdot A \otimes B \otimes C$ without ambiguity.

As an example, consider two qubit states $|\psi\rangle$ and $|\phi\rangle$, occupying systems 1 and 2 respectively, and defined, for $\alpha, \beta, \gamma, \delta \in \mathbb{C}$, as

$$|\psi\rangle_1 \triangleq \alpha \cdot |0\rangle_1 + \beta \cdot |1\rangle_1 \text{ and}$$
$$|\phi\rangle_2 \triangleq \gamma \cdot |0\rangle_2 + \delta \cdot |1\rangle_2.$$

Then, the description of the global state of both systems is given by

$$|\psi\rangle_1 \otimes |\phi\rangle_2 = \alpha\gamma \cdot |0\rangle_1 \otimes |0\rangle_2 + \alpha\delta \cdot |0\rangle_1 \otimes |1\rangle_2 + \beta\gamma \cdot |1\rangle_1 \otimes |0\rangle_2 + \beta\delta \cdot |1\rangle_1 \otimes |1\rangle_2,$$

where, in column vector notation, we write:

$$|0\rangle_1 \otimes |0\rangle_2 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}_{12} \quad |0\rangle_1 \otimes |1\rangle_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}_{12} \quad |1\rangle_1 \otimes |0\rangle_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}_{12} \quad \text{and} \quad |1\rangle_1 \otimes |1\rangle_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}_{12}$$

For simplicity, one can also define:

$$|00\rangle_{12} \triangleq |0\rangle_1 \otimes |0\rangle_2, \quad |01\rangle_{12} \triangleq |0\rangle_1 \otimes |1\rangle_2, \quad |10\rangle_{12} \triangleq |1\rangle_1 \otimes |0\rangle_2, \quad |11\rangle_{12} \triangleq |1\rangle_1 \otimes |1\rangle_2.$$

When the different systems are clear from context, we drop the subscript notation. For instance, given a binary word $x = x_1 \ldots x_n \in \{0,1\}^n$, we can define the state

$$|x\rangle \triangleq |x_1\rangle \otimes \cdots \otimes |x_n\rangle \in \mathbb{C}^{2^n}$$

with length defined as $\ell(|x\rangle) \triangleq n$.

### 1.1.2 Unitary operators

In order to perform computation using quantum systems, we need not only to be able to represent information in terms of quantum systems, but also *evolve* these systems over time. In quantum mechanics, a system's description is given by a *unitary* operator, which consists of a *norm-preserving* transformation, frequently represented by matrix multiplication.

Given matrices $A \in \mathbb{C}^{n\times k}$, $B \in \mathbb{C}^{k\times m}$, let $AB \in \mathbb{C}^{n\times m}$ denote their matrix multiplication. Postulate 3 describes a quantum state's evolution over time.

> **Postulate 3.** The evolution of a closed quantum system is described by a unitary transformation $U(t_1, t_2) : \mathbb{R}^2 \to \mathcal{H} \to \mathcal{H}$, such that $|\psi(t)\rangle = U(t, t_0)\,|\psi(t_0)\rangle$.

This would be the most general description of the evolution of a quantum state. For the purposes of this work, it will suffice to consider only discrete transformations given by a fixed set of possible operators. One example of such an operator is the *Hadamard*, defined as

$$H \triangleq \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \in \mathbb{C}^{2\times 2}, \quad \text{such that} \quad H \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} \alpha + \beta \\ \alpha - \beta \end{pmatrix}.$$

Once we give a precise definition of norm, we will see that these operators are norm-preserving. This also requires that they are *dimension-preserving*, meaning that they have the same amount of lines and columns. Furthermore, since we will only be considering states comprised of qubits (defined by a basis of dimension 2), a quantum state of length $n$ will be described by a vector in $\mathbb{C}^{2^n}$ and a unitary operator acting on states of this length will be defined by a matrix in $\mathbb{C}^{2^n\times 2^n}$.

The set of classical states of length $n$ is defined as $\{|x\rangle : x \in \{0,1\}^n\}$. This set will constitute an orthonormal basis for quantum states of dimension $2^n$. First, we define an *inner product* $\langle\cdot\,|\,\cdot\rangle$ for any two states $|\phi\rangle, |\psi\rangle \in \mathbb{C}^{2^n}$, given in general by a complex number, and satisfying the following properties:

- $\langle\phi|\psi\rangle = \langle\psi|\phi\rangle^*$, (Skew symmetry)
- $\langle\psi|\psi\rangle \in \mathbb{R}_0^+$ $(= 0$ if and only if $|\psi\rangle = \mathbf{0})$, (Positive semidefiniteness)
- $\langle\phi|(\alpha\cdot|\psi_1\rangle + \beta\cdot|\psi_2\rangle) = \langle\phi|\alpha\cdot\psi_1 + \beta\cdot\psi_2\rangle = \alpha\cdot\langle\phi|\psi_1\rangle + \beta\cdot\langle\phi|\psi_2\rangle$, (Linearity in the ket)

where $(a + b\,i)^* = a - b\,i$, for $a, b \in \mathbb{R}$, is the complex conjugate.

This inner product can be defined in the matrix notation by using the notion of a *conjugate transpose* $\cdot^\dagger$, which consists of transposing a matrix and conjugating its entries:

$$A^\dagger = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}^\dagger \triangleq \begin{pmatrix} a_{11}^* & \cdots & a_{m1}^* \\ \vdots & \ddots & \vdots \\ a_{1n}^* & \cdots & a_{mn}^* \end{pmatrix}.$$

The inner product of $|\psi\rangle = \alpha\cdot|0\rangle + \beta\cdot|1\rangle$ and $|\phi\rangle = \gamma\cdot|0\rangle + \delta\cdot|1\rangle$ is then defined as

$$\langle\psi|\phi\rangle \triangleq |\psi\rangle^\dagger\,|\phi\rangle = \begin{pmatrix} \alpha^* & \beta^* \end{pmatrix} \begin{pmatrix} \gamma \\ \delta \end{pmatrix} = \alpha^*\,\gamma + \beta^*\,\delta,$$

with the case for larger dimensions being defined as:

$$\langle\psi_1 \otimes \psi_2\,|\,\phi_1 \otimes \phi_2\rangle \triangleq \langle\psi_1|\phi_1\rangle \cdot \langle\psi_2|\phi_2\rangle.$$

It can be checked that this definition of the inner product satisfies all the desired properties described above. The conjugate transpose transforms a ket into a *bra*:

$$\langle 0| \triangleq |0\rangle^\dagger = \begin{pmatrix} 1 & 0 \end{pmatrix}, \qquad \langle 1| \triangleq |1\rangle^\dagger = \begin{pmatrix} 0 & 1 \end{pmatrix},$$

which can also be seen as functions in $\mathbb{C}^2 \to \mathbb{C}$ which, for an input state $|\psi\rangle$, give the probability amplitude in the corresponding state:

$$\text{if} \quad |\psi\rangle = \alpha \cdot |0\rangle + \beta \cdot |1\rangle, \quad \text{then} \quad \langle 0|\psi\rangle = \alpha, \quad \text{and} \quad \langle 1|\psi\rangle = \beta.$$

The *norm* of a state is then given by the square root of its inner product:

$$\||\psi\rangle\| \triangleq \sqrt{\langle\psi|\psi\rangle} = \sqrt{\alpha^*\alpha + \beta^*\beta} = \sqrt{|\alpha|^2 + |\beta|^2},$$

which, from the fact that the inner product is positive semidefinite, is well defined. We are now able to define the *Hilbert space* of dimension $2^n$, denoted $\mathcal{H}_{2^n}$, as the space spanned by the basis $\{|x\rangle : x \in \{0,1\}^n\}$ and including the inner product. We define the Hilbert space of arbitrary dimension as the union $\mathcal{H} \triangleq \bigcup_{n\in\mathbb{N}} \mathcal{H}_{2^n}$.

A unitary transformation, as described by Postulate 3, is defined such that it preserves the norm of the state. This means that, given an initial state $|\psi\rangle$, the state obtained via a transformation $U$, must satisfy:

$$\||U\,\psi\rangle\| = \sqrt{\langle U\,\psi|U\,\psi\rangle} = \sqrt{\langle\psi|U^\dagger U|\psi\rangle} = \sqrt{\langle\psi|\psi\rangle} = \||\psi\rangle\|.$$

Given that this property must be satisfied over all possible states $|\psi\rangle$, unitary operators can be defined as being those that satisfy

$$U^\dagger U = U\,U^\dagger = \mathbb{1}$$

where $\mathbb{1}$ is the identity matrix of appropriate dimension.

An example of a unitary operator is the Hadamard gate defined earlier. In this case, this gate is its own inverse, i.e. $H^{-1} = H^\dagger = H$. Another example is the CNOT gate, defined as

$$\text{CNOT} : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = |00\rangle\langle 00| + |01\rangle\langle 01| + |10\rangle\langle 11| + |11\rangle\langle 10| \in \mathbb{C}^{4\times 4},$$

which inverses the state of the second qubit whenever the first qubit is in state $|1\rangle$.

If the unitary is only applied on a subpart of the quantum state, this can be represented by a larger operator derived from tensoring the gate with the identity operator of dimension $k$, written $\mathbb{1}_k$. Given an operator $U \in \mathbb{C}^{2^k}$ a size $n$, and an index $1 \le i \le n$, the operator

$$\mathbb{1}_{i-1} \otimes U \otimes \mathbb{1}_{n-i-k+1} \in \mathbb{C}^{2^n \times 2^n}$$

applies the unitary $U$ on the qubits of index $[i, \ldots, i+k-1]$.

A Hilbert space may also be *infinite dimensional*, as in the case of quantum Turing machine configurations (Section 2.1), where the machine tape is infinite.

Postulate 3 described the evolution of *closed* quantum systems. To extract information from a quantum system, however, one needs to perform a *measurement* on the system, which will be described by a new postulate.

### 1.1.3 Measurement

A system's evolution stops being reversible when it is affected upon by an outside system. This is described as a *measurement*, and its behavior is given by the fourth and last postulate.

> **Postulate 4.** Quantum measurements are described by a collection $\{M_m\}$ of measurement operators. These are operators acting on the state space of the system being measured. The index $m$ refers to the measurement outcomes that may occur in the experiment. If the state of the quantum system is $|\psi\rangle$ immediately before the measurement then the probability that result $m$ occurs is given by
>
> $$\text{Prob}(m) = \langle\psi| M_m^\dagger M_m |\psi\rangle, \quad \text{with} \quad \sum_m M_m^\dagger M_m = \mathbb{1}.$$
>
> The system's state immediately after the measurement outcome $m$ is given by
>
> $$\frac{M_m |\psi\rangle}{\sqrt{\langle\psi| M_m^\dagger M_m |\psi\rangle}}.$$

Postulate 4 describes a more general scenario than the one we will consider in this thesis, as we will only be interested by measurements in the *computational basis*, which is the the basis consisting of the classical states of the system. Other measurement basis are given by linear combinations of computational basis vectors.

For instance, if the states $\{|0\rangle, |1\rangle\}$ form a basis of the Hilbert space of states of a given system, then the states $|+\rangle \triangleq \frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $|-\rangle \triangleq \frac{|0\rangle-|1\rangle}{\sqrt{2}}$ also form a basis. Measuring state $|0\rangle$ in the basis $\{|+\rangle, |-\rangle\}$ will give outcome $|+\rangle$ or $|-\rangle$ with uniform probability. If the measurement result is $|+\rangle$ then that is its new state, and the previous superposition is no longer accessible.

The choice of unit norm for quantum states (which is preserved by unitary evolution) ensures that the sum of the probabilities for all measurement outcomes sums to one:

$$\sum_m \text{Prob}(m) = \sum_m \langle\psi| M_m^\dagger M_m |\psi\rangle = \langle\psi| \sum_m M_m^\dagger M_m |\psi\rangle = \||\psi\rangle\|^2 = 1.$$

We will also consider *partial projections* given by a bra whose length is smaller than the projected state. For $y \in \{0,1\}^k$ and $|\psi\rangle \in \mathcal{H}_{2^n}$ such that $n > k$, we define

$$\langle y|\psi\rangle \triangleq \left(\langle y| \otimes \mathbb{1}_{n-k}\right) |\psi\rangle = \sum_{\substack{x \in \{0,1\}^n : \\ x_1 \dots x_k = y}} \langle x|\psi\rangle \cdot |x_{k+1} \dots x_n\rangle.$$

For instance, consider the 3-qubit state $|\psi\rangle = \frac{1}{\sqrt{3}} \cdot \big(|001\rangle + |101\rangle + |110\rangle\big)$. Then, projecting on the first qubit, we obtain

$$\langle 0|\psi\rangle = \frac{1}{\sqrt{3}} \cdot |01\rangle, \qquad \text{and} \qquad \langle 1|\psi\rangle = \frac{1}{\sqrt{3}} \cdot \big(|01\rangle + |10\rangle\big).$$

A quantum state cannot be directly accessed, meaning that, given a state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, there is no way to directly ascertain the probability amplitudes $\alpha$ and $\beta$. In order to obtain information from a quantum system, one must perform a *measurement*, as described by Postulate 4.

A measurement can be seen as a projection onto a certain basis. We will only consider projections onto the classical basis as described above, but this is not required in general. Measuring $|\psi\rangle$ in the classical basis results in outcomes

$$\begin{aligned} &0, \text{ with probability } |\langle 0|\psi\rangle|^2 = |\alpha|^2, \\ &1, \text{ with probability } |\langle 1|\psi\rangle|^2 = |\beta|^2. \end{aligned}$$

Having obtained outcome $i$, the system will now be in state $|i\rangle$. We will also be interested in the outcome of measuring a state only *partially*, for instance, by observing its first few qubits. For instance, given the state $|\psi\rangle$ we gave in the example of partial projections, we have that the probability of observing different values for the first qubit are given by

$$\text{Prob}(0) = \frac{\| \langle 0|\psi\rangle \|}{\||\psi\rangle\|} = \frac{\left|\frac{1}{\sqrt{3}}\right|^2}{1} = \frac{1}{3}, \qquad \text{Prob}(1) = \frac{\| \langle 1|\psi\rangle \|}{\||\psi\rangle\|} = \frac{\left|\frac{1}{\sqrt{3}}\right|^2 + \left|\frac{1}{\sqrt{3}}\right|^2}{1} = \frac{2}{3}.$$

This will prove useful when we consider the outcome of a computation as being given by only a subpart of its outcome. Generally, a quantum computation can be interspersed with partial or total measurements. The *deferred measurement principle* claims that these measurements can always be moved to the end without changing the computation [NC10], and so we can consider only unitary transformations with measurements performed at the end without loss of generality.

### 1.1.4 Entanglement

When considering multi-qubit states, Postulate 2 describes a state comprising all qubits that is created from the tensor product. This is the case because the states of these systems are *known*, or at least guaranteed to be independent from on another.

In general, states of multiple qubits do not admit descriptions where each qubit is described separately, and therefore the state cannot be written as a tensor product of single-qubit states. We call such a state *entangled*, and when the qubit states can be separated and the state can be written as a tensor product we call it *separable*.

Basis states are clearly separable, and a standard example of an entangled state is the *Bell state*:

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} \neq \big(\alpha \cdot |0\rangle + \beta \cdot |1\rangle\big) \otimes \big(\gamma \cdot |0\rangle + \delta \cdot |1\rangle\big),$$

for any values $\alpha, \beta, \gamma, \delta \in \mathbb{C}$. Entangled states are not only a staple of quantum protocols such as *quantum key distribution* [BB14], but they are also essential in the advantage of quantum algorithms [Vid03].

### 1.1.5 Infinite-dimensional Hilbert spaces

So far, we have worked with the definition of a Hilbert space for a finite dimension, in this case the space spanned by a fixed number of qubits. This definition suffices for computational models of finite dimension such as quantum circuits (defined for a fixed number of qubits), but in the case of quantum Turing machines (Section 2.1), the state of the machine is described by a *countably infinite* basis consisting of all classical Turing machine states.

In this case, we build an orthonormal basis consisting of all Turing machine configurations $\gamma$, such that the basis state $|\gamma\rangle$ has norm 1, and given two different configurations $|\gamma\rangle$ and $|\gamma'\rangle$, we have that $\langle\gamma|\gamma'\rangle = 0$. To distinguish this Hilbert space from $\mathcal{H}$ defined from finite sets of qubits, we denote it by $\mathcal{S}$, following the notation of [BV97].

The definition of a unitary operator $U$ in the infinite basis scenario is no longer given by a (finite) matrix, but rather via transition amplitudes $\langle\gamma_i|U|\gamma_j\rangle$. We can define the adjoint $U^\dagger$ of $U$ as the operator satisfying

$$\langle\gamma_i|\,U^\dagger\,|\gamma_j\rangle = \langle\gamma_j|\,U\,|\gamma_i\rangle^*,$$

and the unitarity condition $U^\dagger U = \mathbb{1}$ can be verified for all basis states $\gamma$.

For more background on quantum computation, we direct the reader to [NC10].

# 2

# Models of Quantum Computation

> Computers are ways of instantiating abstract objects and their relationships in physical objects and their motion.
>
> – David Deutsch

Having described the basics of quantum mechanics, we were already able to notice very interesting and unintuitive behaviour, such as entanglement and superpositions. We now turn to describing models of computation based on the laws of quantum mechanics, which will be of use to us in the following chapters.

We start by defining the quantum Turing machine model, which defines the class FBQP of functions that can be efficiently approximated by quantum computers. In this chapter, we discuss the subtle differences between quantum Turing machines and deterministic Turing machines, and in Chapter 5 we give an implicit characterization of FBQP.

We also introduce quantum *random-access* Turing machines which allow for instant access to any cell of the input tape according to an address written on an index tape. This model, described in [YC22, Yam20], allows for avoiding the linear cost incurred by moving the tape head to any designated position, and is at the basis of the definition of FBQPOLYLOG, the polylogarithmic-time version of FBQP, of which we give an implicit characterization in Chapter 6. This model differs from that of the QRAM machine of [Kni96, WY23], which consists of a classical machine controlling operations (i.e. unitaries, measurements) on a quantum memory, and is more akin to the qRAM model of [GLM08, YC22, JR23], where data (quantum or classical) is accessed using memory addresses that can themselves be quantum.

The quantum circuit model is also introduced. This model is of interest not only because it is the most widely-used model for reasoning about quantum computation, but also because it is the closest, among the standard models we consider, to a physical quantum computer implementation. In Part III we focus on strategies that translate programs into quantum circuits of appropriate size and depth for the respective complexity class.

## 2.1 Quantum Turing Machines

The quantum Turing machine model (QTM) was first introduced by David Deutsch [Deu85] as the quantum version of Turing's computational model [Tur37], the Turing machine (TM), which serves as a cornerstone for reasoning about what can be calculated, as well as the relative difficulty of certain types of calculations. Naturally, as the field of quantum computation emerged, different proposals of QTMs were offered. Currently, the standard definition of QTM is the one given in [BV97], which is the following.

**Definition 2.1** ([BV97]). *A $k$-tape QTM, with $k \geq 1$, is defined by a triplet $(\Sigma, Q, \delta)$ where*

- *$\Sigma$ is a finite alphabet including a blank symbol $\#$,*
- *$Q$ is a finite set of states with an initial state $s_0$ and a final state $s_\top \neq s_0$,*
- *$\delta$ is the quantum transition function in $Q \times \Sigma^k \to \mathbb{C}^{Q \times \Sigma^k \times \{L,N,R\}^k}$; $\{L, N, R\}$ being the set of possible movements of a head on a tape.*

*Each tape of the QTM is two-way infinite and contains cells indexed by $\mathbb{Z}$.*

The starting position of the tape heads is the *start cell*, the cell indexed by 0. A configuration $\gamma$ of a $k$-tape QTM is a tuple $(s, \overline{w}, \overline{n})$, where $s$ is a state in $Q$, $\overline{w}$ is a $k$-tuple of words in $\Sigma^*$, and $\overline{n}$ is a $k$-tuple of indexes (head positions) in $\mathbb{Z}$.

An initial (final) configuration $\gamma_{init}$ (resp. $\gamma_{fin}$) is a configuration of the shape $(s_0, \overline{w}, \overline{0})$ (resp. $(s_\top, \overline{w}, \overline{0})$). We use $\gamma(w)$ to denote a configuration $\gamma$ where the word $w$ is written on the input/output tape.

Following [BV97], we write $\mathcal{S}$ to represent the Hilbert space with orthonormal basis given by the infinite set of possible configurations of a TM. A QTM $M$ defines a unitary operator $U_M : \mathcal{S} \to \mathcal{S}$, that outputs a superposition of configurations $\sum_i \alpha_i |\gamma_i\rangle$ (also called a *surface configuration*) obtained by applying a single-step transition of $M$ to a configuration $|\gamma\rangle$ (i.e., $U_M|\gamma\rangle = \sum_i \alpha_i |\gamma_i\rangle$). Let $U_M^t$, for $t \geq 1$, be the $t$-steps transition obtained from $U_M$ as follows:

$$U_M^1 \triangleq U_M \quad \text{and} \quad U_M^{t+1} \triangleq U_M \circ U_M^t.$$

Given a quantum state $|\psi\rangle = \sum_{w \in \{0,1\}^n} \alpha_w |w\rangle$ and a configuration $\gamma$, let $\gamma(|\psi\rangle) \in \mathcal{S}$ be the quantum configuration defined by $\gamma(|\psi\rangle) \triangleq \sum_{w \in \{0,1\}^n} \alpha_w |\gamma(w)\rangle$.

A quantum function $f : \mathcal{H} \to \mathcal{H}$ is computed by the QTM $M$ in time $t$ if for any $|\psi\rangle \in \mathcal{H}$,

$$U_M^t(\gamma_{init}(|\psi\rangle)) = \gamma_{fin}(f(|\psi\rangle)).$$

Given $T : \mathbb{N} \to \mathbb{N}$ and a quantum function $f$, we say that the QTM $M$ computes $f$ in time $T$ if for inputs of length $n$, $M$ computes $f$ in time $T(n)$.

In [BV97], the authors specify that any description of a quantum Turing machine will have to provide a satisfying answer to the following points in which this model necessarily differs from its classical counterpart:

- *Analog transitions.* As in probabilistic TMs, the computational power of QTMs will depend on the set of allowed transition amplitudes. This requirement must be balanced with the *universality* of quantum computation.
- *Superposed termination.* A machine occupying a superposition of its configurations may be partly in a final state (i.e. in a superpositon where some configurations are in a final state and others are not). Our criteria for termination should address this possibility.
- *Reversibility and unitarity.* A QTM's transition function must be described by a unitary (and, in particular, reversible) transformation.

We will discuss each of these points separately.

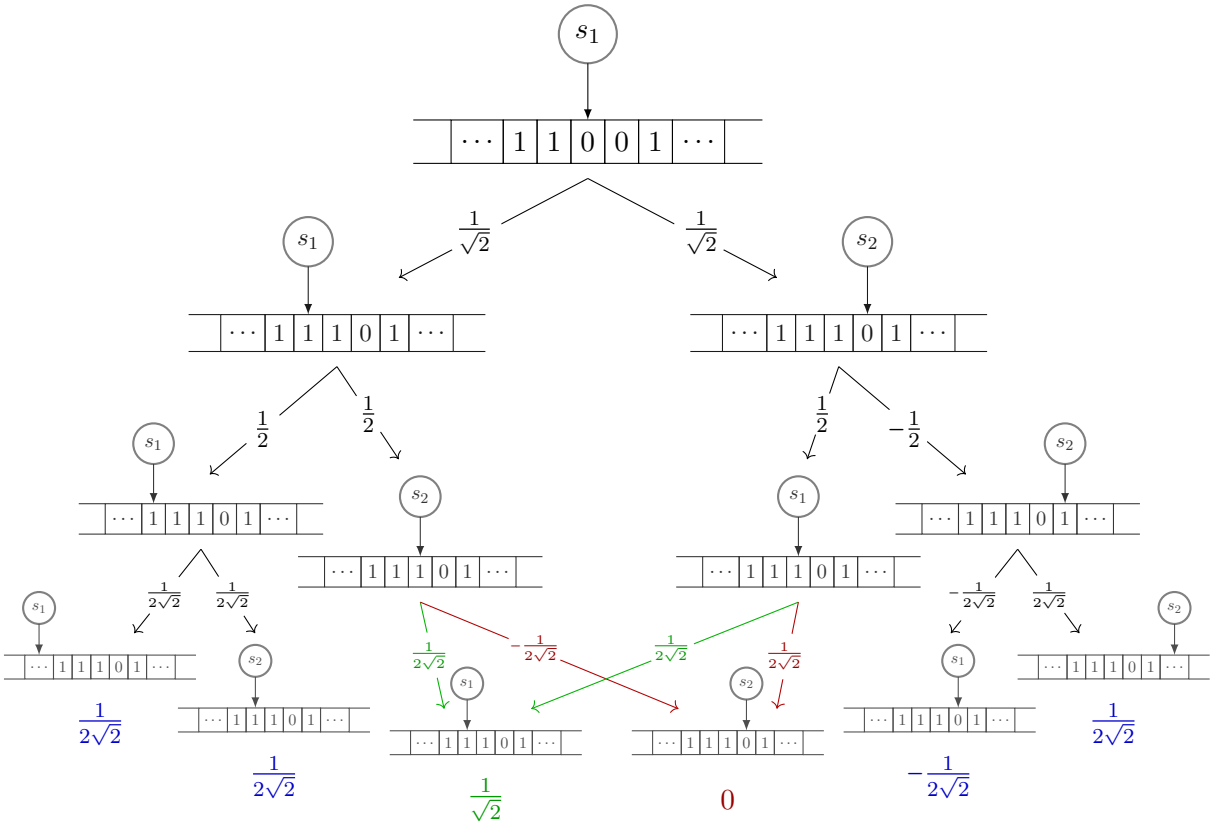



Figure 2.1: Constructive and destructive interference in a QTM computation.

### 2.1.1 Analog transitions

It has been shown that the choice of amplitudes of a quantum Turing machine makes a difference in its computational power, although not in any practical sense. For instance, a first point to make is that it does not change a QTM's expressivity whether its transition amplitudes are real or complex [BV97].

**Definition 2.2.** *Let* $\mathrm{QP}(K)$ *denote the set of functions* $f : \mathcal{H} \to \mathcal{H}$ *such that there exists a polytime-bounded QTM with transition amplitudes in* $K$ *that computes* $f$*.*

**Definition 2.3.** *Given two sets of amplitudes* $K, K'$*, we write* $\mathrm{QP}(K) \subseteq \mathrm{QP}(K')$ *if for any* $\varepsilon > 0$ *and* $f \in \mathrm{QP}(K')$*, there exists a* $g \in \mathrm{QP}(K)$ *such that, for any* $|\psi\rangle \in \mathbb{C}^{2^n}$*,* $\|f(|\psi\rangle) - g(|\psi\rangle)\| < \varepsilon$*. We write* $\mathrm{QP}(K) \sim \mathrm{QP}(K')$ *if* $\mathrm{QP}(K) \subseteq \mathrm{QP}(K')$ *and* $\mathrm{QP}(K') \subseteq \mathrm{QP}(K)$*.*

**Example 2.4** ([BV93, Lemma 3])**.** $\mathrm{QP}(\mathbb{C}) \sim \mathrm{QP}(\mathbb{R})$.

Adleman et al. showed that $\mathrm{QP}(\mathbb{C})$ contains undecidable sets [ADH97, Theorem 5.1], so it seems that the set of all complex numbers is too powerful to be able to define quantum polynomial time. Furthermore, in order to discuss universality, we require that the set of possible QTMs be countable, so that they may possibly be given as input to a universal QTM, which is not the case if the set of possible amplitudes is all the complex (or real) numbers. Therefore, Bernstein and Vazirani defined a class of polytime bounded QTMs where the transition amplitude is either -1, 1 or a rotation using the angle $2\pi \sum_{i=1}^{\infty} 2^{-2^i}$, which we will denote in an ad-hoc notation as $\mathrm{QP}(\mathrm{BV})$.

Bernstein and Vazirani showed that any angle could be efficiently approximated with only a polynomial number of instances of this single rotation [BV93, Lemma 6], and so this model faithfully approximates the initial set of uncountable QTMs. They then put forth the following definition of a class of classical functions that can be approximated with bounded-error probability by polytime QTMs, denoted FBQP.

**Definition 2.5.** *Given a set of functions $\mathcal{F}$, let $\mathcal{F}_{\geq \frac{2}{3}}$ be the set of classical functions $f : \{0,1\}^* \to \{0,1\}^*$ for which there exists a $g$ in $\mathcal{F}$ such that $\|\langle g(x)|f(x)\rangle\|^2 \geq 2/3$.*

**Definition 2.6** ([BV93])**.** $\mathrm{FBQP} \triangleq \mathrm{QP}(\mathrm{BV})_{\geq \frac{2}{3}}$.

A natural question is then, how does this model compare to other models of countable QTMs? If we restrict $\mathbb{C}$ to the set $\tilde{\mathbb{C}}$ of complex numbers whose real and imaginary parts can be computed to within $2^{-n}$ accuracy in time polynomial in $n$, Adleman et al. showed the following result, where $\mathbb{Q}$ denotes the set of rational numbers.

**Lemma 2.7** ([ADH97, Theorem 3.1])**.** $\mathrm{QP}(\tilde{\mathbb{C}}) \sim \mathrm{QP}(\mathbb{Q}) \sim \mathrm{QP}(\{0, \pm\frac{3}{5}, \pm\frac{4}{5}, \pm 1\}) \sim \mathrm{QP}(\mathrm{BV})$.

**Corollary 2.8.** $\mathrm{QP}(\tilde{\mathbb{C}})_{\geq \frac{2}{3}} = \mathrm{QP}(\mathbb{Q})_{\geq \frac{2}{3}} = \mathrm{QP}(\{0, \pm\frac{3}{5}, \pm\frac{4}{5}, \pm 1\})_{\geq \frac{2}{3}} = \mathrm{QP}(\mathrm{BV})_{\geq \frac{2}{3}}$.

Equivalency between these different sets of amplitudes indicates that the definition of quantum polynomial-time QTMs is quite robust. We denote this class of functions merely as QP in the remaining of the thesis, a notation which can be though of as a shorthand for $\mathrm{QP}(\tilde{\mathbb{C}})$.

### 2.1.2 Superposed termination

Let a *final configuration* of a QTM be a configuration in state $s_\intercal$. Since a QTM's control flow is generally quantum, it is possible that it may reach a superposition that is only partially in a final configuration, i.e. where the probability that it is observed in state $s_\intercal$ is strictly between 0 and 1. Should this QTM state count as terminating? Notice that, in this case, we cannot inspect the machine's state without collapsing it and changing its flow of computation [Mye97], as typically the information of whether or not a machine is in state $s_\intercal$ is entangled with the rest of the machine's configuration.

In principle, one could circumvent this problem by classically simulating the QTM, therefore being able to identify whether it has "fully terminated" or not without affecting the computation, although this simulation might take exponential time. This, as argued in [LP98], centers the objective not as an attempt to track a QTM's termination but rather to determine "whether the computer allows useful interference". An added difficulty is the fact that interference in QTMs is sensitive not only to the machine's configuration but also in regards to its time step, and given the requirement of unitarity, a QTM's branch cannot simply wait until another branch terminates to perform interference.

The convention of [BV97], which we will use in this work, is to define the termination of a QTM as arriving at a superposition of *only* final configurations, and where no final configuration was present in the QTM state up to that moment. Put differently, a QTM terminates if and when all of its branches reach a final configuration for the first time, in precisely the same time-step.

**Definition 2.9** (Definition 3.11, [BV97])**.** *If, when QTM $M$ is run with input $x$, at time $T \in \mathbb{N}$ the superposition contains only final configurations, and at any time less than $T$ the superposition contains no final configuration, then $M$ halts with running time $T$ on input $x$. The superposition of $M$ at time $T$ is called the final superposition of $M$ run on input $x$.*

Since QTM interference occurs when two branches reach the same configuration in the same time step (see example in Figure 2.1), it is useful in quantum branching to reason specifically about QTMs whose running time only depends on the input size. Such a machine is called *well-behaved*. Furthermore, if the machine terminates with all of its tape heads back on the start cells, it is called *stationary*.

This fact is important when considering *branching* QTMs, since the interference of the two paths of computation is sensitive to the moment of termination of each machine. Indeed, it is possible to construct a QTM that is well-behaved and stationary that performs the desired branching.

**Lemma 2.10** (Branching lemma, [BV97], restated)**.** *If $M_1$ and $M_2$ are well-behaved QTMs with the same alphabet, then there is a well-behaved QTM $M$ such that if the second track is empty, $M$ runs $M_1$ on its first track and leaves its second track empty, and if the second track has a 1 in the start cell (and all other cells blank), $M$ runs $M_2$ on its first track and leaves the 1 where its tape head ends up. In either case, $M$ takes four more time steps than the appropriate $M_i$.*

An important ingredient in quantum branching machines is the possibility of ensuring that all branches are synchronized, to allow for interference. To guarantee this, the transition function is defined such that the running time of the machine only depends on the input's size. This property can be stated as in the following theorem from [BV97].

**Lemma 2.11** (Synchronization theorem, [BV97], restated)**.** *If $f$ is a function from strings to strings such that both $f$ and $f^{-1}$ can be computed in deterministic polynomial time and such that the length of $f(x)$ depends only on the length of $x$, then there is a polynomial time, stationary, normal form reversible TM which given input $x$, produces output $f(x)$, and whose running time depends only on the length of $x$.*

Since a QTMs computation can be seen as various paths of classical computation performed in superposition, Lemma 2.11 can be used to ensure that the lengths of these computations only depend on the initial input size and therefore terminate simultaneously.

Finally, we consider the fact that the unitary evolution of a QTM must necessarily be reversible, which implies that not all TM transitions are QTM transitions.

### 2.1.3 Reversibility and unitarity

A QTM is said to be *well-formed* if the transition function $\delta$ preserves the norm of the superposition (or, equivalently, if the time evolution of the machine is unitary). A unitary evolution is necessarily reversible, and since deterministic Turing machines can perform irreversible computations, they do not form a subset of all well-formed quantum Turing machines.

While this implies that not all possible transitions are admissable for a QTM, it is always possible to rewrite the transitions so that they are done reversibly.

**Theorem 2.12** ([Ben73], restated)**.** *For every standard one-tape Turing machine, there exists a three-tape reversible, deterministic Turing machine that simulates it.*

Reversibility with one-tape machines is also possible, with the eventual drawback of maintaining separate regions of the tape and incurring a quadratic slowdown in the computation [Ben73].

Importantly, reversibility in a deterministic Turing machine is sufficient condition for its validity as a quantum Turing machine.

**Lemma 2.13** ([BV97, Theorem 4.2], restated)**.** *Any reversible Turing machine is also a (valid) quantum Turing machine.*

Therefore, calculations that are performed in a reversible way can be included in a QTM's definition while maintaining its validity.

We will now consider QTMs which have access to an *index* tape allowing for instantaneous access to a cell in the input tape.

## 2.2 Quantum Random-Access Turing Machines

In Chapter 3 we will consider a complexity class obtained from restricting a QTM's runtime polylogarithmically. Such machines cannot read the entirety of the input, and in order for them to access any input cell requires that they can access the input tape in a different way. For that reason, we introduce here the model of the quantum *random-access* Turing machine (QRATM).

A QRATM has a random-access input tape, an index tape, and $c$ work tapes and is then defined as a triplet $(Q, \Sigma, \delta)$, where $Q$ is a finite set of states containing an initial state $s_0$ and two (disjoint) subsets $Q_{\text{acc}}$ and $Q_{\text{rej}}$ for accepting and rejecting states, $\Sigma = \{0, 1, \#\}$ is the tape alphabet, and the transition function $\delta$ is such that

$$\delta : Q \times \Sigma^{1+c} \to \mathbb{C}^{Q \times \Sigma^{1+c} \times \{L,R,N\}^{1+c}}.$$

This transition maps the state and read symbols on the index tape and work tapes to a function mapping each state, each written symbol, and each head move to an amplitude. Note that the input tape does not have a tape head, hence is not taken into account in this transition function.

To get access to any character of the input, a special transition of the machine is defined: when the machine is in a special state $s_{\text{query}}$, the cell of the input tape indexed by the value written on the index tape is swapped with the cell under the work tape head, and the machine transitions to a state $s_{\text{accept}}$. Note that, in contrast with [Yam22], the input tape is not read-only as we consider a class of functions rather than decision problems, hence having the modified input be part of the output is necessary.

A pure configuration of a QRATM is a tuple

$$(s, w, w_0, w_1, \ldots, w_c, z_0, z_1, \ldots, z_c) \in Q \times \Sigma^{*2+c} \times \mathbb{Z}^{1+c}$$

where $s$ is a state, $w$ the word written on the input tape, assumed to begin in cell 0, $w_0$ is the word on the index tape, $w_1$, ..., $w_c$ the words written on the work tapes, $z_0$ is the position of the index tape head, $z_1$, ..., $z_c$ the tape head positions for the work tapes, all positions are relative to the first character of the word. The initial configuration for input $x$ is $\gamma(x) \triangleq (s_0, x, \epsilon, \ldots, 0, \ldots)$.

We call a superposition of pure configurations a surface configuration. Surface configurations can be written as $\sum_r \alpha_r |r\rangle$, with $r$ ranging over pure configurations, $\alpha_r \in \mathbb{C}$ is the amplitude associated with configuration $r$. QRATMs are also required to satisfy reversibility and well-formedness condition: a configuration may have only one predecessor, and the transition function must preserve the norm of configurations, that means that for all reachable surface configurations, $\sum_r |\alpha_r|^2 = 1$.

A QRATM halts in time $t$ on input $x$ if, starting from the initial configuration $\gamma(x)$, after $t$ steps, its surface configuration is a superposition of pure configurations in accepting states. If for all input $x$, a QRATM $M$ halts on input $x$ in time $T(|x|)$, we say that $M$ halts in time $T$. If a QRATM halts, its output is defined as the linear combination of the words on the input tape and work tapes, using the previous notations, it corresponds to $\sum_r \alpha_r |w^r, w_0^r, w_1^r, \ldots, w_c^r\rangle$.

**Definition 2.14.** *Given a function $f : \{0,1\}^* \to \{0,1\}^*$, we say that a QRATM $M$ approximates $f$ with probability $p$ if for any input $x \in \{0,1\}^*$, starting from the initial configuration $\gamma(x)$, $M$ halts with an output $\sum_r \alpha_r |w^r, w_0^r, w_1^r, \ldots, w_c^r\rangle$ such that $\sum_{r \in Q_{acc} \times \{f(x)\} \times \mathbb{Z}^{2+c}} |\alpha_r|^2 \geq p$.*

We now turn to the circuit model of computation, which differs in many ways from the quantum Turing machine model.

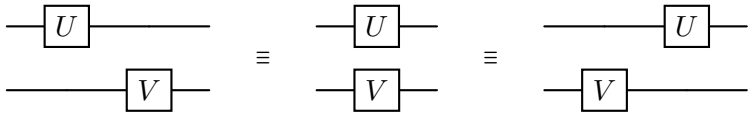
## 2.3 Quantum Circuits

> By the mid- to late-1990s, quantum circuits effectively supplanted quantum Turing machines as the computational model of choice in the study of quantum algorithms and complexity theory – a shift made possible by Yao's proof that the models are equivalent.
>
> – Molina and Watrous [MW19]

Quantum circuits [Deu89, Yao93] are the ubiquitous model for describing quantum computation, and in many ways are considered to be the current standard approach to reasoning about quantum programs. A quantum circuit is defined by an acyclic array of *quantum gates*, applied on *wires* which represent individual qubits.

Each gate is defined by a unitary operator, as it describes the evolution of a qubit state. As a consequence, a gate's number of input qubits is the same as its number of output qubits. A circuit's semantics is given by the composition of the semantics of each gate.

Given two circuits $C_1$, $C_2$, we denote by $C_1 \equiv C_2$ the equivalence between the semantics of two circuits. For instance, we have that




derived from the fact that

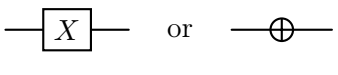
$$(\mathbb{1} \otimes V)(U \otimes \mathbb{1}) = U \otimes V = (U \otimes \mathbb{1})(\mathbb{1} \otimes V).$$

This means that gates can be generally "dragged" through wires without changing the circuit's semantics, and that operations can be applied concurrently. This is important in the distinction between a gate's *size* (number of gates) and *depth* (the number of time steps required to execute all the gates).

### 2.3.1 Quantum gates

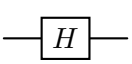
We now consider a few simple gates. An example of a classical gate which is also described by a unitary operator is the *NOT* gate:

$$NOT \triangleq \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \text{or}$$



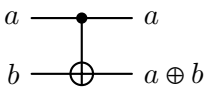

A standard example of a quantum gate is the *Hadamard gate*, defined in Section 1.1.2, which creates a balanced superposition from a classical state:

$$H \triangleq \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad \text{with } H|a\rangle = \frac{1}{\sqrt{2}} \cdot |0\rangle + \frac{(-1)^a}{\sqrt{2}} \cdot |1\rangle$$



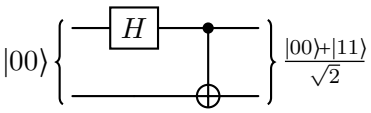
So far we have only considered single-qubit gates. In order to generate *interaction* between qubits, we will need to consider multi-qubit gates. Given a gate described by a unitary $U$, one may define a *controlled-U* gate, where $U$ is applied if and only if a control qubit is in state $|1\rangle$. For instance, a controlled-$NOT$ gate is described as

$$CNOT \triangleq \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

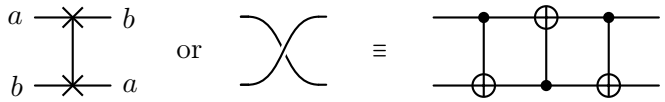



If a gate is only applied if the control qubit is in state $|0\rangle$, this is called a *negative* control and the result gate is said to be *anticontrolled*.

We can compose gates to create larger circuits. For example, a Hadamard and $CNOT$ gate can create an entangled pair of qubits:




The swapping of qubit addresses can be performed with 3 controlled-$NOT$ gates:



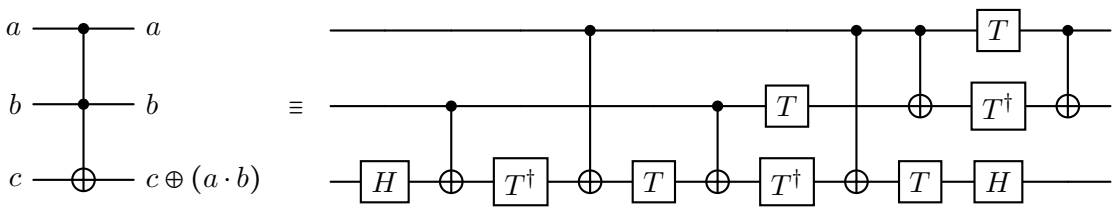

The inverse of unitary gates coincides with the conjugate transpose. So far, all gates we have considered are their own inverse, i.e. we have that $NOT^\dagger = NOT$ and $H^\dagger = H$. Examples of gates that are not their own inverse are *rotations* and *phase-shifts*. For example,

$$R_y(\theta) \triangleq \begin{pmatrix} \cos(\theta/2) & -\sin(\theta/2) \\ \sin(\theta/2) & \cos(\theta/2) \end{pmatrix} \quad \text{and} \quad Ph(\theta) \triangleq \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$

satisfy $R_y(\theta)^\dagger = R_y(-\theta)$ and $Ph(\theta)^\dagger = Ph(-\theta)$.

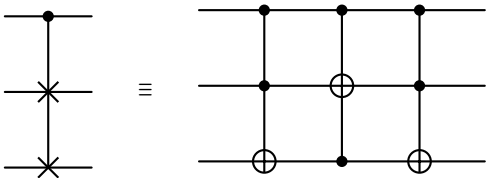
### 2.3.2 A universal set

Many more gates can be defined. However, it is an interesting question to know if at this point we have enough gates to be able to describe any unitary. This is the issue of *gate universality* [BBC+95, BMP+99]. Different sets of gates have been shown to be (approximately or exactly) universal for quantum computing. In classical computing, a universal gate is the 2-bit NAND gate, which returns 0 if both input bits are in state 1, and returns 1 otherwise. Its reversible equivalent is the 3-qubit *Toffoli*, which we can implement using gates we have defined so far:



where $T \triangleq Ph(\frac{\pi}{4})$ is the so-called $T$ gate, corresponding to a phase shift of $\pi/4$.

Using the Toffoli gate, we are able to define a controlled-swap, also called a *Fredkin* gate:



To build the Toffoli gate, we used the set of gates $\{CNOT, H, T\}$ (note that $T^\dagger = T^7$), which has been shown to be approximately universal for quantum computation [BMP+99]. This set is also *minimal* in the sense that removing any gate makes the set no longer universal. This can be seen as each subset falls short of describing general unitaries:

- *Removing* $CNOT$: The set $\{H, T\}$ consists only of single-qubit gates, and therefore is not capable of producing entangled states.
- *Removing* $H$: The set $\{CNOT, T\}$ cannot create superposed states, as a basis state input results in a basis state output (up to a global phase) for both gates.
- *Removing* $T$: Gates $\{CNOT, H\}$ only contain real entries and so cannot be combined to describe a unitary with complex values.

Among these basic gates, the $T$ gate is considered to be the hardest one to implement, depending on hardware, as the $CNOT$ and Hadamard gates belong to the Clifford group, and circuits containing only these gates can be efficiently simulated by a classical computer [Got98, AG04].

In this work, we will consider as *basic gates* the set $\mathcal{B} \triangleq \{CNOT, R_y(\theta), Ph(\theta)\}$, where $R_y(\theta)$ and $Ph(\theta)$ can take any value $\theta \in \mathbb{R}$. Given a circuit $C$ described with gates in $\mathcal{B}$, we denote by $\#C$ its number of gates.

### 2.3.3 Ancilla qubits

A very standard tool of reversible and, in particular, quantum computation is the use of extra bits of information which allow for performing logical operations without erasing information. These ancillary bits, also simply called *ancillas*, add to the circuit's total complexity in the total number of bits (or qubits).

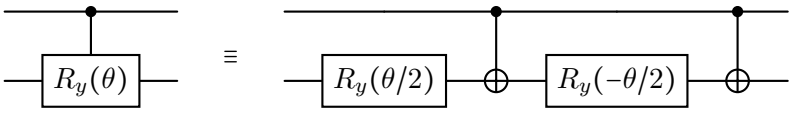
In most applications in quantum computing, and for all intents and purposes in this work, these extra qubits are not part of the output of the computation and should be set back to their initial state once the computation is over (this is called *uncomputation*). This treatment of ancillary qubits corresponds to what are called *clean ancillas*, which differentiates them from *dirty ancillas* or *conditionally clean ancillas* recently defined in [KG25].

In this work, we will only be handling clean ancillas. An easy example of ancilla use is given below in the construction of a multi-controlled $U$ gate, using 4 ancillas in state $|0\rangle$ which are then reset to that state by the end of the circuit.

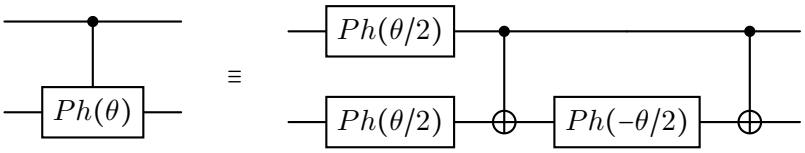
### 2.3.4 (Multi-)controlled gates

We have considered a set of basic gates $\mathcal{B}$ which is universal for quantum computing. We now consider how we can describe arbitrarily-controlled versions of these gates by using the original set $\mathcal{B}$.

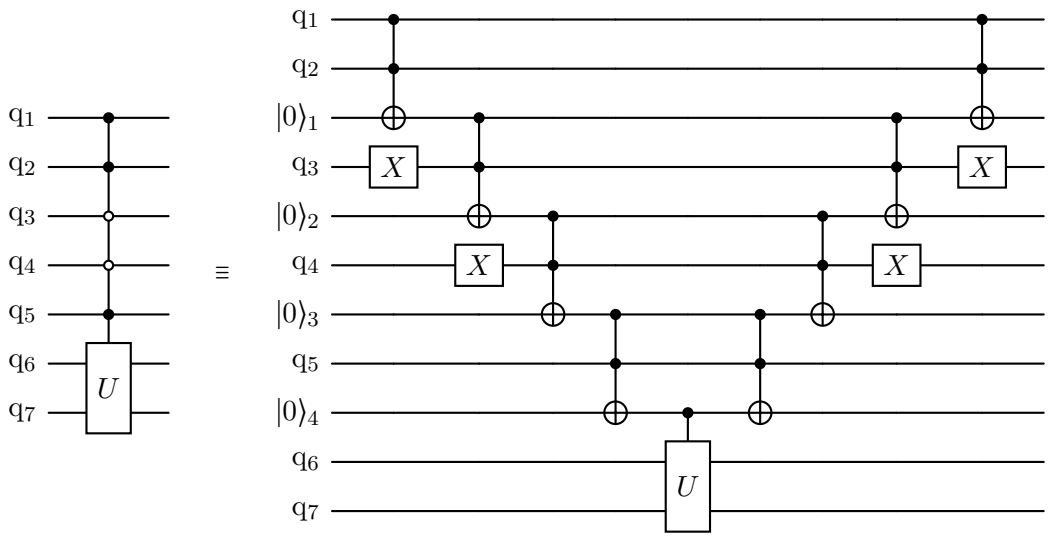
We start by considering gates with a single control qubit. The controlled $y$-rotation gate can be written with $CNOT$s and single-qubit rotations:




A similar construction can also be done for the phase-shift gate:




Given an arbitrary gate $U$ with $n > 1$ control qubits, we are able to implement the multi-controlled $U$ using an instance of $U$ controlled on a single qubit, as well as $n-1$ ancillary qubits and $(2n-2)$ Toffoli gates [NC10]. For example, for $n = 5$:




Where an empty circle ∘ indicates a *negative* control, meaning that the gate is applied when the control qubit is in state $|0\rangle$.

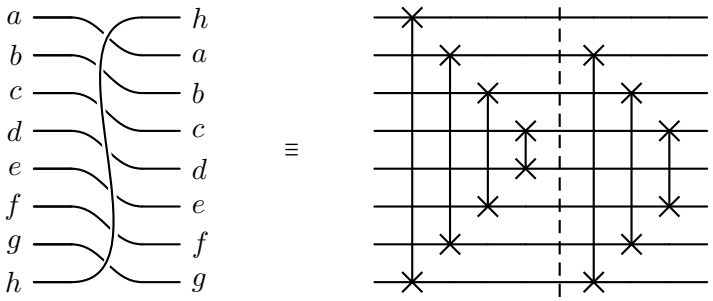
### 2.3.5 Circuit depth

As we claimed at the start of Section 2.3, gates can "move around" in wires without changing the circuit semantics. In practice, this means that, in many cases, gates can be applied concurrently, and their respective qubits are active for a shorter amount of time.

The *fidelity* of a qubit measures how close a given qubit state is to the *target* state, i.e. the intended output state, given its initial condition and the gates that were applied [HYC+19]. Current qubit implementations have much lower fidelities than classical bits, and for this reason the *depth* of a circuit is an important measure of its complexity, as it gives a metric for the maximum number of operations performed on any given qubit in the circuit.

Circuit depth is an important notion even in complexity theory, as a circuit's asymptotic size may be very different from its asymptotic depth. As an example, consider an arbitrary

permutation of qubit positions. In the worst case (where all addresses must be changed) this requires at least a linear number of swap gates, but these gates can be chosen and arranged such that the circuit has depth 2 [MN01]. For example, consider the permutation:




where the swap gates left and right of the dashed line all refer to disjoint pairs of qubits, and therefore the circuit depth is 2.

The final computational model we consider are quantum programming languages.

## 2.4 Quantum Programming Languages

While quantum circuits have become the standard for reasoning about quantum algorithms and their complexity, any given algorithm will not correspond to a particular circuit, but rather to a *family of circuits*, typically parametrized by the input size of the problem. In fact, solving a problem is usually done by reasoning about its structure, using scalable tools such as loops, recursion, and other types of control flow. For these reasons, programming languages are incredibly useful tools, even when one wishes to reason about problems at the circuit level [Sel04].

### 2.4.1 A focus on quantum circuit description

The design of many programming languages is directly inspired by the ubiquitousness of quantum circuits as a paradigm for quantum computation. This is the case of *circuit description languages*, such as Quipper [GLR+13], LIQ$Ui|\rangle$ [WS14], QWIRE [PRZ17], Qiskit [JATK+24], and Cirq [Dev24]. These languages allow for scaling up quantum circuits by combining them in safe ways, all the while making use of classical control flow. Since quantum circuits are *classical data*, they lend themselves to all the typical operations of classical computation, including duplication (cloning) and deletion. This allows the quantum programmer to build large circuits without having to consider the low-level details of the implementation.

Another approach is to create programs by direct use of qubit variables, this is the case, for instance, in quantum lambda-calculi [AG05, SV06, DCM22], and the languages Silq [BBGV20] and Qunity [VLRH23]. These examples vary in the way that they define basic quantum operations, but ensuring that qubit transformations are valid (for instance, that no-cloning is enforced) is generally more complicated than in circuit description languages.

### 2.4.2 Unitarity and reversibility

Ensuring that a program describes a unitary (and therefore reversible) transformation necessarily requires that it satisfies the no-cloning theorem. For instance, a function encoding a $CNOT$ that takes in qubit variables $x$ and $y$ is only valid if $x$ and $y$ reference different qubits. More

intricate examples exist, as in QML [AG05] or Lambda-$\mathcal{S}_1$ [DCM22], where a term encoding an **if** statement

$$\text{if } x \text{ then } s \text{ else } t$$

is well-typed for a qubit variable $x$ only if $s$ and $t$ are *orthogonal* (in cases where $s$ and $t$ are open terms, this means that they are orthogonal for any term substitution where they have adequate types). The definition of orthogonality then depends on the type of the term, and different criteria for orthogonality can be proposed. Ensuring that $s$ and $t$ are orthogonal guarantees that the reduced term $\alpha \cdot s + \beta \cdot t$ satisfies unitarity when $|\alpha|^2 + |\beta|^2 = 1$.

### 2.4.3 Classical versus quantum control

While quantum programming languages strive to be *hardware-independent* in that they can be translated into different architectures without further intervention from the programmer [BCS03], they usually differ on their emphasis of different aspects of quantum computation, one example being that of *control flow*.

Quantum mechanics allows for two possible interactions with qubits – *unitary evolution* of their state, or *measurement*. As a result, there are two approaches to handling control flow in quantum programs:

- *Quantum control flow*: the superposed state of a qubit determines which instruction is applied, resulting in potentially very large unitary transformations;
- *Classical control flow*: the program (possibly) executes different instructions according to the measurement result of a qubit; this makes the program probabilistic.

This distinction is not absolute and should be seen more as a matter of degree. All languages will have some sort of classical control flow, even if they do not allow for measurements, since they may contain **for** or **while** loops, or even perform recursion over classical parameters such as integers or the size of a qubit list.

Likewise, most languages allow for a measure of quantum control, for instance, in the form of adding control qubits to a unitary gate. The quintessential example of quantum control is given by the existence of a quantum **if** statement which, for unitaries $U_0$, $U_1$ defined in the program, performs the transformation

$$|0\rangle_{\text{control}} \otimes |\psi\rangle_{\text{target}} + |1\rangle_{\text{control}} \otimes |\phi\rangle_{\text{target}}$$

$$\downarrow$$

$$|0\rangle_{\text{control}} \otimes U_0 |\psi\rangle_{\text{target}} + |1\rangle_{\text{control}} \otimes U_1 |\phi\rangle_{\text{target}}$$

The interesting case occurs when $U_0$ and $U_1$ are not basic gates, but rather unitary transformations defined by substatements of the program, possibly making further use of classical and quantum control flows. This instruction is not directly available in Qiskit, but is implemented in Quipper, Silq and Qrisp [SBZ+24].

The use of quantum control is not strictly necessary for a language to express all quantum algorithms [DMZ10]. However, it allows for the quantum programmer to make "full use" of quantum superposition by being able to reason about differents sets of instructions being performed according to the superposed state of a qubit. This approach is closer to the quantum Turing machine model, where all properties of the machine (tape symbols, head position, state) exist in superposition and the entire control flow is quantum.

### 2.4.4 Resource estimation and optimization

Given the fragility of quantum computers, many programming languages include tools for *resource estimation* and *optimization*. For instance, Quantinuum's TKET language [SDC+20] offers translation of circuits into diagrams that can be rewritten to obtain simpler, equivalent circuits.

Some circuit description languages allow for typing rules that estimate the complexity of obtained circuits [CDL24, CDL25]. Another example can be found in Quipper, which allows for the reuse of ancillary qubits in a circuit by allowing for declarations of ancilla *creation* and *deallocation* [GLR+13].

In this thesis, we address the problem of optimizing quantum circuits in a way that differs significantly from the approach in typical programming languages. In Part III we study how to improve the *asymptotic complexity* of the quantum circuits by exploiting the structure of the program. This contrasts with the approach of taking an already-compiled circuit and reducing its size or depth via local simplifications.

# 3

# Quantum Complexity Classes

> By any objective standard, the theory of computational complexity ranks as one of the greatest intellectual achievements of humankind – along with fire, the wheel, and computability theory.
>
> – Scott Aaronson, *Quantum Computing Since Democritus*

Having gone over the structure and requirements of different models of quantum computation, in this chapter we will focus in how these models are used to define quantum complexity classes, namely the classes FBQP and FBQPOLYLOG of functions that can be approximated in quantum polynomial and poly-logarithmic time, respectively.

The computational complexity of a given model is defined by its access to resources, usually in time or in space. A quantum Turing machine's *space complexity* corresponds to the number of tape cells necessary to perform all branches of its computation. The *time complexity* of the QTM is the number of steps required for the machine to reach a final configuration.

Given a set of basic gates, a quantum circuit's *size*-complexity is defined as its total number of gates[1], whereas its *depth*-complexity is the number of time-steps required to execute all the gates in the circuit. Circuit *space*-complexity can be defined as its number of wires (including ancillas), but this notion is included, in the asymptotic case, in the definition of circuit size, as wires without gates are irrelevant to the computation.

The quantum Turing machine model was first introduced by Deutsch [Deu85] and later studied by Bernstein and Vazirani [BV93, BV97], who defined the class FBQP by only considering QTMs that terminate in polynomial time (Definition 2.6). As we saw in Chapter 2, ensuring the robustness of the FBQP definition, as well as guaranteeing that QTMs were well-defined, required grappling with different obstacles, namely: restricting the set of transition amplitudes, ensuring that all branches of computation terminate simultaneously, and generally making sure that the transition function is at all times unitary (and, therefore, reversible).

The quantum circuit model is comparatively much simpler, as it lends itself more easily to classical intuitions. While there are important differences between quantum and classical circuits – quantum wires exist in superposition, and gates are described by unitary matrices – they can be handled in similar ways.

In 1993, Yao showed that $t$ steps of a quantum Turing machine on input $n$ where $t \geq n$ can be simulated by a quantum circuit of size quadratic in $t$ [Yao93]. The equivalence between these two models led to quantum circuits becoming the standard approach to describing quantum

[1] A case can be made that the number of *wires* should also be included in the definition of circuit size. However, only considering the number of gates allows us to describe the cases where not all qubits are affected, for instance in constant-time operations.

algorithms. The ideas of the proof will also be useful for proving that a programming language can simulate a polytime quantum Turing machine.

We start by introducing the *big-O* notation which we will use throughout the thesis. Let $f, g : \mathbb{N} \to \mathbb{N}$ be two functions. Then, we write:

$$f(n) = O(g(n)) \text{ if there exist } M, n_0 \in \mathbb{N} \text{ such that } f(n) \leq M\, g(n),\ \forall n \geq n_0,$$

$$f(n) = \Omega(g(n)) \text{ if there exist } M > 0, n_0 \in \mathbb{N} \text{ such that } f(n) \geq M\, g(n),\ \forall n \geq n_0.$$

The following notation is used when the two conditions hold:

$$f(n) = \Theta(g(n)) \text{ if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

We now discuss the class FBQP and its definition using quantum circuits.

## 3.1 Quantum Polynomial Time (FBQP)

We can complement the definition of FBQP given in Chapter 2 by considering its equivalent definition via *uniform quantum circuit families.*

### 3.1.1 Equivalence between QTMs and quantum circuits

In this section we go into the details of Yao's result, and some of its ideas. We start by introducing a standard notion in circuit complexity, that of *uniform families of circuits.*

Circuits are defined for a fixed input size – typically its number of (non-ancillary) wires. Therefore, we consider *families of circuits*, defined as a sequence of circuits $\{C_n\}_{n\in\mathbb{N}}$ where $C_n$ takes in an input of size $n$. Furthermore, we will be interested in families of circuits for which there is a single, efficient classical algorithm that, from the input size $n$, is able to compute a description of the circuit $C_n$.

**Definition 3.1.** *A family of circuits $(C_n)_{n\in\mathbb{N}}$ is said to be* uniform *if there exists a polynomial-time Turing machine that takes $n$ as input and outputs a representation of $C_n$, for all $n \in \mathbb{N}$.*

**Definition 3.2.** *A family of circuits $(C_n)_{n\in\mathbb{N}}$ is said to be* polynomially-sized *with $\alpha \in \mathbb{N} \to \mathbb{N}$ ancilla qubits if there exists a polynomial $P \in \mathbb{N}[X]$ such that, for each $n \in \mathbb{N}$, $\#C_n \leq P(n)$ and the number of ancilla qubits in $C_n$ is exactly $\alpha(n)$.*

Yao's equivalence result can then be stated as follows.

**Theorem 3.3** (Theorem 2, [Yao93], restated)**.** *Given a quantum Turing machine $M$, there exists a uniform family of quantum circuits of size $O(t^2)$ that simulates $t$ steps of $M$.*

**Corollary 3.4.** *Polytime-bounded QTMs and uniformly generated quantum circuits have equivalent computational power, up to a polynomial overhead.*

This equivalence suggests an alternative definition of FBQP to the one given in Definition 2.6 which made use of quantum Turing machines. To give such a definition, we start by bringing the two models closer, by adapting circuits with a fixed input size to the infinite-tape of a QTM.

We define a padding function for the input given to a quantum circuit. This is done in order to simulate a QTM's unbounded writing space. An important detail in this simulation is that, for a given input, a time-bounded QTM only accesses a finite number of cells in its tape.

Furthermore, if the QTM is *polytime-bounded*, the maximum number of cells accessed for any input of size $n$ is bounded by a polynomial $P(n)$.

Furthermore, as is customary, we assume that the size of our target function $f$ is known and that it only depends on the input size. Then, a circuit computes $f(x)$ if the first $|f(x)|$ qubits are in state $f(x)$ with probability at least $\frac{2}{3}$.

**Theorem 3.5** (Adapted from [Yao93] and [NC10])**.** *A function $f : \{0,1\}^\star \to \{0,1\}^\star$ is in* FBQP *iff there exists a uniform polynomially-sized family of circuits $(C_n)_{n\in\mathbb{N}}$ with $\alpha$ ancilla qubits s.t. $\forall x \in \{0,1\}^\star$,*

$$\left\| \langle f(x) \,|\, [\![C_{|x|}]\!] \left( |x\rangle \otimes |0^{\alpha(|x|)}\rangle \right) \right\| \geq \frac{2}{3}.$$

We will now look at the more interesting aspects of Yao's proof, in particular those which will be relevant for our purposes later on in this work.

**Overview of the proof of Theorem 3.3** The entire argument for the simulation scheme is beyond the scope of this work, and so we will focus on its aspects that will be more useful to us in the next chapters. We discuss the main ideas of Yao's proof, with some comparisons to more recent work that improves the asymptotic depth of the simulation [MW19].

Given a single-tape quantum Turing machine $M$, we let $Q = \{1, \ldots, m\}$ be its set of states, and $\Sigma = \{0, 1, \#\}$ its alphabet. In Section 2.1, we described a QTM's transition as the action of an infinite-dimensional unitary acting on its configuration, described by a transition function

$$\delta : Q \times \Sigma \to \mathbb{C}^{Q\times\Sigma\times\{L,N,R\}}$$

which evolves the inner machine state, the tape head position, and the state of its infinite tape in a unitary fashion. In order to describe this unitary as the action of a quantum circuit, one must describe the QTM's transition as *finite* and *local*, so that it may be described by a finite quantum circuit described by the number of constant-size gate operations.

*Finiteness* is given by the fact that we only wish to simulate a fixed number $t$ of transition steps. Starting from the cell with index 0, in $t$ steps the machine $M$ can only reach the cells with index in $\{-t, \ldots, t\}$. *Locality* comes from the fact that, while $M$'s unitary evolution is defined over the entire tape, its description is focused in the tape space surrounding the tape head. Put differently, while the machine has an infinite tape to work with, each step only concerns three tape cells at a time: the cell with the tape head, and the cells adjacent to it, left and right.

We can start by considering the simulation of a single step of $M$. We rewrite $M$'s tape description such that all the relevant information is available locally. For every cell, we wish to have the following information:

- what symbol is written on the cell;
- whether or not the machine head is reading the cell, and
- if so, what is the internal state of the machine.

For this purpose, we denote $M$'s configuration by $2t+1$ pairs $\tau_i = (k_i, \sigma_i)$ where $\sigma_i$ is the tape symbol written in cell $i \in \{-t, \ldots, t\}$, and $k_i \in \{-m, \ldots, m\}$ is defined, for $s \in Q$, as

$$k_i \triangleq \begin{cases} s, & \text{if the machine is in state } s \text{ with the tape head } \textit{active} \text{ on cell } i, \\ -s, & \text{if the machine is in state } s \text{ with the tape head } \textit{inactive} \text{ on cell } i, \\ 0, & \text{if the machine head is not in cell } i. \end{cases}$$

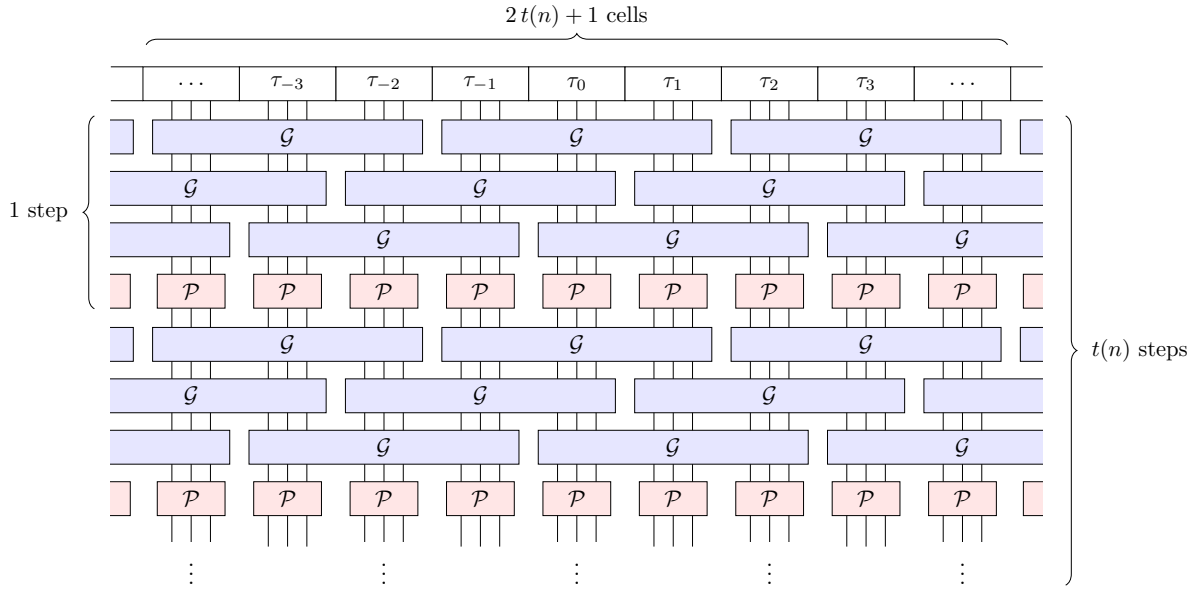



Figure 3.1: Scheme for simulation of a QTM using a quantum circuit.

The distinction between an *active* and *inactive* tape head in the simulation is an essential idea in the proof, as it ensures that a precise number of steps are applied. Let $\mathcal{T}$ be the Hilbert space of pairs $(\sigma, k)$. The initial state of $M$ on input $x_0, \ldots, x_{n-1}$ is defined in the Hilbert space $\mathcal{T}^{2t+1}$ and is encoded as

$$\left(\bigotimes_{i=-t}^{-1} |(\#,0)\rangle_i\right) \otimes |(x_0, s_0)\rangle_0 \otimes \left(\bigotimes_{i=1}^{n-1} |(x_i,0)\rangle_i\right) \otimes \left(\bigotimes_{i=n}^{t} |(\#,0)\rangle_i\right)$$

The first gate is $\mathcal{G} : \mathcal{T}^3 \to \mathcal{T}^3$, defined over the Hilbert space of three pairs, which evolves the state such that it simulates a step of the QTM if the head is active on cell $i$ and then deactivates the tape head. For instance, $\mathcal{G}$ acts trivially on a state of the form

$$|(a_1, b_1)\rangle \otimes |(a_2, 0)\rangle \otimes |(a_3, b_3)\rangle$$

as the head is not in found in the middle register pair. If it is, $\mathcal{G}$ performs the transition:

$$|(a_1,0)\rangle \otimes |(a_2,p_2)\rangle \otimes |(a_3,0)\rangle$$

$$\begin{aligned}
\longrightarrow \quad & \sum_{c_2 \in Q,\, b_2 \in \Sigma} \delta(p_2,a_2)[q_2,b_2,L] \cdot |(a_1,-q_2)\rangle \otimes |(b_2,0)\rangle \otimes |(a_3,0)\rangle \\
& + \sum_{c_2 \in Q,\, b_2 \in \Sigma} \delta(p_2,a_2)[q_2,b_2,N] \cdot |(a_1,0)\rangle \otimes |(b_2,-q_2)\rangle \otimes |(a_3,0)\rangle \\
& + \sum_{c_2 \in Q,\, b_2 \in \Sigma} \delta(p_2,a_2)[q_2,b_2,R] \cdot |(a_1,0)\rangle \otimes |(b_2,0)\rangle \otimes |(a_3,-q_2)\rangle
\end{aligned}$$

The deactivation of the tape head by $\mathcal{G}$ ensures that precisely one step is applied, as once the first $\mathcal{G}$ acts non-trivially the remaining will no longer change the state. Simulating one step of $M$ consists then of applying $\mathcal{G}$ to every triple $\tau_{i-1}\,\tau_i\,\tau_{i+1}$, for $i \in \{-t, \ldots, t\}$.

There remains, however, the problem that the unitariry of $M$'s transition may not be conserved when we restrict the tape to the cells of index $-t$ through $t$. For instance, even the trivial machine that only moves the tape head right is not reversible when we consider its restriction to a finite amount of tape. To handle this restriction, we consider a *looped* tape, where

$$\tau_{t+2} \triangleq \tau_{t-1} \quad \text{and} \quad \tau_{t-2} \triangleq \tau_{t+1}.$$

Yao proved the correctness of this construction for a cascading application of gate $\mathcal{G}$ from left to right [Yao93]. By slightly modifying $\mathcal{G}$, it is possible to arrange gate applications such that the depth of one step is constant [MW19], as in Figure 3.1.

After a round of $\mathcal{G}$ gates is applied, a gate $\mathcal{P}$ resets the tape head in each cell back to active:

$$\mathcal{P} : \mathcal{T} \to \mathcal{T}, \quad \mathcal{P}\,|k\rangle \triangleq |-k\rangle,$$

and a full step of $M$ is simulated. In order to simulate $t$ steps of $M$, one simply composes this circuit $t$ times.

We now consider a class that is more restrictive than FBQP, the class of quantum polylogarithmic-time functions.

## 3.2 Quantum Polylogarithmic Time (FBQPOLYLOG)

*Polylog-time* problems occur in particular circumstances, usually when the input is particularly structured. An example of such a problem is binary search on a sorted list.

A standard quantum Turing machine running in polylog-time would not be able to access the entirety of its input tape by moving the tape head. In order for the machine to have access to any part of the input, the class FBQPOLYLOG makes use of the quantum random-access Turing machine model described in Section 2.2.

**Definition 3.6.** *The class* FBQPOLYLOG *is defined as the set of functions* $f : \{0,1\}^* \to \{0,1\}^*$ *such that there exists a QRATM running in polylogarithmic time that approximates* $f$ *with probability at least* $\frac{2}{3}$.

We will now relate this class to the quantum circuit model via the theory of quantum query complexity.

### 3.2.1 Quantum query complexity

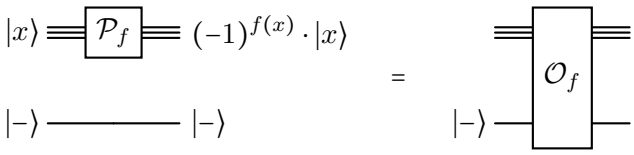
The *query model* of quantum computation is the scenario in which one wishes to compute a function by making use of *black boxes* which are accessed via queries [Amb18]. A black box performs a unitary transformation on the quantum state according to some classical function $f : \{0,1\}^* \to \{0,1\}$. There are generally two types of oracles considered, *phase* oracles and *boolean* oracles. A phase oracle $\mathcal{P}_f$ performs the phase-shift operation

$$|x\rangle \mapsto^{\mathcal{P}_f} (-1)^{f(x)}|x\rangle,$$

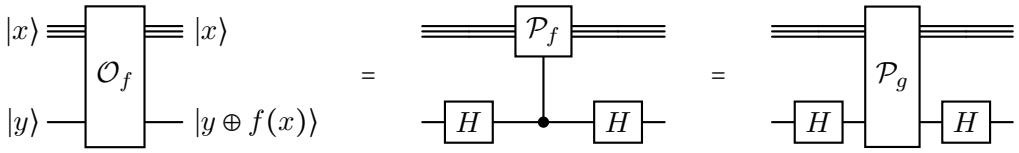
whereas a boolean oracle $\mathcal{O}_f$ is defined, for $x \in \{0,1\}^*$ and $b \in \{0,1\}$, as

$$|x\rangle \otimes |b\rangle \mapsto^{\mathcal{O}_f} |x\rangle \otimes |b \oplus f(x)\rangle.$$

In both cases, the oracle is defined as being executed in a *single step*, and are used as a tool for reasoning about the complexity of algorithms. The two oracles are equivalent, and a choice can be made according to which one better suits a specific algorithm. For instance, a phase-shift oracle can be simulated by a boolean oracle perform the query on a $|-\rangle$ state:




Likewise, a boolean oracle can be simulated by a controlled phase-shift oracle:




where a controlled phase-shift oracle for $f$ is recast as a phase-shift oracle for a function $g$ defined, for $n \geq 1$, as

$$g(x_1, \ldots, x_n, y) \triangleq y \cdot f(x_1, \ldots, x_n).$$

Given a target function $\mathcal{F}$, we denote by $Q(\mathcal{F})$ the function that takes as input $n \in \mathbb{N}$ and gives the least number of queries necessary for a quantum algorithm to approximate $\mathcal{F}$ with bounded-error probability on input size $n$. The function $Q(\mathcal{F})$ gives a *lower bound* on the time complexity of $\mathcal{F}$. For instance, for the OR, AND, and PARITY functions defined as

$$\mathrm{OR}(x) \triangleq \max_{i=1\ldots n} x_i, \qquad \mathrm{AND}(x) \triangleq \min_{i=1\ldots n} x_i, \qquad \text{and} \qquad \mathrm{PARITY}(x) \triangleq \bigoplus_{i=1}^{n} x_i,$$

we have that, while Grover's algorithm [Gro96] allows for a quadratic speedup in the query complexity of AND and OR, no such speedup exists for PARITY.

**Lemma 3.7** ([Zal99])**.** *For $\mathcal{F} \in \{\mathrm{AND}, \mathrm{OR}\}$, we have that $Q(\mathcal{F}) = \Theta(\sqrt{n})$.*

**Lemma 3.8** ([FGGS98, BBC+01])**.** $Q(\mathrm{PARITY}) = \Theta(n)$.

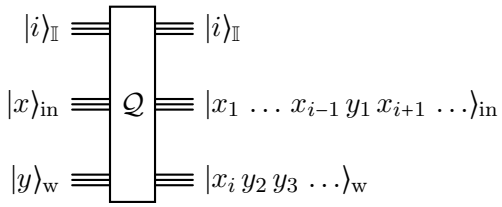
### 3.2.2 The limits of quantum polylogarithmic time

A link between the quantum query model and quantum random-access Turing machines can be made by viewing the read-input transition as an oracle query [Yam22].

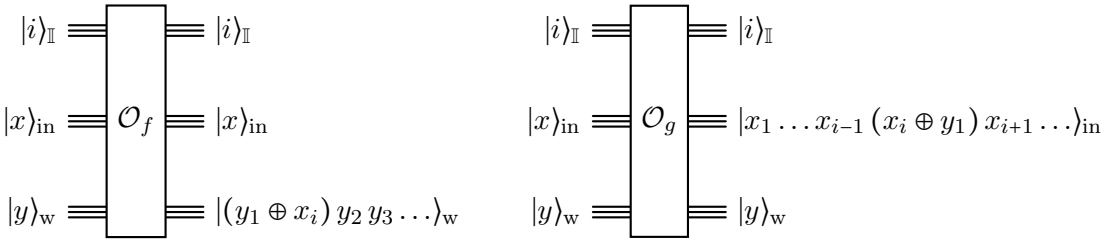
**Lemma 3.9.** *Let $\mathcal{F} \in$ FBQPOLYLOG. There exists $k \in \mathbb{N}$ such that $Q(\mathcal{F}) = O(\log^k n)$.*

*Proof.* Let $\mathcal{F}$ be a function approximated with bounded-error probability by a QRATM $M$ bounded in polylogtime. Let $M'$ denote the machine described by $M$ where the $s_{\text{query}}$ state acts trivially on the tape, meaning that it only changes into state $s_{\text{out}}$ without moving the tape heads or changing any symbols. We have that $M'$ is a valid quantum Turing machine, and therefore we may build a unitary $U_{M'}$ from basic gates that simulates a step of $M'$ in an encoding of the tape as in Yao's proof.

The execution of the QRAM step is then defined using a gate $\mathcal{Q}$, which performs the transformation:




The gate $\mathcal{Q}$ can then be described as an oracle operation. Given the relation between $SWAP$ and $CNOT$ gates, we consider the following two boolean oracles:




and obtain $\mathcal{Q} = \mathcal{O}_f\, \mathcal{O}_g\, \mathcal{O}_f$. Therefore, one application of $\mathcal{G}$ corresponds to a constant number of oracle queries. A single step of $M$ is given by $U_{M'}\, \mathcal{Q}$ and since machine $M$ runs in polylogarithmic time, the entire simulation can be done with a polylog number of gates, including oracle queries. □

We may therefore conclude that AND, OR, and PARITY cannot be computed in quantum polylogarithmic time, even with bounded error.

**Lemma 3.10.** AND, OR, PARITY $\notin$ FBQPOLYLOG.

*Proof.* By Lemmas 3.7, 3.8 and 3.9. □

So far, we have considered the quantum Turing machine and quantum circuit models of computation, as well as the complexity classes FBQP and FBQPOLYLOG. We gave equivalent definitions of FBQP, using polytime-bounded QTMs and uniform families of poly-sized quantum circuits. We now consider two existing *model-free* (or *implicit*) characterizations of quantum polynomial time.

## 3.3 Implicit Characterizations

Quantum polynomial time has been characterized with a lambda calculus [DMZ10] and a function algebra [Yam20]. While these works differ on the precise class being characterized, they share a general structure. First, one considers a system that is able to simulate a polytime quantum Turing machine, and then one imposes the bounded-error condition on the result.

We describe in some detail the characterization of FBQP given in [Yam20] as it provides a point of comparison to the equivalent characterization given in Chapter 5, and because it will be of use in the completeness proof of Theorem 5.24. We start by giving the definition of the function algebra given in [Yam20], which we denote by $\mathscr{Y}_0$.

**Definition 3.11** ([Yam20]). *Let $\mathscr{Y}_0$ be the smallest class of functions including the following basic functions, with $\theta \in [0, 2\pi) \cap \tilde{\mathbb{C}}$,*

$$I(|\psi\rangle) \triangleq |\psi\rangle, \qquad Ph_\theta(|\psi\rangle) \triangleq |0\rangle\langle 0|\psi\rangle + e^{i\theta}|1\rangle\langle 1|\psi\rangle\,,$$

$$NOT(|\psi\rangle) \triangleq |0\rangle\langle 1|\psi\rangle + |1\rangle\langle 0|\psi\rangle\,, \qquad Rot_\theta(|\psi\rangle) \triangleq \cos\theta|\psi\rangle + \sin\theta(|1\rangle\langle 0|\psi\rangle - |0\rangle\langle 1|\psi\rangle)$$

$$SWAP(|\psi\rangle) \triangleq \begin{cases} |\psi\rangle & \text{if } \ell(|\psi\rangle) \leq 1, \\ \sum_{a,b\in\{0,1\}} |ba\rangle\langle ab|\psi\rangle & \text{otherwise,} \end{cases}$$

*that is closed under schemes Comp, Branch, and $kQRec_t$, for $k, t \in \mathbb{N}$, defined as*

$$Comp[F, G](|\psi\rangle) \triangleq F(G(|\psi\rangle)),$$

$$Branch[F, G](|\psi\rangle) \triangleq \begin{cases} |\psi\rangle & \text{if } \ell(|\psi\rangle) \leq 1, \\ |0\rangle \otimes F(\langle 0|\psi\rangle) + |1\rangle \otimes G(\langle 1|\psi\rangle) & \text{otherwise,} \end{cases}$$

$$kQRec_t[F, G, H](|\psi\rangle) \triangleq \begin{cases} F(|\psi\rangle) & \text{if } \ell(|\psi\rangle) \leq t, \\ G\big(\sum_{w\in\{0,1\}^k} |w\rangle \otimes F_w\left(\langle w|H(|\psi\rangle)\right)\big) & \text{otherwise,} \end{cases}$$

*where each $F_w \in \{kQRec_t[F, G, H], I\}$.*

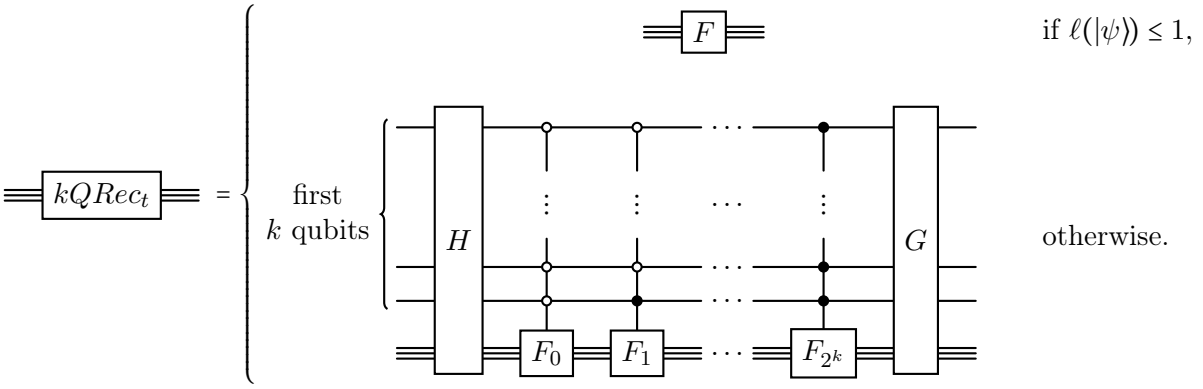
The functions in $\mathscr{Y}_0$ are defined using partial projection on qubit states, as defined in Section 1.1.3. This means that functions are well defined for input states of any dimension, with the transformation being performed on the first qubits of the state. For instance, the $NOT$ function inverts the value of the first input qubit, as shown in the following example.

**Example 3.12.** *Let $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$, then*

$$\begin{aligned} NOT(|\psi\rangle) \triangleq |0\rangle\langle 1|\psi\rangle + |1\rangle\langle 0|\psi\rangle &= \frac{1}{\sqrt{2}}\big(\,|0\rangle\langle 1|00\rangle + |0\rangle\langle 1|11\rangle + |1\rangle\langle 0|00\rangle + |1\rangle\langle 0|11\rangle\,\big) \\ &= \frac{1}{\sqrt{2}}\big(\mathbf{0} + |01\rangle + |10\rangle + \mathbf{0}\big) = \frac{1}{\sqrt{2}}\big(|01\rangle + |10\rangle\big). \end{aligned}$$

The iterative power of $\mathscr{Y}_0$ is expressed in the multi-qubit recursion scheme $kQRec_t$. This scheme takes in a function $F$ for the base case, where the input length is smaller than some constant $t$, and two other functions $G$ and $H$ were $H$ is first applied to the input state, followed by a quantum case statement controlled on the first $k$ qubits, and finally concluded with an application of $G$ to the final state.

We can describe this construction in circuit-like notation as follows, where we write the gate $F_i$ to represent $F_w$ where $i$ is the integer encoded by the binary word $w$.




Each function $F_i$ can either be the identity function or a recursive call to $kQRec_t$, which is performed on a smaller input (in this case, with $k$ fewer input qubits).

Functions in $\mathscr{Y}_0$ are not allowed to increase or pad the input, meaning that the input and output sizes are the same. In order to simulate a QTM with an infinite tape, a polynomial-time padding function is included in the simulation.

**Definition 3.13** (Polytime padding function)**.** *A polytime padding function is any polytime function $\phi : \{0,1\}^* \to \{0,1\}^*$ such that for any $|x\rangle \in \{0,1\}^n$, we have that $\phi(|x\rangle) = |x\rangle \otimes |w(n)\rangle$, where $|w\rangle \in \{0,1\}^*$.*

Padding therefore consists of adding some amount of writing space which only depends on the size of the input. The final class of functions is then defined by composing the functions in $\mathscr{Y}_0$ with polynomial padding.

**Definition 3.14.** *Let $\mathscr{Y}$ denote the following set of functions*

$$\mathscr{Y} \triangleq \left\{ f \circ \phi \text{ such that } f \in \mathscr{Y}_0 \text{ and } \phi \text{ is a polytime padding function.} \right\}$$

The padding allows for simulating the space in the QTM that starts out blank but is nevertheless used during its computation. The fact that a polynomial padding suffices for is given by the argument that a QTM running in $P(n)$ time can only reach $2P(n)+1$ cells, as in Yao's equivalency proof described in Section 3.1.1.

The set of functions $\mathscr{Y}$ can simulate any QTM running in polynomial time, and adding an approximation condition gives us a schematic definition of FBQP.

**Theorem 3.15** ([Yam20])**.** $\mathscr{Y}$ = QP.

**Corollary 3.16.** $\mathscr{Y}_{\geq \frac{2}{3}}$ = FBQP.

The entire argument for the proof of Theorem 3.15 is outside the scope of this thesis, but its structure is of interest as background on implicit characterizations of quantum complexity classes. These kinds of proofs are based on the characterization of quantum Turing machines via quantum circuits, described in Section 3.1.1.

Essentially, one builds a function in the algebra (or a small program, in the case of a programming language) that is capable of executing one local step of the quantum Turing machine: take three adjacent cells, and simulate the transition function. This smaller function is then iterated over the entire tape (using the construction rules in the algebra) so that the step of simulation

is executed in the whole tape (or set of tapes) in such a way that precisely a single step of the machine is executed. Given that the machine is (polynomially) time-bounded, we only need to simulate a finite (polynomial) number of cells. This *single-step* function is then iterated so as to simulate the full execution of the QTM, which will terminate in a polynomial number of steps.

The interest in this result, which partly represents the motivation of this thesis, is in being able to describe quantum polytime without an explicit bound on the number of resources used. The class $\mathscr{Y}$ has the advantages of possessing an intuitive definition: it contains a few basic functions, and then it is simply extended with composition, quantum branching and bounded recursion. Furthermore, there are no explicit resource bounds, nor did we have to handle the restrictions of well-formedness as appeared in the QTM model or uniformity in the case of quantum circuits.

This section concludes the introductory part of the thesis and provides a good opportunity for motivating our work before we begin Part II which will contain our new results.

Our intent in providing a *programming language* characterization of FBQP and FBQPOLYLOG is to bring the tools of implicit complexity closer to the quantum programmer. While the implicit characterizations of [DMZ10, Yam20, Yam22] provide a model-free description of quantum poly- and polylogarithmic-time, they are not of much help to quantum programming, as their restricted expressivity makes it harder to write even relatively simple programs.

Our focus will be not only in showing that our implicit characterizations are sound and complete for the targeted complexity classes, but also that they are able to express typical cases of quantum polytime or polylogtime algorithms. For this reason, we have peppered each section with at least a few examples of relevant programs in order to demonstrate our results and to convince the reader of the expressivity of the language.

Finally, it is relatively standard to use a high-level model such as the quantum Turing machine in order to prove both the soundness and completeness of an implicit characterization. However, in quantum programming one is usually interested in the quantum circuit description of a program, and standard implicit characterizations do not typically provide compilation algorithms, making them less practical for the quantum programmer.

Given the equivalency between quantum circuits and QTMs (Theorem 3.3), the soundness of a characterization also means the existence of a uniform family of quantum circuits that implements each program, and an implicit characterization should provide a compilation algorithm for its programs into circuits of the desired complexity (according to the complexity class). We start Part III by showing that defining such a compilation strategy is a non-trivial problem, and then we provide an algorithm that gives a circuit description of any program in our characterizations of FBQP and FBQPOLYLOG, respecting adequate complexity bounds.

# Part II

# Quantum Implicit Computational Complexity

# 4

# A First-Order Quantum Programming Language (FOQ)

> The limits of my language mean the limits of my world.
>
> – Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*

In this chapter, we introduce FOQ, a First-Order Quantum programming language, which will lay the foundations for Chapters 5 and 6, where we will introduce its fragments PFOQ and LFOQ, respectively characterizing quantum polynomial and polylogarithmic time.

We discuss here FOQ's syntax and semantics, as well as the notions of *well-foundedness* and *reversibility* of a program, and introduce some syntactic sugar to be used in the remaining of this thesis. We exemplify FOQ with two well-known examples of quantum programs, namely the quantum Fourier transform (Figure 4.6) and Grover's search algorithm (Figure 4.7), requiring a polynomial and exponential number of operations, respectively.

## 4.1 Syntax

The syntax of FOQ, given in Figure 4.1, includes basic data types such as Integers, Booleans, Qubits and Operators. A FOQ program has the ability to call first-order (recursive) procedures taking lists of qubits as input parameters.

Let x denote an integer variable and $\bar{q}$ denote a qubit list. The size of the list $\bar{q}$ will be denoted by $|\bar{q}|$. We can refer to the $i$-th qubit in $\bar{q}$ as $\bar{q}[i]$, with $1 \leq i \leq |\bar{q}|$. The empty list, of size 0, will be denoted by nil and $\bar{q} \ominus [i]$ will denote the list obtained by removing the qubit of index $i$ in $\bar{q}$. If no such index exists, then $\bar{q} \ominus [i]$ corresponds to the empty list. This definition of qubit removal ensures that $\ominus$ strictly reduces the size of any non-empty qubit list.

For convenience, we also extend this notation in straightforward ways. Let $n_1, \ldots, n_k \in \mathbb{N}$ be a sequence of integers. Then, we denote by $s \ominus [n_1, \ldots, n_k]$ the list obtained from s by removing the qubits of indexes $n_1, \ldots, n_k$. We also allow for the use of negative integers, such as s[-1], by defining $s[-i] \triangleq s[|s| - i + 1]$, for any $1 \leq i$.

The language also includes some constructs $U^f$ to represent (unary) unitary operators, for some total function $f \in \mathbb{Z} \to [0, 2\pi) \cap \tilde{\mathbb{R}}$ required to be polynomial-time approximable [KF82] (see discussion of this restriction in Section 2.1).

A FOQ program P consists of a sequence of *procedure declarations* D followed by a *program statement* S, with $\varepsilon$ denoting the empty sequence. Let $var(S)$ be the set of variables appearing in the statement S. Let |P| be the size of program P, that is, the total number of symbols in P.

| | | | |
|---|---|---|---|
| (Integers) | i | $\triangleq$ | $n \mid \mathrm{x} \mid \mathrm{i} + n \mid \mathrm{i} - n \mid \lvert\mathrm{s}\rvert \mid \mathrm{i}/2, \quad \text{with } n \in \mathbb{Z}$ |
| (Booleans) | b | $\triangleq$ | $\mathrm{i} > \mathrm{i} \mid \mathrm{i} \geq \mathrm{i} \mid \mathrm{i} = \mathrm{i} \mid \mathrm{b} \wedge \mathrm{b} \mid \mathrm{b} \vee \mathrm{b} \mid \neg \mathrm{b}$ |
| (Qubit Lists) | s | $\triangleq$ | $\mathrm{nil} \mid \bar{\mathrm{q}} \mid \mathrm{s} \ominus [\mathrm{i}] \mid \mathrm{s}^{\boxplus} \mid \mathrm{s}^{\boxminus}$ |
| (Qubits) | q | $\triangleq$ | $\mathrm{s}[\mathrm{i}]$ |
| (Operators) | $\mathrm{U}^f(\mathrm{i})$ | $\triangleq$ | $\mathrm{NOT} \mid \mathrm{R}_Y^f(\mathrm{i}) \mid \mathrm{Ph}^f(\mathrm{i}), \quad \text{with } f \in \mathbb{Z} \to [0, 2\pi) \cap \tilde{\mathbb{R}}$ |
| (Statements) | S | $\triangleq$ | $\mathbf{skip};$ $\mid \mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i});$ $\mid \mathrm{S}\ \mathrm{S}$ $\mid \mathbf{if}\ \mathrm{b}\ \mathbf{then}\ \{\mathrm{S}\}\ \mathbf{else}\ \{\mathrm{S}\}$ $\mid \mathbf{qcase}\ \mathrm{q}\ \mathbf{of}\ \{0 \to \mathrm{S}, 1 \to \mathrm{S}\}$ $\mid \mathbf{call}\ \mathtt{proc}[\mathrm{i}](\mathrm{s}_1, \ldots, \mathrm{s}_k);$ |
| (Proc. declar.) | D | $\triangleq$ | $\varepsilon \mid \mathbf{decl}\ \mathtt{proc}[\mathrm{x}](\bar{\mathrm{q}}_1, \ldots, \bar{\mathrm{q}}_k)\{\mathrm{S}\}, \mathrm{D}$ |
| (Programs) | P | $\triangleq$ | $\mathrm{D} :: \mathrm{S}$ |

Figure 4.1: Syntax of FOQ programs.

A procedure declaration $\mathbf{decl}\ \mathtt{proc}[\mathrm{x}](\bar{\mathrm{q}}_1, \ldots, \bar{\mathrm{q}}_k)\{\mathrm{S}\}$ takes as input the qubit lists $\bar{\mathrm{q}}_1, \ldots, \bar{\mathrm{q}}_k$ and an (optional) integer parameter x. S is called the *procedure statement*, proc is the *procedure name* and belongs to a countable set Procedures. We will write $\mathrm{S}^{\mathtt{proc}}$ to refer to S and $\mathtt{proc} \in \mathrm{P}$ holds if proc is declared in D.

Besides the no-op instruction **skip**;, the most basic statement is the operation of an (unary) operator on a qubit ($\mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i});$). The considered operators are NOT, $\mathrm{R}_Y^f(\mathrm{i})$, and $\mathrm{Ph}^f(\mathrm{i})$, which will correspond to the *NOT*, rotation and phase-shift gates introduced in Section 2.3.1.

Statements also include sequences, (classical) conditionals, *quantum cases*, and *procedure calls* which take an integer input i and a qubit list s ($\mathbf{call}\ \mathtt{proc}[\mathrm{i}](\mathrm{s});$). A quantum case $\mathbf{qcase}\ \mathrm{q}\ \mathbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ provides a quantum control feature that will execute statements $\mathrm{S}_0$ and $\mathrm{S}_1$ in superposition according to the state of q. A quick and easy example of the use of the **qcase** is in simulating the *CNOT* gate, both with positive and negative controls, as defined in Figure 4.2. We further elaborate on the use of the **qcase** in the remaining examples of the same figure, namely in the simulation of the *SWAP* and Toffoli gates. For ease of notation, the standard *CNOT* and Toffoli gates are denoted by $\mathrm{CNOT}(\mathrm{q}_1, \mathrm{q}_2) \triangleq \mathrm{CNOT}_1(\mathrm{q}_1, \mathrm{q}_2)$ and $\mathrm{TOF}(\mathrm{q}_1, \mathrm{q}_2, \mathrm{q}_3) \triangleq \mathrm{TOF}_{11}(\mathrm{q}_1, \mathrm{q}_2, \mathrm{q}_3)$.

Furthemore, we may extend the use of the quantum control case in order to handle multiple control qubits in a straightforward way. For instance,

$$\begin{aligned}
&\mathbf{qcase}\ \mathrm{s}[\mathrm{i}_1, \mathrm{i}_2]\ \mathbf{of}\ \{\ 00 \to \mathrm{S}_{00}, \\
&\qquad\qquad 01 \to \mathrm{S}_{01}, \\
&\qquad\qquad 10 \to \mathrm{S}_{10}, \\
&\qquad\qquad 11 \to \mathrm{S}_{11}\ \}
\end{aligned}
\quad \triangleq \quad
\begin{aligned}
&\mathbf{qcase}\ \mathrm{s}[\mathrm{i}_1]\ \mathbf{of}\ \{\ 0 \to \mathbf{qcase}\ \mathrm{s}[\mathrm{i}_2]\ \mathbf{of}\ \{0 \to \mathrm{S}_{00}, 1 \to \mathrm{S}_{01}\}, \\
&\qquad\qquad 1 \to \mathbf{qcase}\ \mathrm{s}[\mathrm{i}_2]\ \mathbf{of}\ \{0 \to \mathrm{S}_{10}, 1 \to \mathrm{S}_{11}\}\ \}
\end{aligned}$$

We will restrict our study to *well-formed* programs, where procedure names declared in D are pairwise distinct and, for each procedure call, the procedure name is declared in D.

It is often desirable to build large programs from smaller ones, and to be able to reason about their properties. To that end, we introduce the following rules that allow for combining FOQ programs using the syntax for statements.

**Definition 4.1** (Syntactic sugar for programs)**.** *For $i \neq j$, let $\mathrm{D}_i$ and $\mathrm{D}_j$ denote procedure declarations with no collisions in procedure names. Then, for $\mathrm{P}_i = \mathrm{D}_i :: \mathrm{S}_i$, we define the following*

| | | |
|---|---|---|
| *Controlled-NOT* | $\mathrm{CNOT}_0(\mathrm{q}_1,\mathrm{q}_2)$ | $\triangleq$ **qcase** $\mathrm{q}_1$ **of** $\{0 \to \mathrm{q}_2 \mathrel{*}= \mathrm{NOT};, 1 \to \textbf{skip};\}$ |
| | $\mathrm{CNOT}_1(\mathrm{q}_1,\mathrm{q}_2)$ | $\triangleq$ **qcase** $\mathrm{q}_1$ **of** $\{0 \to \textbf{skip};, 1 \to \mathrm{q}_2 \mathrel{*}= \mathrm{NOT};\}$ |
| *Swap* | $\mathrm{SWAP}(\mathrm{q}_1,\mathrm{q}_2)$ | $\triangleq \mathrm{CNOT}_1(\mathrm{q}_1,\mathrm{q}_2)\ \mathrm{CNOT}_1(\mathrm{q}_2,\mathrm{q}_1)\ \mathrm{CNOT}_1(\mathrm{q}_1,\mathrm{q}_2)$ |
| *Toffoli* | $\mathrm{TOF}_{0y}(\mathrm{q}_1,\mathrm{q}_2,\mathrm{q}_3)$ | $\triangleq$ **qcase** $\mathrm{q}_1$ **of** $\{0 \to \mathrm{CNOT}_y(\mathrm{q}_2,\mathrm{q}_3), 1 \to \textbf{skip};\}$ |
| | $\mathrm{TOF}_{1y}(\mathrm{q}_1,\mathrm{q}_2,\mathrm{q}_3)$ | $\triangleq$ **qcase** $\mathrm{q}_1$ **of** $\{0 \to \textbf{skip};, 1 \to \mathrm{CNOT}_y(\mathrm{q}_2,\mathrm{q}_3)\}$ |

Figure 4.2: FOQ syntactic sugar.

*rules for program creation, where*

$$\mathrm{P}_1\ \mathrm{P}_2 \triangleq \mathrm{D}_1, \mathrm{D}_2 :: \mathrm{S}_1\,\mathrm{S}_2,$$
$$\textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{P}_1\ \textbf{else}\ \mathrm{P}_2 \triangleq \mathrm{D}_1, \mathrm{D}_2 :: \textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2,$$
$$\textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{P}_1, 1 \to \mathrm{P}_2\} \triangleq \mathrm{D}_1, \mathrm{D}_2 :: \textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_1, 1 \to \mathrm{S}_2\}.$$

*Furthermore, let* $\mathtt{proc}$ *be a procedure with body* $\mathrm{S}^{\mathtt{proc}}$*, such that* $\mathrm{P}_1, \ldots, \mathrm{P}_k \in \mathrm{S}^{\mathtt{proc}}$*. Then*

$$\mathrm{D}_0, \textbf{decl}\ \mathtt{proc}[\mathrm{x}]\{\mathrm{S}^{\mathtt{proc}}\} :: \mathrm{S} \triangleq \mathrm{D}_0, \mathrm{D}_1, \ldots, \mathrm{D}_k, \textbf{decl}\ \mathtt{proc}[\mathrm{x}]\{\mathrm{S}^{\mathtt{proc}}[\mathrm{S}_i/\mathrm{P}_i]\} :: \mathrm{S},$$

*where in* $\mathrm{S}^{\mathtt{proc}}[\mathrm{S}_i/\mathrm{P}_i]$ *we replace each instance of* $\mathrm{P}_i$ *with* $\mathrm{S}_i$*, for* $i = 1, \ldots, k$*.*

In the following chapters, we will see that these construction rules naturally preserve certain program properties. For instance, they will allow for us to quickly conclude that combining polytime programs in safe ways results in a larger polytime program.

## 4.2 Semantics

We now define the *semantics* of FOQ programs, starting with the most basic statement: the unitary operation.

Each operator U of a unitary application $\mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i});$ comes with a function $[\![\mathrm{U}]\!]$ assigning a unitary matrix $[\![\mathrm{U}]\!](f)(n) \in \tilde{\mathbb{C}}^{2\times 2}$ to each integer $n$ and polytime-approximable total function $f \in \mathbb{Z} \to \tilde{\mathbb{C}}$ (sometimes also written $g$ when this is clear from context). We restrict ourselves to three kinds of gates: the phase gate $Ph$, rotation gate $\mathrm{R_Y}$ and $NOT$ gate NOT, with semantics defined as follows:

$$[\![\mathrm{Ph}]\!](f)(n) \triangleq \begin{pmatrix} 1 & 0 \\ 0 & e^{ig(n)} \end{pmatrix} \quad [\![\mathrm{R_Y}]\!](f)(n) \triangleq \begin{pmatrix} \cos(f(n)) & -\sin(f(n)) \\ \sin(g(n)) & \cos(f(n)) \end{pmatrix} \quad [\![\mathrm{NOT}]\!](\cdot)(\cdot) \triangleq \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

This choice of gates is inspired by the implicit characterization of [Yam20] and justified by the fact that and $\mathrm{Ph}^f(\mathrm{i})$ as basic operators is justified by the fact that, for the constant and polynomial-time approximable function $g(x) \triangleq \pi/4 \in \mathbb{Z} \to [0, 2\pi) \cap \tilde{\mathbb{R}}$, these operators simulate the following set of one-qubit unitary gates

$$[\![\mathrm{R}_Y^g]\!](n) = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}, \qquad [\![\mathrm{Ph}^g]\!](n) \triangleq \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix},$$

which, combined with a construction for controlled gates, is universal for quantum computing [BMP+99], as discussed in Section 2.3.2. For instance, the Hadamard gate can be derived as

$$H \triangleq \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} = [\![\mathrm{NOT}]\!](\cdot)(\cdot) \times [\![\mathrm{R}_Y]\!](g)(0).$$

Given the usefulness of the Hadamard, we will make use of the following shorthand notation:

$$\mathrm{q} \mathrel{*}= \mathrm{H}; \quad \triangleq \quad (\mathrm{q} \mathrel{*}= \mathrm{R}_Y^g(0);\ \mathrm{q} \mathrel{*}= \mathrm{NOT};).$$

Let $\mathbb{B}$ be the set of Boolean values $b \in \{\mathbf{false}, \mathbf{true}\}$. For a given set $X$, let $\mathcal{L}(X)$ be the set of lists of elements in $X$. Let $l = [x_1, \ldots, x_m]$, with $x_1, \ldots, x_m \in X$, denote a list of $m$ elements in $\mathcal{L}(X)$ and $[\,]$ be the empty list (when $m = 0$). Lists of integers will be used to represent pointers to qubits in a global memory.

We interpret each basic data type $\tau$ as follows:

$$[\![\text{Integers}]\!] \triangleq \mathbb{Z}, \qquad [\![\text{Booleans}]\!] \triangleq \mathbb{B}, \qquad [\![\text{Qubit Lists}]\!] \triangleq \mathcal{L}(\mathbb{N}), \qquad [\![\text{Qubits}]\!] \triangleq \mathbb{N}.$$

Each basic operation $\mathrm{op} \in \{+, -, /2, >, \geq, =, \wedge, \vee, \neg\}$ of arity $n$, with $1 \leq n \leq 2$, has a type signature

$$\mathrm{op} : \tau_1 \times \ldots \times \tau_n \to \tau$$

fixed by the program syntax and computes a fixed total function

$$[\![\mathrm{op}]\!] \in [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to [\![\tau]\!].$$

For example,

$$\begin{aligned} +&: \text{Integers} \times \text{Integers} \to \text{Integers} & [\![+]\!] &\triangleq (n, m) \mapsto n + m,\\ /2&: \text{Integers} \to \text{Integers} & [\![/2]\!] &\triangleq n \mapsto \lceil n/2 \rceil. \end{aligned}$$

Constants are treated as particular operators of arity 0. For each basic type $\tau$, the reduction $\Downarrow_{[\![\tau]\!]}$ is a map

$$\Downarrow_{[\![\tau]\!]} : \tau \times (Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N})) \to [\![\tau]\!].$$

Intuitively, it maps an expression of type $\tau$ and a context function $f$ to the value of the expression in $[\![\tau]\!]$. These reductions are defined in Figure 4.3, where e and d denote either an integer expression i or a boolean expression b. For instance,

$$\begin{aligned} (\bar{\mathrm{q}}[2], \bar{\mathrm{q}} \mapsto [1,4,5]) &\Downarrow_{\mathbb{N}} 4 && \text{the second qubit of the list has global index 4}\\ (\bar{\mathrm{q}} \ominus [3], \bar{\mathrm{q}} \mapsto [1,4,5]) &\Downarrow_{\mathcal{L}(\mathbb{N})} [1,4] && \text{the third qubit has been removed}\\ (\bar{\mathrm{q}} \ominus [4], \bar{\mathrm{q}} \mapsto [1,4,5]) &\Downarrow_{\mathcal{L}(\mathbb{N})} [\,] && [\,] \text{ is used for errors on type } \mathcal{L}(\mathbb{N})\\ (\bar{\mathrm{q}}[4], \bar{\mathrm{q}} \mapsto [1,4,5]) &\Downarrow_{\mathbb{N}} 0 && \text{index 0 is used for error on type } \mathbb{N} \end{aligned}$$

Note that, in rule ($\mathrm{Rm}_\notin$), if we try to delete an undefined index then we return the empty list, and, in rule ($\mathrm{Qu}_\notin$), if we try to access an undefined qubit index then we return the value 0 (defined indexes will always be positive).

Given a program P with input qubit list lengths $n_i$, let the *length* of P be a function mapping each qubit variable $\bar{\mathrm{q}}_i \in Var(\mathrm{P})$ to an integer $\mathtt{len}(\bar{\mathrm{q}}_i) = n_i$. We write $\mathtt{len}_\mathrm{P}$ as a shorthand for $\sum_{\bar{\mathrm{q}} \in Var(\mathrm{P})} \mathtt{len}(\bar{\mathrm{q}})$ and define

$$\mathtt{len}_\mathrm{P}^{<\bar{\mathrm{q}}} \triangleq \sum_{\bar{\mathrm{q}}' \in Var(\mathrm{P}),\ \bar{\mathrm{q}}' < \bar{\mathrm{q}}} \mathtt{len}(\bar{\mathrm{q}}').$$

$$\frac{\forall i \le n,\ (\mathrm{e}_i, f) \Downarrow_{[\![\tau_i]\!]} x_i}{(\mathrm{op}(\mathrm{e}_1, \ldots, \mathrm{e}_n), f) \Downarrow_{[\![\mathrm{op}(\tau_1,\ldots,\tau_n)]\!]} [\![\mathrm{op}]\!](x_1, \ldots, x_n)} \qquad \frac{(\mathrm{s}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_n]}{(|\mathrm{s}|, f) \Downarrow_{\mathbb{Z}} n}$$

$$\frac{\bar{\mathrm{q}} \in Var(\mathrm{P})}{(\bar{\mathrm{q}}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} f(\bar{\mathrm{q}})} \qquad \frac{(\mathrm{s}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \quad (\mathrm{i}, f) \Downarrow_{\mathbb{Z}} k \in \{1, \ldots, m\}}{(\mathrm{s} \ominus [\mathrm{i}], f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_{k-1}, x_{k+1}, \ldots, x_m]}$$

$$\frac{(\mathrm{s}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \quad (\mathrm{i}, f) \Downarrow_{\mathbb{Z}} k \notin \{1, \ldots, m\}}{(\mathrm{s} \ominus [\mathrm{i}], f) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]} \qquad \frac{(\mathrm{s}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \quad m > 1}{(\mathrm{s}^{\boxminus}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_{\lceil m/2 \rceil}]}$$

$$\frac{(\mathrm{s}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \quad m > 1}{(\mathrm{s}^{\boxplus}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_{\lceil m/2 \rceil + 1}, \ldots, x_m]} \qquad \frac{(\mathrm{s}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \quad (\mathrm{i}, f) \Downarrow_{\mathbb{Z}} k \notin \{1, \ldots, m\}}{(\mathrm{s}[\mathrm{i}], f) \Downarrow_{\mathbb{N}} 0}$$

$$\frac{(\mathrm{s}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} l \quad |l| \le 1}{(\mathrm{s}^{\pm}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]} \qquad \frac{(\mathrm{s}, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [x_1, \ldots, x_m] \quad (\mathrm{i}, f) \Downarrow_{\mathbb{Z}} k \in \{1, \ldots, m\}}{(\mathrm{s}[\mathrm{i}], f) \Downarrow_{\mathbb{N}} x_k}$$

Figure 4.3: Semantics of expressions.

A *configuration* $c$ of program P over $\mathtt{len}_{\mathrm{P}}$ qubits is of the shape

$$(\mathrm{S}, |\psi\rangle, A, f) \in (\text{Statements} \cup \{\top, \bot\}) \times \mathcal{H}_{2^{\mathtt{len}_{\mathrm{P}}}} \times (Var(\mathrm{P}) \to \mathcal{P}(\mathbb{N})) \times (Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N})),$$

where $\top$ and $\bot$ are two special symbols denoting termination and error, respectively, where $|\psi\rangle \in \mathcal{H}_{2^{\mathtt{len}_{\mathrm{P}}}}$ is a quantum state, and where, for each qubit list $\bar{\mathrm{q}} \in Var(\mathrm{P})$, $A(\bar{\mathrm{q}})$ is the set of qubit pointers accessible from $\bar{\mathrm{q}}$ and $f(\bar{\mathrm{q}})$ is the list of qubit pointers assigned to $\bar{\mathrm{q}}$.

Given a qubit q such that $Var(\mathrm{q}) = \{\bar{\mathrm{q}}\}$, we write $A(\mathrm{q})$ as a shorthand for $A(\bar{\mathrm{q}})$ and we write $A_{\mathrm{q}\setminus\{n\}}$ for the function $A' \in Var(\mathrm{P}) \to \mathcal{P}(\mathbb{N})$ defined by $A'(\bar{\mathrm{q}}') \triangleq A(\bar{\mathrm{q}}')$, $\forall \bar{\mathrm{q}}' \neq \bar{\mathrm{q}}$, and $A'(\bar{\mathrm{q}}) \triangleq A(\bar{\mathrm{q}})\setminus\{n\}$.

Given a program $\mathrm{P} \triangleq \mathrm{D} :: \mathrm{S}$, with $n = \mathtt{len}_{\mathrm{P}}$, let $\mathrm{Conf}_n$ be the set of configurations over $n$ qubits. The *initial configuration* in $\mathrm{Conf}_n$ on input state $|\psi\rangle \in \mathcal{H}_{2^n}$ is given by

$$c_{init}(|\psi\rangle) \triangleq (\mathrm{S}, |\psi\rangle, \bar{\mathrm{q}} \mapsto \{1, \ldots, \mathtt{len}(\bar{\mathrm{q}})\}, \bar{\mathrm{q}} \mapsto [1, \ldots, \mathtt{len}(\bar{\mathrm{q}})]).$$

### 4.2.1 Operational semantics

The program big-step semantics $\longrightarrow$, described in Figure 4.4, is defined as a relation in $\bigcup_{n\in\mathbb{N}} \mathrm{Conf}_n \times \mathrm{Conf}_n$. A program is said to be *error-free* if there is no initial configuration $c_{init}(|\psi\rangle)$ such that

$$c_{init}(|\psi\rangle) \longrightarrow (\bot, |\psi'\rangle, A, l).$$

We write $[\![\mathrm{P}]\!](|\psi\rangle) = |\psi'\rangle$, whenever $c_{init}(|\psi\rangle) \xrightarrow{m} (\top, |\psi'\rangle, A, l)$ holds for some $m$. $(\top, |\psi'\rangle, A, l)$ is called a *terminal configuration*. Let $\mathcal{H} \triangleq \bigcup_n \mathcal{H}_{2^n}$, a program *terminates* if $[\![\mathrm{P}]\!]$ is a total function in $\mathcal{H} \to \mathcal{H}$.

We now give a brief intuition on the rules of Figure 4.4.

$$\frac{}{(\textbf{skip};,|\psi\rangle,A,f)\xrightarrow{0}(\top,|\psi\rangle,A,f)}\ (\text{Skip}) \qquad \frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\notin A(\mathrm{q})}{(\mathrm{q}\ *\!=\mathrm{U}^{g}(\mathrm{j});,|\psi\rangle,A,f)\xrightarrow{0}(\bot,|\psi\rangle,A,f)}\ (\text{Asg}_{\bot})$$

$$\frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\in A(\mathrm{q}) \qquad (\mathrm{i},f)\Downarrow_{\mathbb{N}} m \qquad j=\mathtt{len}_{\mathrm{P}}^{<\mathrm{q}}+n}{(\mathrm{q}\ *\!=\mathrm{U}^{g}(\mathrm{i});,|\psi\rangle,A,f)\xrightarrow{0}(\top,I_{2^{j-1}}\otimes[\![\mathrm{U}]\!](g)(m)\otimes I_{2^{\mathtt{len}_{\mathrm{P}}-j}}|\psi\rangle,A,f)}\ (\text{Asg}_{\top})$$

$$\frac{(\mathrm{S}_0,|\psi\rangle,A,f)\xrightarrow{m_1}(\top,|\psi'\rangle,A,f) \qquad (\mathrm{S}_1,|\psi'\rangle,A,l)\xrightarrow{m_2}(\diamond,|\psi''\rangle,A,f) \qquad \diamond\in\{\top,\bot\}}{(\mathrm{S}_0\ \mathrm{S}_1,|\psi\rangle,A,f)\xrightarrow{m_1+m_2}(\diamond,|\psi''\rangle,A,f)}\ (\text{Seq}_{\diamond})$$

$$\frac{(\mathrm{S}_0,|\psi\rangle,A,f)\xrightarrow{m}(\bot,|\psi\rangle,A,f)}{(\mathrm{S}_0\ \mathrm{S}_1,|\psi\rangle,A,f)\xrightarrow{m}(\bot,|\psi\rangle,A,f)}\ (\text{Seq}_{\bot})$$

$$\frac{(\mathrm{b},f)\Downarrow_{\mathbb{B}} b \qquad (\mathrm{S}_b,|\psi\rangle,A,f)\xrightarrow{m_b}(\diamond,|\psi'\rangle,A,f) \qquad \diamond\in\{\top,\bot\}}{(\textbf{if}\ \mathrm{b}\ \textbf{then}\ \{\mathrm{S}_1\}\ \textbf{else}\ \{\mathrm{S}_0\},|\psi\rangle,A,f)\xrightarrow{m_b}(\diamond,|\psi'\rangle,A,f)}\ (\text{If})$$

$$\frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\in A(\mathrm{q}) \qquad \forall k\in\mathbb{B},\ (\mathrm{S}_k,|\psi\rangle,A_{\mathrm{q}\setminus\{n\}},f)\xrightarrow{m_k}(\top,|\psi_k\rangle,A_{\mathrm{q}\setminus\{n\}},f) \qquad j=\mathtt{len}_{\mathrm{P}}^{<\mathrm{q}}+n}{(\textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0\to\mathrm{S}_0,1\to\mathrm{S}_1\},|\psi\rangle,A,f)\xrightarrow{\max_{k\in\{0,1\}}m_k}(\top,\textstyle\sum_{k\in\{0,1\}}|k\rangle_j\langle k|_j|\psi_k\rangle,A,f)}\ (\text{Case}_{\top})$$

$$\frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\in A(\mathrm{q}) \qquad \exists k\in\mathbb{B},\ (\mathrm{S}_k,|\psi\rangle,A_{\mathrm{q}\setminus\{n\}},f)\xrightarrow{m_k}(\bot,|\psi_k\rangle,A_{\mathrm{q}\setminus\{n\}},f)}{(\textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0\to\mathrm{S}_0,1\to\mathrm{S}_1\},|\psi\rangle,A,f)\xrightarrow{\max_{k\in\{0,1\}}m_k}(\bot,|\psi\rangle,A,f)}\ (\text{Case}_{\bot})$$

$$\frac{(\mathrm{q},f)\Downarrow_{\mathbb{N}} n\notin A(\mathrm{q})}{(\textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0\to\mathrm{S}_0,1\to\mathrm{S}_1\},|\psi\rangle,A,f)\xrightarrow{0}(\bot,|\psi\rangle,A,f)}\ (\text{Case}_{\notin})$$

$$\frac{\forall j\le n,\ (\mathrm{s}_j,f)\Downarrow_{\mathcal{L}(\mathbb{N})} l_j\neq[\,] \qquad (\mathrm{S}^{\mathtt{proc}}\{\mathrm{s}_j/\bar{\mathrm{q}}_j\},|\psi\rangle,A,f)\xrightarrow{m}(\diamond,|\psi'\rangle,A,f) \qquad \diamond\in\{\top,\bot\}}{(\textbf{call}\ \mathtt{proc}(\mathrm{s}_1,\ldots,\mathrm{s}_n);,|\psi\rangle,A,f)\xrightarrow{m+1}(\diamond,|\psi'\rangle,A,f)}\ (\text{Call}_{\diamond})$$

$$\frac{\exists j\le n,\ (\mathrm{s}_j,f)\Downarrow_{\mathcal{L}(\mathbb{N})}[\,]}{(\textbf{call}\ \mathtt{proc}(\mathrm{s}_1,\ldots,\mathrm{s}_n);,|\psi\rangle,A,f)\xrightarrow{m+1}(\top,|\psi\rangle,A,f)}\ (\text{Call}_{\diamond})$$

Figure 4.4: Semantics of statements.

- Rules ($\text{Asg}_\perp$) and ($\text{Asg}_\top$) evaluate the application of a unitary operator, corresponding to $\text{U}^f(\text{j})$, to a qubit s[i]. For that purpose, they evaluate the index $n$ of s[i] in the global memory. Rule ($\text{Asg}_\perp$) deals with the error case, where the corresponding qubit is not allowed to be accessed. Rule ($\text{Asg}_\top$) deals with the success case: the new quantum state is obtained by applying the result of tensoring the evaluation of $\text{U}^f(\text{j})$ to the right index.
- Rules ($\text{Seq}_\diamond$) and ($\text{Seq}_\perp$) evaluate the sequence of statements, depending on whether an error occurs or not.
- The (If) rule deals with classical conditionals in a standard way.
- The three rules ($\text{Case}_\top$), ($\text{Case}_\perp$), and ($\text{Case}_\notin$) evaluate the qubit index $n$ of the control qubit s[i]. They then check whether this index belongs to the set of accessible qubits (is $n$ in $A$?). If so, intuitively, the two statements $\text{S}_0$ and $\text{S}_1$ are evaluated in superposition, on the projected state $\langle 0|_n|\psi\rangle$ and $\langle 1|_n|\psi\rangle$, respectively. During these evaluations, the index $n$ cannot be accessed anymore.
- The rule ($\text{Call}_{[\,]}$) treats the base case of a procedure call when the qubit list parameter is empty. In the non-empty case, rule ($\text{Call}_\diamond$) evaluates the qubit list parameter s to $l'$ and the integer parameter x to $n$. It returns the result of evaluating the procedure statement $\text{S}^{\texttt{proc}}\{n/\text{x}\}$, where $n$ has been substituted to x, with regards to the updated qubit pointer list $l'$.

Note that if a program terminates then it is obviously error-free but the converse property does not hold. Every program P can be efficiently transformed into an error-free program $\text{P}_{\neg\perp}$ such that $\forall|\psi\rangle$, if $[\![\text{P}]\!](|\psi\rangle)$ is defined then $[\![\text{P}]\!](|\psi\rangle) = [\![\text{P}_{\neg\perp}]\!](|\psi\rangle)$. For example, an assignment s[i] $*=$ $\text{U}^f(\text{j})$; can be transformed into the conditional statement

$$\textbf{if } ((0 < \text{i}) \wedge (\text{i} \leq |\text{s}|)) \textbf{ then } \text{s[i]} *= \text{U}^f(\text{j}); \textbf{ else skip};$$

For this reason, we will only consider error-free programs in the rest of this work. Furthermore, to ensure that there is no cloning of qubit variables, we enforce that in each procedure call **call** proc[x]($\text{s}_1, \ldots, \text{s}_k$);, that $\forall i \neq j$, $Var(\text{s}_i) \neq Var(\text{s}_j)$.

### 4.2.2 Derivation Tree and Runtime of a Program

Given a configuration $c$ with regards to a fixed program P, $\pi_\text{P} \rhd c$ denotes the *derivation tree* of P, the tree of root $c$ whose children are obtained by applying the rules of Figures 4.3 and 4.4 on configuration $c$ with respect to P. We write $\pi$ instead of $\pi_\text{P} \rhd c$ when P and $c$ are clear from the context. Note that a derivation tree $\pi$ can be infinite in the particular case of a non-terminating computation. We will write $\pi \trianglelefteq \pi'$ to denote that $\pi$ is a subtree of $\pi'$.

In the case of a terminating computation $\pi \rhd c$, there exists a terminal configuration $c'$ and a level $m \in \mathbb{N}$ such that $c \xrightarrow{m} c'$ holds. In this case, the level of $\pi$ is defined as $\text{lv}_\pi \triangleq m$. Given a FOQ program $\text{P}(\bar{\text{q}}_1, \ldots, \bar{\text{q}}_k)$ that terminates, $\text{Time}_\text{P}$ is a total function in $\mathbb{N}^k \to \mathbb{N}$ defined as

$$\text{Time}_\text{P}([n_1, \ldots, n_k]) \triangleq \max_{|\psi\rangle \in \mathcal{H}_{2^{n_1+\cdots+n_k}}} \text{lv}_{\pi_\text{P} \rhd c_{init}(|\psi\rangle)}.$$

Intuitively, $\text{Time}_\text{P}([n_1, \ldots, n_k])$ corresponds to the maximal number of non-superposed procedure calls in any program execution where the input qubit list corresponding to each variable $\bar{\text{q}}_i$ has size $n_i$.

**Example 4.2.** *For all* $n \in \mathbb{N}$, $\mathrm{Time}_{\mathtt{QFT}}([n]) = \frac{(n+1)(n+2)}{2} + \lfloor \frac{n}{2} \rfloor + 1$. *Indeed, on a qubit list of size* $n$*, procedure* `rec` *is called recursively* $n+1$ *times and makes* $n+1$ *calls to procedure* `rot` *on qubit lists of size* $n$*,* $n-1$*, ..., and* $1$*. On qubit lists of size* $n$*,* `rot` *performs* $n$ *recursive calls. Hence, the total number of calls to* `rot` *is equal to* $\sum_{i=1}^{n} i$*. Finally, on a qubit list of size* $n$*, procedure* `inv` *does* $\lfloor \frac{n}{2} \rfloor + 1$ *recursive calls.*

We now show that a FOQ program, if it terminates and therefore has a well-defined semantics for every input, admits a QTM that simulates it, whose time-complexity is related to the $\mathrm{Time}_{\mathrm{P}}$ function. For simplicity, we consider programs with only one qubit list variable.

**Lemma 4.3.** *For any terminating* FOQ *program* P*, there exists a stationary QTM* $M$ *that computes* $[\![\mathrm{P}]\!]$ *in time* $O(n + n \times \mathrm{Time}_{\mathrm{P}}(n))$.

*Proof.* Consider a terminating FOQ program $\mathrm{P} = \mathrm{D} :: \mathrm{S}$. We build a 4-tape QTM $M$ computing $[\![\mathrm{P}]\!]$ inductively on the statement S. Fix $\Sigma \triangleq \{0, 1, \#, \|, \&\}$, where $\#$ is the blank symbol and where $\|$ and $\&$ are special separation symbols for encoding stacks. The input tape $t_{in}$ of $M$ contains a word in $\{0,1,\#\}^n$ encoding the quantum state. The 3 working tapes are $t_{call}$, $t_l$, and $t_{\mathbb{K}}$ for storing the integer values of a procedure call, the list of qubit pointers, and intermediate classical computations, respectively, as words in $\Sigma^*$. The configurations of $M$ will be in $Q \times (\Sigma^*)^4 \times \mathbb{Z}^4$, for some finite set of states $Q$. In particular, the initial configuration is

$$(s_0, w, \varepsilon, \varepsilon, \varepsilon, 0, 0, 0, 0),$$

with $w \in \{0,1\}^n$ encoding a quantum state of length $n \in \mathbb{N}$; the tapes $t_{call}$, $t_l$, and $t_{\mathbb{K}}$ are initially empty ($\varepsilon$). The tape heads all start on the first cells indexed by 0. For $m \in \mathbb{Z}$, let $t(m)$ denote the symbol at position $m$ on tape $t$. Given a word $w \in \Sigma^*$ and a tape $t$, $tw$ denotes that the content of $t$ ends with the word $w$.

By abuse of notation, let $[\![\mathrm{e}]\!]$ denote the result of evaluating the expression e with respect to the machine current configuration. Also, we will assume that deterministic computations, such as taking tape $t\|[\![\mathrm{i}]\!]$ and appending $f([\![\mathrm{i}]\!])$, for any function $f$, are done by a reversible Turing machine [Ben73], as reversible TMs are well-formed QTMs [BV97, Theorem 4.2].

We now describe a QTM $M$ simulating P inductively on the statement S.

- The **skip**; statement is trivial.
- If $\mathrm{S} = \mathrm{q} \mathrel{*}= \mathrm{U}^g(\mathrm{j});$, $M$ appends $[\![\mathrm{q}]\!]$ to $t_{\mathbb{K}}$. As the program terminates, $t_{in}([\![\mathrm{q}]\!]) \neq \#$ and the transition function is set to:

  $$\forall a \in \{0,1\},\ \delta(s_{\mathrm{S}}, t_{in}([\![\mathrm{q}]\!]), s_{next(\mathrm{S})}, a, N) \triangleq \langle t_{in}([\![\mathrm{q}]\!]) | [\![\mathrm{U}]\!](g)([\![\mathrm{j}]\!]) | a \rangle$$

  where $s_{\mathrm{S}}$ is the state before executing the assignment when the head of the input tape has been moved to position $[\![\mathrm{q}]\!]$, and $s_{next(\mathrm{S})}$ is the state just after executing the assignment. Finally, the machine erases $\|[\![\mathrm{q}]\!]$ at the end of $t_{\mathbb{K}}$, leaves its head in the last non-blank cell of $t_{\mathbb{K}}$, and moves the head in $t_{in}$ back to the initial cell. Program P has $\mathrm{Time}_{\mathrm{P}}(n) = 0$ and the simulating machine runs in time $O(n)$.

For the remaining statements, assume by induction hypothesis the existence of two stationary QTMs $M_1$ and $M_2$ that compute functions $[\![\mathrm{P}_1]\!]$ and $[\![\mathrm{P}_2]\!]$, respectively, with $\mathrm{P}_1 \triangleq \mathrm{D} :: \mathrm{S}_1$ and $\mathrm{P}_2 \triangleq \mathrm{D} :: \mathrm{S}_2$. By induction hypothesis $M_1$ and $M_2$ run in time $O(n + n \times \mathrm{Time}_{\mathrm{P}_i}(n))$. States of $M_i$ will be denoted by $s^i$, for $i \in \{1,2\}$. We assume without loss of generality that machines $M_i$ are synchronized (by Lemma 2.11) and therefore that they halt in exactly the same time for any quantum input of equal length.

- Consider the case $\mathrm{S} = \mathrm{S}_1\,\mathrm{S}_2$. Machine $M$ is defined as in [BV97, Dovetailing Lemma], with the initial state $s_0 \triangleq s_0^1$, its final state $s_\top \triangleq s_\top^2$, and the two machines are composed by setting $s_\top^1 = s_0^2$. The machine $M$ is stationary, well-formed and it is well-behaved since the running time of $M_2$ only depends on $n$ and the output of $M_1$ contains a superposition of equally sized quantum states. $M$ computes $[\![\mathrm{P}]\!]$ in time

$$O(n + n \times \mathrm{Time}_{\mathrm{P}_1}(n))) + O(n + n \times \mathrm{Time}_{\mathrm{P}_2}(n)) = O(n + n \times \mathrm{Time}_{\mathrm{P}}(n)).$$

- For the conditional $\mathrm{S} = \textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{S}_1\ \textbf{else}\ \mathrm{S}_2$, we build a machine $M$ that concatenates $\|[\![\mathrm{b}]\!]$ on the working tape $t_{\mathbb{K}}$ and runs $M_1$ or $M_2$ depending on the value of $[\![\mathrm{b}]\!]$, using the [BV97, Branching Lemma]. Then we erase $[\![\mathrm{b}]\!]$ from the end of tape $t_{\mathbb{K}}$. $M$ computes $[\![\mathrm{P}]\!]$ in time

$$\max_i(O(n + n \times \mathrm{Time}_{\mathrm{P}_i}(n))) = O(n + n \times \mathrm{Time}_{\mathrm{P}}(n)).$$

- For the quantum case $\mathrm{S} = \textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_1, 1 \to \mathrm{S}_2\}$, the machine appends $\|[\![\mathrm{q}]\!]$ on tape $t_{\mathbb{K}}$. It reads $t_{in}([\![\mathrm{q}]\!])$, sets it to $\#$, and if it reads 0 runs $M_1$, if it reads 1, runs $M_2$. Finally, from state $s_\top^1$, it writes 0 in $t_{in}([\![\mathrm{q}]\!])$, moves the head to index 0, and transitions to $s_\top$; similarly, from state $s_\top^2$, the machines writes 1 in $t_{in}([\![\mathrm{q}]\!])$ before moving the head and transitioning to $s_\top$. We have that $M$ computes $[\![\mathrm{P}]\!]$ in time

$$\max_i(O(n + n \times \mathrm{Time}_{\mathrm{P}_i}(n))) = O(n + n \times \mathrm{Time}_{\mathrm{P}}(n)).$$

We now consider the case of a (possibly recursive) procedure call. For the procedure call $\mathrm{S} = \textbf{call}\ \texttt{proc}[\mathrm{i}](\mathrm{s})$, we start by creating a state $s_{\texttt{proc}}$. If `proc` is a constant-time procedure (that is to say, it does not contain any procedure calls), we can simply inductively define a machine $M_{\texttt{proc}}$ that executes the procedure and replace $s_{\texttt{proc}}$ with this machine.

If the procedure body of `proc` contains other procedure calls, then we proceed as follows: we inductively create a machine that executes the procedure body where every procedure call to procedure `proc'` is defined as a transition into state $s_{\texttt{proc'}}$. If $\texttt{proc} \succ \texttt{proc'}$, then the machine will not reenter state $s_{\texttt{proc}}$ and we can build it separately. Otherwise, if the procedure call is recursive (for a simple case, consider $\texttt{proc'} = \texttt{proc}$), it is replaced with a transition back into the state $s_{\texttt{proc}}$.

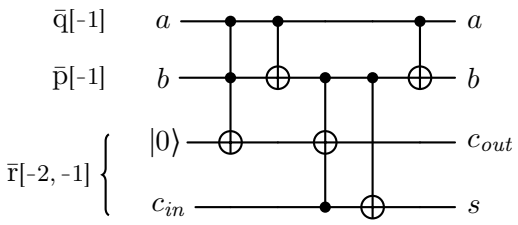
When entering or leaving machines simulating procedure calls, we update $t_{call}$ by appending $[\![\mathrm{i}]\!]$, and update $t_l$ by adding the qubit pointer indices excluded in $[\![\mathrm{s}]\!]$, separating them using the symbol &. The machine $M_{\texttt{proc}}$ then computes the function $[\![\mathrm{P}_{\texttt{proc}}]\!]$, for $\mathrm{P}_{\texttt{proc}} \triangleq \mathrm{D} :: \mathrm{S}^{\texttt{proc}}\{[\![\mathrm{i}]\!]/\mathrm{x}, \mathrm{s}/\bar{\mathrm{q}}\}$, in time $O(m + m \times \mathrm{Time}_{\mathrm{P}_{\texttt{proc}}}(m))$, with $m \triangleq [\![|\mathrm{s}|]\!] \leq n$, afterwards erasing $[\![\mathrm{i}]\!]$ and the new indices of $t_{in}$. As $\mathrm{Time}_{\mathrm{P}_{\texttt{proc}}}(n) = O(\mathrm{Time}_{\mathrm{P}}(n))$ and the complexity of $M$ is $O(n + n \times \mathrm{Time}_{\mathrm{P}}(n))$. This concludes the proof. □

We now show some examples of FOQ programs. Quantum arithmetic is one of the most fundamental building blocks of quantum computing, and one of its simplest examples is the Quantum Ripple-Carry Adder (QRCA) [VBE96, CDKM04, TK05].

**Example 4.4** (Quantum Ripple-Carry Adder)**.** *To define a QRCA in* FOQ*, we can start by defining a procedure* `add` *which performs the addition of two qubit inputs, while taking in a carry-in and outputting a carry-out.*

*The addition is performed reversibly using Controlled-NOTs and Toffoli gates, with which procedure* `add` *computes* $c_{out} = (a \cdot b) \oplus (c_{in} \cdot (a \oplus b))$ *and* $s = a \oplus b \oplus c_{in}$.

```
1  decl add(q̄, p̄, r̄) {
2    TOF(q̄[-1], p̄[-1], r̄[-2])
3    CNOT(q̄[-1], p̄[-1])
4    TOF(p̄[-1], r̄[-1], r̄[-2])
5    CNOT(p̄[-1], r̄[-1])
6    CNOT(q̄[-1], p̄[-1]) }
```




*To perform addition over two input strings of size $n \geq 1$, we can iterate* `add` *over the strings where each iteration takes as carry-in the corresponding carry-out of the previous iteration. To do so, we define a program containing a procedure* `fullAdder`*, as defined in program* `QRCA` *in Figure 4.5, that calls itself recursively while discarding the qubits which it no longer needs for the addition.*

`QRCA`

```
1..  decl add(q̄, p̄, r̄) {...}
7    decl fullAdder(q̄, p̄, r̄) {
8      if |r̄| > 1 then
9        call add(q̄, p̄, r̄);
10       call fullAdder(q̄ ⊖ [-1],
                        p̄ ⊖ [-1],
                        r̄ ⊖ [-1]);
11     else skip;}
12   ::
13   call fullAdder(q̄, p̄, r̄);}
```

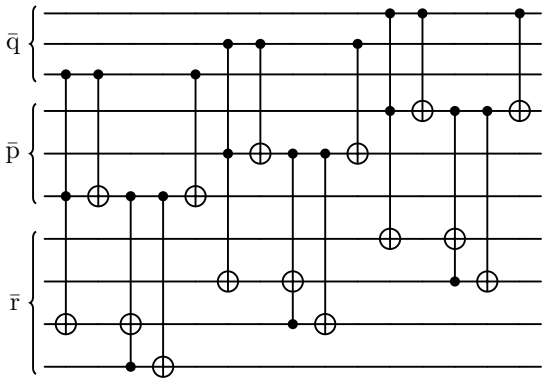



Figure 4.5: Program `QRCA` for quantum addition.

The quantum Fourier transform (QFT), used as a subroutine in Shor's algorithm [Sho94], can also be simulated in FOQ.

**Example 4.5** (Quantum Fourier Transform)**.** *We describe the following notation for the controlled-phase gate:*

$$\text{CPHASE}(q_1, q_2, i) \triangleq \textbf{qcase } q_1 \textbf{ of } \{0 \to \textbf{skip};, 1 \to q_2 \ *= \text{Ph}^{\lambda x.\pi/2^x}(i-1);\}$$

*and define the program* `QFT` *as given in Figure 4.6. Note that $\lambda x.\pi/2^x$ is a total function in $\mathbb{Z} \to [0, 2\pi) \cap \tilde{\mathbb{R}}$ that is polynomial-time approximable.*

Let $f : \{0,1\}^n \to \{0,1\}$ be a black-box function such that, for precisely one input $x \in \{0,1\}^n$, we have that $f(x) = 1$. The problem of finding the value of $x$ (with bounded probability) requires $\Theta(2^n)$ queries to $f$ in the classical case (i.e. there is no better strategy than simply testing all possible inputs until finding $x$).

Grover's algorithm for quantum search allows for a quadratic speedup [Gro96], where the number of queries to an oracle for $f$ is given by $O(2^{\frac{n}{2}})$. Grover's algorithm is asymptotically

QFT

```
1  decl rec(q̄){                5  decl rot[x](q̄){              10  decl inv(q̄){
2    q̄[1] *= H;                6    if |q̄| > 1 then           11    if |q̄| > 1 then
3    call rot[2](q̄);           7      CPHASE(q̄[2], q̄[1], x)   12      SWAP(q̄[1], q̄[-1])
4    call rec(q̄ ⊖ [1]); },     8      call rot[x + 1](q̄ ⊖ [2]); 13     call inv(q̄ ⊖ [1, -1]);
                                9    else skip; },             14    else skip; }
15 ::
16 call rec(q̄); call inv(q̄);
```

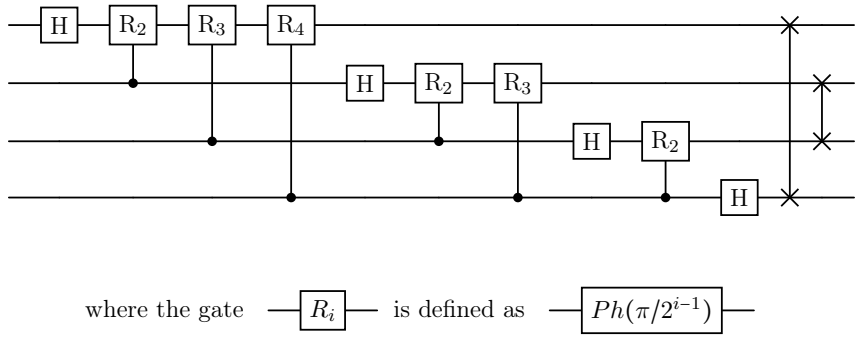



Figure 4.6: Program QFT for the quantum Fourier transform.

optimal [BBBV97] for unstructured search, and provides a quadratic speedup over classical methods for a variety of problems.

**Example 4.6** (Grover's algorithm)**.** *The program* GROVER *given in Figure 4.7 implements the quantum search algorithm. Given an input of $n$ qubits in state $|0\dots0\rangle$, the algorithm starts by creating a uniform superposition of states, followed by the application of $O(2^{\frac{n}{2}})$* Grover iterations*, containing a call to an oracle for $f$ that performs the transformation $|x\rangle \mapsto (-1)^{f(x)}|x\rangle$. The Grover iteration also includes a* diffusion operator *defined by $\mathbb{1} - 2|0\dots0\rangle\langle0\dots0|$.*

As a consequence of FOQ programs defining unitary transformations, they are necessarily reversible. We now show that FOQ programs can be statically inverted.

### 4.2.3 Reversibility

We start by defining the following transformation over programs.

**Definition 4.7** (Inverse transformation)**.** *Let $\cdot^{-1}$ : Programs → Programs denote the transformation that inverts a* FOQ *program. We define it inductively as follows. By abuse of notation, given*

**GROVER**

```
1  decl h(q̄){
2    q̄[1] *= H;
3    call h(q̄ ⊖ [1]); },

4  decl diffusion(q̄){
5    if |q̄| > 1 then
6      qcase q̄[1] of {
7        0 → call diffusion(q̄ ⊖ [1]);
8        1 → skip; }
9    else
10     q̄[1] *= NOT; q̄[1] *= Ph^{λx.π}(1);
11     q̄[1] *= NOT; }

12 decl grover[x](q̄){
13   if x > 0 then
14     call grover[x - 1](q̄);
15     call grover[x - 1](q̄);
16   else
17     call oracle(q̄);
18     call h(q̄);
19     call diffusion(q̄);
20     call h(q̄); }

21 :: call h(q̄); call grover[|q̄|/2](q̄);
```

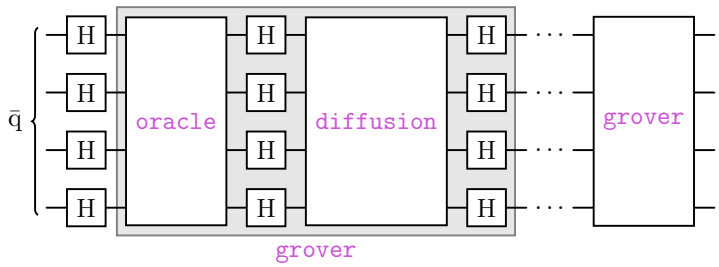



Figure 4.7: Program GROVER for quantum search.

proc ∈ Procedures, *we denote by* $\texttt{proc}^{-1}$ *the name of the* inverse procedure *relative to* proc.

$$
\begin{aligned}
(\mathrm{D} :: \mathrm{S})^{-1} &\triangleq \mathrm{D}^{-1} :: \mathrm{S}^{-1} \\
(\textbf{decl}\ \texttt{proc}[\mathrm{x}](\bar{\mathrm{q}}_1,\ldots,\bar{\mathrm{q}}_k)\{\mathrm{S}\},\ \mathrm{D})^{-1} &\triangleq \textbf{decl}\ \texttt{proc}^{-1}[\mathrm{x}](\bar{\mathrm{q}}_1,\ldots,\bar{\mathrm{q}}_k)\{\mathrm{S}^{-1}\},\ \mathrm{D}^{-1} \\
\varepsilon^{-1} &\triangleq \varepsilon \\
\textbf{skip;}^{-1} &\triangleq \textbf{skip;} \\
(\mathrm{q} \mathrel{*}= \mathrm{U}^f(\mathrm{i});)^{-1} &\triangleq \mathrm{q} \mathrel{*}= (\mathrm{U}^f(\mathrm{i}))^{\dagger}; \\
(\mathrm{S}_1\ \mathrm{S}_2)^{-1} &\triangleq \mathrm{S}_2^{-1}\ \mathrm{S}_1^{-1} \\
(\textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{S}_{\textbf{true}}\ \textbf{else}\ \mathrm{S}_{\textbf{false}})^{-1} &\triangleq \textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{S}^{-1}_{\textbf{true}}\ \textbf{else}\ \mathrm{S}^{-1}_{\textbf{false}} \\
(\textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\})^{-1} &\triangleq \textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_0^{-1}, 1 \to \mathrm{S}_1^{-1}\} \\
(\textbf{call}\ \texttt{proc}[\mathrm{i}](\mathrm{s}_1,\ldots,\mathrm{s}_k);)^{-1} &\triangleq \textbf{call}\ \texttt{proc}^{-1}[\mathrm{i}](\mathrm{s}_1,\ldots,\mathrm{s}_k);,
\end{aligned}
$$

*with*

$$
\mathrm{NOT}^{\dagger} \triangleq \mathrm{NOT}, \qquad \mathrm{R}_Y^f(\mathrm{i})^{\dagger} \triangleq \mathrm{R}_Y^{2\pi-f}(\mathrm{i}), \qquad \textit{and} \qquad \mathrm{Ph}^f(\mathrm{i})^{\dagger} \triangleq \mathrm{Ph}^{2\pi-f}(\mathrm{i}).
$$

We show that the inverse transformation is sound and that it does not change the complexity of the program.

$\underline{\texttt{QFT}^{-1}}$

```
1  decl rec⁻¹(q̄){                5  decl rot⁻¹[x](q̄){                10  decl inv⁻¹(q̄){
2    call rec⁻¹(q̄ ⊖ [1]);          6    if |q̄| > 1 then                11    if |q̄| > 1 then
3    call rot⁻¹[2](q̄);             7      call rot⁻¹[x + 1](q̄ ⊖ [2]);  12      call inv⁻¹(q̄ ⊖ [1, -1]);
4    q̄[1] *= H; },                 8      CPHASE⁻¹(q̄[2], q̄[1], x)      13      SWAP(q̄[1], q̄[-1])
                                    9    else skip; },                  14    else skip; }
15 ::
16 call inv⁻¹(q̄); call rec⁻¹(q̄);
```

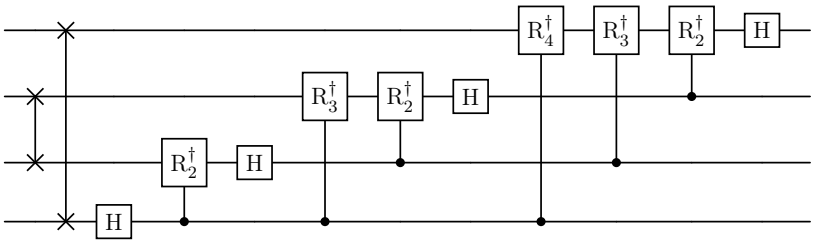



Figure 4.8: Program $\texttt{QFT}^{-1}$ for the inverse quantum Fourier transform.

**Theorem 4.8.** *Let* P *be a terminating* FOQ *program, then*

$$[\![\mathrm{P}^{-1}]\!] \circ [\![\mathrm{P}]\!] = \mathbb{1} \qquad \textit{and} \qquad \mathrm{Time}_{\mathrm{P}}(n) = \mathrm{Time}_{\mathrm{P}^{-1}}(n).$$

As an example, let us consider the inverse of the QFT, which is used in phase estimation and arithmetic.

**Example 4.9** (Inverse Quantum Fourier Transform)**.** *First, we notice that the inverse operation does not change the semantics of some commands, which allows for some simplifications. For instance, since the Hadamard gate and the swap operation are their own inverse, we may use the following rules:*

$$(\mathrm{q} \mathrel{*}= \mathrm{H};)^{-1} \triangleq \mathrm{q} \mathrel{*}= \mathrm{H};$$
$$(\mathrm{SWAP}(\mathrm{q}_1, \mathrm{q}_2, \mathrm{i}))^{-1} \triangleq \mathrm{SWAP}(\mathrm{q}_1, \mathrm{q}_2, \mathrm{i})$$

*Likewise, we define a shorthand notation for the inverse of the controlled-phase gate:*

$$\begin{aligned}(\mathrm{CPHASE}(\mathrm{q}_1, \mathrm{q}_2, \mathrm{i}))^{-1} &= \textbf{qcase } \mathrm{q}_1 \textbf{ of } \{0 \to \textbf{skip};, 1 \to \mathrm{q}_2 \mathrel{*}= \mathrm{Ph}^{\lambda x.2\pi - \pi/2^{x-1}}(\mathrm{i}-1);\} \\ &\triangleq \mathrm{CPHASE}^{-1}(\mathrm{q}_1, \mathrm{q}_2, \mathrm{i})\end{aligned}$$

*The inverse QFT program,* $\texttt{QFT}^{-1}$*, is given in Figure 4.8.*

In this chapter, we introduced the syntax and semantics of the FOQ programming language. We showed that FOQ admits a QTM simulation with only a polynomial slowdown compared to the time complexity of the program, as measured by the maximum number of procedure calls performed in depth (Lemma 4.3). We gave two examples of FOQ programs, one in exponential

time on the input size (Grover's search algorithm, Figure 4.7) and one in polynomial time (quantum Fourier transform, Figure 4.6). Finally, we showed that the inverse of a FOQ program is another FOQ program which can be statically obtained.

We will now turn to identifying fragments of FOQ, obtained via syntactical restrictions, that capture quantum complexity classes.

# 5

# A Characterization of Quantum Polynomial Time

> The class of functions, [FBQP] (functions computable with bounded error, given quantum resources, in polynomial time), has been defined in three distinct but equivalent ways: via quantum Turing machines, quantum circuits, and modular functors. [...] We may now propose a "thesis" in the spirit of Alonzo Church: All 'reasonable' computational models which add the resources of quantum mechanics (or quantum field theory) to classical computation yield (efficiently) inter-simulable classes: there is one quantum theory of computation.
>
> – *Topological Quantum Computing* [FKLW03]

In this chapter, we restrict the set of FOQ programs to a strict subset, named PFOQ, that is sound and complete for the quantum complexity class FBQP. To achieve this, we define two criteria: one ensuring that a program terminates and another preventing a terminating program from having an exponential runtime.

## 5.1 Well-foundedness

The first criterion we will consider is one which ensures program termination, by requiring that recursive procedure calls strictly reduce the number of qubits that may be accessed.

Given two statements S and S′, we write $\mathrm{S} \in \mathrm{S}'$ to mean that S is a substatement of S′ and, for a procedure proc, we have that $\texttt{proc} \in \mathrm{S}$ holds if there are i and s such that **call** proc[i](s); $\in \mathrm{S}$. Given a program P = D :: S, we define the relation $>_\mathrm{P} \subseteq \text{Procedures} \times \text{Procedures}$ by $\texttt{proc}_1 >_\mathrm{P} \texttt{proc}_2$ if $\texttt{proc}_2 \in \mathrm{S}^{\texttt{proc}_1}$, for any two procedures $\texttt{proc}_1, \texttt{proc}_2 \in \mathrm{S}$. Let the partial order $\geq_\mathrm{P}$ be the transitive and reflexive closure of $>_\mathrm{P}$ and define the equivalence relation $\sim_\mathrm{P}$ by

$$\texttt{proc}_1 \sim_\mathrm{P} \texttt{proc}_2 \text{ if } (\texttt{proc}_1 \geq_\mathrm{P} \texttt{proc}_2 \wedge \texttt{proc}_2 \geq_\mathrm{P} \texttt{proc}_1).$$

Define also the strict order $>_\mathrm{P}$ as

$$\texttt{proc}_1 >_\mathrm{P} \texttt{proc}_2 \text{ if } (\texttt{proc}_1 \geq_\mathrm{P} \texttt{proc}_2 \wedge \texttt{proc}_1 \not>_\mathrm{P} \texttt{proc}_2).$$

**Definition 5.1.** *Let* WF *be the set of* FOQ *programs* P *that satisfy the following condition*

$$\forall \texttt{proc} \in \mathrm{P},\ \forall \textbf{call}\ \texttt{proc'}[\mathrm{i}](\mathrm{s}_1, \ldots, \mathrm{s}_k); \in \mathrm{S}^{\texttt{proc}},$$
$$\texttt{proc} \sim_\mathrm{P} \texttt{proc'} \Rightarrow \textit{there exists } \mathrm{s}_i \textit{ such that } \forall f : Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N}),\ (|\mathrm{s}_i|, f) \Downarrow_{\mathcal{L}(\mathbb{N})} n < |f(\bar{\mathrm{q}}_i)|,$$

*meaning that, for at least one argument, the number of accessible qubits is strictly reduced. If* $\mathrm{P} \in$ WF *we say that* P *is* well-founded.

The well-foundedness condition can be checked relatively easily in a FOQ program as the $s \ominus [i]$, $s^{\boxminus}$ and $s^{\boxplus}$ constructs strictly reduce the number of qubits in s, and no qubits can be added. We check this condition for the example programs given in the previous chapter, in Examples 5.3, 5.4 and 5.5.

We start by showing that this condition is enough to ensure the termination of a FOQ program.

**Lemma 5.2** (Termination)**.** *If* $P \in \mathrm{WF}$*, then* P *terminates.*

*Proof.* Without loss of generality, we consider that all procedures have the same inputs, given in the same order. This is because an input given to a procedure only gives it *access* to said input, and so a procedure can have access to inputs it does not use. For a given program P, we define a partial order $\gg_P$ between configurations whose statements are procedure calls, as

$$(\textbf{call } \texttt{proc}[i](s_1, \ldots, s_k);, |\psi\rangle, A, f) \gg_P (\textbf{call } \texttt{proc'}[i'](s'_1, \ldots, s'_k);, |\psi'\rangle, A', f')$$

if

$$(\texttt{proc} >_P \texttt{proc'}) \vee \left(\texttt{proc} \sim_P \texttt{proc'} \wedge \sum_{j=1}^{k} n_j > \sum_{j=1}^{k} n'_j\right), \text{ for } (|s_j|, f) \Downarrow_{\mathcal{L}(\mathbb{N})} n_j \text{ and } (|s'_j|, f) \Downarrow_{\mathcal{L}(\mathbb{N})} n'_j.$$

Given a program P and $\pi \rhd (\textbf{call } \texttt{proc}[i](s);, |\psi\rangle, A, l)$ and $\pi' \rhd (\textbf{call } \texttt{proc'}[i'](s');, |\psi'\rangle, A', f)$, we show that, if $\pi' \trianglelefteq \pi$, then it holds that

$$(\textbf{call } \texttt{proc}[i](s_1, \ldots, s_k);, |\psi\rangle, A, f) \gg_P (\textbf{call } \texttt{proc'}[i'](s'_1, \ldots, s'_k);, |\psi'\rangle, A', f').$$

Assume that $\pi' \trianglelefteq \pi$, then it holds that $\texttt{proc} \geq_P \texttt{proc'}$. Hence, either $\texttt{proc} >_P \texttt{proc'}$ or $\texttt{proc} \sim_P \texttt{proc'}$. In this latter case, by transitivity of $\sim_P$, there exists an index $j$ such that $s'_j = s_j \ominus [i_1, \ldots, i_k]$, for some $k > 0$. It implies that the evaluation of $|s'_j|$ is strictly smaller than the evaluation of $|s_j|$, by transitivity. Given that the input cannot be increased, the sum of the input sizes is strictly smaller. We conclude by observing that $\gg_P$ is a well-founded order. □

Programs `QRCA` (Figure 4.5) and `QFT` (Figure 4.6) introduced in Chapter 4 are well-founded. Program `GROVER` (Figure 4.7), introduced in the same chapter, is *not* well-founded, even though it always terminates.

**Example 5.3.** *Consider the program* `QRCA` *of Figure 4.5. The recursivity relations between the procedure in the program are* $\texttt{fullAdder} \sim_{\texttt{QRCA}} \texttt{fullAdder}$ *and* $\texttt{fullAdder} >_{\texttt{QRCA}} \texttt{add}$*, therefore mutually recursive calls are only present in the procedure statement of* `fullAdder`*, and since the inputs of the procedure call are* $\bar{q} \ominus [-1]$, $\bar{p} \ominus [-1]$*, and* $\bar{r} \ominus [-1]$*, we have that* $\texttt{QRCA} \in \mathrm{WF}$ *and we conclude by Lemma 5.2 that it terminates.*

**Example 5.4.** *Consider the program* `QFT` *of Figure 4.6. The statements of the procedure declarations define the following relation:* $\texttt{rec} >_{\texttt{QFT}} \texttt{rec}$, $\texttt{rec} >_{\texttt{QFT}} \texttt{rot}$, $\texttt{rot} >_{\texttt{QFT}} \texttt{rot}$*, and* $\texttt{inv} >_{\texttt{QFT}} \texttt{inv}$*. Consequently,* $\texttt{rec} \sim_{\texttt{QFT}} \texttt{rec}$, $\texttt{rot} \sim_{\texttt{QFT}} \texttt{rot}$, $\texttt{inv} \sim_{\texttt{QFT}} \texttt{inv}$*, and* $\texttt{rec} >_{\texttt{QFT}} \texttt{rot}$ *hold. For each call to an equivalent procedure, we check that the argument decreases:* $\bar{q} \ominus [1]$ *in* `rec`*,* $\bar{q} \ominus [2]$ *in* `rot`*, and* $\bar{q} \ominus [1, |\bar{q}|]$ *in* `inv`*. Consequently,* $\texttt{QFT} \in \mathrm{WF}$*, and by Lemma 5.2, we have that* `QFT` *terminates.*

**Example 5.5.** *In the program* `GROVER` *defined in Figure 4.7, we have that* $\texttt{grover} \sim_{\texttt{GROVER}} \texttt{grover}$ *and that the corresponding mutually recursive procedure call does not reduce the input size. Therefore,* $\texttt{GROVER} \notin \mathrm{WF}$*, even if one can check that the program always terminates.*

## 5.2 Bounded width

We now add a further restriction on mutually-recursive procedure calls for guaranteeing polynomial time using a notion of width. This restriction will ensure that, for a given program, no procedure in the program can perform *duplication*, which prevents the program from executing an exponential number of operations.

**Definition 5.6.** *Given a program* P *and a procedure* proc ∈ P*, the* width of proc in P*, noted* $\mathrm{width}_{\mathrm{P}}(\mathtt{proc})$*, is defined as* $\mathrm{width}_{\mathrm{P}}(\mathtt{proc}) \triangleq w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}^{\mathtt{proc}})$*, where* $w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S})$ *is the* width of the procedure proc in P relative to statement S*, defined inductively as:*

$$
\begin{aligned}
w_{\mathrm{P}}^{\mathtt{proc}}(\textbf{skip};) &\triangleq 0,\\
w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{q} \mathrel{*}= \mathrm{U}^{f}(\mathrm{i});) &\triangleq 0,\\
w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}_1\ \mathrm{S}_2) &\triangleq w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}_1) + w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}_2),\\
w_{\mathrm{P}}^{\mathtt{proc}}(\textbf{if}\ \mathrm{b}\ \textbf{then}\ \mathrm{S}_{\textbf{true}}\ \textbf{else}\ \mathrm{S}_{\textbf{false}}) &\triangleq \max(w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}_{\textbf{true}}), w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}_{\textbf{false}})),\\
w_{\mathrm{P}}^{\mathtt{proc}}(\textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}) &\triangleq \max(w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}_0), w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}_1)),\\
w_{\mathrm{P}}^{\mathtt{proc}}(\textbf{call}\ \mathtt{proc'}[\mathrm{i}](\mathrm{s});) &\triangleq \begin{cases} 1 & \textit{if } \mathtt{proc} \sim_{\mathrm{P}} \mathtt{proc'},\\ 0 & \textit{otherwise.} \end{cases}
\end{aligned}
$$

Consider the width of procedures in the example programs of Chapter 4.

**Example 5.7.** $\mathrm{width}_{\mathtt{QRCA}}(\mathtt{add}) = 0$*, and* $\mathrm{width}_{\mathtt{QRCA}}(\mathtt{fullAdder}) = 1$.

**Example 5.8.** $\mathrm{width}_{\mathtt{QFT}}(\mathtt{rec}) = \mathrm{width}_{\mathtt{QFT}}(\mathtt{rot}) = \mathrm{width}_{\mathtt{QFT}}(\mathtt{inv}) = 1$*, since* $\mathtt{rec} >_{\mathtt{QFT}} \mathtt{rot}$ *holds.*

**Example 5.9.** $\mathrm{width}_{\mathtt{GROVER}}(\mathtt{grover}) = 2$.

We define a restriction on FOQ programs according the maximum width of a procedure that appears in the program.

**Definition 5.10.** *We denote by* $\mathrm{WIDTH}_{\leq 1}$ *the set of* FOQ *programs* P *that satisfy*

$$\max_{\mathtt{proc} \in \mathrm{P}} \mathrm{width}(\mathtt{proc}) \leq 1.$$

Given the previous examples, we have that QRCA, QFT ∈ $\mathrm{WIDTH}_{\leq 1}$ and GROVER ∉ $\mathrm{WIDTH}_{\leq 1}$. We now turn to showing that by combining the WF and $\mathrm{WIDTH}_{\leq 1}$ conditions, we obtain a fragment of FOQ programs that characterizes quantum polynomial time functions.

## 5.3 A Polynomial First-Order Quantum Language (PFOQ)

We define the polynomial fragment of FOQ, consisting of well-founded programs that also have bounded width.

**Definition 5.11** (PFOQ)**.** PFOQ *is the set of programs defined as:*

$$\mathrm{PFOQ} \triangleq \mathrm{WF} \cap \mathrm{WIDTH}_{\leq 1}.$$

*If* P ∈ PFOQ*, then we call* P *a* PFOQ *program.*

**Example 5.12.** *The program* QFT *defined in Figure 4.6 is well-founded (Example 5.4) and has width 1 (Example 5.8). Therefore,* QFT ∈ PFOQ.

PFOQ inclusion can be determined efficiently on the size of the program.

**Theorem 5.13.** *For each* FOQ *program* $\mathrm{P}$*, it can be decided in time* $O(|\mathrm{P}|^2)$ *whether* $\mathrm{P} \in \mathrm{PFOQ}$.

*Proof.* The relations $\sim_{\mathrm{P}}$, $\geq_{\mathrm{P}}$, and $>_{\mathrm{P}}$ can be computed in time $O(|\mathrm{P}|^2)$. After this computation, determining whether $\mathrm{P} \in \mathrm{WF}$ and computing the width of each procedure can be done in time $O(|\mathrm{P}|)$. □

Another useful property is that the inverse of a PFOQ program, obtained via the construction in Definition 4.7, is also a PFOQ program.

**Lemma 5.14.** *Let* $\mathrm{P} \in \mathrm{PFOQ}$*. Then* $\mathrm{P}^{-1} \in \mathrm{PFOQ}$*, where* $\cdot^{-1}$ *is the inverse transformation given in Definition 4.7.*

*Proof.* $\mathrm{P}^{-1}$ can be shown to be in PFOQ since it preserves the relation $>$, that is, $\mathtt{proc}_1^{-1} >_{\mathrm{P}^{-1}} \mathtt{proc}_2^{-1}$ if and only if $\mathtt{proc}_1 >_{\mathrm{P}} \mathtt{proc}_2$. Furthermore, the width of each procedure is preserved, therefore $\mathrm{P}^{-1} \in \mathrm{WIDTH}_{\leq 1}$. $\mathrm{P}^{-1}$ is also well-founded since the input qubit list of each procedure call is preserved when generating the inverse program. □

A relevant example is the inverse QFT, given in Figure 4.8.

**Example 5.15.** $\mathtt{QFT}^{-1} \in \mathrm{PFOQ}$.

We now show that one can easily combine PFOQ programs while being able to maintain the syntactical restrictions of WF and $\mathrm{WIDTH}_{\leq 1}$.

**Lemma 5.16.** *Let* $\mathrm{P}_1, \ldots, \mathrm{P}_k$ *be* PFOQ *programs appearing in program* $\mathrm{P}$ *according to the syntactic sugar of Definition 4.1, and let* $\mathrm{P}_{\mathbf{skip}}$ *denote* $\mathrm{P}$ *with all instances of* $\mathrm{P}_i$ *replaced by the* **skip** *statement. If* $\mathrm{P}_{\mathbf{skip}}, \mathrm{P}_1, \ldots, \mathrm{P}_k \in \mathrm{PFOQ}$*, then* $\mathrm{P} \in \mathrm{PFOQ}$.

*Proof.* In the case of the first three rules, it is easy to check that the disjoint union of procedure declarations is enough to preserve the $>$ relation and width of each procedure, and therefore the conclusion follows. For the case of the procedure declaration, the disjoint union ensures that $\mathtt{proc}$ does not appear on any program $\mathrm{P}_i$, and therefore statements in $\mathrm{S}^{\mathtt{proc}}$ will not change the width of $\mathtt{proc}$ and the entire program is in PFOQ as long as $\mathtt{proc}$ has width at most one from recursive calls to itself and to the procedures in D. □

The quantum Fourier transform can be used to perform algebraic operations, such as addition and multiplication [Dra00, RPGE17]. We define PFOQ programs for these operations by combining smaller programs, including the QFT and its inverse.

**Example 5.17** (QFT Addition)**.** *Let* $\mathtt{QFTAdd}$ *be the program defined in Figure 5.1. Given that* $\mathtt{QFT}, \mathtt{QFT}^{-1} \in \mathrm{PFOQ}$*, by Lemma 5.16, since* $\mathtt{ADD} \in \mathrm{PFOQ}$*, we have that* $\mathtt{QFTAdd} \in \mathrm{PFOQ}$.

**Example 5.18** (QFT Multiplication)**.** *Consider the program* $\mathtt{QFTMult}$ *defined in Figure 5.2. When we replace* $\mathtt{ADD}(\bar{\mathrm{p}}, \bar{\mathrm{r}})$ *in* $\mathrm{S}^{\mathtt{adding\_loop}}$ *with the no-op statement, we have that*

$$\mathrm{width}_{\mathtt{QFTMult}}(\mathrm{S}^{\mathtt{adding\_loop}}[\mathbf{skip}; /\mathtt{ADD}]) = 1$$

*and this new procedure respects the* WF *restriction. Since* $\mathtt{ADD}, \mathtt{QFT}$ *and* $\mathtt{QFT}^{-1}$ *are all* PFOQ *programs, we conclude that* $\mathtt{QFTMult} \in \mathrm{PFOQ}$.

Let ADD denote the following program

```
1 decl inner_loop[x](q̄, r̄){                    4 decl outer_loop(q̄, r̄){
2   CPHASE(r̄[1], q̄[1], x)                      5   call inner_loop[1](q̄, r̄);
3   call inner_loop[x + 1](q̄, r̄ ⊖ [1]); },     6   call outer_loop(q̄ ⊖ [1], r̄ ⊖ [1]); }
7 ::
8 call outer_loop(q̄, r̄);
```

Then $\texttt{QFTAdd}(\bar{q},\bar{r}) \triangleq \texttt{QFT}(\bar{q})\ \texttt{ADD}(\bar{q},\bar{r})\ \texttt{QFT}^{-1}(\bar{q})$

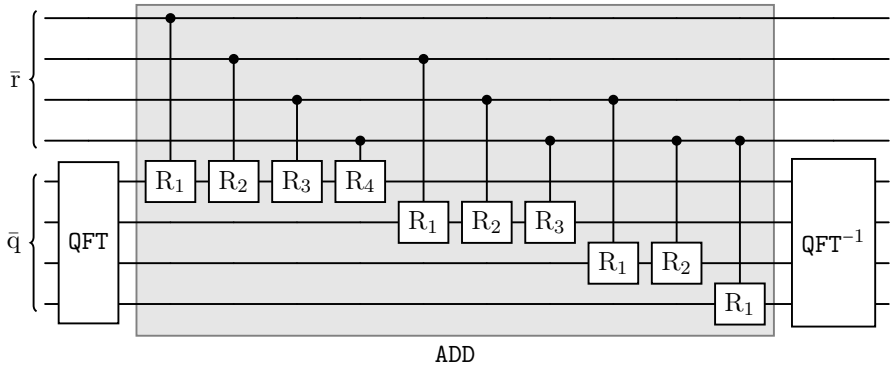



Figure 5.1: PFOQ program QFTAdd for addition using the QFT and its inverse.

### 5.3.1 Soundness

We now show that the time complexity of a PFOQ program is bounded by a polynomial in the length of its input. We start by defining the notion of *rank of a procedure* and *rank of a program*.

Intuitively, the rank of a procedure is its level of recursion in a given program: a non-recursive procedure has rank 0, a procedure that only calls itself will has rank 1, a recursive procedure that calls another procedure of rank 1 will have rank 2, and so on. The rank of a program is then the maximum rank among all of its procedures.

**Definition 5.19** (Rank of a procedure)**.** *Given a* FOQ *program* P*, the rank of a procedure in* P *is defined as:*

$$rk(\texttt{proc}) \triangleq 0 \qquad \text{if } \nexists\, \texttt{proc'},\ \texttt{proc} >_{\mathrm{P}} \texttt{proc'},$$
$$rk(\texttt{proc}) \triangleq \max\{rk(\texttt{proc'}) + 1 \mid \texttt{proc} >_{\mathrm{P}} \texttt{proc'}\} \qquad \text{otherwise.}$$

**Definition 5.20** (Rank of a program)**.** *The rank of program* P *is defined by*

$$rk(\mathrm{P}) \triangleq \max_{\texttt{proc} \in \mathrm{P}} rk(\texttt{proc}).$$

As was shown in Lemma 4.3, any terminating FOQ program can be simulated by a QTM. The time complexity of this simulation depends on the length of the input quantum state and on the runtime of the program.

**QFTMult**

```
1  decl adding_loop(q̄, p̄, r̄){
2    qcase q̄[-1] of {
3      0 → skip;,
4      1 → ADD(p̄, r̄) }
5    call adding_loop(q̄ ⊖ [-1], p̄, r̄ ⊖ [-1]); }
6  ::
7  QFT(r̄)
8  call adding_loop(q̄, p̄, r̄);
9  QFT^{-1}(r̄)
```

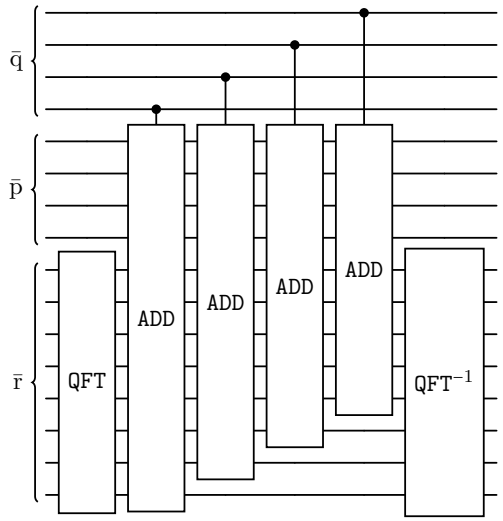



Figure 5.2: PFOQ program `QFTMult` for multiplication using the QFT and its inverse.

**Lemma 5.21.** *For any* PFOQ *program* P, *we have that* $\mathrm{Time}_{\mathrm{P}}(n) = O(n^{rk(\mathrm{P})+1})$.

*Proof.* Consider a PFOQ program P = D :: S. We show the result by induction on the rank of procedure calls in S. Take **call** proc[i](s); ∈ S. If $rk(\mathtt{proc}) = 0$ and the procedure is not recursive then there is only 1 call to a procedure. If the procedure is recursive, it can be called at most once in each branch of a quantum case statement. Hence there can be at most $n+1$ such calls in the full quantum case branch of the derivation tree and it holds that $\mathrm{Time}_{\mathrm{D::\mathbf{call}\ proc[i](s);}}(n) = O(n)$.

Induction hypothesis: assume that any procedure proc' such that $rk(\mathtt{proc'}) \leq k$ satisfies

$$\mathrm{Time}_{\mathrm{D::\mathbf{call}\ proc'[i](s);}}(n) = O(n^{k+1}).$$

Consider a procedure proc such that $rk(\mathtt{proc}) = k+1$. If the procedure is not recursive then it can call a constant number (bounded by the size of the program) of procedures of strictly smaller rank. By induction hypothesis,

$$\mathrm{Time}_{\mathrm{D::\mathbf{call}\ proc[i](s);}}(n) = \sum_{\mathtt{proc'} <_{\mathrm{P}} \mathtt{proc}} O(n^{rk(\mathtt{proc'})+1}) = O(n^{rk(\mathtt{proc})}).$$

If the procedure is recursive, it can be called at most once in each branch of a quantum case statement. Hence there can be at most $n+1$ such calls in the full quantum case branch of the derivation tree. Moreover, each of these calls can perform a constant number of calls to procedures of strictly smaller rank (since each procedure statement contains a finite number of procedure calls that does not depend on the input size). Consequently,

$$\mathrm{Time}_{\mathrm{D::\mathbf{call}\ proc[i](s);}}(n) = O(n) + \sum_{i=0}^{n} \sum_{\mathtt{proc'} <_{\mathrm{P}} \mathtt{proc}} O(n^{rk(\mathtt{proc'})+1}) = O(n^{rk(\mathtt{proc})+1}).$$

We observe that for a program P = D :: $\mathrm{S}_1$ … $\mathrm{S}_m$, for a constant $m \in \mathbb{N}$, it holds that

$$\mathrm{Time}_{\mathrm{P}}(n) = O(\sum_{i=1}^{m} \mathrm{Time}_{\mathrm{D::S}_i}(n)) = O(n^{rk(\mathrm{P})+1}).$$

This concludes the proof. □

Since the time complexity of a PFOQ program is always polynomial, we conclude that any PFOQ program can be simulated by a polynomial-time QTM.

**Definition 5.22.** *Let*

$$[\![\text{PFOQ}]\!]^{\text{poly}} \triangleq \left\{ [\![\text{P}]\!] \circ \phi \text{ such that } \text{P} \in \text{PFOQ} \text{ and } \phi \text{ is a polytime padding function}, \right\}$$

*where a polytime padding function follows Definition 3.13.*

**Theorem 5.23.** $[\![\text{PFOQ}]\!]^{\text{poly}} \subseteq \text{QP}$.

*Proof.* This is a direct consequence of Lemmas 4.3 and 5.21. □

### 5.3.2 Completeness

In this section, we show that PFOQ is expressive enough to encode any quantum polytime function. Toward this end, we demonstrate that PFOQ is expressive enough to simulate a function algebra that is sound and complete for quantum polynomial time, introduced in [Yam20] and described in Section 3.3.

**Theorem 5.24.** $\text{QP} \subseteq [\![\text{PFOQ}]\!]$.

*Proof.* By Theorem 3.15 we have that $\text{QP} = \mathscr{Y}$. It suffices to show that, for any function in $\mathscr{Y}_0$, there exists a PFOQ program that simulates it.

We prove this result by structural induction on a function on the function algebra. The basic function $I$ can be simulated by $\text{P}(\bar{\text{q}}) \triangleq \varepsilon :: \textbf{skip};$. $F \in \{Ph_\theta, ROT_\theta, NOT\}$ can be simulated using an assignment. In these cases $\text{P}(\bar{\text{q}}) = \varepsilon :: \bar{\text{q}}[1] \mathrel{*}= \text{U}^f(0);$ with $f$ such that $[\![\text{U}^f]\!](0) = F$. The basic initial function $SWAP$ can be simulated by the program $\text{P}(\bar{\text{q}}) = \varepsilon :: \text{SWAP}(\bar{\text{q}}[1], \bar{\text{q}}[2])$, with the SWAP statement as defined in Figure 4.2.

We now simulate the $Comp$, $Branch$ and $kQRec_t$ schemes. For that purpose, assume the existence of PFOQ programs $\texttt{F}$, $\texttt{G}$ and $\texttt{H}$ simulating the $\mathscr{Y}$ functions $F$, $G$, and $H$, respectively. We will construct new programs using these initial ones by applying the rules in Definition 4.1. By Lemma 5.16 we have that our obtained programs are also in PFOQ.

The composition of functions, $Comp[F, G]$, is simulated by composition in the program algebra, given by $\texttt{F}(\bar{\text{q}})\,\texttt{G}(\bar{\text{q}})$. Quantum branching, given by the function $Branch[F, G]$, can be simulated by $\textbf{qcase}\ \bar{\text{q}}[1]\ \textbf{of}\ \{0 \to \texttt{F}(\bar{\text{q}} \ominus [1]);\ 1 \to \texttt{G}(\bar{\text{q}} \ominus [1])\}$.

The multi-qubit quantum recursion function, $kQRec_t[F, G, H]$, is simulated in PFOQ by

$$\begin{array}{l}
\textbf{decl}\ \texttt{kQRec}_t(\bar{\text{q}})\{ \\
\quad \textbf{if}\ |\bar{\text{q}}| > t\ \textbf{then} \\
\quad\quad \textbf{call}\ \texttt{H}(\bar{\text{q}}); \\
\quad\quad \textbf{qcase}\ \bar{\text{q}}[1 \ldots k]\ \textbf{of}\ \{w \to \text{S}_w\} \\
\quad\quad \textbf{call}\ \texttt{G}(\bar{\text{q}}); \\
\quad \textbf{else} \\
\quad\quad \textbf{call}\ \texttt{F}(\bar{\text{q}}); \} \\
:: \ \textbf{call}\ \texttt{kQRec}_t(\bar{\text{q}});
\end{array}$$

$$\text{where } \text{S}_w \triangleq \begin{cases} \textbf{call}\ \texttt{kQRec}_t(\bar{\text{q}} \ominus [1, \ldots, k]); & \text{if } F_w = kQRec_t, \\ \textbf{skip}; & \text{if } F_w = I. \end{cases}$$

By induction hypothesis, $\mathtt{F}, \mathtt{G}, \mathtt{H} \in$ PFOQ. The only procedure that is not derived from these programs is $\mathtt{kQRec}_t$, and its recursive calls are of the shape **call** $\mathtt{kQRec}_t(\bar{\mathtt{q}} \ominus [1, \ldots, k])$;, and there is only one call per branch of a **qcase**. Therefore, the program is in PFOQ. □

By Theorems 5.23 and 5.24, we conclude that $[\![\text{PFOQ}]\!] = \text{QP}$. Therefore, the set of functions in FBQP (Definition 2.6) can be characterized as the set of classical functions that are approximated by PFOQ programs with bounded probability.

**Theorem 5.25.** $[\![\text{PFOQ}]\!]^{\text{poly}}_{\geq \frac{2}{3}} = \text{FBQP}$.

*Proof.* From Theorems 5.23 and 5.24, we have that $[\![\text{PFOQ}]\!]^{\text{poly}} = \text{QP}$, and by the definition of FBQP (Definition 2.6) we obtain the result. □

In this chapter, we defined two fragments of the FOQ language introduced in Chapter 4, namely the subset of programs that strictly reduce their qubit access, denoted WF for well-founded programs, and the subset $\text{WIDTH}_{\leq 1}$ of programs that avoid recursive duplication. The intersection of these two fragments, denoted PFOQ, was shown to be sound and complete for the set of functions that can be computed by quantum Turing machines running in polynomial time. This, in turn, was used to provide the first characterization of the class FBQP using a programming language.

As a tool for reasoning about quantum programs, we showed that PFOQ is expressive enough to contain standard examples of programs such as the quantum Fourier transform, as well as circuits for algebraic operations, and that large programs can be constructed from smaller ones in ways that preserve PFOQ properties.

# 6

# A Characterization of Quantum Polylogarithmic Time

In this chapter, we turn our attention to a more restrictive class of quantum programs: those that can be executed in *polylogarithmic* time. In order to do so, we introduce a new syntactic restriction for FOQ programs which ensures that recursive procedure calls reduce at least one input qubit list by half.

## 6.1 Halving

Consider the following subset of FOQ programs.

**Definition 6.1.** *A* FOQ *program* P *is said to be* recursively-halving *if*

$$\forall \texttt{proc} \in \mathrm{P},\ \forall \textbf{call}\ \texttt{proc'}[\mathrm{i}](\mathrm{s}_1,\ldots,\mathrm{s}_k); \in \mathrm{S}^{\texttt{proc}},$$
$$\texttt{proc} \sim_{\mathrm{P}} \texttt{proc'} \Rightarrow \textit{there exists } \mathrm{s}_i \textit{ such that } \forall f : Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N}),\ (|\mathrm{s}_i|, f) \Downarrow_{\mathcal{L}(\mathbb{N})} n < \frac{|f(\bar{\mathrm{q}}_i)|}{2}.$$

*We denote the set of recursively-halving programs as* HALF.

This restriction ensures that in every recursive procedure call at least one of the input qubit lists is cut in half. This ensures a polylogarithmic depth of recursive calls.

We now present two examples of recursively-halving programs, one which performs a binary search, and another which computes the last index in which a certain entry occurs. Both of these examples take in sorted strings so that the problems can be solved in logarithmic time.

**Example 6.2** (Binary search)**.** *Let $x \in 0^*1^*2^*$ be a sorted string and $\hat{x}$ denote the encoding of $x$ as a binary string given by $\hat{0} \triangleq 00$, $\hat{1} \triangleq 01$, and $\hat{2} \triangleq 10$. Program* SEARCH *in Figure 6.1 computes the function*

$$[\![\texttt{SEARCH}]\!](|\hat{x}\rangle_{\bar{\mathrm{q}}} \otimes |0\rangle_{\bar{\mathrm{r}}}) = |\hat{x}\rangle_{\bar{\mathrm{q}}} \otimes |b\rangle_{\bar{\mathrm{r}}},$$

*where $b \in \{0,1\}$ indicates whether $x$ contains a 1 or not. Since every recursive procedure in* SEARCH *takes as input either $\bar{\mathrm{q}}^{\boxplus}$ or $\bar{\mathrm{q}}^{\boxminus}$, we conclude that* SEARCH $\in$ HALF.

**Example 6.3** (Counting)**.** *Let $x \in 0^*1^*$ be a sorted binary string. Program* COUNT *defined in Figure 6.1 performs the transformation*

$$[\![\texttt{COUNT}]\!](|x\rangle_{\bar{\mathrm{q}}} \otimes |0^{\lceil \log(|x|) \rceil}\rangle_{\bar{\mathrm{r}}}) = |x\rangle_{\bar{\mathrm{q}}} \otimes |k\rangle_{\bar{\mathrm{r}}},$$

SEARCH

```
1  decl search(q̄, r̄){
2    if |q̄| > 1 then
3      qcase q̄[|q̄|/2, |q̄|/2 + 1] of
4        00 → call search(q̄^⊞, r̄);
5        01 → r̄[1] *= NOT;
6        10 → call search(q̄^⊟, r̄);
7        11 → skip;
8    else skip; }
9  ::
10 call search(q̄, r̄);
```

COUNT

```
1  decl count(q̄, r̄){
2    if |q̄| > 1 then
3      qcase q̄[|q̄|/2] of
4        0 → call count(q̄^⊞, r̄ ⊖ [1]);
5        1 → r̄[1] *= NOT;
6            call count(q̄^⊟, r̄ ⊖ [1]);
7    else skip; }
8  ::
9  call count(q̄, r̄);
```

Figure 6.1: Programs SEARCH and COUNT for binary search and counting sorted elements.

*where $k$ is a binary word encoding the number of instances of zero in the word (equivalently, smallest index with a 1). As in the previous example, it is easy to check that* $\texttt{COUNT} \in \textsc{half}$.

Recursively-halving programs are well-founded (Definition 5.1), as $\textsc{half} \subsetneq \textsc{wf}$. Therefore, by Lemma 5.2, they terminate.

**Example 6.4** (Parity)**.** *The program* PARITY *in Figure 6.2 computes the function*

$$[\![\texttt{PARITY}]\!](|x\rangle_{\bar{q}} \otimes |y\rangle_{\bar{r}}) = |x\rangle_{\bar{q}} \otimes |y \oplus x_1 \oplus \cdots \oplus x_n\rangle_{\bar{r}}$$

*for $x = x_1 \dots x_n \in \{0,1\}^n$. We have that* $\texttt{PARITY} \in \textsc{half}$*, as both its recursive procedure calls take as input $\bar{q}^{\boxplus}$ or $\bar{q}^{\boxminus}$, and therefore* PARITY *terminates.*

We now consider how the HALF restrictions can ensure the polylogarithmic-time termination of programs, when combined with the $\textsc{width}_{\leq 1}$ condition.

## 6.2 A (Poly-)Logarithmic First-Order Quantum Language (LFOQ)

In order to restrict FOQ to a subset of programs that terminate in polylogarithmic time, we introduce the following fragment.

**Definition 6.5.** *We define the polylogarithmic-time fragment of* FOQ *as*

$$\textsc{lfoq} \triangleq \textsc{half} \cap \textsc{width}_{\leq 1},$$

*where* $\textsc{width}_{\leq 1}$ *is the width restriction given in Definition 5.10.*

As we will see shortly (Theorem 6.9), the restriction to LFOQ programs ensures that they can be simulated by quantum random-access Turing machines running in polylogarithmic time. For example, programs SEARCH and COUNT introduced in the previous section are LFOQ programs.

PARITY

```
1  decl parity(q̄, r̄){
2    if |q̄| > 1 then
3      call parity(q̄^⊞, r̄);
4      call parity(q̄^⊟, r̄);
5    else
6      CNOT(q̄[1], r̄[1]) }
7  ::
8  call parity(q̄, r̄);
```

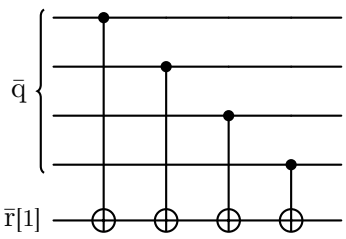



Figure 6.2: Program PARITY ∈ HALF for computing the parity of an input string.

**Example 6.6.** *Both programs* SEARCH *and* COUNT *of Figure 6.1 can be shown to be in* LFOQ. *We consider only the case of* COUNT. *This program only contains one procedure* count, *that is a recursive procedure. We verify that* COUNT ∈ WIDTH$_{\leq 1}$ *by computing* width$_{\texttt{COUNT}}$(count)*:*

$$\begin{aligned}\text{width}_{\texttt{COUNT}}(\texttt{count}) &= \dots \\ &= \max(w^{\texttt{count}}_{\texttt{COUNT}}(\textbf{qcase}\ \dots), 0) \\ &= \max(w^{\texttt{count}}_{\texttt{COUNT}}(\textbf{call}\ \texttt{count}(\bar{q}^{\boxminus}, \bar{r} \ominus [1]);), w^{\texttt{count}}_{\texttt{COUNT}}(\textbf{call}\ \texttt{count}(\bar{q}^{\boxplus}, \bar{r} \ominus [1]);)) \\ &= \max(1,1) = 1.\end{aligned}$$

*A similar analysis can be performed to determine that* SEARCH ∈ LFOQ.

While LFOQ is a subset of PFOQ, none of the PFOQ programs of Chapter 5 were in LFOQ, as none followed the HALF restriction. While important, the HALF condition by itself does not ensure polylogarithmic-time termination, as the doubling of procedure calls can lead to, for instance, linear-time iteration.

**Example 6.7.** PARITY ∉ LFOQ, *as it does not satisfy the* WIDTH$_{\leq 1}$ *condition:*

$$\begin{aligned}\text{WIDTH}_{\texttt{PARITY}}(\texttt{parity}) &= \max(w^{\texttt{parity}}_{\texttt{PARITY}}(\textbf{call}\ \texttt{parity}\dots), 0) \\ &= w^{\texttt{parity}}_{\texttt{PARITY}}(\textbf{call}\ \texttt{parity}(\bar{q}^{\boxplus}, \bar{r});) + w^{\texttt{parity}}_{\texttt{PARITY}}(\textbf{call}\ \texttt{parity}(\bar{q}^{\boxminus}, \bar{r});) = 2.\end{aligned}$$

*Furthermore,* $\text{Time}_{\texttt{PARITY}}([n_1, n_2]) = n_1 + 1$, *for* $n_2 > 0$, *therefore it is not a polylogtime program.*

It is not surprising that PARITY ∉ LFOQ, as the problem of determining the parity of a string cannot be solved (even approximatively) in quantum polylogarithmic time, as was shown in Lemma 3.10.

Another property of LFOQ is that we can also use the syntactic sugar given in Definition 4.1 to create larger LFOQ programs, similarly to the construction of larger PFOQ programs (Lemma 5.16).

**Lemma 6.8.** LFOQ *is closed for the program construction rules in Definition 4.1 whenever the procedure calls already satisfy the* HALF *and* WIDTH$_{\leq 1}$ *conditions whenever the subprograms are replaced with* **skip**.

We now consider the relation between FBQPOLYLOG and LFOQ. We start by showing that any LFOQ program can be simulated by a QRATM running in polylogarithmic time.

### 6.2.1 Soundness

First, we consider the fact that the largest possible number of procedure calls that can be performed in LFOQ programs is bounded polylogarithmically.

**Theorem 6.9** (Polylogarithmic time)**.** *Let* $\mathrm{P} \in \textsc{lfoq}$*. Then, there exists a* $k \in \mathbb{N}$ *such that, for all* $n \in \mathbb{N}$*,* $\mathrm{Time}_{\mathrm{P}}(n) = O(\log^k(n))$.

*Proof.* Let $\mathrm{P} \triangleq \mathrm{D} :: \mathrm{S}$ and let **call** proc(s); $\in \mathrm{S}$. If no such procedure exists the program can be shown to run in constant time.

We now consider the set of (mutually) recursive procedure calls originating from proc. From the $\textsc{width}_{\leq 1}$ and HALF conditions, there are only $O(\log n)$ such procedure calls and therefore at most $O(\log n)$ calls to procedures of strictly smaller rank. The level is then bounded by $O(\log^k n)$ where $k$ is the largest rank of a procedure in P. □

We now show that any LFOQ program admits a QRATM that simulates it. One question that we do not address in this thesis is what is an appropriate restriction of amplitude transitions in the polylogarithmic case. This question was discussed in Chapter 2 in the case of polytime QTMs, but no such result exists for the polylogarithmic time classes. For our purposes, we will only ensure that, for some set of amplitudes $K$, an LFOQ program with unitaries in $K^{2\times 2}$ can be simulated by a QRATM with transition amplitudes in $K$, and vice-versa.

For this reason, we may denote by $\textsc{lfoq}_K$ the set of LFOQ programs with amplitudes in $K$, but we mostly leave this notation implicit.

**Lemma 6.10.** *For all* $\mathrm{P} \in \textsc{lfoq}_K$*, there exist* $T : \mathbb{N} \to \mathbb{N}$ *and a QRATM* $M$ *with amplitudes in* $K$ *that computes* $[\![\mathrm{P}]\!]$ *in time* $T$ *with* $T(n) = O(\log(n) + \log(n) \times \mathrm{Time}_{\mathrm{P}}(n))$.

*Proof.* This can be shown with an argument along the lines of the proof of Lemma 4.3. To handle the fact that the machine must be able to run in sublinear time, the following changes can be made:

- **Qubit addresses:** instead of updating the list of qubits entering each procedure call, we can equivalently consider that the information about the list is kept in the form of its transformations, namely a sequence of qubit removals such as first half ($\bar{\mathrm{q}}^{\boxminus}$), second half ($\bar{\mathrm{q}}^{\boxplus}$), or removal of specific indices ($\bar{\mathrm{q}} \ominus [\mathrm{i}]$) such that this information can be kept in a polylogarithmic length string. Qubit addresses are then obtained by computing the specific address from this string, which can be done in logarithmic time.

- **Qubit state evolution** To apply a unitary evolution to a qubit, the QRAM uses the query state to access the qubit address. The machine enters the *query* state and swaps the last bit of $t_{\mathrm{QRAM}}$ with $t_{\mathrm{in}}([\![\mathrm{q}]\!])$, transitioning to state $s_{\mathrm{accept}}$. Then, the machine performs a transition according to the unitary operator. For entry state $s$ and exit state $s'$, we have

$$\forall a \in \{0,1\}, \quad \delta(s, t_{in}([\![\mathrm{q}]\!]), s', a) \triangleq \langle t_{in}([\![\mathrm{q}]\!]) | [\![\mathrm{U}]\!](g)([\![\mathrm{j}]\!]) | a \rangle.$$

  Afterwards, the machine reenters the query state and swaps the (modified) last symbol of $t_{\mathrm{QRAM}}$ with the current value of $t([\![\mathrm{q}]\!])$. Finally, the machine erases $\|[\![\mathrm{q}]\!]$ at the end of $t_{\mathbb{K}}$ and leaves its head in the last non-blank cell of $t_{\mathbb{K}}$.

- **Size of qubit lists:** the size of a list of qubits can also be computed in time linear on the *length* of the list, either by reducing the value by one or dividing by two, according to the list description.

- **Set of amplitudes:** as in the proof of Lemma 4.3, if the amplitudes in the program are in $K$, so are the amplitudes of the simulating QRATM.

The remaining details of the proof follow the ones of Lemma 4.3 for normal quantum Turing machines. □

Having shown that LFOQ programs are sound for FBQPOLYLOG, we now prove that the language is *complete* for this class.

### 6.2.2 Completeness

In order to show that LFOQ programs are complete for quantum polylogarithmic time, we demonstrate how any QRATM running in polylogtime can be simulated by an LFOQ program.

This proof is more involved than the one for the completeness of PFOQ in the polynomial-time case (Theorem 5.24), as we attempt to simulate a QRATM directly with an LFOQ program, as opposed to proving completeness with regards to an already-existing implicit characterization.

An added difficulty also comes from simulating the *query* state of a QRATM, where the machine can access any cell on the input tape in a single transition, according to the address written on the index tape. To simulate this step, we perform a binary search of the input cell by following each symbol in the index tape.

**Theorem 6.11** (Completeness)**.** $\mathrm{FBQPOLYLOG} \subseteq [\![\mathrm{LFOQ}]\!]_{\geq \frac{2}{3}}$.

*Proof.* We define an LFOQ program simulating $M$. Without loss of generality we consider a single-tape QRATM. Let $\delta : Q \times \{0,1,\#\}^2 \to K^{Q \times \{0,1,\#\}^2 \times \{L,R\}^2}$ be the transition function of $M$. We obtain a program simulating $M$ by simulating the iteration of $\delta$ a polylogarithmic number of times. To that end, we will define two procedures: `access_input` and `iterate`. These procedures will make use of the following qubit lists and variables: the qubit lists $\bar{\mathrm{q}}_{\mathbb{K}}, \bar{\mathrm{q}}_{\mathrm{in}}, \bar{\mathrm{q}}_{\mathrm{out}}$ encode an index tape, input and output tapes, respectively. The main procedures will work as follows:

- `local_step`: simulates a constant-time transition of $M$ locally on three adjacent cells of the index tape and the work tape.
- `access_input`: allows for QRAM-like access to the input tape by performing quantum branching on each cell of the index tape. The correct cell on the input tape to be read is determined by "splitting" in half the set of possible input tape addresses according to the value of each index tape cell. This procedure is used in `local_step`.
- `full_step`: performs `local_step` iteratively to simulate a transition of $M$ over the entirety of the index and work tapes.
- `iterate`: executes `full_step` a polylogarithmic number of times, simulating the entire run of $M$.

We now define all of these procedures and show how they can be combined to obtain an LFOQ program simulating $M$. To do so, we encode $M$'s tapes in a way that allows for the transition function $\delta$ to be applied *locally* [Yam22], as in the simulation of QTMs via quantum circuits [Yao93, MW19].

**Encoding a QTM's state and tapes into qubit lists.** In order to simulate $M$ using an LFOQ program, for a natural number $n \in \mathbb{N}$, we define as $\widetilde{n}$ the encoding of $n$ into its bitstring.

We encode the internal configurations of $M$ as follows. Let $\bar{q}_{\mathbb{K}}, \bar{q}_{\text{in}}, \bar{q}_{\text{w}}$ be qubit lists for encoding the index, input and work tapes, respectively. The state of $\bar{q}_{\mathbb{K}}$ and $\bar{q}_{\text{w}}$ also encode the head positions and current state of the machine. If the QTM is in state $s$ with its head in the $i_{\mathbb{K}}$-the cell on the index tape, and in the $j_{\text{w}}$-th cell on the work tape, then the states of $\bar{q}_{\mathbb{K}}$ and $\bar{q}_{\text{w}}$ are encoded as

$$|\psi_{\mathbb{K}}^{(i_{\mathbb{K}},\pm s)}\rangle \triangleq |(\widetilde{x_i}, \sigma_i)_{i\in 1\ldots|t_{\mathbb{K}}|}\rangle$$

and

$$|\phi_{\text{w}}^{(i_{\text{w}})}\rangle \triangleq |(y_i, \tau_i)_{i\in 1\ldots|t_{\text{w}}|}\rangle,$$

with $x_i \triangleq \pm s$ if $i = i_{\mathbb{K}}$ and $x_i \triangleq 0$ otherwise, and likewise, $y_i \triangleq 1$ if $i = i_{\mathbb{K}}$ and $x_i \triangleq 0$ otherwise. The sign of $\pm s$ serves to indicate whether the single-step transition has already been performed.

**Possible types of local tape transformations.** The machine transition can then be separated into two cases. If it is not in the query state, then it performs a unitary evolution of its state, index and work tapes, and moves its tape heads to an adjacent cell. Without loss of generality, we can consider only the following types of transitions [BV97, BBC+95]. Let $s \neq s_{\text{query}}, s' \in Q$, and $\sigma_{\mathbb{K}}, \sigma'_{\mathbb{K}}, \sigma_{\text{w}}, \sigma'_{\text{w}} \in \{0,1\}$. Then, the transition function is defined by one of two possibilities.

- $|s, \sigma_{\mathbb{K}}, \sigma_{\text{w}}\rangle \to^{\delta} e^{\imath\theta}|s', \sigma'_{\mathbb{K}}, \sigma'_{\text{w}}, d_{\mathbb{K}}, d_{\text{w}}\rangle$, and
- $|s, \sigma_{\mathbb{K}}, \sigma_{\text{w}}\rangle \to^{\delta} \cos\theta|s', \sigma'_{\mathbb{K}}, \sigma'_{\text{w}}, d_{\mathbb{K}}, d_{\text{w}}\rangle + \sin\theta|s'', \sigma''_{\mathbb{K}}, \sigma''_{\text{w}}, d'_{\mathbb{K}}, d'_{\text{w}}\rangle$,

for $\theta \in [0, 2\pi)$. We extend $\delta$ to handle $s_{\text{query}}$ with the rule $|s_{\text{query}}, \sigma_{\mathbb{K}}, \sigma_{\text{w}}\rangle \to^{\delta} |p, \sigma_{\mathbb{K}}, \sigma_{\text{w}}, 0, 0\rangle$, where $p$ is a designated state which, to ensure unitarity, can only be accessed via $s_{\text{query}}$.

Given our encoding of $M$'s index and work tapes, this can be done by taking each triple of cells in $t_{\mathbb{K}}$ and $t_{\text{w}}$ and checking for the head positions, where, for instance, in the case of a phase-shift, if $d_{\mathbb{K}} = 1$, then

$$|(\tilde{0}, \sigma_{i-1})\rangle|(\widetilde{s}, \sigma_i)\rangle|(\tilde{0}, \sigma_{i+1})\rangle \to^{U^{\delta}} e^{\imath\theta}|(\tilde{0}, \sigma_{i-1})\rangle|(\tilde{0}, \sigma'_i)\rangle|(\widetilde{-s'}, \sigma_{i+1})\rangle,$$

and, in the case of a rotation, if $d_{\mathbb{K}} = 1$ and $d'_{\mathbb{K}} = -1$,

$$\begin{aligned}|(\tilde{0}, \sigma_{i-1})\rangle|(\widetilde{s}, \sigma_i)\rangle|(\tilde{0}, \sigma_{i+1})\rangle \to^{U^{\delta}} \cos\theta \cdot |(\tilde{0}, \sigma_{i-1})\rangle|(\tilde{0}, \sigma'_i)\rangle|(\widetilde{-s'}, \sigma_{i+1})\rangle \\ + \sin\theta \cdot |(\widetilde{-s''}, \sigma_{i-1})\rangle|(\tilde{0}, \sigma''_i)\rangle|(\tilde{0}, \sigma_{i+1})\rangle.\end{aligned}$$

These transitions correspond to phase-shifts and rotations, respectively, which for a specific machine $M$ can be encoded by a composition of statements without any procedure call. Let `unitary_step`$(\bar{q}_{\mathbb{K}}, \bar{q}_{\text{out}})$ be the constant-time procedure that executes $U^{\delta}$. We use this procedure to define a larger one, `local_step`$(\bar{q}_{\mathbb{K}}, \bar{q}_{\text{in}}, \bar{q}_{\text{w}})$ which applies the unitary transformation or the input access in case $M$ is in the query state.

```
1  decl local_step(q̄_K, q̄_in, q̄_w){
2    qcase q̄_w[1..⌈log(|Q|)⌉ + 1] of {
3      |s_query⟩ → call access_input(q̄_K, q̄_in, q̄_w ⊖ [1, ..., ⌈log(|Q|)⌉ + 1]);
4      |s̃⟩ → skip; }
5    call unitary_step(q̄_K, q̄_w); }
```

We now describe how we may simulate the access to the input state using LFOQ recursion. Using each subsequent value in the index tape, we search either on the left or on the right of the input tape, until we find the correct cell and perform a swap operation with the correct work cell. This is described in the procedure $\texttt{access\_input}(\bar{q}_{\mathbb{K}}, \bar{q}_{\text{in}}, \bar{q}_{\text{w}};)$.

```
1  decl access_input(q̄_K, q̄_in, q̄_w){
2    if |q̄_K| = 1 then qcase q̄_K[1] of { //base case: QRAM access
3      0 → SWAP(q̄_w[1], q̄_in[0])
4      1 → SWAP(q̄_w[1], q̄_in[1])}
5    else qcase q̄_K[1] of{ //recursive case: binary search according to index bit
6      0 → call access_input(q̄_K ⊖ [1], q̄_in^⊟, q̄_w);
7      0 → call access_input(q̄_K ⊖ [1], q̄_in^⊞, q̄_w); } }
```

This procedure performs two recursive calls on different branches, and on each of them it halves one of the input qubit lists (in this case, $\bar{q}_{\text{in}}$, i.e. the one representing the input tape).

**Iterating over the tapes.** To obtain a single transition of $M$, it now suffices to iterate `local_step` over the entirety of the work and index tapes. Let $n$ be the size of the input, and $k \triangleq \lceil \log n \rceil$. We may consider the index as having size $k$ and the work tape as having size $O(k^d)$, for some constant $d \in \mathbb{N}$. Then, we can iterate simultaneously over the two tapes by composing `local_step` a $O(k^{d+1})$ number of times. This can be done by using a nested recursion in LFOQ of depth $d+1$. Since $d$ is a constant, we can define functions $\texttt{iterate}_1, \ldots, \texttt{iterate}_{d+1}$, in the following way for $i = 1, \ldots, d$,

```
1  decl iterate_i(·, q̄_1, ..., q̄_d){
2    call iterate_i(·, q̄_1, ..., q̄_i^⊞, ..., q̄_d);
3    call iterate_{i+1}(·, q̄_1, ..., q̄_d); }
```

These procedures are congruent with LFOQ's restrictions, as the procedures satisfy $\texttt{iterate}_i > \texttt{iterate}_{i+1}$ and therefore $\text{width}(\texttt{iterate}_i) = 1$, for each $i$. Then, if the qubit lists $\bar{q}_1, \ldots, \bar{q}_d$ all have size $n$, the result of a call to $\texttt{iterate}_1(\cdot, \bar{q}_1, \ldots, \bar{q}_d)$ is precisely $\log^{d+1} n$ calls to $\texttt{iterate}_{d+1}$.

**The full simulation.** To finalize the simulation of $M$, we simply put together the procedures we have so far created. Let $d \in \mathbb{N}$ be such that $M$ on an input of size $n$ runs for $\log^d n$ steps. We define `iterate` as a procedure that composes `full_step` a number $\log^d n$ times, using the technique shown above, by defining

$$\texttt{iterate} \triangleq \texttt{iterate}_1 \qquad \text{and} \qquad \texttt{full\_step} \triangleq \texttt{iterate}_d.$$

Similarly, the procedure body of `full_step` will be defined with nested iteration performing the $\log^{d+1} n$ composition of `local_step`.

To perform the full simulation, for an input of size $n$, if $k \triangleq \lceil \log n \rceil$, we make use of qubit lists $\bar{q}_{\mathbb{K}}, \bar{q}_{\text{in}}, \bar{q}_{\text{w}}$ of size $k$, $n$, and $k^d$, as well as $2d-1$ qubit lists of size $n$ used to control the iteration of `full_step` and `iterate`.

This concludes the proof. □

In this chapter, we defined a fragment of FOQ, namely LFOQ, that is sound and complete for quantum polylogarithmic time. This provides an implicit characterization of the class FBQPOLYLOG, whose definition is based on a polylogtime-bounded quantum random-access Turing machine. The decision-problem equivalent class BQPOLYLOG was implicitly defined via a function algebra in [Yam22], and we claim that our language is simpler and more expressive.

LFOQ was also shown to satisfy certain desirable properties. For instance, like PFOQ, LFOQ is closed for the inverse transformation given in Definition 4.7 (the argument is not repeated as the proof is similar to the one of Lemma 5.14 for PFOQ programs). As claimed in Lemma 6.8, we also saw that larger LFOQ programs can be built from smaller ones using standard syntax, which is also a testament to the robustness of the chosen syntactical restrictions.

We were also able to bound the power of LFOQ by separating quantum polynomial and polylogarithmic-time, using the quantum query complexity bound of Lemma 3.10. For this reason, we can conclude that no LFOQ program is capable of approximating the AND, OR and PARITY. Conversely, it is not difficult to define PFOQ programs for these functions.

In Part III, we will tackle the problem of compiling LFOQ programs into quantum circuits of appropriate complexity. In the case of polylogarithmic time, it is not expected that corresponding circuits are of polylogarithmic *size*, but rather of polylogarithmic *depth*. This is because a polylogtime program can still refer to all the qubits in the input over all its branches of computation. This is evident in the example programs `SEARCH` and `COUNT` in Figure 6.1.

# Part III

# Quantum circuit compilation

# 7

# A Resource-Preserving and Tractable Compilation Algorithm

So far, we have concerned ourselves only with connecting FOQ and its fragments, PFOQ and LFOQ, with the QTM model of quantum computation. This model, while intricate in its details, is useful for reasoning about programs and their complexity. This is due to the fact that the QTM model lends itself quite well to reasoning over programming primitives. For instance, the composition of two QTMs can be defined with a bigger QTM, whose time complexity is the sum of the time complexities of the two, smaller QTMs [BV97, Dovetailing Lemma].

Likewise, one can reason about a machine that, depending on the state of the symbol it reads, branches in a superposition of simulating one machine or another on the remaining tape. The running time of such a machine is the maximum running time of the two smaller QTMs [BV97, Branching Lemma].

We will see that this reasoning cannot be directly imported into quantum programs and their circuits, namely when it comes to *quantum branching*. Quantum branching is generally defined as the use of a quantum state as the guard of the control flow of a program. For instance, consider the FOQ statement

$$\textbf{qcase of } \mathrm{q} \textbf{ then } \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}.$$

This statement indicates that, depending on the quantum state of the qubit address corresponding to q, we execute statments $\mathrm{S}_0$ and $\mathrm{S}_1$ in superposition. Since we cannot observe the qubit's state mid-computation, a circuit implementation will require that we include the circuits of both statements. Therefore, the circuit complexity in this case is not the maximum circuit size between $\mathrm{S}_0$ and $\mathrm{S}_1$, but instead their sum. This is the case whenever we have no information about the structure of $\mathrm{S}_0$ and $\mathrm{S}_1$, and no simplification is apparent.

We now motivate this problem with a specific example of a program that performs recursive quantum branching.

## 7.1 The problem of exponential blow-up

This mismatch between our intuitive reasoning about the **qcase** statement and its naïve circuit implementation poses a serious problem for the quantum programmer, as in many cases a program's circuit complexity can be exponentially larger than the expected one, derived from a straightforward reasoning about its primitives.

Given the importance of preserving the time complexity of a program in its circuit implementation, there is an interest in discovering general techniques that avoid the problem of branch

```
1  decl pairs(q̄){
2    if |q̄| ≥ 2 then
3      qcase q̄[1,2] of {
4        00 → call pairs(q̄ ⊖ [1,2]);
5        01 → skip;
6        10 → skip;
7        11 → call pairs(q̄ ⊖ [1,2]);
8        }
9    else q̄[1] *= NOT;}
10 :: call pairs(q̄);
```

Figure 7.1: Program PAIRS.

sequentialization. We will show that this can be done partly for PFOQ programs, and optimally for a fragment of PFOQ that is complete for QP.

### 7.1.1 Motivating example

To demonstrate how the problem of recursive branching can incur an exponential blow-up, we consider the simple example of the PFOQ program PAIRS defined in Figure 7.1.

With $\bar{x} \in \{0,1\}^*$ and $y \in \{0,1\}$, given the input state $|\bar{x}y\rangle$, pairs will apply a $NOT$ gate to $y$ if and only if $\bar{x}$ is a string consisting only of sequences of 00 and 11. Put another way, pairs encodes the unitary transformation that inverts the state of the last qubit of an input when $\bar{x}$ belongs to the regular language defined by $(00 \mid 11)^*$.

From the point-of-view of the programmer, pairs is a procedure with linear complexity since it performs at most one recursive call per branch, and consumes 2 qubits from its input while doing so, and therefore its runtime is bounded linearly. This complexity analysis is equivalent to estimating the program's Time function (see Section 4.2): given that a procedure may only execute a constant number of instructions (excluding its procedure calls) the asymptotic complexity of the program is bounded by *the maximum number of procedure calls performed in depth*. For instance, it is easy to see that $\mathrm{Time}_{\texttt{PAIRS}}(n) = \lfloor \frac{n}{2} \rfloor + 1$.

However, when finding an approach to compiling PAIRS, different strategies, while semantically equivalent, can produce circuits that differ in asymptotic complexity. Consider, for instance, the compilation strategies given in Figure 7.2 for the recursive case of pairs, i.e. where the input size is larger than 2.

Strategy **(a)**, which we will call the "in-depth" approach, consists of compiling the circuits for the two recursive calls to pairs in sequence, controlled on the first two qubits according to the corresponding branches. Each instance of pairs then produces two more, in sequence, with an input containing 2 fewer qubits, and results in a circuit of size $\Theta(n2^n)$.

Strategy **(b)**, which we call the "in-width" approach, applies QRAM techniques to parallelize the two calls to pairs, by making use of two registers $r_{00}$ and $r_{11}$, to perform each branch in parallel and then recombine the results in the same register. In this case, the final circuit is linear in depth but requires an exponential number of gates and ancillas to be generated.

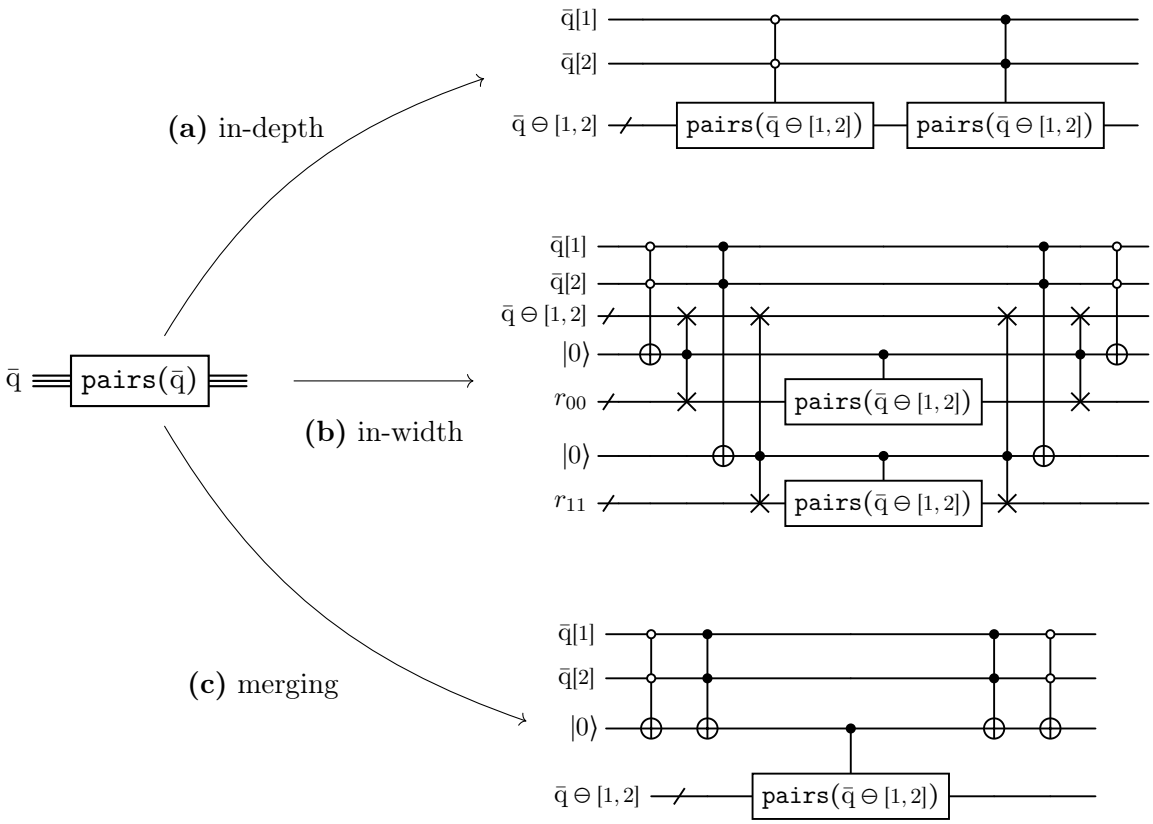



Figure 7.2: Compilation strategies.

Both strategies **(a)** and **(b)** ignore the structure of the branches of the **qcase** statement, namely the fact that the two statements are identical and therefore encode the same unitary transformation. These represent automatic strategies for compiling a quantum control statement.

One relatively simple idea for making use of the symmetry between the two branches is to merge them into a single procedure call to `pairs`, by using an ancilla that controls the procedure call and whose state depends on the two relevant branches. This approach corresponds to strategy **(c)** and results in a final circuit of size $\Theta(n)$.

While strategies **(a)**, **(b)** and **(c)** produce equivalent circuits, on an input of size $n$, strategy **(a)** results in a circuit with an exponential number of gates, and strategy **(b)** results in a circuit with an exponential number of gates and ancillas, while strategy **(c)** results in a circuit of size $\Theta(n)$, making use of $\Theta(n)$ ancillas. Therefore, while straightforward approaches to compilation may result in an exponential blow-up of circuit complexity, the structure of the program can sometimes be used to find a compilation strategy that preserves the program's complexity.

The exponential blow-up present in strategies **(a)** and **(b)** is given by the fact the the two compilation strategies did not make use of the structure of the two branches, and in both cases the circuit size of the **qcase** statement is given by the maximum of each branch, whereas in strategy **(c)** it is given asymptotically by the maximum.

The term *branch sequentialization* is coined in [YC22] in the context of compiling a quantum program for finding an element in a sorted binary tree. The authors notice that the program written in the most straightforward way results in a circuit of linear complexity, instead of the (expected) logarithmic complexity, due to the fact that the procedure is recursive and calls itself on different quantum branches.

In order to obtain the correct complexity, the authors rewrite the program so that it creates an ancilla that is set to the superposition according to the subtree the program should recursively search. This removes the procedure call from inside the **qcase** and avoids the exponential blow-up in the circuit compilation.

Informally, branch sequentialization occurs whenever a compilation strategy applied on a quantum control case, results in a circuit where the size of the circuit scales as the sum of the complexities of each branch, rather than the maximum. A compilation strategy for a given language can be said to avoid branch sequentialization in general if it avoids branch sequentialization on every program.

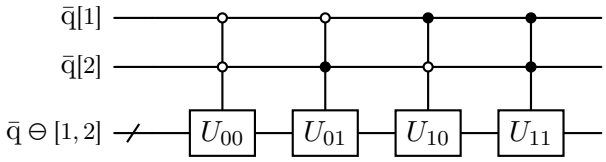
In both `PAIRS` and the binary tree traversal program in [YC22], branch sequentialization is not too dificult to avoid by rewriting the program. However, there are currently no general techniques ensuring no exponential blow-up from the use of **qcase** statements. In this chapter we will show how an exponential blow-up can be avoided for all PFOQ programs, and in Chapter 8 we introduce a fragment of PFOQ, denoted $\text{PFOQ}^{\text{UNIF}}$, that is computationally equivalent to PFOQ but where the asymptotic complexity of a **qcase** statement is given by the maximum of each branch, as in the QTM model. Therefore, there is a strategy that avoids branch sequentialization on a language that is complete for quantum polynomial time.

### 7.1.2 The limits of quantum demultiplexing

The problem of avoiding the sequentialization of operations in a quantum case – i.e. a **qcase**-like statement – has been widely studied using models of quantum random-access memory (QRAM). The problem is usually stated in the following way. Given $n$ control qubits and $k$ target qubits, with $N \triangleq 2^n$ possible control qubit states, what is the complexity of performing the unitary

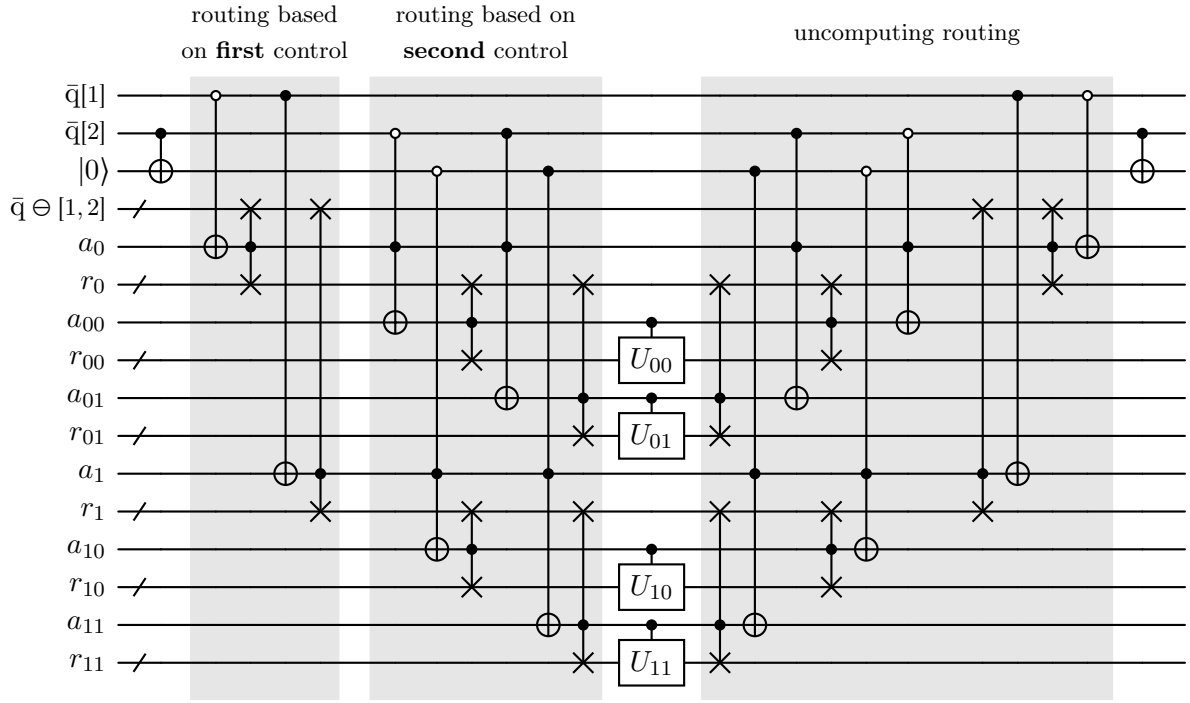
$$\sum_{i=0}^{N-1} |i\rangle\langle i| \otimes U_i$$

where each $U_i \in \mathcal{H}_{2^k} \times \mathcal{H}_{2^k}$ has time-complexity $T_i$. One straightforward approach is implementing each controlled-$U_i$ in sequence, as in strategy **(a)** in Figure 7.2, with time complexity $O(\sum_{i=1}^{n} T_i)$. For instance, given $n = 2$, this corresponds to the circuit




Another approach is to *parallelize* the instances of $U_i$, via a generalization of the QRAM protocol. An example of such a construction is the following circuit, where $r_{ij}$ for $i, j \in \{0, 1\}$ indicate registers of size $k$ where the respective $U_{ij}$ is applied, and $r_0$ and $r_1$ are intermediate registers used in the routing procedure [LS01, STY+23, ZLY22].

The routing is structured like a binary tree. In this case, the most significant bit, given by $\bar{q}[1]$, is used to send the target qubits either into register $r_0$ or $r_1$, depending on its value. This is done with two controlled swaps. Then, depending on the state $j \in \{0, 1\}$ of the next most significant bit, $\bar{q}[2]$, the register $r_i$ is swapped with register $r_{ij}$. This is done with 4 Fredkin gates which can be parallelized into two time-steps. For controls of arbitary size, each level of the binary tree can be performed in constant time, by employing an exponential number of ancillas. In Figure 7.2, the quantum demultiplexer would roughly correspond to strategy **(b)**.




In some scenarios, such a construction can be shown to be optimal [ZLY22]. The exponential space needed for the circuit is unavoidable, given that we have no information on the structure of each $U_i$.

Both *in-depth* and *in-width* strategies result in circuits of exponential size in the case of a simple program such as `PAIRS`. The merging technique results in a circuit of linear size as it allows the compiler to factor out repeated procedure calls. Our objective in this last part of the

thesis will be to exploit the structure of PFOQ and LFOQ programs in order to obtain circuits of adequate asymptotic size.

We will show that PFOQ programs allow for such techniques and, after formalizing the notion of branch sequentialization introduced in [YC22], we show that we can restrict PFOQ without reducing its expressivity and obtain a fragment where branch sequentialization is avoided altogether for polynomial-time programs.

## 7.2 Anchoring and merging

In this section, we discuss how we can extend the idea for the *merging* compilation strategy for `PAIRS` (Figure 7.1) to arbitrary programs in PFOQ. In compiling a procedure call to `pairs`, we made use of the fact that the two branches in the **qcase** statement were identical, in that they contained two calls to `pairs` on precisely the same input. We can consider more general scenarios: different inputs, different procedure calls, procedures making use of classical inputs, and the general structure of the **qcase** substatements being more complex than a procedure call.

Our strategy for compilation, introduced in the next section, can be described as *anchoring and merging*. It consists of performing the following two types of operations while compiling the program:

- *anchoring*: in the case of a *new* procedure call, an ancilla is created that contains its quantum control information, and is used to control the procedure statement;
- *merging*: if a *repeated* procedure call occurs, instead of being compiled, the information of its quantum control is sent to the compatible ancilla created from anchoring.

The notion of *new* or *repeated* procedure calls is defined by compatibility. Two procedure calls **call** `proc`$[\mathrm{i}](\mathrm{s}_1,\dots,\mathrm{s}_k)$; and **call** `proc'`$[\mathrm{i}'](\mathrm{s}'_1,\dots,\mathrm{s}'_k)$; in contexts $f$ and $f'$ respectively are said to be *compatible* if they correspond to the same procedure, have the same classical input and the quantum inputs have the same size:

$$\texttt{proc} = \texttt{proc'} \quad \wedge \quad (\mathrm{i}, f), (\mathrm{i}', f') \Downarrow_{\mathbb{N}} n \quad \wedge \quad (|\mathrm{s}_i|, f), (|\mathrm{s}'_i|, f') \Downarrow_{\mathbb{N}} m_i.$$

What interests us is that compatible procedure calls represent the same unitary transformation, *up to a permutation of input qubits*. In the case of `pairs`, the permutation was trivial, as the input qubits were the same in both procedure calls appearing in the different branches of the **qcase**. In general, all qubit addresses may have to be changed.

Multiple compatible procedure calls occurring in separate **qcase** branches can be simplified into a single instance, by performing a controlled permutation over the inputs.

**Lemma 7.1.** *Any controlled permutation of $n$ qubits can be performed with a quantum circuit of size $O(n)$ and depth $O(\log n)$.*

*Proof.* Any permutation can be written as the composition of two sets of disjoint transpositions, and therefore any permutation can be performed in constant time, using two time steps [MN01]. To perform a controlled permutation, it suffices to create $O(n)$ ancillas with the correct control, which can be done in $O(\log n)$ depth with $O(n)$ gates. □

As an example, consider the controlled permutation given in Figure 7.3, where the vertical dashed lines separate time slices containing gates which can be implemented concurrently.

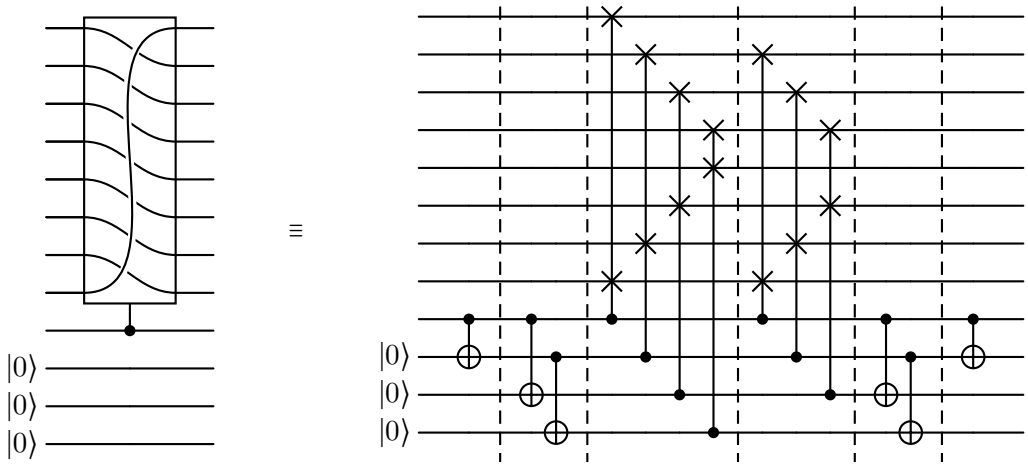



Figure 7.3: Implementation of a controlled permutation in logarithmic depth.

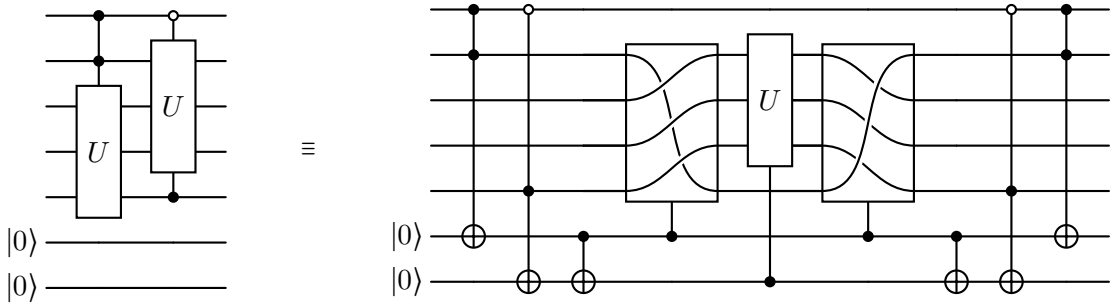



Figure 7.4: Merging orthogonally-controlled statements.

The merging of two compatible procedure calls can then be seen as the merging of unitaries with orthogonal controls. An example is given in Figure 7.4.

Anchoring and merging allows for the compilation procedure to compile only the *unique* procedure calls (those that are not compatible as defined above). While a PFOQ program can result in an exponential number of procedure calls over different branches, only a *polynomial* number of such calls are unique.

**Lemma 7.2.** *Let* $\pi_{\mathrm{P}} \rhd (\textbf{call}\ \texttt{proc}[\mathrm{i}](\mathrm{s}_1, \ldots, \mathrm{s}_k);, |\psi\rangle, A, f)$ *where* $\mathrm{P} \in \textsc{pfoq}$ *and* $\ell(|\psi\rangle) = n$*. The number of subtrees* $\pi' \trianglelefteq \pi$ *with unique procedure calls* $\pi' \rhd (\textbf{call}\ \texttt{proc'}[\mathrm{i}'](\mathrm{s}', \ldots, \mathrm{s}'_k);, |\psi\rangle', A', f')$ *where* $\texttt{proc} \sim_{\mathrm{P}} \texttt{proc'}$ *is bounded by* $O(n^{k+1})$*.*

*Proof.* Non-compatible procedure calls can only differ in procedure name, classical input, and input sizes. There is only a fixed number of procedure names in a given program. The classical input value can only be increased by depth of recursion, which by the WF condition can only be linear. Given that there are $O(n^k)$ possible input values, we obtain our upper bound. □

Having motivated the problem and some of the techniques that will be used, we are now ready to present our first compilation algorithm.

## 7.3 Compilation algorithm

In this section, we introduce a compilation algorithm that ensures polynomial bounds for PFOQ programs, therefore avoiding the exponential blow-up from branch sequentialization.

The compilation algorithm **compile** (Algorithm 1) is described by the rewrite rules of Figure 7.5. The input of compile is a triple

$$(\mathrm{P}, f, cs) \in \text{Programs} \times (Var \to \mathcal{L}(\mathbb{N})) \times (\mathbb{N} \to \{0,1\})$$

where P is a program, $f \in Var \to \mathcal{L}(\mathbb{N})$ is a function that maps qubit list variables to their espective addresses, and $cs$ is a *control structure*.

A control structure $cs$ is a partial map from $\mathbb{N} \to \{0,1\}$ that maps wire numbers to values coding negative or positive control. The empty control structure is denoted by $\{\}$, more generally, we will often use a set extension to exhibit the mapping of a control structure. Given a control structure $cs$, we write $cs[n \coloneqq b]$ to represent the new control structure $cs \cup \{n \mapsto b\}$.

Let a *controlled statement* be a pair of the shape $(cs, \mathrm{S})$, for some control structure $cs$ and some statement S. In the compilation process, controlled statements $(cs, \mathrm{S})$ are used to represent a statement S that remains to be compiled into a quantum circuit $C$ together with a control structure $cs$, representing the qubits controlling the circuit $C$.

$M(cs, n)$ denotes the circuit that has a single controlled gate $M$ acting on wire $n$, controlled under $cs$. For example, $\mathrm{NOT}(\{1 \mapsto 1, 2 \mapsto 1\}, 3)$ denotes a $CCNOT$ with positive control on wires 1 and 2 and acting on wire 3.

Given two lists of qubit pointers (integers) having the same length, we define

$$\mathrm{SWAP}([x_1, .., x_k], [x'_1, .., x'_k])$$

as the permutation gate that maps each $x_i$ to $x'_i$. For simplicity, we extend this notation to lists of lists of qubit pointers. For $l, l' \in \mathcal{L}(X)$, $l@l'$ denotes the concatenation of $l$ and $l'$. Then:

$$\mathrm{SWAP}([\mathrm{s}_1, .., \mathrm{s}_n], [\mathrm{s}'_1, .., \mathrm{s}'_n]) \triangleq \mathrm{SWAP}(\mathrm{s}_1@\cdots@\mathrm{s}_n, \mathrm{s}'_1@\cdots@\mathrm{s}'_n).$$

As for unary gates, the controlled version is defined by passing a control structure as first argument: $\mathrm{CSWP}(cs, [\mathrm{s}_1, \ldots], [\mathrm{s}'_1, \ldots])$. Note that by Lemma 7.1, such a control swap gate can be performed by a circuit of linear size and logarithmic depth.

In addition to control structures, the algorithm also manipulates lists (denoted with square brackets) and dictionaries. Functions *hd* and *tl* give access to the head and the tail of the list, respectively. Given a dictionary $d$, we can test if it has a given key $k$ with $k \in d$. Then we can assign or access the values with the syntax $d[k]$.

Generating the circuit corresponding to the program P = D :: S will consist in running **compile** on the controlled statement $(\{\,\}, \mathrm{S})$, with a function $f : Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N})$ corresponding to the map of qubit pointers lists used in the semantics. Given a program P with variables $Var(\mathrm{P}) = \{\bar{\mathrm{q}}_1, \ldots, \bar{\mathrm{q}}_k\}$, we use the following shorthand notation

$$\textbf{compile}(\mathrm{P}, [n_1, \ldots, n_k]) \triangleq$$
$$\textbf{compile}(\mathrm{P}, \{\bar{\mathrm{q}}_1 : [1, \ldots, n_1], \bar{\mathrm{q}}_2 : [n_1 + 1, \ldots, n_1 + n_2], \ldots, \bar{\mathrm{q}}_k : [\sum_{i=1}^{k-1} n_i + 1, \ldots, \sum_{i=1}^{k} n_i]\}, \{\,\})$$

for the circuit obtained for program P on input sizes $|\bar{\mathrm{q}}_1| = n_1, \ldots, |\bar{\mathrm{q}}_k| = n_k$.

The algorithm generates the quantum circuit corresponding to a PFOQ program inductively on the statement S. Given the rules of Figure 7.5 when **compile** is called on a given controlled

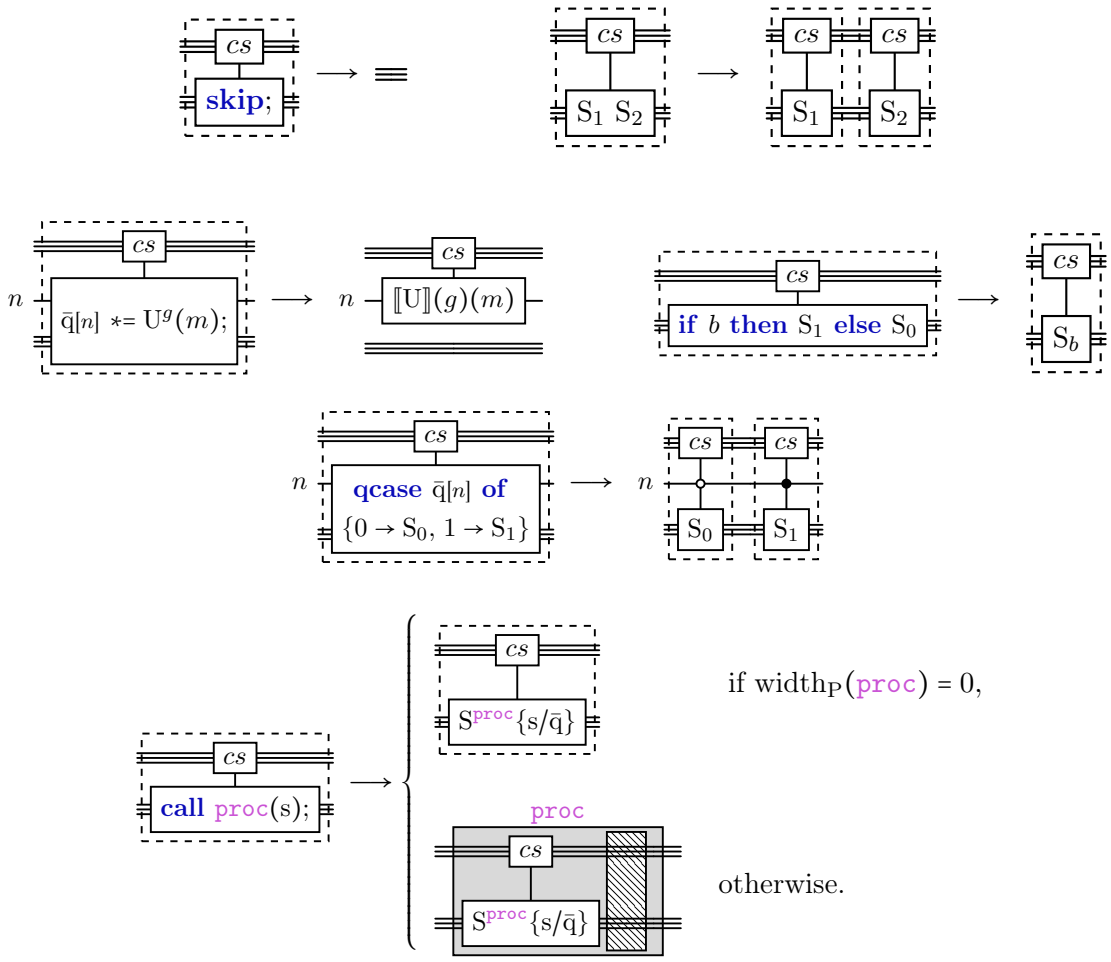



Figure 7.5: Rewrite rules of **compile**.

statement $(cs, \mathrm{S})$, an inductive call to **compile** is performed on controlled statements whose statements are sub-statements of S. The two base cases are the rules for the skip statement and the unitary application. In these cases, the compilation just outputs the identity circuit and a controlled gate computing the unitary, respectively.

The rules for sequence and quantum case perform two inductive calls to **compile** on each branch of the statement. The rule for quantum case is the only rule that directly performs changes on the control structure. In the particular case of a call to a recursive procedure, i.e., when $\mathrm{width}_{\mathrm{P}}(\texttt{proc}) > 0$, **compile** calls the **optimize** subroutine (Algorithm 2) to perform anchoring and merging. This call to **optimize** is highlighted in Figure 7.5 through the use of a shaded square ▨, which takes the procedure name `proc` as superscript. We call this process the *optimization* of procedure `proc`.

The recursive call case in Figure 7.5 also includes a striped rectangle. For the purposes of describing **compile** and **optimize**, this rectangle can be ignored, as it will only be of use in an improved version of the algorithm, **compile**$^+$ and **optimize**$^+$, both described in Section 8.3.

The rewrite rules for the subroutine **optimize** on procedure `proc` are described in Figure 7.6.

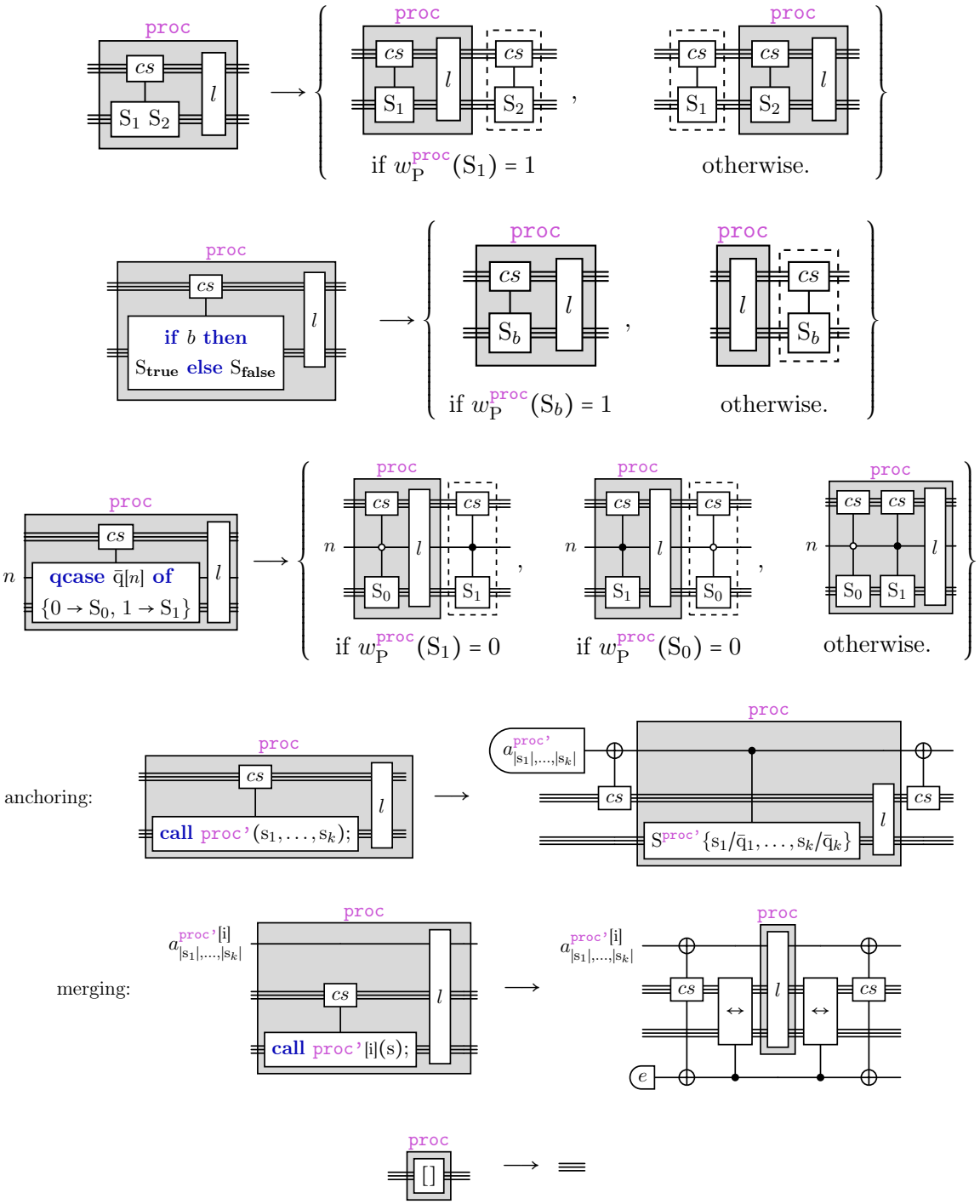



Figure 7.6: Rewrite rules of **optimize**.

The algorithm **optimize** evolves a list $l$ of controlled statements, based on the $\text{WIDTH}_{\leq 1}$ condition. In Figure 7.6, the rules are applied by considering the first controlled statement in the list $l$. We just specify the most interesting cases below:

- for a controlled statement $(cs, \mathrm{S}_1\ \mathrm{S}_2)$, two distinct rules can be applied. In the first scenario,

where $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_1) = 1$, we have that $\mathrm{S}_1$ contains a recursive procedure call and $\mathrm{S}_2$ does not (this is a consequence of the $\textsc{width}_{\leq 1}$ condition in PFOQ), or the converse. Depending on this, the rule selects on which control statement $(cs, \mathrm{S}_1)$ or $(cs, \mathrm{S}_2)$ to perform the optimization and, appends the compiled circuit of the other controlled statement to the left or to the right.

- for an **if** statement, we precompute the boolean value $b$ (the value is computable as it only depends on classical data), and according to whether the corresponding branch contains a recursive call or not, we either add it to $l$ or compile it separately.
- for a **qcase** statement, one or two of the branches are added to the list $l$ depending on whether they are both recursive or not.
- for a controlled statement with a procedure call **call** `proc'`(s); we perform either anchoring or merging depending on whether the ancilla $a_{|\mathrm{s}|}^{\texttt{proc'}}$ exists or not. We denote by $\boxed{a}$ or $\boxed{e}$ a completely fresh ancilla created in state $|0\rangle$.
- when the list of control statement is empty (i.e., $l = []$), the algorithm terminates. The controlled gate $\boxed{\leftrightarrow}$ encodes the swapping of qubit positions, for instance using the construction given in Lemma 7.1.

One simple but important detail of the compilation is that the statements in $l$ must be handled by non-increasing order of number of accessible qubits, to ensure that merging of procedures is semantically valid. This can be seen by the fact that the validity of the merging step, in the proof of Lemma 7.4, assumes that at the moment of merging an instance of a procedure `proc` on input size $n$, that the corresponding procedure statement $\mathrm{S}^{\texttt{proc}}$ is present in $l$, which is only true in general if the statement has not yet been treated in **optimize**. This can be ensured by keeping a partial order of accessible qubits over the elements in $l$.

### 7.3.1 Correctness of the algorithm

In this section, we discuss the validity of the compilation algorithm. One first observation should be that, given a a program, the compilation necessarily terminates. For instance, in **compile** (Figure 7.5), all rules besides the procedure call rewrite the controlled statement into either a circuit or into instances of **compile** of smaller statements. In the case of a procedure call, the rewriting of the procedure body produces a finite number of calls to procedures of lower recursive level.

In **optimize**, a recursive procedure will result in a finite number of calls to mutually-recursive procedures – this is ensured by the well-foundedness condition WF, that requires that recursive procedure calls reduce the size of the input, therefore procedure calls either reduce the level of recursion or the number of available qubits.

The soundness of the compilation algorithm is given by an orthogonality invariant in **optimize**. Let $cs$, $cs'$ be two control structures. We say that $cs$ and $cs'$ are *orthogonal* if there exists $i \in \mathrm{dom}(cs) \cap \mathrm{dom}(cs')$ such that $cs(i) = 1 - cs'(i)$. Two controlled statements are orthogonal if their control structures are orthogonal.

**Lemma 7.3.** *During the compilation of a* PFOQ *program* P*, for each optimization of a (recursive) procedure* `proc`*, all controlled statements* $l$ *are pairwise orthogonal.*

*Proof.* This invariant can be easily inspected from the rewrite rules in Figure 7.6. For instance, in the case of a sequence of instructions or a classical **if** statement, the control structure $cs$ is either kept in the list as it was before, or removed. In any of these two cases, the invariant is maintained.

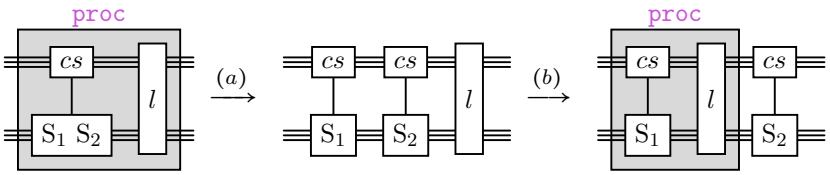
The only rule where the number of instances of $cs$ increases is the one for the **qcase** statement. In this case, control statements $cs[n := 0]$ and $cs[n := 1]$ can still appear in the list of orthogonally-controlled statements. Since, by induction, $(cs, \mathrm{S})$ was orthogonal to all other controlled statements, we can conclude that the same is true for $cs[n := 0]$ and $cs[n := 1]$ separately, and clearly these two control structures are mutually orthogonal. □

The orthogonality invariant ensures the validity of **optimize**, and the soundness of the compilation algorithm. It is also a consequence of the $\textsc{width}_{\leq 1}$ restriction in $\textsc{pfoq}^{\textsc{unif}}$, as by the definition of width, at most one recursive call may appear per branch of a recursive procedure. This ensures that two recursive calls on a given procedure always occur in orthogonal branches and can be simply combined in the same ancilla. Given a circuit $C$, we define its semantics $[\![C]\!]$ naturally as the composition of the semantics of each gate.

**Theorem 7.4** (Correctness of compilation)**.** *Given a* PFOQ *program* P *and a quantum state* $|\psi\rangle \in \mathcal{H}_{2^n}$ *we have that* $[\![\mathbf{compile}(\mathrm{P}, n)]\!](|\psi\rangle) = [\![\mathrm{P}]\!](|\psi\rangle)$.

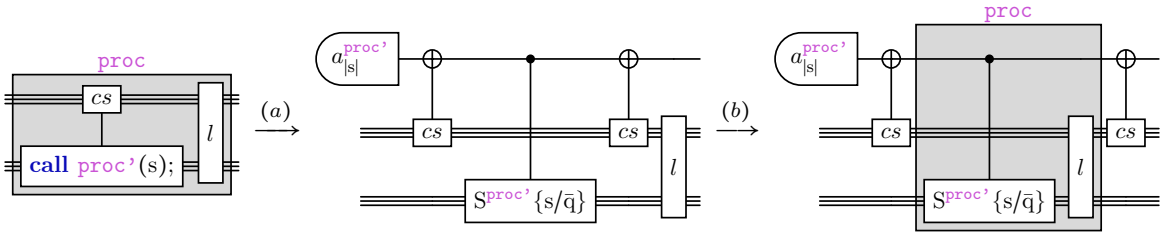
*Proof.* By induction on the structure of the program $\mathrm{P} = \mathrm{D} :: \mathrm{S}$. One can check by inspection of each case that the **compile** rules for non-recursive statements corresponds to the straightforward circuit semantics of quantum programs.

Likewise, the rewriting rules of **optimize**, given in Figure 7.6, can be easily checked using the orthogonality invariant of Lemma 7.3. For instance, consider the case of the sequential statement $\mathrm{S} = \mathrm{S}_1\ \mathrm{S}_2$. In the case where $w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}_1) = 1$, we can derive the rule for the statement with the following steps:



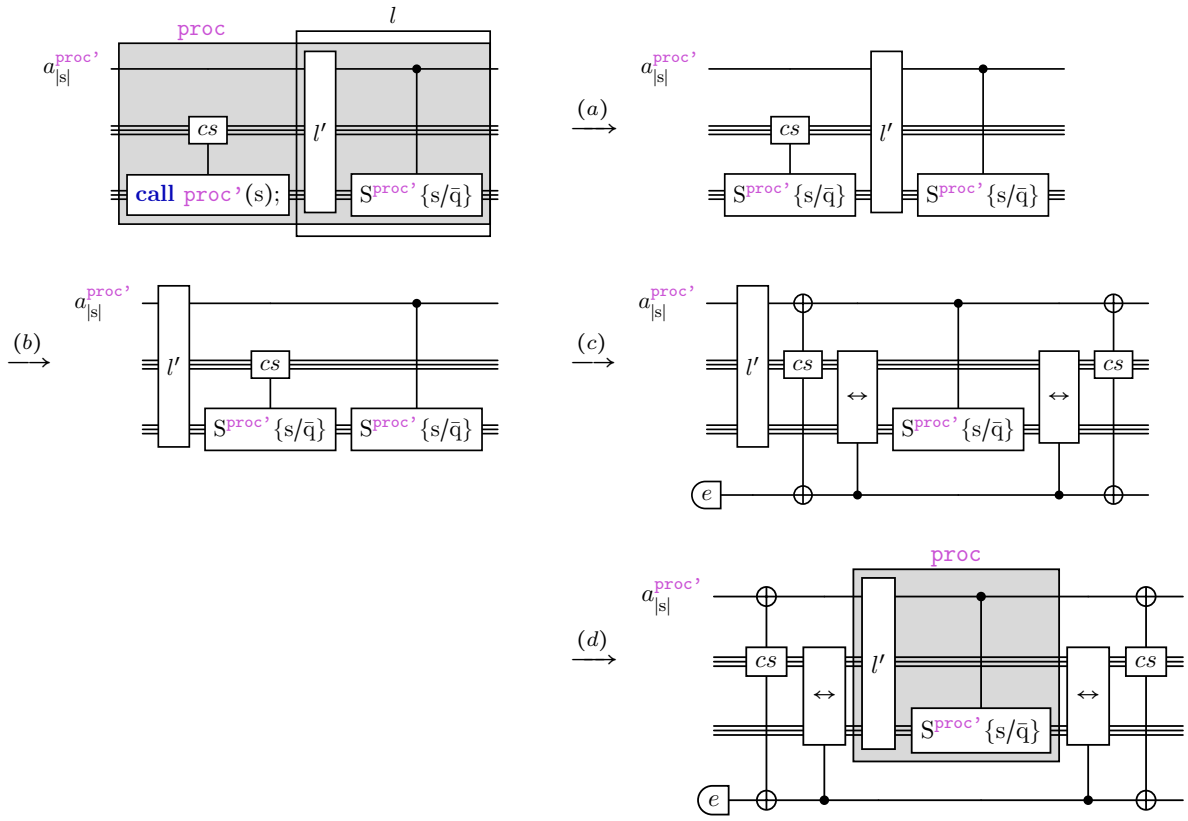

where $(a)$ corresponds to the definition of the sequential statement, and in $(b)$ we make use of the fact that $(cs, \mathrm{S}_2)$ is orthogonal to all controlled statements in $l$ and therefore can be commuted in the circuit.

Other cases can be inspected to follow a short sequence of steps, such as described for the sequential statement. For instance, in the case of anchoring, we consider the following composition of rules




where, likewise, $(a)$ corresponds to the typical circuit semantics of a procedure call (with an added anchoring ancilla), and $(b)$ makes use of the orthogonality of $cs$ with the other controlled

statements in $l$. The validity of merging and also be checked:




Step $(a)$ can be seen as the definition of the procedure call. Since we are in the merging scenario, a similar procedure call has already been performed and is anchored to its corresponding ancilla. Without loss of generality (since all controlled statements in $l$ are orthogonal and therefore commute) we assume that this call appears at the end of $l$. Rule $(b)$ simply indicates that since $cs$ is orthogonal to other control structures in $l$, we may move the controlled statements so that they are adjacent, and where we apply step $(c)$ to perform merging. Since $cs$ is orthogonal to $\cdot[a^{\texttt{proc'}}_{|\mathrm{s}|} = 1]$ the control is added to the ancilla as expected. Finally, orthogonality of $cs$ with other control structures means that we may move the two controlled-$NOT$s to the edges of the circuit.

All other cases can be similarly checked to be valid □

**Example 7.5. compile**(`QFT`, [4]) *outputs the circuit provided in Figure 4.6 corresponding to the* PFOQ *program* `QFT`. *Notice that there are no extra ancillas as no procedure call appears in the branch of a quantum case.*

We now turn to showing that the compilation strategy ensures a polynomial bound on the size of the obtained circuits.

### 7.3.2 Size Bounds

A bound on the size of the circuits obtained from **compile** can be obtained by reasoning about the number of *different* procedure calls performed (meaning that they cannot be merged) during the course of the compilation.

**Algorithm 1** (**compile**)
**Input:** $(\mathrm{P}, f, cs) \in \mathrm{Programs} \times (Var \to \mathcal{L}(\mathbb{N})) \times (\mathbb{N} \to \{0,1\})$

**Let** $\mathrm{D} :: \mathrm{S} = \mathrm{P}$ **in**

**if** $\mathrm{S} =$ **skip**; **then**
  $C \leftarrow \mathbb{1}$ ▷ Identity circuit

**else if** $\mathrm{S} = \mathrm{q} \mathrel{*}= \mathrm{U}^g(\mathrm{j})$; **and** $(\mathrm{q}, f) \Downarrow_{\mathbb{N}} n$ **and** $(\mathrm{U}^g(\mathrm{j}), f) \Downarrow_{\mathbb{C}^{2\times 2}} M$ **then**
  $C \leftarrow M(cs, n)$ ▷ Controlled gate

**else if** $\mathrm{S} = \mathrm{S}_0\,\mathrm{S}_1$ **then**
  $C \leftarrow$ **compile**$(\mathrm{D} :: \mathrm{S}_0, f, cs) \circ$ **compile**$(\mathrm{D} :: \mathrm{S}_1, f, cs)$ ▷ Composition

**else if** $\mathrm{S} =$ **if** b **then** $\{\mathrm{S}_{\mathbf{true}}\}$ **else** $\{\mathrm{S}_{\mathbf{false}}\}$ **and** $(\mathrm{b}, f) \Downarrow_{\mathbb{B}} b$ **then**
  $C \leftarrow$ **compile**$(\mathrm{D} :: \mathrm{S}_b, f, cs)$ ▷ Conditional

**else if** $\mathrm{S} =$ **qcase** q **of** $\{0 \to \mathrm{S}_0,\ 1 \to \mathrm{S}_1\}$ **and** $(\mathrm{q}, f) \Downarrow_{\mathbb{N}} n$ **then**
  $C \leftarrow$ **compile**$(\mathrm{D} :: \mathrm{S}_0, f, cs[n := 0]) \circ$ **compile**$(\mathrm{D} :: \mathrm{S}_1, f, cs[n := 1])$ ▷ Quantum case

**else if** $\mathrm{S} =$ **call** proc$(\mathrm{s}_1, \ldots, \mathrm{s}_n)$; **and** $\exists i$ such that $(\mathrm{s}_i, f) \Downarrow_{\mathcal{L}(\mathbb{N})} [\,]$ **then**
  $C \leftarrow \mathbb{1}$ ▷ Nil call

**else if** $\mathrm{S} =$ **call** proc$(\mathrm{s}_1, \ldots, \mathrm{s}_n)$; **and** $\forall i, (\mathrm{s}_i, f) \Downarrow_{\mathcal{L}(\mathbb{N})} l'_i \neq [\,]$ **then**
  **if** $\mathrm{width}_{\mathrm{P}}(\mathrm{proc}) = 0$ **then**
    $C \leftarrow$ **compile**$(\mathrm{D} :: \mathrm{S}^{\mathrm{proc}}\{\mathrm{s}_j/\bar{\mathrm{q}}_j\}, f, cs)$ ▷ Non-recursive call

  **else if** $\mathrm{width}_{\mathrm{P}}(\mathrm{proc}) = 1$ **then**
    $C \leftarrow$ **optimize**$(\mathrm{D}, [(cs, \mathrm{S}^{\mathrm{proc}}\{\mathrm{s}_j/\bar{\mathrm{q}}_j\})], \mathrm{proc}, f)$ ▷ Recursive call
  **end if**

**end if**
**return** C

**Algorithm 2 (optimize)** Build circuit for recursive procedure proc
**Inputs:** $(\mathrm{D}, \mathcal{L}_{\mathrm{Cst}}, \texttt{proc}, f) \in \mathrm{Decl} \times \mathcal{L}(\mathrm{Cst}) \times \mathrm{Procedures} \times (Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N}))$

$C_{\mathrm{L}} \leftarrow \mathbb{1}$; $C_{\mathrm{R}} \leftarrow \mathbb{1}$; $\mathrm{P} \leftarrow \mathrm{D} :: \textbf{skip}$; $\mathrm{Anc} \leftarrow \{\}$
**while** $\mathcal{L}_{\mathrm{Cst}} \neq [\,]$ **do**
  $(cs, \mathrm{S}) \leftarrow hd(\mathcal{L}_{\mathrm{Cst}})$; $\mathcal{L}_{\mathrm{Cst}} \leftarrow tl(\mathcal{L}_{\mathrm{Cst}})$
  **if** $\mathrm{S} = \mathrm{S}_1\,\mathrm{S}_2$ **then**
    **if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_1) = 1$ **then** $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs, \mathrm{S}_1)]$; $C_{\mathrm{R}} \leftarrow \textbf{compile}(\mathrm{D} :: \mathrm{S}_2, f, cs) \circ C_{\mathrm{R}}$
    **else** $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs, \mathrm{S}_2)]$; $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ \textbf{compile}(\mathrm{D} :: \mathrm{S}_1, f, cs)$
    **end if**

  **else if** $\mathrm{S} = \textbf{if}\ \mathrm{b}\ \textbf{then}\ \{\mathrm{S}_{\textbf{true}}\}\ \textbf{else}\ \{\mathrm{S}_{\textbf{false}}\}$ **and** $(\mathrm{b}, f) \Downarrow_{\mathbb{B}} b$ **then**
    **if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_b) = 1$ **then** $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs, \mathrm{S}_b)]$
    **else** $C_L \leftarrow C_L \circ \textbf{compile}(\mathrm{D} :: \mathrm{S}_b, f, cs)$
    **end if**

  **else if** $\mathrm{S} = \textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ **and** $(\mathrm{q}, f) \Downarrow_{\mathbb{N}} n$ **then**
    **if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_0) = 1$ **and** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_1) = 1$ **then**
      $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs[n := 0], \mathrm{S}_0), (cs[n := 1], \mathrm{S}_1)]$
    **else if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_1) = 0$ **then**
      $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs[n := 0], \mathrm{S}_0)]$;
      $C_{\mathrm{R}} \leftarrow \textbf{compile}(\mathrm{D} :: \mathrm{S}_1, f, cs[n := 1]) \circ C_{\mathrm{R}}$
    **else if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_0) = 0$ **then**
      $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs[n := 1], \mathrm{S}_1)]$;
      $C_{\mathrm{R}} \leftarrow \textbf{compile}(\mathrm{D} :: \mathrm{S}_0, f, cs[n := 0]) \circ C_{\mathrm{R}}$
    **end if**

  **else if** $\mathrm{S} = \textbf{call}\ \texttt{proc'}[\mathrm{j}](\mathrm{s}_1, \ldots, \mathrm{s}_n)$ **with** $(\mathrm{j}, f) \Downarrow_{\mathbb{N}} m$ **and** $\forall i, (\mathrm{s}_i, f) \Downarrow_{\mathcal{L}(\mathbb{N})} l'_i \neq [\,]$ **then**
    **if** $(\texttt{proc'}, m, [|l'_1|, \ldots, |l'_n|]) \in \mathrm{Anc}$ **then**
      **Let** $(a, [l''_1, \ldots, l''_n]) = \mathrm{Anc}[\texttt{proc'}, m, [|l'_1|, \ldots, |l'_n|]]$ **in**
      $e \leftarrow$ **new** $ancilla()$;
      $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ \mathrm{NOT}(cs, e) \circ \mathrm{NOT}(\{e \mapsto 1\}, a) \circ \mathrm{CSWP}(\{e \mapsto 1\}, [l'_1, .., l'_n], [l''_1, .., l''_n])$;
      $C_{\mathrm{R}} \leftarrow \mathrm{CSWP}(\{e \mapsto 1\}, [l'_1, .., l'_n], [l''_1, .., l''_n]) \circ \mathrm{NOT}(\{e \mapsto 1\}, a) \circ \mathrm{NOT}(cs, e) \circ C_{\mathrm{R}}$
    **else**
      $a \leftarrow$ **new** $ancilla()$;
      $\mathrm{Anc}[\texttt{proc}', m, [|l'_1|, \ldots, |l'_n|]] \leftarrow (a, [l'_1, \ldots, l'_n])$;
      $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ \mathrm{NOT}(cs, a)$; $C_{\mathrm{R}} \leftarrow \mathrm{NOT}(cs, a) \circ C_{\mathrm{R}}$;
      $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(\{a \mapsto 1\}, \mathrm{S}^{\texttt{proc'}}\{m/\mathrm{x}, \mathrm{s}_i/\bar{\mathrm{q}}_i\})]$
    **end if**

  **end if**
**end while**
**return** $C_L \circ C_R$

**Theorem 7.6.** *For any* P *in* LFOQ *where* $Var(\mathrm{P}) = \{\bar{q}_1, \ldots, \bar{q}_k\}$*, we have that,*

$$\#\textbf{compile}(\mathrm{P}, [n_1, \ldots, n_k]) = O\left(n^{(k+1)\cdot rk(\mathrm{P})+1}\right),$$

*where* $n \triangleq n_1 + \cdots + n_k$*.*

*Proof.* We first show that an execution of the **optimize** subroutine performs $O(n^{k+1})$ calls to **compile**. We use a simple counting argument. The dictionary Anc ensures that ancillas related to the same $(\texttt{proc}, \mathrm{i}, [\mathrm{j}_1, \ldots, \mathrm{j}_k]) \in \text{Procedures} \times \mathbb{Z} \times \{1, \ldots, n\}^k$ will only be created once.

The classical parameter can either be updated to a new constant or by adding (or subtracting) a constant and this can be done $O(n)$ times as procedure calls also remove at least an element from the qubit list parameter. So the range of values for i is $O(n)$, and there are $O(n^k)$ possible combinations of input sizes. Therefore, the space of possible values has size $O(n^{k+1})$. This is essentially the statement of Lemma 7.2. Therefore, at most $O(n^{k+1})$ ancillas can be created, which also bounds the number of calls to **compile**.

Second, consider an execution of **compile**. Subroutine **compile** calls itself directly only outside of recursive procedure calls, meaning that these cases consist of constant time circuit constructions. On the case of a call to a recursive procedure, it does a single call to **optimize** which creates $O(n^{k+1})$ calls back to **compile**.

Each of these instances of **compile** either does a non-recursive circuit construction of constant size, or does a call to **optimize** for a procedure of strictly lower rank, as defined in the proof of Lemma 5.21. Since there are $O(n^{k+1})$ such calls, the maximum total number of procedure calls is $O(n^{(k+1)\,rk(\mathrm{P})})$, where $rk(\mathrm{P})$ is the rank of P (Definition 5.20).

Each of these recursive procedure calls can have a constant number of merge constructions with linear complexity, therefore the total number of gates is $O(n^{(k+1)rk(\mathrm{P})+1})$. □

While Theorem 7.6 gives an upper bound on circuit complexity, it is not tight. This can be seen for the example of program `PAIRS` defined in Figure 7.1.

**Example 7.7.** *The program* `PAIRS` *in Figure 7.1 has a single qubit variable, and* $rk(\texttt{PAIRS}) = 1$*, therefore Theorem 7.6 gives a quadratic bound on the circuit obtained from* **compile**. *However, we have that* $\#\textbf{compile}(\texttt{PAIRS}, [n]) = \Theta(n)$*, with the circuit given in Figure 7.7, where ancilla* $a_i^{\texttt{pairs}}$ *anchors procedure* `pairs` *on input size* $i$*.*

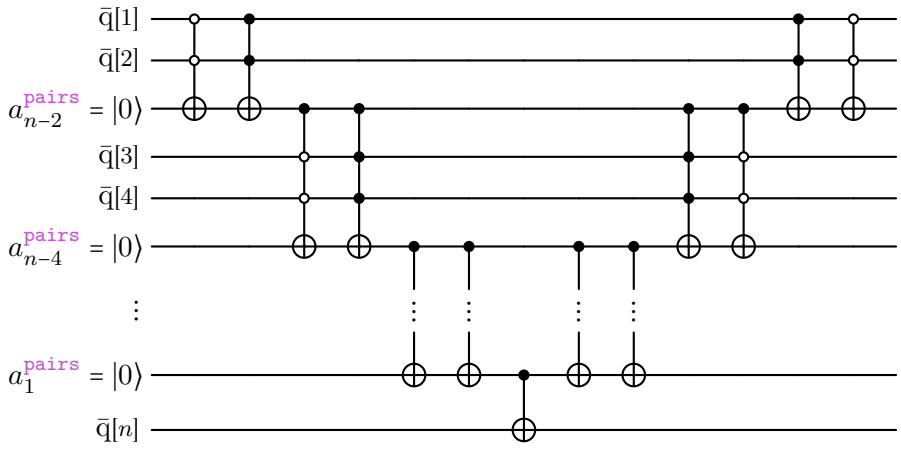



Figure 7.7: Circuit for program `PAIRS` on odd input size $n$.

Note that, since $\mathrm{Time}_{\mathrm{P}}(n) \triangleq \lfloor \frac{n}{2} \rfloor + 1$ this is the correct asymptotic bound for the circuit.

We now make an argument for the size and depth of circuits resulting from LFOQ programs. Many of the arguments are similar to the PFOQ case, and so we only highlight the small differences.

**Theorem 7.8.** *For any* P *in* PFOQ *where* $Var(\mathrm{P}) = \{\bar{\mathrm{q}}_1, \ldots, \bar{\mathrm{q}}_k\}$ *such that* $n = \sum_{\bar{\mathrm{q}} \in Var(\mathrm{P})} |\bar{\mathrm{q}}|$*, the quantum circuit produced by* **compile** *is of size* $O(n\,\mathrm{polylog}(n))$ *and depth* $O(\mathrm{polylog}(n))$*.*

*Proof.* Similar to the proof of Theorem 7.6, but in the case of LFOQ the possible number of different procedures is bounded *polylogarithmically*, therefore only a number $O(\mathrm{polylog}(n))$ of procedure calls are performed. This can be seen by the fact that in each recursive procedure call at least one of the input sizes must decrease by half. Therefore, there is only a poylogarithmic number of possible integer inputs, as well as a polylogarithmic number of input sizes.

Therefore **optimize** makes at most a polylogarithmic number of calls to **compile**, i.e., to procedures of lower rank.

Since merging can be done in logarithmic depth, the polylogarithmic bound remains true for the entire circuit depth. Since merging requires a linear number of ancillas and gates, a linear factor is added to the circuit size. □

We have described the **compile** and **optimize** algorithms both as programs (Algorithms 1 and 2) and with rewrite systems (Figures 7.5 and 7.6). We saw that by exploiting the orthogonality between different branches of a **qcase** statement, we were able to construct circuits whose size depends on the number of *unique* procedure calls created, rather than their total number.

We now consider two examples of the compilation algorithm, one with a linear-time program and another with a logarithmic-time program.

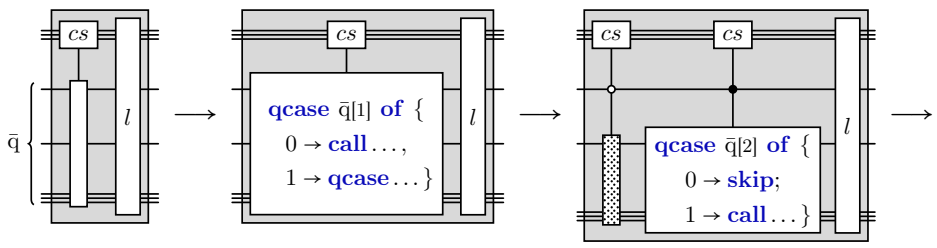
## 7.4 Two compilation examples

We will exemplify the compilation procedure with programs `REC` of Figure 7.8 and `SEARCH` of Figure 6.1. The first example will allow us to follow the application of anchoring and merging when the input sizes do not match immediately (as was the case for `PAIRS`), while the second one will demonstrate the use of the logarithmic-depth swapping of qubit addresses.

### 7.4.1 A linear-time example

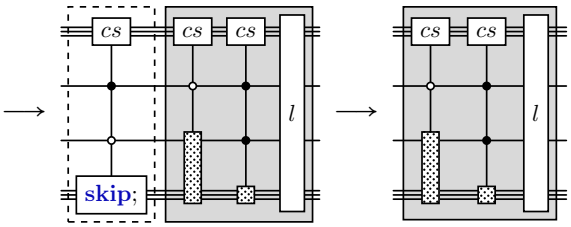
Consider the PFOQ program `REC` defined in Figure 7.8. Given that `rec` is a recursive procedure, its compilation is performed within **optimize**. We will perform compilation steps using the rewrite rules of Figure 7.6.

We denote by the empty box □ the procedure statement $\mathrm{S}^{\texttt{rec}}$ and by the dotted box ▨ the statement **call** `rec`(s);. Applying rewrite rules to the statement of `rec` we obtain the transitions:



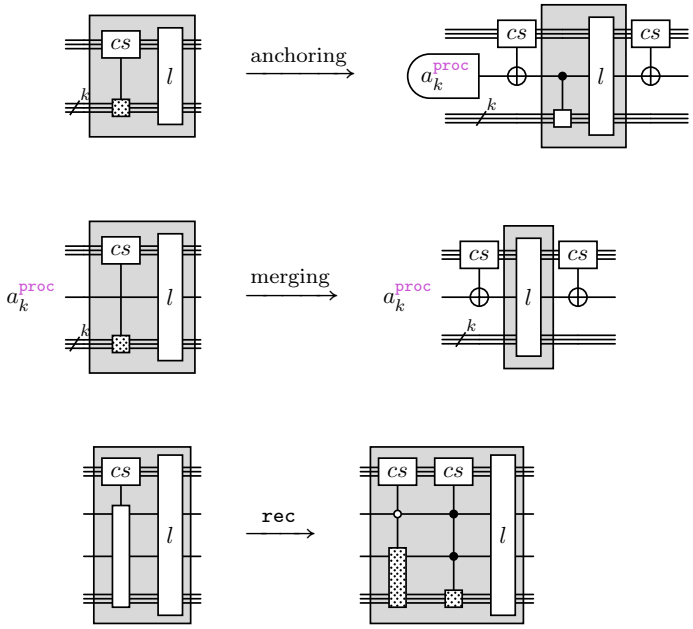
```
1  decl rec(q̄){
2    if |q̄| > 2 then
3      qcase q̄[1] of{
4        0 → call rec(q̄ ⊖ [1]);
5        1 → qcase q̄[2] of{
6              0 → skip;
7              1 → call rec(q̄ ⊖ [1,2]);
8              }
9        }
10   else q̄[1] *= U; }
11 :: call rec(q̄);
```

Figure 7.8: Program REC.




The first step is obtained by applying the rule of **if** statements, where we assume that $|\bar{q}| > 2$. Afterwards, we apply twice the rule for the **qcase** statement, adding the two qubits $\bar{q}[1]$ and $\bar{q}[2]$ to the control structure. Finally, the compilation of the **skip** statement corresponds to the empty circuit. We denote by $\xrightarrow{\texttt{rec}}$ the application of this sequence of rewrite rules.

The compilation can then be defined by the use of three different steps, two related to anchoring and merging, and a third one implementing the rec procedure statement:



The merging step does not require controlled-swaps since, given the structure of `rec`, when two procedure calls have the same input size, they refer to precisely the same qubits. More precisely, all instances of procedure `rec` on input size $n$ will always apply to the last $n$ qubits.

We show the first few compilations steps for `rec` on Figure 7.9, where we assume that the input size $|\bar{q}|$ is large enough that we always find ourselves in the recursive case.

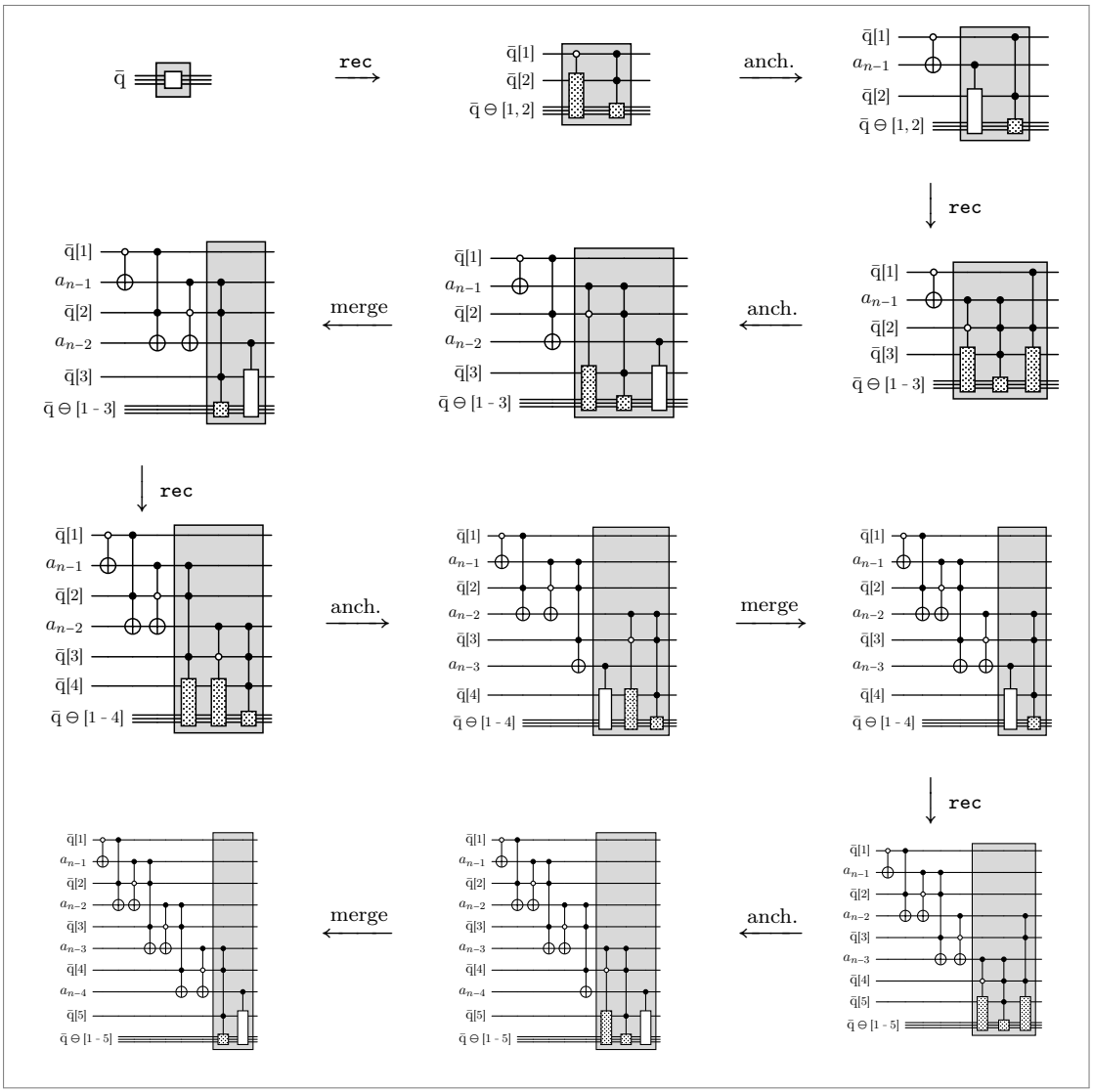



Figure 7.9: Example of anchoring and merging for the program `REC` in Figure 7.8.

One important thing to notice in Figure 7.9 is that the compilation strategy does not avoid branch sequentialization *locally* but rather *asymptotically* in the construction of the circuit. In other words, the **qcase** statement in rule `rec` does generate two instances of $S^{\texttt{rec}}$ in the circuit in sequence, one for each branch. However, the merging of calls to `rec` on inputs of the same size ensures that only one instance per input size needs to be compiled, and therefore this strategy achieves linear complexity in the number of gates.

We now consider the case of a compiling an LFOQ program.

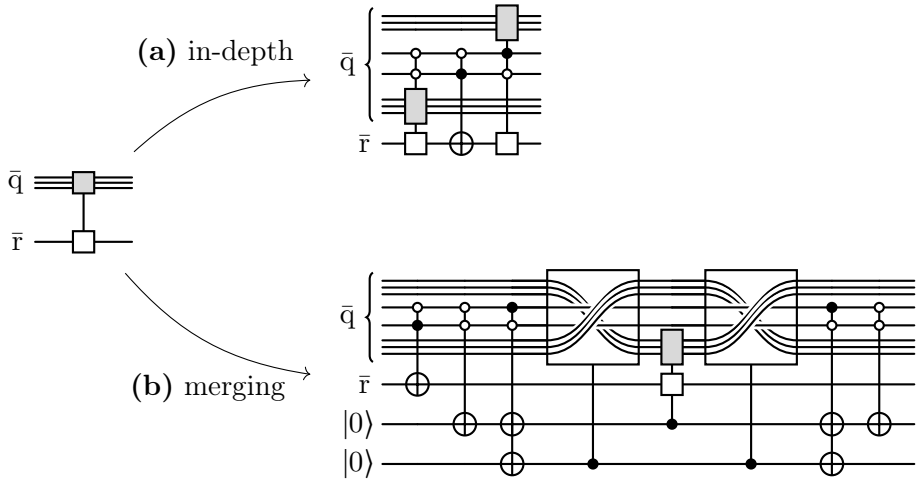



Figure 7.10: Compilation strategies for `search` defined in Figure 6.1.

### 7.4.2 A polylogarithmic-time example

We revisit the program `SEARCH` in Figure 6.1, now with the objective of obtaining a logarithmic-depth circuit. We can again consider multiple strategies, and notice that the *in-depth* strategy generates a duplication of instances of `search`, which is represented in Figure 7.10 by two connected white and gray gates. Such a strategy results in a circuit of linear size and depth.

The *merging* strategy of **optimize** requires that we swap the addresses of the qubits between the first and second halves of the set $\bar{q}$. This requires changing the addresses of $\frac{|\bar{q}|-2}{2}$ qubits and therefore using a linear number of gates. According to Lemma 7.1, this can be performed in linear size and log depth, and so the merging strategy outputs a circuit of polylogarithmic depth (in this case, $O(\log^2 n)$ size).

In Figure 7.11 we show the circuit obtained for an input of size $|\bar{q}| = 14$ and $|\bar{r}| = 1$. For this input size, two ancillas are created for controlling the procedure call to `search`, namely on inputs size $|\bar{q}| = 6$ and $|\bar{q}| = 2$ (the size of $\bar{r}$ is unchanged during the procedure calls).

With these two examples, we demonstrate the use of the compilation strategy introduced in this section, as it allows to obtain circuits that satisfy the complexity bounds of the respective classes: polynomial size of circuits in the case of PFOQ, and circuits of polylogarithmic depth in the case of LFOQ. This contrasts with the approaches of naïve compilation (i.e., *in-depth*) and the QRAM approach described by the *in-width* strategy (Figure 7.2).

In the next chapter, we present a modification of the compilation strategy that strictly improves the asymptotic bounds. We start by noticing that, by focusing on recursive calls, the strategy of **compile** performs less merging of procedures than what is possible, and that this can lead to a polynomial slowdown in the circuit complexity.

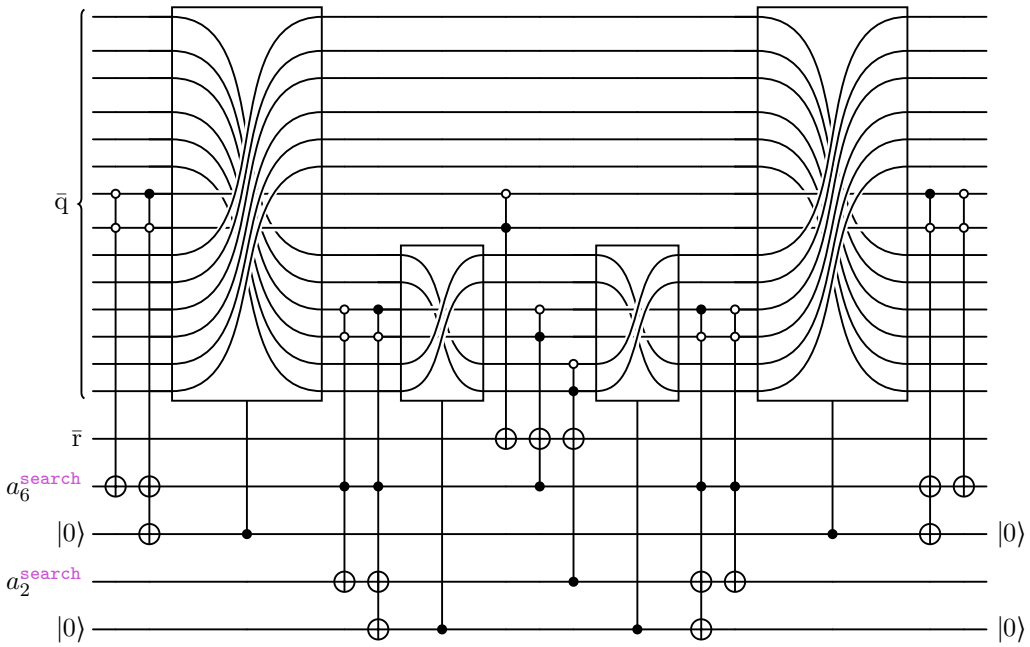



Figure 7.11: Quantum circuit resulting of compiling the `SEARCH` program on input size $|\bar{q}| = 14$.

# 8

# An Asymptotically-Optimal Compilation Strategy

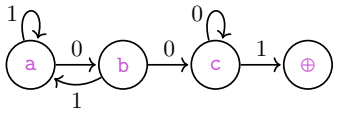
In the previous chapter, we showed that PFOQ restrictions allow us to compile polynomial-time programs without incurring the exponential blow-up from branch sequentialization. However,the application of anchoring and merging, as used in **optimize**, can result in a circuit whose asymptotic size differs from the program's time complexity by a polynomial factor.

In this section, we identify the source of the polynomial slowdown, and propose a refinement of **optimize**. Furthermore, we identify a fragment of PFOQ programs, denoted PFOQ$^{\text{UNIF}}$, that is sound and complete for quantum polynomial time and for which branch sequentialization can be avoided altogether. Therefore, PFOQ$^{\text{UNIF}}$ constitutes the first programming language that is complete for quantum polynomial time and for which there exists a compilation strategy such that the complexity of the **qcase** statement is the maximum of the branches, and not the sum.

## 8.1 Motivation: a polynomial slowdown

To exemplify how **optimize** can incur a polynomial slowdown due to branch sequentialization, we consider the example for identifying a substring on a given input.

**Example 8.1** (Detecting substring)**.** *Consider a program for detecting a substring* 001 *occurring in an input. We define a program with procedures* a*,* b*,* c *and* ⊕*, with the graph:*




*The two outgoing edges of a node indicate the two branches of a* **qcase** *statement in the corresponding procedure, controlled on the first input qubit. Procedures consist only of a* **qcase** *statement, for the exception of* ⊕*:*

$$
\begin{aligned}
\mathrm{D} \triangleq\ & \textbf{decl}\ \mathtt{a}(\bar{\mathrm{q}})\{\,\textbf{qcase}\ \bar{\mathrm{q}}[1]\ \textbf{of}\ \{0 \to \textbf{call}\ \mathtt{b}(\bar{\mathrm{q}} \ominus [1]);\,,\ 1 \to \textbf{call}\ \mathtt{a}(\bar{\mathrm{q}} \ominus [1]);\,\}\,\},\\
& \textbf{decl}\ \mathtt{b}(\bar{\mathrm{q}})\{\,\textbf{qcase}\ \bar{\mathrm{q}}[1]\ \textbf{of}\ \{0 \to \textbf{call}\ \mathtt{c}(\bar{\mathrm{q}} \ominus [1]);\,,\ 1 \to \textbf{call}\ \mathtt{b}(\bar{\mathrm{q}} \ominus [1]);\,\}\,\},\\
& \textbf{decl}\ \mathtt{c}(\bar{\mathrm{q}})\{\,\textbf{qcase}\ \bar{\mathrm{q}}[1]\ \textbf{of}\ \{0 \to \textbf{call}\ \mathtt{c}(\bar{\mathrm{q}} \ominus [1]);\,,\ 1 \to \textbf{call}\ \oplus(\bar{\mathrm{q}} \ominus [1]);\,\}\,\},\\
& \textbf{decl}\ \oplus(\bar{\mathrm{q}})\{\,\bar{\mathrm{q}}[\text{-}1]\ \ast\!\!=\ \mathrm{NOT};\,\}
\end{aligned}
$$

*The program then defined as* $\texttt{SubS\_001}(\bar{q}) \triangleq D :: \textbf{call}\ a(\bar{q})$; *and performs the transformation, for* $x \in \{0,1\}^*$, $y \in \{0,1\}$*:*

$$[\![\texttt{SubS\_001}]\!](|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus b\rangle$$

*where* $b$ *is* 1 *if the string* 001 *appears in* $x$ *and* 0 *otherwise.*

Analysing program `SubS_001`, we check that it is in PFOQ and that it satisfies the relations

$$a \sim b \succ c \succ \oplus.$$

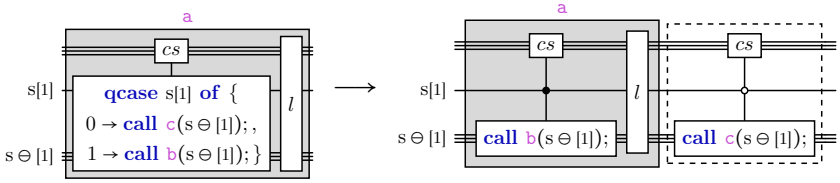
Furthermore, we have that $\text{Time}_{\texttt{SubS\_001}}(n) = n + 1$, but if we compile the program using the strategy described in Chapter 7, we obtain a circuit that grows as the *square* of its input size.

**Example 8.2.** $\#\textbf{compile}(\texttt{SubS\_001}, [n]) = \Theta(n^2)$.

How can we explain this difference in complexity? The reason is that, for instance, in the first call to **optimize** in the context of procedure `a`, since `a` and `b` are mutually recursive, we encounter the procedure statement of `b`:

$$\textbf{qcase}\ \text{s}[1]\ \textbf{of}\ \{0 \to \textbf{call}\ c(\text{s} \ominus [1]);, 1 \to \textbf{call}\ b(\text{s} \ominus [1]);\}$$

as one of the controlled statements in the list. Since `a` ∼ `b` and `a` ≻ `c`, the rule for the **qcase** states the following transformation:




which means that each instance of procedure `c` derived from `b` is compiled separately. As a consequence, the complexity of the **qcase** statement is the sum of the branches whenever they contain calls to procedure at a lower level of recursion. This is because the information about the orthogonality between the call to `c` and the call to `b` is lost when the rule is applied.

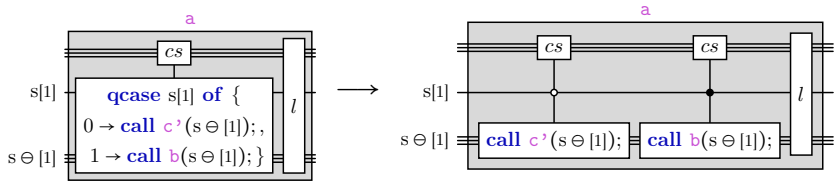
To stress this point, consider a modified program where we add a call to `a` that never occurs in procedures `c` and `d`:

$$\textbf{decl}\ c'(\bar{q})\{\textbf{if}\ 0 = 1\ \textbf{then call}\ a(\bar{q});\ \textbf{else}\ S^c\},$$
$$\textbf{decl}\ d'(\bar{q})\{\textbf{if}\ 0 = 1\ \textbf{then call}\ a(\bar{q});\ \textbf{else}\ S^d\}$$

where $S^c$ and $S^d$ are the original procedure statements of procedures `c` and `d` respectively. In this new, functionally equivalent program, we have that

$$a \sim b \sim c' \succ \oplus$$

and procedure calls from `b` to `c'` no longer ignore the orthogonality between the branches of the **qcase**, since they are both kept in the **optimize** list:



and the circuit we obtained is of linear complexity, coinciding with $\mathrm{Time}_{\texttt{SubS\_001}}(n)$.

While this transformation gives us a clue as to how to improve on the algorithm given in Chapter 7, it does not work for all the cases that interest us as a PFOQ program may no longer be in PFOQ after the transformation, in the case where two calls to procedures of lower rank are performed in sequence.

In this chapter, we will not only look to improve the **optimize** algorithm given in Chapter 7, but we will also identify a fragment of PFOQ where the new compilation strategy that completely avoids branch sequentialization.

## 8.2 A uniformity condition (UNIF)

We introduce a set of syntactical restrictions which ensure that the size of the circuit is bounded by the semantical complexity of the program, that is, by its function $\mathrm{Time}_{\mathrm{P}}(n)$.

**Definition 8.3.** *Let* UNIF *denote the set of* FOQ *programs* P *with* $Var(\mathrm{P}) = \{\bar{\mathrm{q}}\}$ *satisfying the conditions:*

*i) procedures do not make use of classical inputs,*

*ii) there is a qubit list* $\mathrm{s}_0$ *such that, for any procedure call* $\texttt{proc}(\mathrm{s})$; *occurring in* P,

$$\textit{either} \quad \mathrm{s} = \mathrm{s}_0 \quad \textit{or} \quad \mathrm{s} = \bar{\mathrm{q}}.$$

**Definition 8.4.** $\text{PFOQ}^{\text{UNIF}} \triangleq \text{UNIF} \cap \text{PFOQ}$.

It trivially holds that $\text{PFOQ}^{\text{UNIF}} \subsetneq \text{PFOQ} \subsetneq \text{FOQ}$. We will show that $\text{PFOQ}^{\text{UNIF}}$ is extensionally equivalent to PFOQ, i.e. that $[\![\text{PFOQ}^{\text{UNIF}}]\!] = [\![\text{PFOQ}]\!]$. As an example, we show how to rewrite the QFT program with the UNIF restrictions.

**Example 8.5** ($\texttt{QFT} \notin \text{PFOQ}^{\text{UNIF}}$)**.** *The* QFT *program defined in Figure 4.6 is not in* $\text{PFOQ}^{\text{UNIF}}$ *since, for instance, it makes use of integer inputs.*

We define a new program $\texttt{QFTu} \in \text{PFOQ}^{\text{UNIF}}$ (Figure 8.1) such that $[\![\texttt{QFT}]\!] = [\![\texttt{QFTu}]\!]$. We make use of the fact that we can reorder the controlled rotation gates without changing their effect, and perform the qubit swapping by performing a linear number of *shift* operations.

We can check that QFTu requires asymptotically the same number of procedure calls in depth as QFT, i.e. this transformation has not altered the asymptotic complexity of the program.

**Lemma 8.6.** $\mathrm{Time}_{\texttt{QFTu}}(n) = \Theta(n^2)$.

The uniform fragment of PFOQ is also sound and complete for quantum polynomial time.

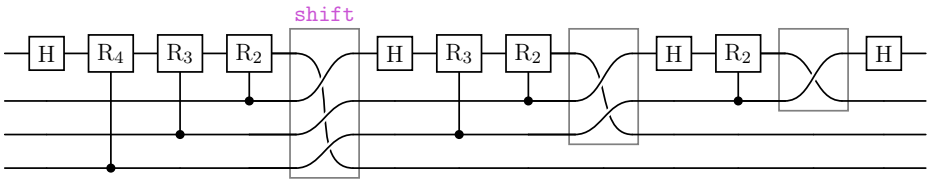
**Theorem 8.7.** $[\![\text{PFOQ}^{\text{UNIF}}]\!] = \text{QP}$.

*Proof.* Soundness is trivial since $\text{PFOQ}^{\text{UNIF}}$ is contained in PFOQ. Completeness can be seen by the fact that the PFOQ programs created in the proof of Theorem 5.24 can be easily adapted to fit UNIF restrictions. In this case, since the first qubits are always the ones that are removed in recursive calls, one can define a pattern of qubit removal always using the input $\bar{\mathrm{q}} \ominus [1]$. □

We now introduce a new compilation strategy that improves the complexity bounds for PFOQ programs and, in particular, results in an optimal bound for the $\text{PFOQ}^{\text{UNIF}}$ fragment.

QFTu

```
1  decl rec(q̄){
2    q̄[1] *= H;
3    call rot(q̄);
4    call shift(q̄);
5    call rec(q̄ ⊖ [-1]); },
6  decl rot(q̄){
7    if |q̄| > 1 then
8      CPHASE(q̄[-1], q̄[1], |q̄|)
9      call rot(q̄ ⊖ [-1]);
10   else skip; },
11 decl shift(q̄){
12   if |q̄| > 1 then
13     SWAP(q̄[1], q̄[-1])
14     call shift(q̄ ⊖ [-1]);
15   else skip; }
16 ::
17 call rec(q̄);
```




Each shift gate consists of a linear number of $SWAP$ operations:

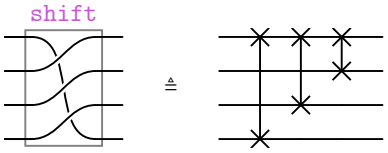



Figure 8.1: PFOQ$^{\text{UNIF}}$ program QFTu for the quantum Fourier transform.

## 8.3 An improved compilation algorithm

We modify the **optimize** algorithm (Algorithm 2) to obtain **optimize**$^+$, defined as Algorithm 3, and with rewrite rules given in Figure 8.2. We define **compile**$^+$ to be the same as **compile** but with **optimize**$^+$ replacing **optimize**.

In order to make use of the orthogonality information for procedure calls between procedures of different rank (i.e. when proc ≻ proc'), **optimize**$^+$ manages a contextual list $l_{\mathrm{M}}$ of controlled statements that are orthogonal to those in $l$ but that do not contain recursive procedure calls. In the rewriting rules of Figure 8.2, the contextual list $l_{\mathrm{M}}$ is denoted by a striped box ▨. A gate is placed inside the dashed box to indicate that a controlled statement was added to the list $l_{\mathrm{M}}$.

At the end of **optimize**$^+$, we rearrange the contextual list in the following way. Let $l_{\mathrm{M}} = [(cs_1, \mathrm{S}_1), \ldots, (cs_k, \mathrm{S}_k)]$ be the state of the contextual list at the end of **optimize**$^+$. We may rewrite each controlled statement as a list of its atomic elements. This transformation, denoted seq, can be described inductively, as follows:

$$\begin{aligned} \mathrm{seq}(cs, \textbf{skip};, f) &\triangleq [], \\ \mathrm{seq}(cs, \mathrm{q} \mathrel{*}= \mathrm{U}^g(\mathrm{j});, f) &\triangleq [(cs, \mathrm{q} \mathrel{*}= \mathrm{U}^g(\mathrm{j});, f)], \\ \mathrm{seq}(cs, \mathrm{S}_1\ \mathrm{S}_2, f) &\triangleq \mathrm{seq}(cs, \mathrm{S}_1, f) @ \mathrm{seq}(cs, \mathrm{S}_1, f), \end{aligned}$$

For the case of the classical **if** statement, when $(\mathrm{b}, f) \Downarrow_{\mathbb{B}} b$, we have,

$$\mathrm{seq}(cs, \textbf{if b then } \mathrm{S}_{\textbf{true}} \textbf{ else } \mathrm{S}_{\textbf{false}}, f) \triangleq \mathrm{seq}(cs, \mathrm{S}_b, f)$$

The **qcase** statement is treated as follows. When $(\mathrm{q}, f) \Downarrow_{\mathbb{N}} n$,

$$\mathrm{seq}(cs, \textbf{qcase q of } \{0 \to \mathrm{S}_0,\ 1 \to \mathrm{S}_1\}, f) \triangleq \mathrm{seq}(cs[n := 0], \mathrm{S}_0, f) @ \mathrm{seq}(cs[n := 1], \mathrm{S}_1, f)$$

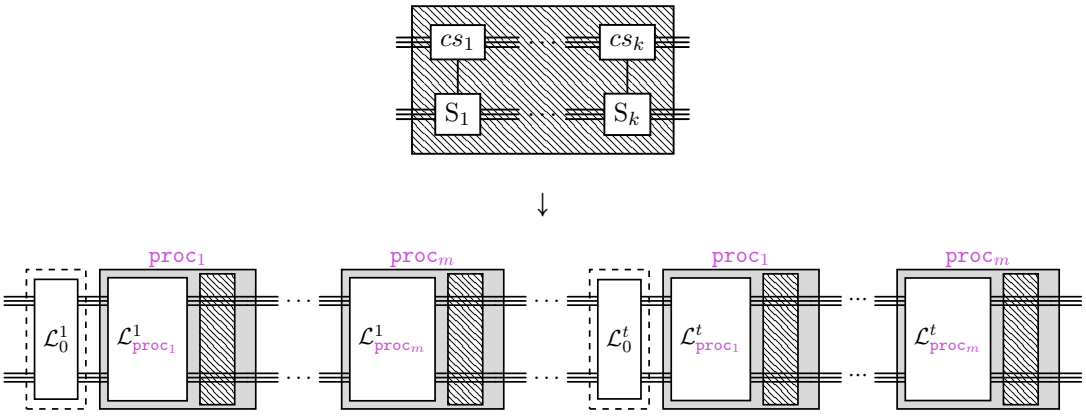
Finally, in the case of the procedure call, we have

$$\mathrm{seq}(cs, \textbf{call } \mathtt{proc}(\mathrm{s});, f) \triangleq [(cs, \textbf{call } \mathtt{proc}(\mathrm{s});, f)].$$

This separation of statements allows for a partitioning according to the type of procedure call appearing in the statement. Given a list of controlled statements $\mathcal{L}$, we denote by procedure_split($\mathcal{L}$) the list $[\mathcal{L}_0, \mathcal{L}_1, \ldots, \mathcal{L}_m]$ where, for $\mathtt{proc}_1$, ..., $\mathtt{proc}_m$ are procedures that are not mutually recursive.

$$\mathcal{L}_0 \triangleq \{(cs, \mathrm{S}, f) \in \mathcal{L} : \nexists\ \mathtt{proc} \text{ such that } w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}^{\mathtt{proc}}) = 1 \text{ and } w_{\mathrm{P}}^{\mathtt{proc}}(\mathrm{S}) = 1\}.$$
$$\mathcal{L}_{\mathtt{proc}_i} \triangleq \{(cs, \mathrm{S}, f) \in \mathcal{L} : w_{\mathrm{P}}^{\mathtt{proc}_i}(\mathrm{S}) = 1\}, \quad i = 1, \ldots, m.$$

Given our choice of procedures and the controled statements obtained from seq, this partition is well-defined. Performing these two partitions (first in terms of sequential order of statements, and then according to procedure calls), we are able to rewrite the list $l_{\mathrm{M}}$ in the following way:




The different instances of **optimize**$^+$ contain calls to procedures that are mutually recursive, which will allow for further anchoring and merging.

The soundness of **optimize**$^+$ is based on an extension of the orthogonality invariant of Lemma 7.3. The validity of the circuit construction is based on the fact that controlled statements in $l \cup l_{\mathrm{M}}$ are pairwise orthogonal.

**Lemma 8.8** (Orthogonality invariant of **optimize**$^+$)**.** *During the compilation of a* PFOQ$^{\mathrm{UNIF}}$ *program* P*, for each optimization of a (recursive) procedure* $\mathtt{proc}$*, all controlled statements in the union of list* $l$ *and the contextual list* $l_M$ *are pairwise orthogonal.*

**Theorem 8.9** (Soundness of **compile**$^+$)**.** *Given a* PFOQ$^{\mathrm{UNIF}}$ *program* P *and a quantum state* $|\psi\rangle \in \mathcal{H}_{2^n}$ *we have that* $[\![\textbf{compile}^+(\mathrm{P}, n)]\!](|\psi\rangle) = [\![\mathrm{P}]\!](|\psi\rangle)$.

**Algorithm 3 (optimize$^+$)** Build circuit for recursive procedure proc
**Inputs:** $(\mathrm{D}, \mathcal{L}_{\mathrm{Cst}}, \texttt{proc}, f) \in \mathrm{Decl} \times \mathcal{L}(\mathrm{Cst}) \times \mathrm{Procedures} \times (Var(\mathrm{P}) \to \mathcal{L}(\mathbb{N}))$

$C_{\mathrm{L}} \leftarrow \mathbb{1}$; $C_{\mathrm{M}} \leftarrow \mathbb{1}$; $C_{\mathrm{R}} \leftarrow \mathbb{1}$; $l_{\mathrm{M}} \leftarrow [\,]$; $\mathrm{Anc} \leftarrow \{\}$; $\mathrm{P} \leftarrow \mathrm{D} :: \textbf{skip}$;
**while** $\mathcal{L}_{\mathrm{Cst}} \neq [\,]$ **do**
  $(cs, \mathrm{S}) \leftarrow hd(\mathcal{L}_{\mathrm{Cst}})$; $\mathcal{L}_{\mathrm{Cst}} \leftarrow tl(\mathcal{L}_{\mathrm{Cst}})$

  **if** $\mathrm{S} = \mathrm{S}_1\,\mathrm{S}_2$ **then**
    **if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_1) = 1$ **then** $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs, \mathrm{S}_1)]$; $C_{\mathrm{R}} \leftarrow \textbf{compile}^+(\mathrm{D} :: \mathrm{S}_2, f, cs) \circ C_{\mathrm{R}}$
    **else** $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs, \mathrm{S}_2)]$; $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ \textbf{compile}^+(\mathrm{D} :: \mathrm{S}_1, f, cs)$
    **end if**

  **else if** $\mathrm{S} = \textbf{if}\ \mathrm{b}\ \textbf{then}\ \{\mathrm{S}_{\textbf{true}}\}\ \textbf{else}\ \{\mathrm{S}_{\textbf{false}}\}$ **and** $(\mathrm{b}, f) \Downarrow_{\mathbb{B}} b$ **then**
    **if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_b) = 1$ **then** $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs, \mathrm{S}_b)]$
    **else** $l_{\mathrm{M}} \leftarrow l_{\mathrm{M}}@[(cs, \mathrm{S}_b)]$
    **end if**

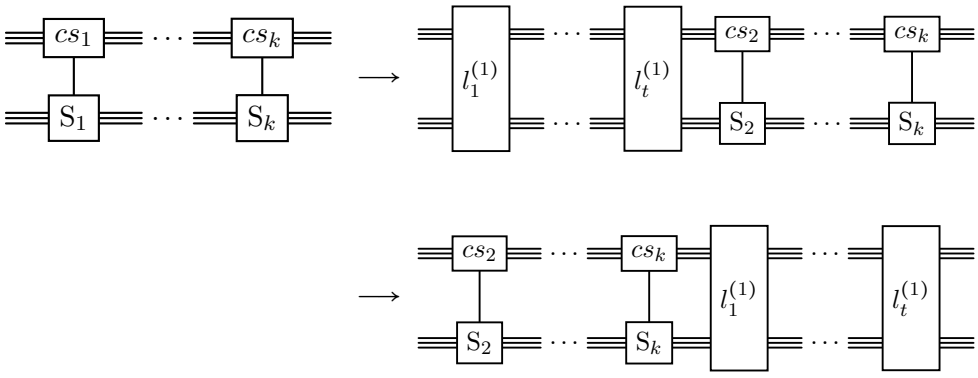
  **else if** $\mathrm{S} = \textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ **and** $(\mathrm{q}, f) \Downarrow_{\mathbb{N}} n$ **then**
    **if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_0) = 1$ **and** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_1) = 1$ **then**
      $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs[n := 0], \mathrm{S}_0), (cs[n := 1], \mathrm{S}_1)]$
    **else if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_1) = 0$ **then**
      $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs[n := 0], \mathrm{S}_0)]$; $l_{\mathrm{M}} \leftarrow l_{\mathrm{M}}@[(cs[n := 1, \mathrm{S}_1)]$
    **else if** $w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}_0) = 0$ **then**
      $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(cs[n := 1], \mathrm{S}_1)]$; $l_{\mathrm{M}} \leftarrow l_{\mathrm{M}}@[(cs[n := 0, \mathrm{S}_0)]$
    **end if**

  **else if** $\mathrm{S} = \textbf{call}\ \texttt{proc'}[\mathrm{j}](\mathrm{s}_1, \ldots, \mathrm{s}_n)$; **with** $(\mathrm{j}, f) \Downarrow_{\mathbb{N}} m$ **and** $\forall i, (\mathrm{s}_i, f) \Downarrow_{\mathcal{L}(\mathbb{N})} l'_i \neq [\,]$ **then**
    **if** $(\texttt{proc'}, m, [|l'_1|, \ldots, |l'_n|]) \in \mathrm{Anc}$ **then**
      **Let** $(a, [l''_1, \ldots, l''_n]) = \mathrm{Anc}[\texttt{proc'}, m, [|l'_1|, \ldots, |l'_n|]]$ **in**
      $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ \mathrm{NOT}(cs, a)$
      $C_{\mathrm{R}} \leftarrow \mathrm{NOT}(cs, a) \circ C_{\mathrm{R}}$
    **else**
      $a \leftarrow$ **new** $ancilla()$;
      $\mathrm{Anc}[\texttt{proc}', m, [|l'_1|, \ldots, |l'_n|]] \leftarrow (a, [l'_1, \ldots, l'_n])$;
      $C_{\mathrm{L}} \leftarrow C_{\mathrm{L}} \circ \mathrm{NOT}(cs, a)$; $C_{\mathrm{R}} \leftarrow \mathrm{NOT}(cs, a) \circ C_{\mathrm{R}}$;
      $\mathcal{L}_{\mathrm{Cst}} \leftarrow \mathcal{L}_{\mathrm{Cst}}@[(\{a \mapsto 1\}, \mathrm{S}^{\texttt{proc'}}\{m/\mathrm{x}, \mathrm{s}_i/\bar{\mathrm{q}}_i\})]$
    **end if**

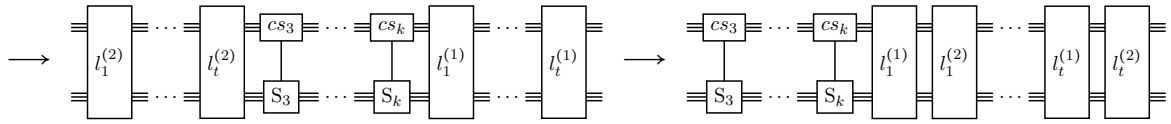
  **end if**
**end while**
$T = \max_{(cs,\mathrm{S}) \in l_{\mathrm{M}}}(|\mathrm{sec}(cs, \mathrm{S})|)$
**for** $1 \leq t \leq T$ **do**
  $\mathcal{L} \leftarrow \bigcup_{(cs,\mathrm{S},l) \in l_{\mathrm{M}}} \mathrm{sec}(cs, \mathrm{S}, l)[t]$
  $\mathcal{L}_0, \ldots \mathcal{L}_m = \mathrm{procedure\_split}(\mathcal{L})$ /* $m$ = number of recursive procedure families */
  $C_{\mathrm{M}} \leftarrow C_{\mathrm{M}} \circ \left( \circ_{(cs,\mathrm{S},l) \in \mathcal{L}_0} \textbf{compile}^+(\mathrm{D} :: \mathrm{S}, l, cs) \right) \circ \left( \circ_{i=1}^{m} \textbf{optimize}^+(\mathrm{D}, \mathcal{L}_i, \texttt{proc}_i) \right)$
**end for**
**return** $C_{\mathrm{L}} \circ C_{\mathrm{M}} \circ C_{\mathrm{R}}$

*Proof.* The sole difference of the algorithm is in the case of **optimize**$^+$. The validity of the rules in this case is done by the inspection of the rewrite rules given in Figure 8.2 with the use of the invariant of Lemma 8.8. This is similar to the reasoning presented in the proof of Theorem 7.4.

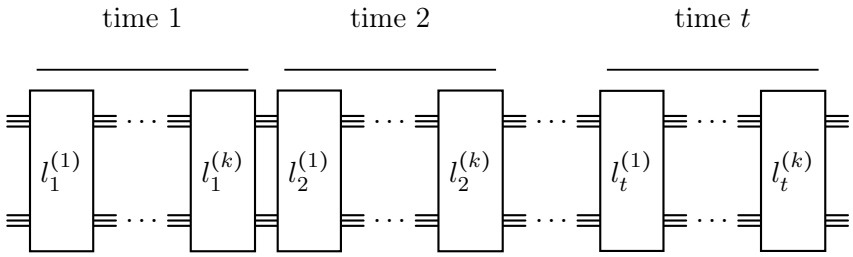
Finally, we consider the validity of the rule for the contextual circuit. Consider a list of mutually-orthogonal controlled statements, where the sequential form of $(cs_i, \mathrm{S}_i)$ is given by the lists of controlled statements $l_1^{(i)}, \ldots, l_t^{(i)}$, then we have that:




The final step comes from the fact that all $l_j^{(1)}$ are orthogonal to $cs_2$, …, $cs_k$ since they include $cs_1$. Using the sequential form of $(cs_2, \mathrm{S}_2)$, we perform the following transitions:




where we make use of the orthogonality between $l_j^{(1)}$ and $l_{j'}^{(2)}$. Performing the same transitions for $(cs_3, \mathrm{S}_3)$, …, $(cs_k, \mathrm{S}_k)$, we obtain the following arrangement:

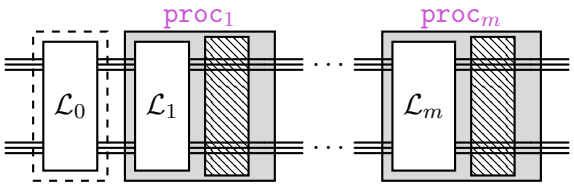



Given a certain $1 \leq j \leq t$, we have that all controlled statements in $\cup_{i=1}^{k} l_j^{(i)}$ (that is, all controlled statements occurring in time $j$) are pairwise orthogonal. Therefore, we may rearrange their order according to their recursivity, and in doing so we may consider each time separately. For instance, in time 1, let $\mathcal{L} \triangleq \cup_{i=1}^{k} l_1^{(i)}$, and let $\texttt{proc}_1, \texttt{proc}_2, \ldots \texttt{proc}_m$ denote procedures belonging to different recursion families. Then, we perform the following partition:

$$\mathcal{L}_0 \triangleq \{(cs, \mathrm{S}) \in \mathcal{L} : \nexists\, \texttt{proc} \text{ such that } w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}^{\texttt{proc}}) = 1 \text{ and } w_{\mathrm{P}}^{\texttt{proc}}(\mathrm{S}) = 1\}.$$
$$\mathcal{L}_i \triangleq \{(cs, \mathrm{S}) \in \mathcal{L} : w_{\mathrm{P}}^{\texttt{proc}_i}(\mathrm{S}) = 1\}, \quad i = 1, \ldots, m.$$

By the definition of the sequential form of each controlled statement, we note that the partition is well-defined (e.g. there are no statements containing calls to more than one procedure).

Therefore, we are able to rearrange $\mathcal{L}$ and perform calls to **optimize**$^+$ in the following way:




Performing this operation on each time block and composing the circuits we obtain the rule for procedure_split. This concludes the proof. □

We now show that the compilation strategy **optimize**$^+$ applied to PFOQ$^{\text{UNIF}}$ programs ensures that the circuit complexity of a program is given by its Time function. That is to say, the complexity of the **qcase** is given by the maximum complexity of the branches.

**Theorem 8.10** (No branch sequentialization)**.** *For all* $\mathrm{P} \in \text{PFOQ}^{\text{UNIF}}$*, we have that*

$$\#\textbf{compile}^+(\mathrm{P}, n) = O(\mathrm{Time}_{\mathrm{P}}(n)).$$

*Proof.* The theorem can be shown by structural induction on the program body. All cases are straightforward except the one of the quantum control case. We make use of the following two facts regarding PFOQ$^{\text{UNIF}}$ programs:

(a) The size of the circuit is directly proportional to its total number of *unique* procedure calls (in the sense required for merging), and

(b) a recursive procedure call results in $O(n)$ unique calls to procedures of the same rank. This is because unique calls may only differ on procedure name (of which there is a constant amount) or input size (for which there is a linear number of possibilites).

Consider a quantum control statement $\mathrm{S} = \textbf{qcase}\ \mathrm{q}\ \textbf{of}\ \{0 \to \mathrm{S}_0, 1 \to \mathrm{S}_1\}$ appearing in **optimize** in the context of a recursive procedure proc. By (a), the circuit size for $\mathrm{S}_0$ and $\mathrm{S}_1$ are proportional to the (total) number of procedure calls in each statement, separately. We check that the number of ancillas created for S is bounded by the maximum number of ancillas between $\mathrm{S}_0$ and $\mathrm{S}_1$.

We proceed by induction on the rank $r$ of the procedure. The base case of $r = 1$ is given by (b), and so we may consider $r > 1$. For the inductive case, we consider two scenarios:

- $w^{\mathrm{P}}_{\mathtt{proc}}(\mathrm{S}_0) = w^{\mathrm{P}}_{\mathtt{proc}}(\mathrm{S}_1) = 1$. In this case, both $\mathrm{S}_0$ and $\mathrm{S}_1$ are of rank $r$, and all their recursive procedure calls may be merged. Therefore, the asymptotic number of such calls is bounded between the maximum between $\mathrm{S}_0$ and $\mathrm{S}_1$ (consider that, if there is no overlap between the ancillas needed, their number is still bounded linearly). Applying the IH on the procedures of rank $r - 1$ we obtain the desired result.

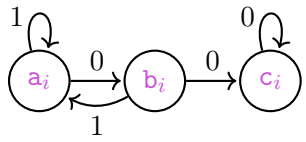
- $w^{\mathrm{P}}_{\mathtt{proc}}(\mathrm{S}_0) = 0$ and $w^{\mathrm{P}}_{\mathtt{proc}}(\mathrm{S}_1) = 1$. In this case, $\mathrm{S}_0$ contains calls to procedures of rank $r' < r$ whereas $\mathrm{S}_1$ contains calls to procedures of rank $r$. According to **optimize**, statement $\mathrm{S}_0$ is kept in the contextual circuit until no more statements are recursive relative to proc. The statements of rank $r'$ which are present in $\mathrm{S}_0$ are then merged with the equivalent procedures that were derived from $\mathrm{S}_1$ and also added to $l_{\mathrm{M}}$. The number of procedures of rank $r'$ is bounded asymptotically by the maximum between those in $\mathrm{S}_0$ and $\mathrm{S}_1$, therefore we obtain our result. □

PFOQ$^{\text{UNIF}}$, then, represents the first programming language sound and complete for quantum polynomial time, for which branch sequentialization can be avoided. To show the impact of the improved algorithm, we use the example of program `SubS_001` to create programs with an arbitrarily large polynomial slowdown.

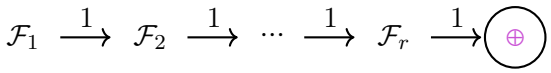
**Theorem 8.11.** *For any $k \in \mathbb{N}$, there exists a program $\mathrm{P}_k \in$ PFOQ$^{\text{UNIF}}$ such that*

$$\#\mathbf{compile}(\mathrm{P}_k, [n]) = \Omega(n^k \cdot \mathrm{Time}_{\mathrm{P}_k}(n)).$$

*Proof.* We revisit the `SubS_001` program of Example 8.1. We add a subscript to each procedure name, and denote the following set of nodes and edges by $\mathcal{F}_i$:

$\mathcal{F}_i \triangleq$ 


We define program $\mathrm{P}_r$ by chaining $r$ instances of $\mathcal{F}_i$ as follows:




where an arrow from $\mathcal{F}_i$ to $\mathcal{F}_{i+1}$ indicates an arrow from $\mathtt{d}_i$ to $\mathtt{a}_{i+1}$. The final program then consists of a call to procedure $\mathtt{a}_1$ on input $\bar{\mathrm{q}}$. It is easy to check that $\mathrm{P}_r \in$ PFOQ$^{\text{UNIF}}$, and that $\mathrm{Time}_{\mathrm{P}_r}(n) = n$, for any $r$. Program $\mathrm{P}_r$ performs the transformation

$$[\![\mathrm{P}_r]\!](|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus b\rangle$$

for $x \in \{0,1\}^*$ and $y, b \in \{0,1\}$ where $b$ is 1 if and only if $x$ contains at least $r$ instances of 001 as a substring. We have that $\#\mathbf{compile}(\mathrm{P}_r, [n]) = \Theta(n^{2r})$, therefore we may choose $r \geq \frac{k+1}{2}$ to obtain our desired result. □

Another example, which could also have been used to show a polynomial slowdown between **compile** and **compile**$^+$, is the problem of computing the Hamming weight of an input binary string.

**Example 8.12.** *Consider the decision problem of checking if an input bitstring $x \in \{0,1\}^n$ satisfies $\sum_{i=1}^n x_i = r$. For instance, if $r = 3$, we may define a* PFOQ$^{\text{UNIF}}$ *program* `SUM_3` *as:*

```
1  decl zero(q̄){ qcase q̄[1] of {0 → call zero(q̄ ⊖ [1]);, 1 → call one(q̄ ⊖ [1]); } },
2  decl one(q̄){ qcase q̄[1] of {0 → call one(q̄ ⊖ [1]);, 1 → call two(q̄ ⊖ [1]); } },
3  decl two(q̄){ qcase q̄[1] of {0 → call two(q̄ ⊖ [1]);, 1 → call three(q̄ ⊖ [1]); } },
4  decl three(q̄){
5    if |q̄| = 1 then
6        call ⊕(q̄);
7    else
8        qcase q̄[1] of {0 → call three(q̄ ⊖ [1]);, 1 → skip; } },
9  decl ⊕(q̄){ q̄[-1] *= NOT; }
10  :: call zero(q̄);
```

*We have that* $\mathrm{Time}_{\mathtt{SUM\_3}}(n) = n$.

In this chapter, we introduced PFOQ$^{\text{UNIF}}$ as a fragment of PFOQ that is still sound and complete for quantum polynomial time (Theorem 8.7). This fragment imposes a further restriction on procedures calls and the use of integer inputs that was not satisfied by programs such as `QFT`, but we were able to describe a semantically equivalent program `QFTu` that did satisfy the UNIF restrictions, and whose asymptotic time complexity was the same as for `QFT` (Lemma 8.6).

We demonstrated in Theorem 8.10 that circuits corresponding to PFOQ$^{\text{UNIF}}$ programs and obtained via the **optimize**$^+$ have a size complexity that scales asymptotically with the time complexity of the program. That is to say, the complexity of compiling the **qcase** statement is the maximum of the circuits corresponding to each branch, instead of the sum, as is the case for existing compilation strategies.

In the next chapter, we provide a Python implementation of **compile** and **compile**$^+$ that compiles PFOQ and LFOQ programs into Qiskit circuits. This allows us to empirically verify some results of the thesis.

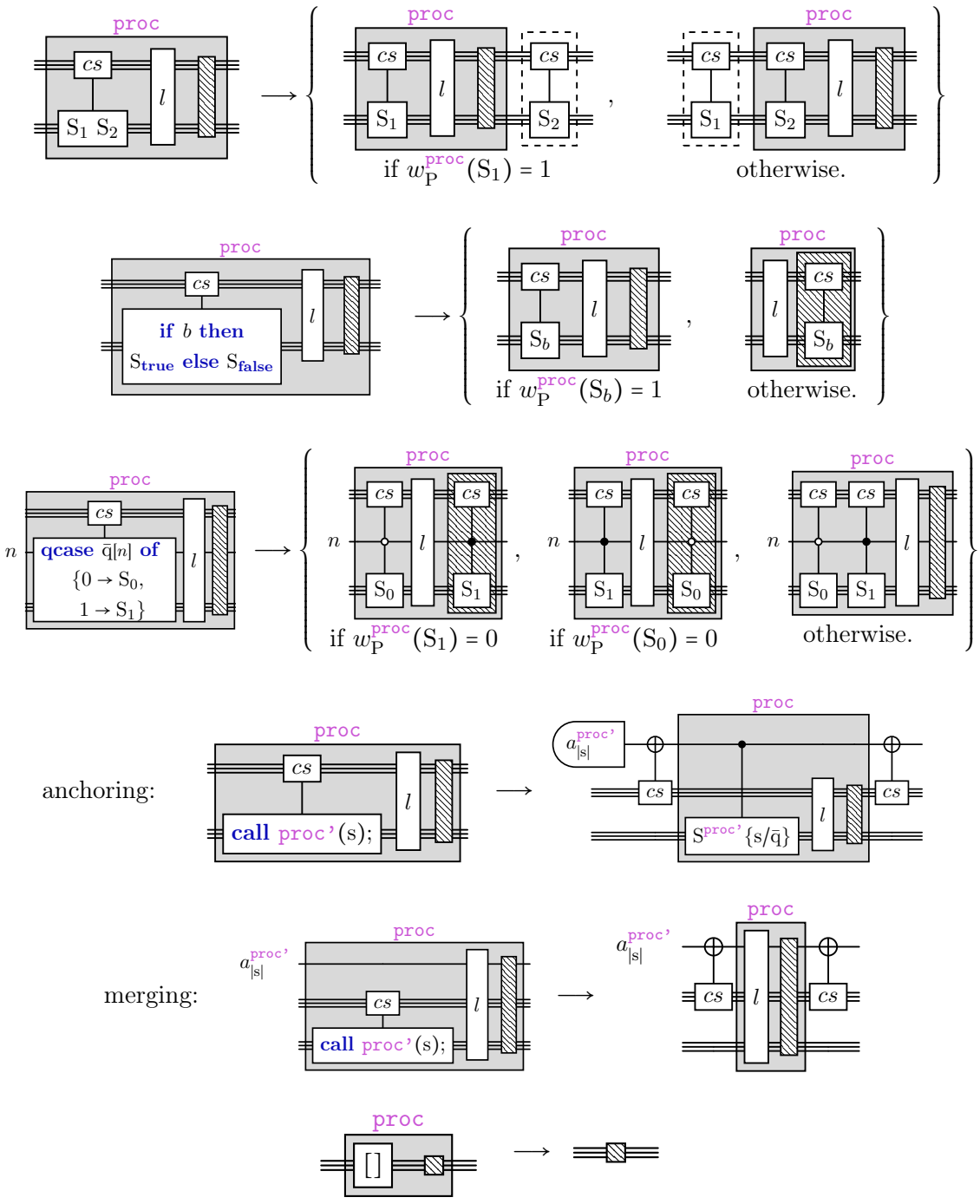



Figure 8.2: Rewrite rules of **optimize**$^+$.

# 9

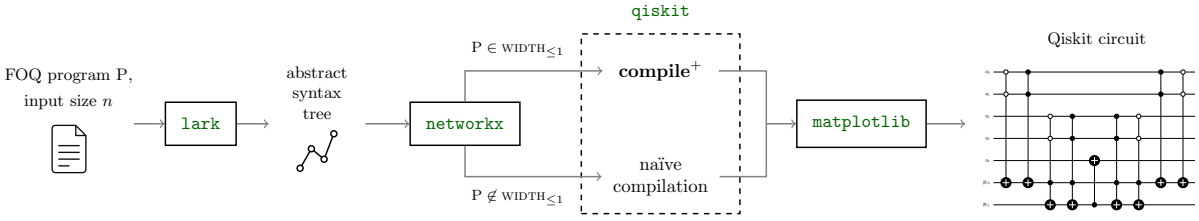
# Compiling FOQ Programs Into Qiskit

We introduce a compiler for FOQ programs that implements the compilation strategies of Chapters 7 and 8. The compiler generates Qiskit [JATK+24] circuits encoding FOQ programs and its code implementation is available at `https://gitlab.inria.fr/mmachado/pfoq-compiler`. It was developed in a collaboration with Alexandre Guernut, and comprises around 1350 lines of code in total.

The compiler is written in `python 3.10`, and uses the following additional packages:

- `typing_extensions 4.13.1`: allows for the use of type hints in the program;
- `lark 1.2.2`: performs the parsing of the FOQ program, resulting in an *abstract syntax tree* that is used to compile the circuit;
- `networkx 3.4.2`: used for generating the call-graph of the program, allowing to check if a program is in $\text{WIDTH}_{\leq 1}$; this package also helps supplement the abstract syntax tree with 'width' information that guides the compilation algorithm;
- `qiskit 2.0.0`: provides the interface with the Qiskit language;
- `matplotlib 3.10.1`: used for drawing the compiled circuit.

The structure of the compiler is roughly the following:




The syntax is similar to that of FOQ programs (Figure 4.1). Given a list of procedure declarations `D` and a program statement `S`, the program is written as

$$\texttt{D} :: \texttt{V} :: \texttt{S},$$

where `V` defines the quantum list variables in the program, which works to simplify the parsing.

The size of each input qubit list is given to the compiler as argument to the option `-i`, and the file to be compiled is passed to the option `-f`. Given $k$ variables, the circuit for program

`example.pfoq` can be obtained from the command

$$\texttt{python compiler.py -f example.pfoq -i n}_1\ \texttt{n}_2\ \ldots\ \texttt{n}_k,$$

where each $\texttt{n}_1, \ldots, \texttt{n}_k$ is a natural number denoting the size of the register corresponding to each variable declared in `V`.

## 9.1 Illustrating the use of the compiler via examples

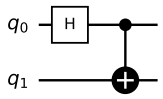
We provide a number of examples that help illustrate the different properties of the FOQ compiler, starting with the application of basic quantum gates, as well as the syntax for some larger, but particularly useful gates. We also consider example with recursion and multiple inputs, and in particular the case of PFOQ, LFOQ and UNIF programs.

### 9.1.1 Gate application and quantum control

Consider the following example program, that, given an input list of qubits `q`, applies a Hadamard gate to the first qubit and a CNOT gate to the first two qubits.

```
::
define q;
::
q[0] *= H;
qcase (q[0]) of {
  0 -> skip;,
  1 -> q[1] *= NOT; }
```

```
python compiler.py -f Bell_qcase.pfoq -i 2
```




We should highlight two aspects: the indexing of the compiler is 0-based, as opposed to the FOQ syntax which was 1-based. This allows for better interfacing with Qiskit, which is 0-based.

Another detail is that the CNOT is written using a `qcase` statement. Since this is a very standard statement, we include it as a basic instruction in the syntax. Therefore, the following program results in precisely the same circuit.

```
::
define q;
::
q[0] *= H;
CNOT(q[0],q[1]);
```

```
python compile.py -f Bell_CNOT.pfoq -i 2
```

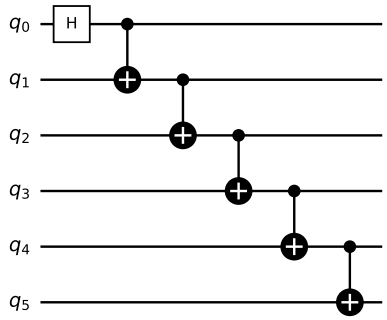
Other standard instructions include the `SWAP` and `TOF` which correspond to the SWAP and Toffoli gates. Unitary gates also include the `Rot` and `Ph`, which encode the rotation and phase-shift gates, respectively. In the case where there is a syntax error in the file, `lark` will raise an exception and point to the place in the file where the parsing failed.

### 9.1.2 Iteration

In order to be able to iterate over an input qubit list, we can define recursive functions as is allowed in FOQ, as demonstrated in the following example.

```
decl cnots(q){
  if (|q|>1) then {
    CNOT(q[0],q[1]);
    call cnots(q-[0]); }
  else { skip; } }
::
define q;
::
q[0] *= H;
call cnots(q);
```

```
python compiler.py -f cat_state.pfoq -i 6
```

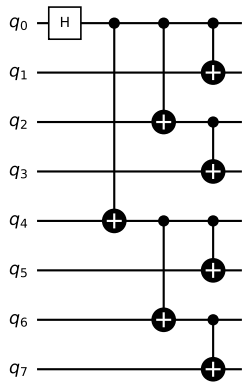



By default, the compiler places invisible barriers between gates that preserve the ordering defined by the cSompiler, and stops Qiskit from changing the gate order according to parallelization or commutativity relations between gates. This ensures that the circuit visualization correctly depicts the circuit as described by **compile**$^+$.

However, to verify our claims about the depth of circuits, specifically in the case of LFOQ programs, we allow the circuit representation to depict this. To allow Qiskit to parallelize gates, one may use the `--no-barriers` flag at the moment of compilation.

```
decl cnots_par(q){
  if (|q|>1) then {
    CNOT(q[0],q[(|q|)/2]);
    call cnots_par((q)^-);
    call cnots_par((q)^+); }
  else { skip; } }
::
define q;
::
q[0] *= H;
call cnots_par(q);
```

```
python compiler.py -f cat_state_parallel.pfoq -i 8 --no-barriers
```



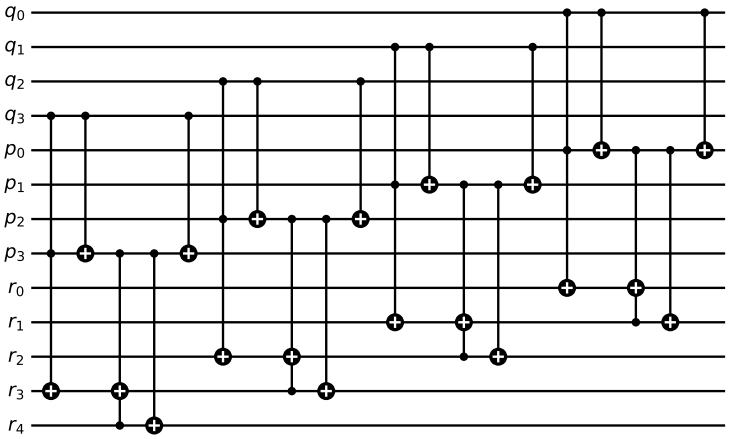

For the previous program, the compiler prints out the message "`Procedure cnots_par has width 2. Turning off optimization.`" indicating that the program does not satisfy the $\text{WIDTH}_{\leq 1}$ condition. For such programs, the optimization subroutine is not well-defined and so is not called. The program is then compiled using only the rules of **compile** (Figure 7.5), where we always consider the first case for procedure calls. Compilation without optimization is also accessible via the `--no-optimize` option given to the compiler.

The number of arguments given should correspond to the number of variables defined in the program. For example, in the `QRCA` program of Figure 4.5, the program takes in 3 different inputs.

```
decl add(q,p,r){
  TOF(q[-1],p[-1],r[-2]);
  CNOT(q[-1],p[-1]);
  TOF(p[-1],r[-1],r[-2]);
  CNOT(p[-1],r[-1]);
  CNOT(q[-1],p[-1]); }

decl full_adder(q,p,r){
  if (|r|>1) then {
  call add(q,p,r);
  call full_adder(q-[-1],
                  p-[-1],
                  r-[-1]); }}
::
define q p r;
::
call full_adder(q,p,r);
```

```
python compiler.py -f QRCA.pfoq -i 4 4 5
```

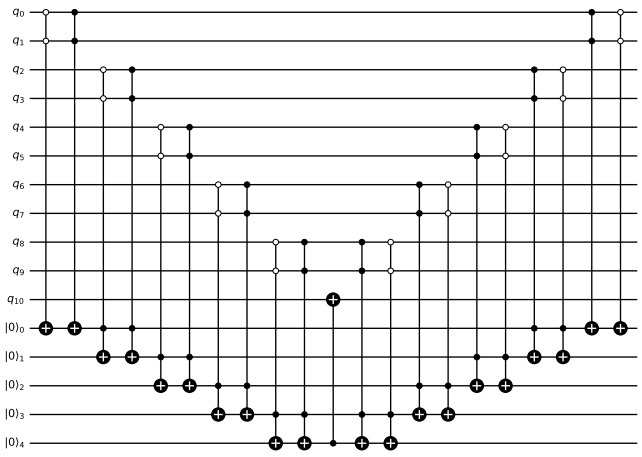



### 9.1.3 Anchoring and merging

In the case where a procedure is recursive and satisfies the $\text{WIDTH}_{\leq 1}$ condition, the compilation is performed with the **optimize**$^+$ routine (Algorithm 3). This generally requires the use of ancilla qubits, as in the following example of program `PAIRS` (Figure 7.1).

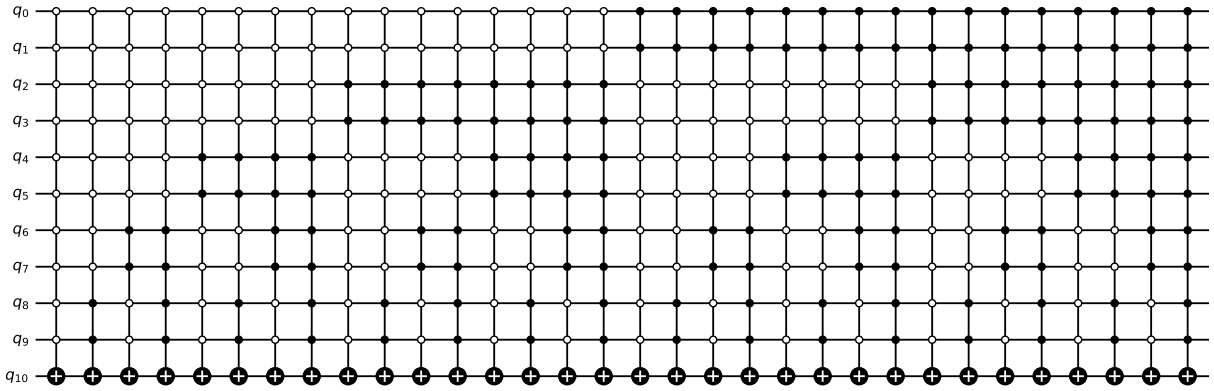
```
decl pairs(q){
  if (|q|>1) then {
    qcase(q[0]) of {
      0 -> qcase (q[1]) of {
        0 -> call pairs(q-[0,1]);,
        1 -> skip; },
      1 -> qcase(q[1]) of {
        0 -> skip;,
        1 -> call pairs(q-[0,1]); }
    }
  } else { q[0] *= NOT; }
}
::
define q;
::
call pairs(q);
```

```
python compiler.py -f pairs.pfoq -i 11
```




The compiler automatically determines a sufficient amount of ancillas for the compilation, which depends on the program and the input size. We can compare the circuit for `PAIRS` obtained without optimization (the result of the "in-depth" strategy of Figure 7.2). In this case, the circuit corresponds to $2^{\frac{n-1}{2}}$ Toffoli gates, each with $n-1$ control-qubits.

```
python compiler.py -f pairs.pfoq -i 11 --no-optimize
```




We now consider programs where merging requires the use of controlled-swaps. Since `PAIRS` $\in$ PFOQ$^{\text{UNIF}}$, all merged procedures have the same input qubits and the merging complexity is $O(1)$. Generally, merging procedure calls involves the rearrangement of qubit addresses. In this case, the compiler performs the controlled-swapping technique described in Lemma 7.1 which adds to the circuit a controlled permutation done in logarithmic depth.

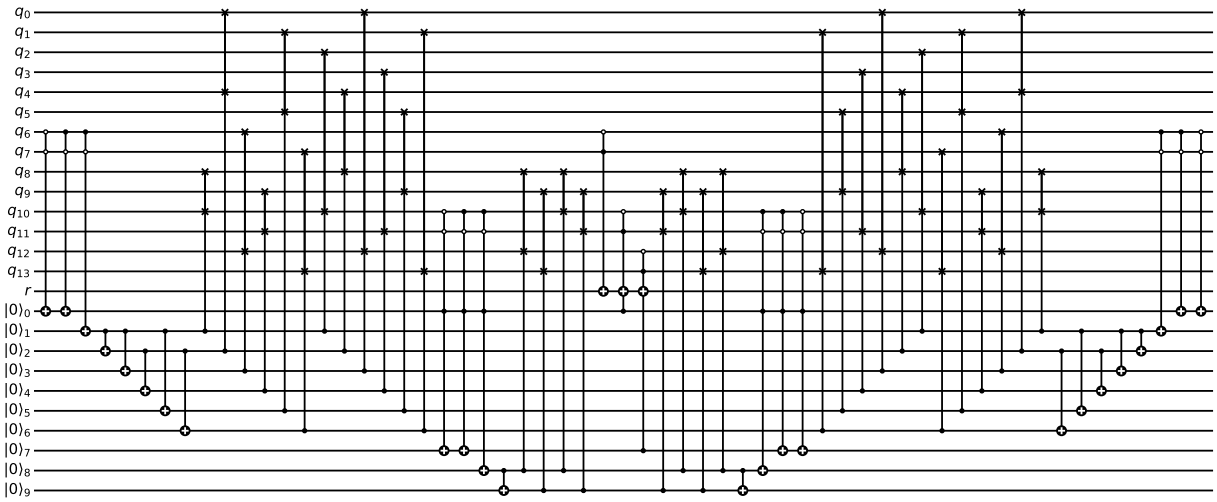
An example of such a program with procedure calls in different registers is the binary search algorithm `SEARCH` defined in Figure 6.1. The program makes use of two qubit list variables, `q` and `r`. The first half of list `q` is given by `(q)^-` and the second half as `(q)^+`, corresponding to the FOQ syntax of $\bar{\mathrm{q}}^{\boxminus}$ and $\bar{\mathrm{q}}^{\boxplus}$.

```
decl search(q,r){
  if (|q|>1) then {
    qcase (q[|q|/2-1]) of {
      0 -> qcase(q[|q|/2]) of { 0 -> call search((q)^+ -[0],r);,
                                1 -> r[0] *= NOT; },
      1 -> qcase(q[|q|/2]) of { 0 -> call search((q)^- -[-1],r);,
                                1 -> skip; } }
  } else { skip; } }
::
define q r;
::
call search(q,r);
```

For input size `|q|` of 14, the first merging occurs between the call to `search` on the first and second halves of `q` which, after having removed the middle qubits. The compiler then prepares a controlled permutation between these two qubit indices by setting 6 ancillas into the appropriate control state, which can be done in parallel with 4 time-steps, and then the two sets of disjoint transpositions are executed in two time-steps. This occurs again when the two instances of `search` on inputs `q[12,13]` and `q[9,8]` are merged.

This circuit can be compared with the one given in Figure 7.11 where the controlled-swapping of qubit addresses was left implicit.

```
python compiler.py -f search.pfoq -i 14 1
```



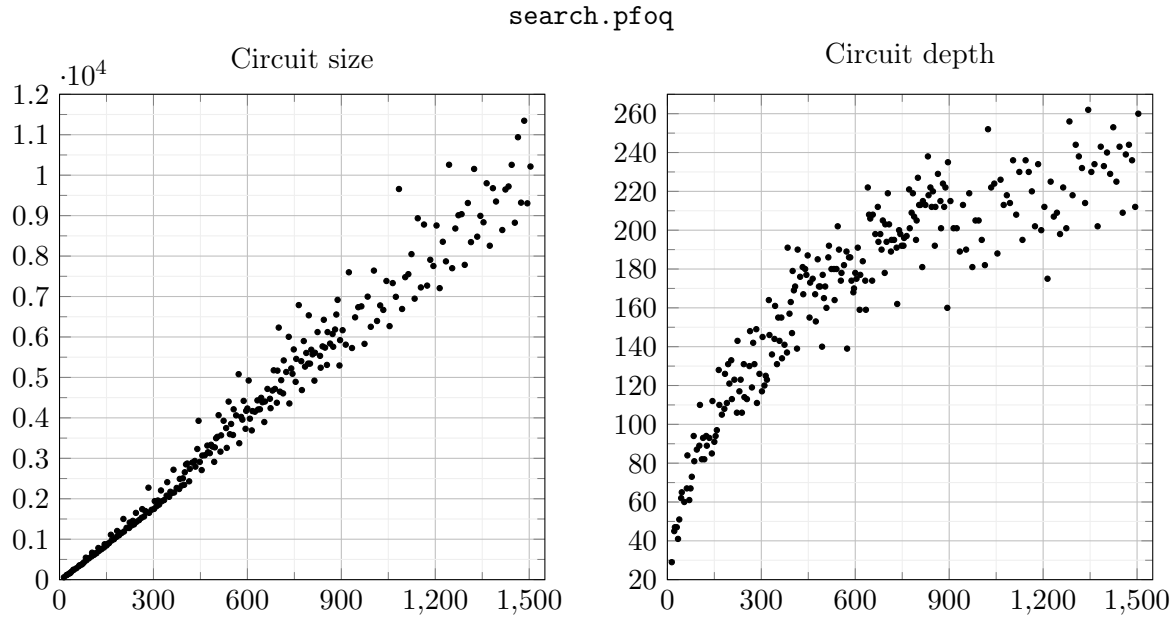

We now verify the properties of the compilation procedure given in the previous chapters.

## 9.2 Properties of the compilation algorithm

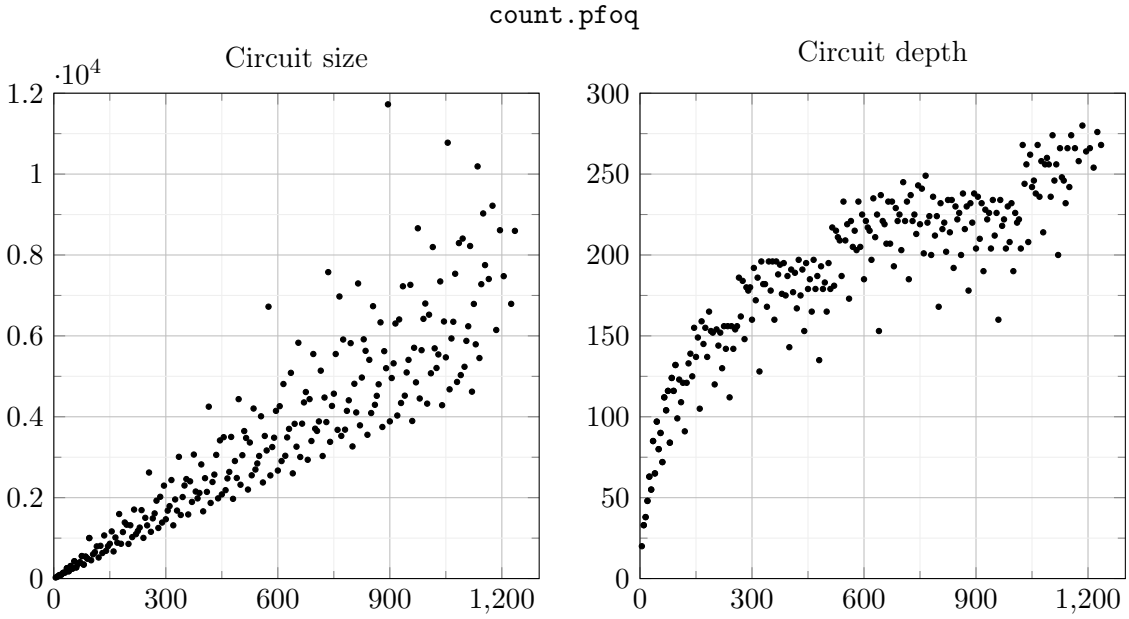
We start by considering the asymptotic size and depth of circuits. All values correspond to a compilation using **compile**$^+$, unless stated otherwise.

### 9.2.1 Size and depth bounds

We can empirically test the circuit bounds given in Chapters 7 and 8 by checking the asymptotic sizes of the obtained circuits. For instance, for the `SEARCH` program of Figure 6.1 the circuit size grows in $O(n \log n)$ but the depth grows as $O(\log^2 n)$.



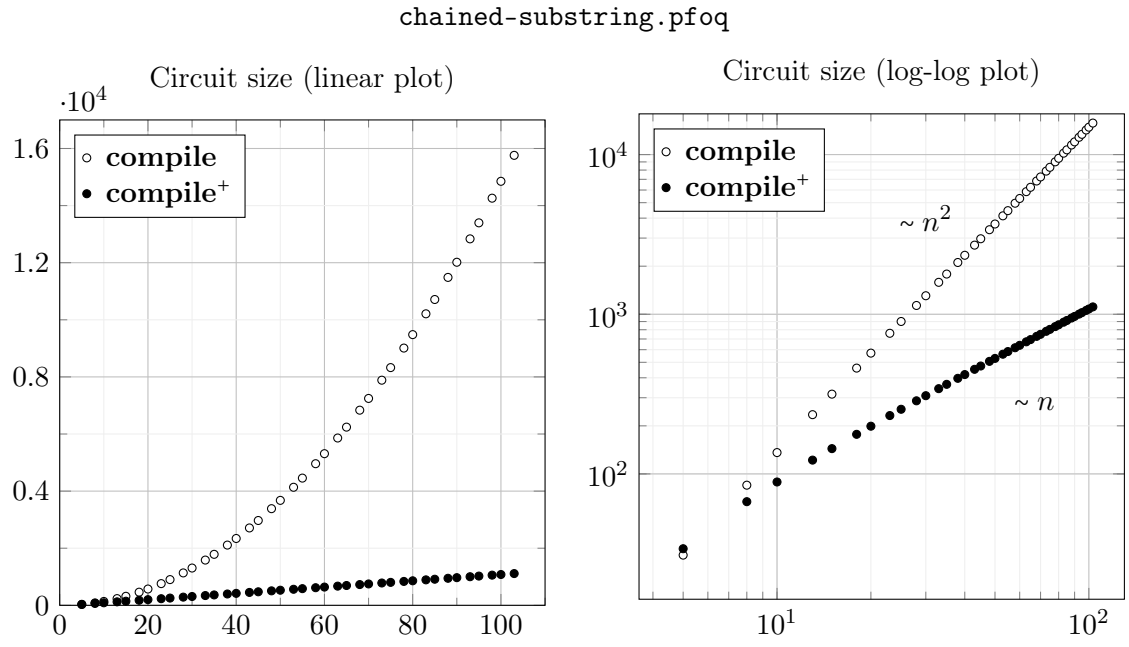
We can verify a similar property for the `COUNT` program, also defined in Figure 6.1.

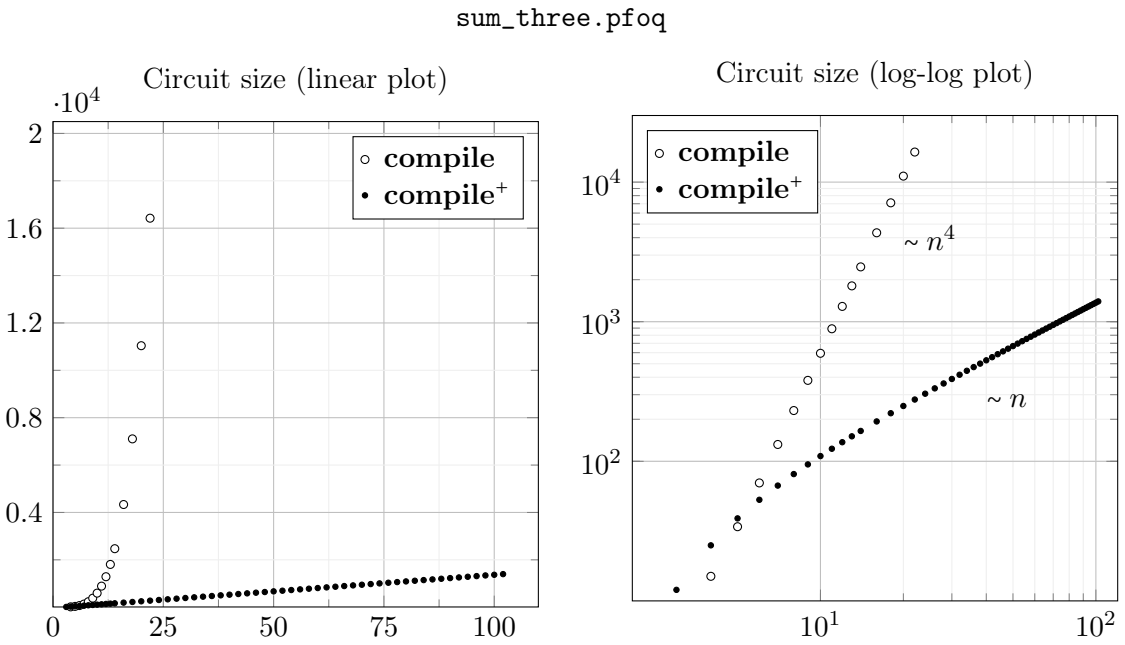



### 9.2.2 Polynomial slowdown

In Theorem 8.11 we give a separation between the circuit complexity obtained from **compile** and **compile**$^+$ when considering PFOQ$^{\text{UNIF}}$ programs. We can show this separation empirically by considering the programs for the chained substring problem (proof of Theorem 8.11) and detecting if the weight of an input strings is a given value (Example 8.12).

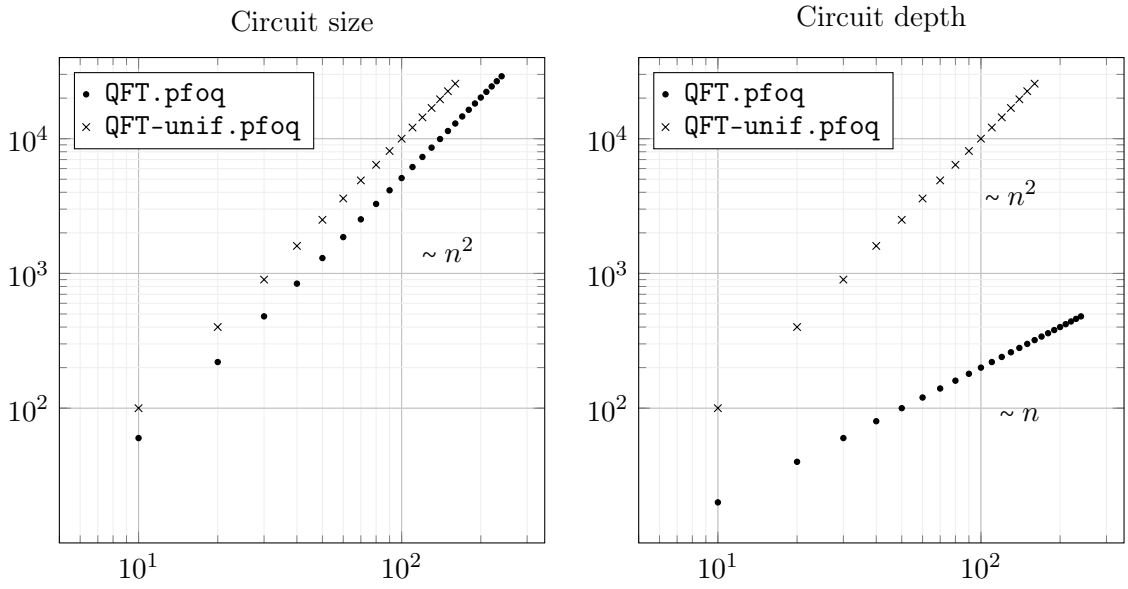
We represent the circuit sizes obtained from **compile** and **compile**$^+$ in a *log-log plot*, which allows not only for determining that the sizes increase polynomially, but we are also able to estimate its degree via linear regression.






In a log-log plot, a polynomial function with leading coefficient $an^k$ will be represented by a graph that asymptotically approaches the line $y = k\log(n) + \log(a)$. Therefore, we are able to estimate the leading power for the circuit sizes obtained for the programs by analysing the slope of the line in the log-log plot.

Another interesting comparison is that of `QFT` and `QFTu`. We notice that, while the asymptotic circuit complexity is the same, the circuit depth is worse for `QFTu`.




This difference in depth complexity highlights an important aspect of our work and the balance of the PFOQ, LFOQ and UNIF restrictions. The fact that `QFTu` $\in$ PFOQ$^{\text{UNIF}}$ allows us to conclude that (a) it is a polytime program and (b) it may be compiled into a family of circuits with size that scales linearly with the time complexity of the program. Analysing the call graph of the program, we can further conclude that the time complexity is quadratic.

However, there exists an equivalent program, `QFT`, that does *not* satisfy the UNIF condition and which results in circuits with better asymptotic depth. This illustrates the fact that, while program restrictions may enable us to prove strong results about program complexity, they may exclude approaches that result in better results for specific cases. Therefore, it is important to strike a balance between the strength of our results and the expressivity of the restricted fragments.

This chapter presented a Python compiler of FOQ programs that allowed us to verify some of the claims of the thesis, such as the size and depth bounds of the circuits obtained from the **compile** and **compile**$^+$ strategies. This prototype constitutes a proof-of-concept for allowing implicit characterizations of quantum complexity classes to inform compilation techniques.

# Conclusion

The purpose of this PhD thesis was to study the resource use of quantum programs from the point of view of their structure, and to provide new implicit characterizations of efficient quantum complexity classes that could be of use to the quantum programmer. With this objective, we introduced a first-order quantum programming language FOQ (Chapter 4) for which we defined fragments characterizing quantum polynomial and polylogarithmic time.

The fragments were defined via syntactical restrictions, more precisely a limit on the depth of recursion using a reduction of available qubits (*well-foundedness*, Definition 5.1), and a bound on the number of recursive calls that can be performed in sequence (*bounded width*, Definition 5.10).

In the case of polynomial time, the fragment PFOQ imposes a *linear* reduction on available qubits, with at least one qubit being removed per recursive call, whereas in the case of polylogarithmic time, LFOQ programs ensure that the reduction is *exponential*, with at least half the accessible qubits being removed in each recursive call (*recursively-halving*, Definition 6.1). We show that the syntactic restrictions satisfy desirable properties, such as being preserved when the programs are inverted, and allowing for the simple combination of programs while ensuring the restrictions.

We demonstrated the expressivity of the PFOQ and LFOQ fragments via a number of examples. In the case of PFOQ, we gave programs for addition and multiplication (Examples 4.4, 5.17 and 5.18), and the quantum Fourier transform and its inverse (Examples 4.5 and 4.9), among others. For the case of the LFOQ fragment, we showed that we can write programs performing quantum binary search (Example 6.2) and counting of elements on sorted lists (Example 6.3). The main examples of programs are depicted in Figure 1.

The proofs of soundness and completeness used two types of techniques, namely the simulation of quantum Turing machines (both in the standard QTM model and in the random-access model), and the technique of simulating another implicit characterization.

We also considered the problem of ensuring an efficient translation from PFOQ and LFOQ programs into quantum circuits of appropriate size and depth. From a theoretical point of view, this can be seen as an alternative proof of soundness, but it also allows for the practical use of the fragments. This is a property which was not present in previously-existing implicit characterizations of quantum complexity classes.

The objective of providing a direct compilation technique that was resource-preserving proved to be non-trivial, in particular due to fact that FOQ includes quantum control statements. Existing approaches to compiling these **qcase** statements treat the complexity of the **qcase** statement as the *sum* of its branches instead of the maximum, by compiling all branches separately, either in depth or in width. As we saw in Section 7.1, this can lead to an *exponential blow-up* in circuit complexity, even for very simple programs.

In Chapter 7, we solved this problem by introducing a compilation technique called *anchoring and merging* which simplifies recursive calls performed in different branches of a **qcase** statement, and we show that this technique ensures correct resource bounds for PFOQ programs (polynomial

size) and also for LFOQ programs (polylogarithmic depth).

Finally, we show in Chapter 8 that, while the compilation technique of Chapter 7 ensures asymptotic bounds within that respect the complexity class of the program, it imposes a *polynomial slowdown* due to the use of the **qcase** statement. We introduce a stricter fragment of PFOQ that satisfies a *uniformity* condition on the removal of qubits in procedure calls (Definition 8.3). We unconditionally improve the compilation technique of Chapter 7 and prove that, for programs in the uniform PFOQ fragment, the circuit complexity of the **qcase** statement is given by the maximum of each branch.

We conclude the thesis by presenting a Python implementation of a compiler prototype that implements the compilation strategies of Chapters 7 and 8 and allows for the empirical verification of various claims made in this work about these fragments.

## Future work

**Compilation strategies.** The problem of compiling the **qcase** statement has been gaining attention recently, particularly in the context of problems involving quantum control [YC22]. Regarding the compilation technique of Chapter 8, one interesting research direction is relatively apparent, namely the relaxation of the UNIF restrictions (Definition 8.3), so that the class becomes more expressive.

As the definition stands, the composition of two UNIF programs (in the sense of Definition 4.1) is not itself a UNIF program. This is obviously a desirable property, and can be satisfied, for instance, by extending the UNIF definition such that input qubit list restriction is not unique over the program but may differ for procedures that are in different connected components of the call graph. More extensions can be described that would include programs such as the `QFT`, as described in Figure 4.6, which, as we saw in Chapter 9, has a better asymptotic depth.

**Higher-order implicit characterizations.** As the name suggests, FOQ is a first-order language, meaning that its procedures only take as input first-order values, such as qubit lists or integers (and not, for instance, other procedures). An interesting research direction is the extension of our work to the case of *higher-order computation*.

This work is already under way [DCHPS24], with a collaboration with Alejandro Díaz-Caro, based on an extension of the Lambda-$\mathcal{S}_1$ typing system [DCM22] with polymorphic types from Dual Light Affine Logic [BT04], resulting in a system where unitarity of qubit transformations is ensured, and all terms reduce to a normal form in a polynomial number of steps.

**Circuit parallelization.** The class QNC is the quantum equivalent of NC, and corresponds to the set of functions computed by uniform families of quantum circuits with polynomial size and polylogarithmic depth [MN01, Coo23]. To our knowledge, no implicit characterization of QNC exists, and it would be interesting to investigate if extensions of the FOQ language could provide such a characterization.

One possible strategy, to enable reasoning about the parallelization of circuits is, for instance, introducing a statement allowing for recursive calls to be performed on disjoint sets of qubits. The LFOQ language also constitutes a good starting point for a characterization of QNC, since LFOQ programs always correspond to polylogarithmic-depth circuits.

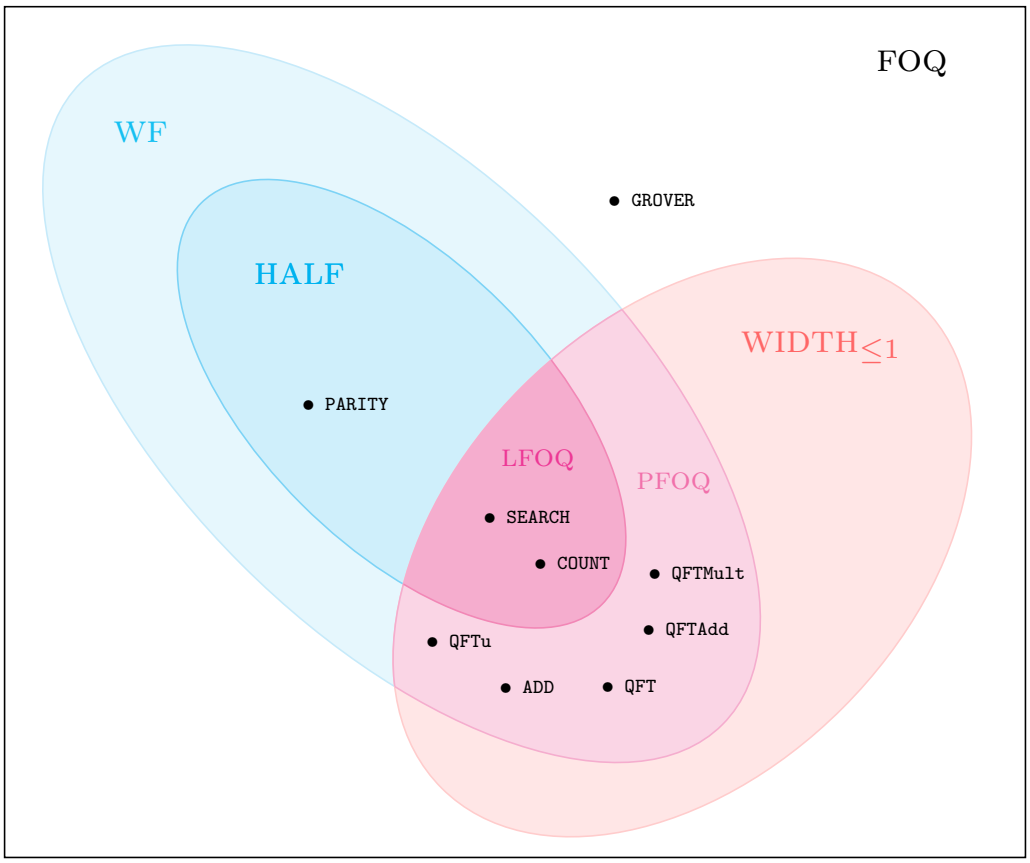



Figure 1: Venn diagram of FOQ restrictions.

# Bibliography


[ADH97] Leonard M. Adleman, Jonathan DeMarrais, and Ming-Deh A. Huang. Quantum computability. *SIAM Journal on Computing*, 26(5):1524–1540, 1997. doi:10.1137/S0097539795293639.

[AG04] Scott Aaronson and Daniel Gottesman. Improved simulation of stabilizer circuits. *Physical Review A—Atomic, Molecular, and Optical Physics*, 70(5):052328, 2004. doi:10.1103/PhysRevA.70.052328.

[AG05] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 249–258, 2005. doi:10.1109/LICS.2005.1.

[AL18] Tameem Albash and Daniel A Lidar. Adiabatic quantum computation. *Reviews of Modern Physics*, 90(1):015002, 2018. doi:10.1103/RevModPhys.90.015002.

[Amb18] Andris Ambainis. Understanding quantum algorithms via query complexity. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*, pages 3265–3285. World Scientific, 2018. doi:10.1142/9789813272880_0181.

[BB14] Charles H. Bennett and Gilles Brassard. Quantum cryptography: public key distribution and coin tossing. *Theoretical Computer Science*, 560:7–11, dec 2014. doi:10.1016/j.tcs.2014.05.025.

[BBBV97] Charles H. Bennett, Ethan Bernstein, Gilles Brassard, and Umesh Vazirani. Strengths and weaknesses of quantum computing. *SIAM journal on Computing*, 26(5):1510–1523, 1997. doi:10.1137/S0097539796300933.

[BBC+95] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Physical Review A*, 52(5):3457–3467, November 1995. doi:10.1103/physreva.52.3457.

[BBC+01] Robert Beals, Harry Buhrman, Richard Cleve, Michele Mosca, and Ronald De Wolf. Quantum lower bounds by polynomials. *Journal of the ACM (JACM)*, 48(4):778–797, 2001. doi:10.1145/502090.502097.

[BBD+09] Hans J. Briegel, David E. Browne, Wolfgang Dür, Robert Raussendorf, and Maarten Van den Nest. Measurement-based quantum computation. *Nature Physics*, 5(1):19–26, 2009. doi:10.1038/nphys1157.

[BBGV20] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 286–300, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3386007.

[BCS03] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *The European Physical Journal D - Atomic, Molecular and Optical Physics*, 25(2):181–200, August 2003. doi:10.1140/epjd/e2003-00242-2.

[Ben73] Charles H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973. doi:10.1147/rd.176.0525.

[BMP+99] Patrick Oscar Boykin, Tal Mor, Matthew Pulver, Vwani Roychowdhury, and Farrokh Vatan. On universal and fault-tolerant quantum computing, 1999. doi:10.48550/ARXIV.QUANT-PH/9906054.

[BT04] Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004*, pages 266–275. IEEE, 2004. doi:10.1016/j.ic.2008.08.005.

[BV93] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory (conference version). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 11–20, New York, NY, USA, 1993. Association for Computing Machinery. doi:10.1145/167088.167097.

[BV97] Ethan Bernstein and Umesh Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997. doi:10.1137/S0097539796300921.

[CDKM04] Steven A. Cuccaro, Thomas G. Draper, Samuel A. Kutin, and David Petrie Moulton. A new quantum ripple-carry addition circuit, 2004. arXiv:quant-ph/0410184.

[CDL24] Andrea Colledan and Ugo Dal Lago. Circuit width estimation via effect typing and linear dependency. In Stephanie Weirich, editor, *Programming Languages and Systems*, pages 3–30, Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-57267-8_1.

[CDL25] Andrea Colledan and Ugo Dal Lago. Flexible type-based resource estimation in quantum circuit description languages. *Proceedings of the ACM on Programming Languages*, 9(POPL):1386–1416, 2025. doi:10.1145/3704883.

[Coo23] Stephen A. Cook. *Towards a Complexity Theory of Synchronous Parallel Computation*, page 219–244. Association for Computing Machinery, New York, NY, USA, 1 edition, 2023. doi:10.1145/3588287.3588301.

[DBK+22] Andrew J. Daley, Immanuel Bloch, Christian Kokail, Stuart Flannigan, Natalie Pearson, Matthias Troyer, and Peter Zoller. Practical quantum advantage in quantum simulation. *Nature*, 607(7920):667–676, Jul 2022. doi:10.1038/s41586-022-04940-6.

[DCHPS24] Alejandro Díaz-Caro, Emmanuel Hainry, Romain Péchoux, and Mário Silva. A feasible and unitary quantum programming language. working paper or preprint, March 2024. URL: https://inria.hal.science/hal-04266203.

[DCM22] Alejandro Díaz-Caro and Octavio Malherbe. Quantum control in the unitary sphere: Lambda-$\mathcal{S}_1$ and its categorical model. *Logical Methods in Computer Science*, Volume 18, Issue 3, September 2022. doi:10.46298/lmcs-18(3:32)2022.

[Deu85] David Deutsch. Quantum theory, the Church–Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400(1818):97–117, 1985. doi:10.1098/rspa.1985.0070.

[Deu89] David Deutsch. Quantum computational networks. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 425(1868):73–90, 1989. doi:10.1098/rspa.1989.0099.

[Dev24] Cirq Developers. *Cirq*. Zenodo, May 2024. doi:10.5281/ZENODO.4062499.

[DL12] Ugo Dal Lago. *A short introduction to implicit computational complexity*, pages 89–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-31485-8_3.

[DMZ10] Ugo Dal Lago, Andrea Masini, and Margherita Zorzi. Quantum implicit computational complexity. *Theoretical Computer Science*, 411(2):377–409, 2010. doi:10.1016/j.tcs.2009.07.045.

[Dra00] Thomas G. Draper. Addition on a quantum computer, 2000. arXiv:quant-ph/0008033.

[Fey82] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, Jun 1982. doi:10.1007/BF02650179.

[FGGS98] Edward Farhi, Jeffrey Goldstone, Sam Gutmann, and Michael Sipser. Limit on the speed of quantum computation in determining parity. *Physical Review Letters*, 81(24):5442–5444, December 1998. doi:10.1103/physrevlett.81.5442.

[FHPS25] Florent Ferrari, Emmanuel Hainry, Romain Péchoux, and Mário Silva. Quantum programming in polylogarithmic time. In Paweł Gawrychowski, Filip Mazowiecki, and Michał Skrzypczak, editors, *50th International Symposium on Mathematical Foundations of Computer Science (MFCS 2025)*, volume 345 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 47:1–47:17, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2025.47.

[FKLW03] Michael Freedman, Alexei Kitaev, Michael Larsen, and Zhenghan Wang. Topological quantum computation. *Bulletin of the American Mathematical Society*, 40(1):31–38, 2003. doi:10.1090/S0273-0979-02-00964-3.

[GLM08] Vittorio Giovannetti, Seth Lloyd, and Lorenzo Maccone. Quantum random access memory. *Physical Review Letters*, 100(16), April 2008. doi:10.1103/physrevlett.100.160501.

[GLR+13] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: a scalable quantum programming language. In *PLDI 2013*, pages 333–342. ACM, 2013. doi:10.1145/2491956.2462177.

[Got98] Daniel Gottesman. The Heisenberg representation of quantum computers. *Proc. XXII International Colloquium on Group Theoretical Methods in Physics, 1998*, pages 32–43, 1998. URL: https://cir.nii.ac.jp/crid/1573950400750455808.

[Gro96] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237814.237866.

[GS22] Walther Gerlach and Otto Stern. Der experimentelle Nachweis der Richtungsquantelung im Magnetfeld. *Zeitschrift für Physik*, 9(1):349–352, Dec 1922. doi:10.1007/BF01326983.

[HPS23] Emmanuel Hainry, Romain Péchoux, and Mário Silva. A programming language characterizing quantum polynomial time. In Orna Kupferman and Pawel Sobocinski, editors, *Foundations of Software Science and Computation Structures*, pages 156–175, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-30829-1_8.

[HPS25] Emmanuel Hainry, Romain Péchoux, and Mário Silva. Branch sequentialization in quantum polytime. In Maribel Fernández, editor, *10th International Conference on Formal Structures for Computation and Deduction (FSCD 2025)*, volume 337 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:22, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2025.22.

[HYC+19] W. Huang, C. H. Yang, K. W. Chan, T. Tanttu, B. Hensen, R. C. C. Leon, M. A. Fogarty, J. C. C. Hwang, F. E. Hudson, K. M. Itoh, A. Morello, A. Laucht, and A. S. Dzurak. Fidelity benchmarks for two-qubit gates in silicon. *Nature*, 569(7757):532–536, May 2019. doi:10.1038/s41586-019-1197-0.

[JATK+24] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, et al. Quantum computing with Qiskit. 2024. doi:10.48550/arXiv.2405.08810.

[JR23] Samuel Jaques and Arthur G. Rattew. QRAM: A survey and critique, 2023. arXiv:2305.10310.

[KF82] Ker-I. Ko and Harvey Friedman. Computational complexity of real functions. *Theoretical Computer Science*, 20(3):323–352, 1982. doi:10.1016/S0304-3975(82)80003-0.

[KG25] Tanuj Khattar and Craig Gidney. Rise of conditionally clean ancillae for efficient quantum circuit constructions. *Quantum*, 9:1752, 2025. doi:10.22331/q-2025-05-21-1752.

[KMN+07] Pieter Kok, W. J. Munro, Kae Nemoto, T. C. Ralph, Jonathan P. Dowling, and G. J. Milburn. Linear optical quantum computing with photonic qubits. *Rev. Mod. Phys.*, 79:135–174, Jan 2007. doi:10.1103/RevModPhys.79.135.

[Kni96] E Knill. Conventions for quantum pseudocode. Technical report, Los Alamos National Lab., NM (United States), 06 1996. doi:10.2172/366453.

[LP98] Noah Linden and Sandu Popescu. The halting problem for quantum computers. 1998. doi:10.48550/arXiv.quant-ph/9806054.

[LS01] Gui-Lu Long and Yang Sun. Efficient scheme for initializing a quantum register with an arbitrary superposed state. *Phys. Rev. A*, 64:014303, Jun 2001. doi:10.1103/PhysRevA.64.014303.

[MN01] Cristopher Moore and Martin Nilsson. Parallel quantum computation and quantum codes. *SIAM Journal on Computing*, 31(3):799–815, 2001. doi:10.1137/S0097539799355053.

[MW19] Abel Molina and John Watrous. Revisiting the simulation of quantum Turing machines by quantum circuits. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 475(2226):20180767, June 2019. doi:10.1098/rspa.2018.0767.

[Mye97] John M. Myers. Can a universal quantum computer be fully quantum? *Phys. Rev. Lett.*, 78:1823–1824, Mar 1997. doi:10.1103/PhysRevLett.78.1823.

[NC10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. doi:10.1017/CBO9780511976667.

[PRZ17] Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE: a core language for quantum circuits. In Giuseppe Castagna and Andrew D. Gordon, editors, *POPL 2017*, pages 846–858. ACM, 2017. doi:10.1145/3009837.3009894.

[RPGE17] Lidia Ruiz-Perez and Juan Carlos Garcia-Escartin. Quantum arithmetic with the quantum Fourier transform. *Quantum Information Processing*, 16(6), April 2017. doi:10.1007/s11128-017-1603-1.

[Ruz81] Walter L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, 1981. doi:10.1016/0022-0000(81)90038-6.

[SBZ+24] Raphael Seidel, Sebastian Bock, René Zander, Matic Petrič, Niklas Steinmann, Nikolay Tcholtchev, and Manfred Hauswirth. Qrisp: A framework for compilable high-level programming of gate-based quantum computers. 2024. doi:10.48550/arXiv.2406.14792.

[SDC+20] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology*, 6(1):014003, nov 2020. doi:10.1088/2058-9565/ab8e92.

[Sel04] Peter Selinger. Towards a quantum programming language. *Mathematical Structures in Computer Science*, 14(4):527–586, 2004. doi:10.1017/S0960129504004256.

[SFMZC23] Mário Silva, Ricardo Faleiro, Paulo Mateus, and Emmanuel Zambrini Cruzeiro. A coherence-witnessing game and applications to semi-device-independent quantum key distribution. *Quantum*, 7:1090, August 2023. doi:10.22331/q-2023-08-22-1090.

[Sho94] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. doi:10.1109/SFCS.1994.365700.

[STY+23] Xiaoming Sun, Guojing Tian, Shuai Yang, Pei Yuan, and Shengyu Zhang. Asymptotically optimal circuit depth for quantum state preparation and general unitary synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42(10):3301–3314, 2023. doi:10.1109/TCAD.2023.3244885.

[SV06] Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *Math. Struct. Comput. Sci.*, 16(3):527–552, 2006. doi:10.1017/S0960129506005238.

[TK05] Yasuhiro Takahashi and Noboru Kunihiro. A linear-size quantum circuit for addition with no ancillary qubits. *Quantum Info. Comput.*, 5(6):440–448, September 2005. doi:10.26421/QIC5.6-2.

[Tur37] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937. doi:10.1112/plms/s2-42.1.230.

[VBE96] Vlatko Vedral, Adriano Barenco, and Artur Ekert. Quantum networks for elementary arithmetic operations. *Physical Review A*, 54(1):147–153, July 1996. doi:10.1103/physreva.54.147.

[Vid03] Guifré Vidal. Efficient classical simulation of slightly entangled quantum computations. *Physical Review Letters*, 91(14), October 2003. doi:10.1103/physrevlett.91.147902.

[VLRH23] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. Qunity: A unified language for quantum and classical computing. *Proceedings of the ACM on Programming Languages*, 7(POPL):921–951, 2023. doi:10.1145/3571225.

[WS14] Dave Wecker and Krysta M. Svore. LIQUi|>: A software design architecture and domain-specific language for quantum computing, 2014. arXiv:1402.4467.

[WY23] Qisheng Wang and Mingsheng Ying. Quantum random access stored-program machines. *Journal of Computer and System Sciences*, 131:13–63, 2023. doi:10.1016/j.jcss.2022.08.002.

[Yam20] Tomoyuki Yamakami. A schematic definition of quantum polynomial time computability. *J. Symb. Log.*, 85(4):1546–1587, 2020. doi:10.1017/jsl.2020.45.

[Yam22] Tomoyuki Yamakami. Expressing power of elementary quantum recursion schemes for quantum logarithmic-time computability. In Agata Ciabattoni, Elaine Pimentel, and Ruy J. G. B. de Queiroz, editors, *Logic, Language, Information, and Computation*, pages 88–104, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-15298-6_6.

[Yao93] Andrew Chi-Chih Yao. Quantum circuit complexity. In *Proceedings of 1993 IEEE 34th Annual Foundations of Computer Science*, pages 352–361, 1993. doi:10.1109/SFCS.1993.366852.

[YC22] Charles Yuan and Michael Carbin. Tower: data structures in quantum superposition. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):259–288, Oct 2022. doi:10.1145/3563297.

[Zal99] Christof Zalka. Grover's quantum searching algorithm is optimal. *Physical Review A*, 60(4):2746, 1999. doi:10.1103/PhysRevA.60.2746.

[ZLY22] Xiao-Ming Zhang, Tongyang Li, and Xiao Yuan. Quantum state preparation with optimal circuit depth: Implementations and applications. *Physical Review Letters*, 129(23):230504, 2022. doi:10.1103/PhysRevLett.129.230504.

## Résumé

L'informatique quantique est un paradigme émergent de calcul, dans lequel des phénomènes physiques quantiques tels que l'intrication et la superposition sont exploités pour obtenir un avantage sur le calcul classique. Bien que le programmeur quantique dispose d'un large choix de langages de programmation, aucun ne permet de garantir la faisabilité de ses programmes. À cette fin, nous introduisons un langage de programmation quantique du premier ordre (FOQ) qui permet de raisonner sur la réalisabilité physique et la complexité des programmes quantiques.

Nous introduisons des restrictions, vérifiables statiquement, sur les programmes FOQ qui nous permettent d'identifier des fragments cohérents et complets pour le temps polynomial quantique (PFOQ) et le temps polylogarithmique (LFOQ). Nous fournissons plusieurs exemples de programmes en temps polynomial et polylogarithmique capturés par ces fragments, représentant des fonctions quantiques pertinentes, telles que la transformée de Fourier quantique, l'arithmétique quantique, ainsi que des exemples comme la recherche dichotomique.

Nous introduisons également de nouvelles techniques de compilation permettant de traduire des programmes PFOQ et LFOQ en circuits de complexité raisonnable, évitant l'explosion exponentielle pouvant résulter de l'utilisation récursive d'une instruction de contrôle quantique. Nous améliorons encore cette technique de compilation et définissons un fragment de FOQ qui est cohérent et complet pour le temps polynomial quantique, où la complexité en circuit de l'instruction conditionnelle quantique correspond au maximum des branches, et non leur somme. Nous développons un compilateur prototype implémentant ces idées pour PFOQ et LFOQ.



**Mots-clés:** informatique quantique, complexité implicite, temps polynomial, temps polylogarithmique, techniques de compilation



## Abstract

Quantum computing is a paradigm of computation where quantum physical phenomena such as entanglement and superposition are used to obtain an advantage over classical computation. While the quantum programmer has a large choice of programming language at their disposal, none allow for ensuring the feasibility of their programs. To this end, we introduce a first-order quantum programming language (FOQ) that allows for reasoning about the physical realizability and complexity of its programs.

We introduce statically-checked restrictions over FOQ program that allow us to identify fragments that are sound and complete for quantum polynomial (PFOQ) and polylogarithmic time (LFOQ). We provide a number of examples of polynomial and polylogarithmic time programs that are captured by the fragments, and constitute relevant quantum functions, such as the quantum Fourier transform, quantum arithmetic, and examples like binary search.

We also introduce new compilation techniques that allow for translating PFOQ and LFOQ programs into circuits of adequate complexity, avoiding the exponential blow-up that can occur from the recursive use of a quantum control statement. We further improve this compilation technique and are able to define a FOQ fragment that is sound and complete for quantum polynomial time where the circuit complexity of the quantum case statement is the maximum of the branches, instead of their sum. We develop a prototype compiler that implementing these ideas over PFOQ and LFOQ programs.



**Keywords:** quantum computing, implicit complexity, polynomial time, polylogarithmic time, compilation techniques