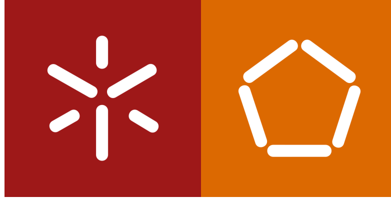
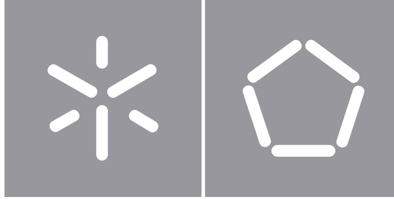


University of Minho
School of Engineering

Maria Inês Machado Correia Brioso Dias

An Interpreter for a Concurrent Quantum Language

january 2024



University of Minho
School of Engineering

Maria Inês Machado Correia Brioso Dias

An Interpreter for a Concurrent Quantum Language

Masters Dissertation
Master's in Engineering Physics

Physics of Information
Dissertation supervised by
Renato Neves
Luís Soares Barbosa

january 2024

Copyright and Terms of Use for Third Party Work

This dissertation reports on academic work that can be used by third parties as long as the internationally accepted standards and good practices are respected concerning copyright and related rights.

This work can thereafter be used under the terms established in the license below.

Readers needing authorization conditions not provided for in the indicated licensing should contact the author through the RepositóriUM of the University of Minho.

License granted to users of this work:

CC BY

https://creativecommons.org/licenses/by/4.0/

Acknowledgements

First of all, I would like to thank my supervisors, Prof. Renato Neves and Prof. Luís Soares Barbosa, for their guidance and all their availability to help, as well as for everything they have taught me. I would also like to thank my co-supervisor, Vítor Fernandes, for all the feedback, guidance and helpful suggestions provided throughout this process, and also for his promptness to help whenever needed.

I would like to thank my colleagues from Lab 2.17 of INESC TEC, next to whom I have spent a lot of time working on this project. I thank them for contributing to a great work environment and for the continuous encouragement. I particularly want to thank Rui Carvalho and Juliana Souza for their helpful advice.

I would like to thank INESC TEC for the research initiation grant attributed for developing this dissertation project, with reference 10107/BII-E_B4/2023; I also thank them for the pleasant and welcoming work conditions they have created.

Last but not least, I would like to express my gratitude to my family, for their valuable and continuous support, patience and advice. I would also like to thank my friends, for providing me with motivation, support and for making this journey much more enjoyable.

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020, DOI 10.54499/LA/P/0063/2020 | https://doi.org/10.54499/LA/P/0063/2020.

This work was financed by National Funds through FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference 10.54499/PTDC/CCI-COM/4280/2021.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, january 2024

Maria Inês Machado Correia Brioso Dias

Abstract

Despite all progress being made over the years in the Quantum Computing field, quantum noise remains a challenge for the realization of quantum computers. A possible way of minimizing the effect of noise in quantum computing is to reorder the instructions that are set for execution in a quantum computer. By introducing concurrency to quantum programs, together with an appropriate scheduler that decides the order of execution of these instructions, it is possible to realize this reordering. In this dissertation project, we implement in Haskell a concurrent quantum language. This task involved the implementation of a parser using Parsec and the implementation of the operational semantics of the language. The goal of this implementation is to study to what extent concurrency (and, specifically, reordering) can reduce noise in quantum computing. Specifically, this implementation allows to simulate the execution of programs of the language. For a given program and an initial state, it is possible to obtain all the possible final results of the execution, as well as an histogram that represents the results of several executions. Therefore this implementation is useful for evaluating if the introduction of concurrency in a program does not change its input-output behaviour.

Keywords Concurrent Quantum Language, Quantum Computing, Concurrent Computing, Programming Language Theory, Operational Semantics, Haskell, Parsec

Resumo

Apesar de todo o progresso a ser feito ao longo dos anos na área da Computação Quântica, o ruído quântico permanece um desafio para a concretização de computadores quânticos. Uma maneira possível de minimizar o efeito do ruído na computação quântica é reordernar as instruções que são dadas a um computador quântico para serem executadas. Através da introdução de concorrência nos programas quânticos, juntamente com um *scheduler* apropriado que decide a ordem de execução destas instruções, é possível realizar esta reordenação. Neste projeto de dissertação, implementamos em Haskell uma linguagem quântica concorrente. Esta tarefa involveu a implementação de um *parser* usando o Parsec a implementação a semântica operacional da linguagem. O objetivo desta implementação é estudar até que ponto a concorrência (e, especificamente, a reordenação) conseguem reduzir o ruído na computação quântica. Especificamente, esta implementação permite simular a execução de programas da linguagem. Dados um programa e um estado inicial, é possível obter todos os estados finais da execução possíveis, bem como um histograma que representa os resultados de várias execuções. Consequentemente, esta implementação é útil para avaliar se a introdução de concorrência num programa não modifica o seu *output* para um dado *input*.

Palavras-chave Linguagem Quântica Concorrente, Computação Quântica, Computação Concorrente, Teoria de Linguagens de Programação, Semântica Operational, Haskell, Parsec

Contents

I Introductory material 1

1 Introduction 2

1.1 Motivation and Context . . . 2

1.2 Contributions . . . 3

1.3 Document Structure . . . 4

2 Parsing Tools and Interpreters 5

2.1 An Overview . . . 5

2.2 The Tool Parsec . . . 6

3 Basics of Probabilistic Concurrency 9

3.1 A Basic Parallel Language and its Semantics . . . 9

3.2 Adding Probabilistic Choice operations into the mix . . . 14

4 Quantum Programming 22

4.1 Basic Notions of Quantum Computing . . . 22

4.2 A Concurrent Quantum Language . . . 29

II Implementation and Case Study 34

5 Parser 35

5.1 Basic Parallel Language . . . 35

5.2 Concurrent Quantum Language . . . 45

6 Semantics 52

6.1 Basic Parallel Language . . . 52

6.2 Basic Parallel Language with Probabilistic Choice 56
6.3 Concurrent Quantum Language 63

7 Examples and Case Study **74**
7.1 Examples 74
7.2 Case study: Quantum Teleportation 77

8 Conclusions and future work **81**
8.1 Conclusions 81
8.2 Future work 82

Appendices **88**

A User Manual **89**

B Minor implementation details **92**
B.1 Implementation of parsers 92
B.2 Implementation of the semantics 95

C Examples and Case-study **108**

D The Kronecker product **113**

List of Figures

1 Histogram with the results of executing command $H(q); (\texttt{Meas}(q) \to (\texttt{skip}, \texttt{skip})$ 10^5 times. Notice that the labels in the vertical axis of the histogram are in the range 49750 to 50250. 3

2 Transition rules for commands relative to the parallel language introduced in Brookes [1996]. 13

3 Transition rules for commands in the basic parallel language with probabilistic choice. . 16

4 Transition rules for commands relative to **CQL**. 32

5 Multiple results of bigStep applied to configuration $\langle (a := 0 \parallel a := 1) \, , \, [a = 2] \rangle$. . 63

6 Content of file cql1.txt. 74

7 Result of bigStepListFile applied to file cql1.txt, linking function l and state state0. 75

8 Result of (histBigStepFile 100000 "cql1.txt" l state0). Each result <conf x> in Figure 8a, with x being an integer, has a caption in Figure 8b, with the command and state (in matrix form) corresponding to the result. 75

9 Content of file cql2.txt. 76

10 Result of bigStepListFile applied to file cql2.txt, linking function l and state statePlus. 76

11 Result of (histBigStepFile 100000 "cql2.txt" l statePlus). Each result <conf x> in Figure 11a, with x being an integer, has a caption in Figure 11b, with the command and state (in matrix form) corresponding to the result. 77

12 Content of file qTelepSeq.txt. 77

13 Histogram plotted by (histBigStepFile 100000 "qTelepSeq.txt" lT qTelepInitState). Notice that the labels in the vertical axis of the histogram are in the range 24850 to 25200. Each result <conf x> in this histogram, with x being an integer, has a caption in Figure 20 of Appendix C, with the command and state (in matrix form) corresponding to the result. 78
14 Content of file qTelepAttempt.txt. 79
15 Histogram plotted by (histBigStepFile 100000 "qTelepAttempt.txt" lT qTelepInitState). Notice that the labels in the vertical axis of the histogram are in the range 0 to 30000. Each result <conf x> in this histogram, with x being an integer, has a caption, which is shown in Figures 21, 22 and 23 of Appendix C, with the command and state (in matrix form) corresponding to the result. 79
16 Result of bigStepListFile applied to file cql1.txt, linking function l and state state0. The content of this file is presented in Figure 17, l attributes integer 1 to variable q and state0 corresponds to state $|0\rangle$. 89
17 Content of file cql1.txt. 89
18 Result of (histBigStepFile 100000 "cql1.txt" l state0). Each result <conf x> in Figure 18a, with x being an integer, has a caption in Figure 18b, with the command and state (in matrix form) corresponding to the result. 91
19 Result of bigStepListFile applied to file qTelepSeq.txt, linking function lT and state qTelepInitState. 108
20 Caption produced by (histBigStepFile 100000 "qTelepSeq.txt" lT qTelepInitState), relative to the histogram in Figure 13. 109
21 Part 1 of the caption produced by (histBigStepFile 100000 "qTelepAttempt.txt" lT qTelepInitState), relative to the histogram in Figure 15. Parts 2 and 3 of this caption are in Figures 22 and 23, respectively. 110
22 Part 2 of the caption produced by (histBigStepFile 100000 "qTelepAttempt.txt" lT qTelepInitState), relative to the histogram in Figure 15. Parts 1 and 3 of this caption are in Figures 21 and 23, respectively. 111
23 Part 3 of the caption produced by (histBigStepFile 100000 "qTelepAttempt.txt" lT qTelepInitState), relative to the histogram in Figure 15. Parts 1 and 2 of this caption are in Figures 21 and 22, respectively. 112

Part I

Introductory material

Chapter 1

Introduction

1.1 Motivation and Context

Over the years, new advances have been revealing the promising nature of quantum computers Nielsen and Chuang [2010]. In particular, quantum algorithms have been proven capable of easily solving certain problems that are considered to be difficult to tackle by classical algorithms Preskill [2018]. For example, an efficient quantum algorithm was presented by Peter Shor for prime factorisation of integers, which is believed to be a difficult problem for classical computers Shor [1999].

Recent advances in quantum computing include the development of Google's *Sycamore* 54-qubit quantum processor Martinis and Boixo [2019]. More recently, in December 2023, IBM presented the new *IBM Quantum Heron*, deeming the device as the world highest-performing quantum processor IBM [2023a]. On the same day, IBM has revealed as well the *IBM Quantum System Two* quantum computer, already operating with three *IBM Heron* processors IBM [2023a].

A quantum computer relies on qubits, which correspond to quantum systems Nielsen and Chuang [2010]. Real quantum systems are not isolated from their surroundings, which leads them to have undesired interactions with their environment Nielsen and Chuang [2010]. Such interactions result in quantum noise in these computers, which has the capability of changing the state of qubits Nielsen and Chuang [2010].

Various quantum algorithms are formulated under the assumption that the states of qubits do not suffer unintentional changes. Thus quantum noise adds unreliability to their execution. Indeed, quantum noise is the greatest obstacle for quantum computers to reach their full potential Kim et al. [2023].

The present state of affairs was called in 2018 NISQ, which stands for *Noisy Intermediate- Scale Quantum* Preskill [2018]. NISQ computers are characterised by containing between 50 and a few hundred qubits, and by being subject to noise Preskill [2018].

One possible way of minimizing the effects of quantum noise is to reorder the list of instructions that

is set for execution in a quantum computer. The goal of this reordering is to reduce the amount of time during which a certain qubit is needed, in order to minimize the probability of noise affecting the result of quantum computations. In other words, its objective is to reduce the probability of using the state of qubits after their coherence time has passed.

Introducing concurrency into quantum programs, accompanied by an appropriate scheduler for deciding the order in which instructions are to be executed, allows for this reordering.

1.2 Contributions

The main contribution of this project is the implementation in Haskell of an interpreter for a concurrent quantum language (**CQL**), developed in Fernandes [2024]. Specifically we implemented a parser for **CQL** and its operational semantics, which instructs how a command of the language should run step-by-step. The implementation of this semantics allows to simulate the execution of a command of this language.

Another contribution is the implementation of a tool that runs a command of the language multiple times and then produces histograms documenting the obtained results. This brings automation into the analysis of **CQL** commands. In Figure 1, an example is presented of an output of that tool. In this example, the tool executes 10^5 times a program corresponding to the application of an Hadamard quantum gate to a certain qubit, followed by its measurement, and outputs an histogram in which each result corresponds to a different final state of the qubit. The results of the histogram show that, in these 10^5 executions, the frequency of each final state is close to 50%. This example is discussed in detail in Subsection 7.1.1.

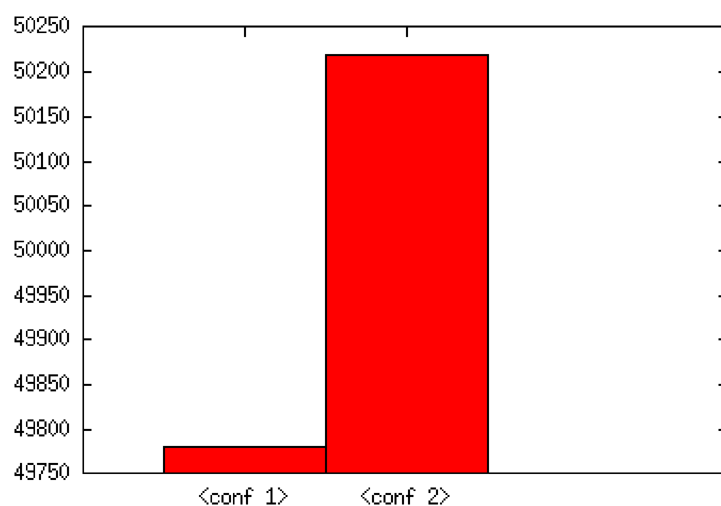


Figure 1: Histogram with the results of executing command $H(q); (\texttt{Meas}(q) \to (\texttt{skip}, \texttt{skip})$ 10^5 times. Notice that the labels in the vertical axis of the histogram are in the range 49750 to 50250.

Lastly, through the exploration of a case-study, we discuss how our implementation can be used in order to check if the introduction of concurrency in a program does not change its input-output behaviour.

The fact that this implementation allows for such verification is useful for some future work, which involves checking if the introduction of concurrency along with an appropriate scheduler does not change the intended output of programs.

1.3 Document Structure

The document is divided into two parts: Part I presents the introductory concepts related to this project, Part II details the implementation of the **CQL** intepreter. Further details are collected in the appendices.

More specifically, Part I is structured as follows: in Chapter 2 we introduce the concept of parsing and describe the tool Parsec, which we selected for developing a parser for **CQL**. In Chapter 3 we explain some concepts related to concurrent programming and present the parallel language (with no quantum features) in Brookes [1996], as well as some programming language theory concepts such as syntax and semantics. This language serves as a basis for **CQL** – the latter corresponds to an extension of the former by adding quantum features. Chapter 3 also discusses probabilistic choice in programming languages, and presents a language with this property. Chapter 4 focuses on quantum programming – it introduces some basic notions of quantum computing and presents **CQL**, the concurrent quantum language this project focuses on.

In Part II, Chapter 5 focuses on the implementations of two parsers – that of the parallel language proposed in Brookes [1996] and that of **CQL**. On the other hand, Chapter 6 focuses on the implementation of the operational semantics of several languages: the parallel language in Brookes [1996], the one discussed in Subsection 3.2.1 and **CQL**. Chapter 7 focuses on examples and a case study related with our implementation, and Chapter 8 discusses the conclusions of our project and possible future work.

Chapter 2

Parsing Tools and Interpreters

2.1 An Overview

A programming language is high-level if it is independent from the machine in which its programs are executed Fernández [2014]. Thus, **CQL** is considered to be a high-level programming language. The implementation of a high-level language allows to execute its programs and can be achieved by implementing an interpreter, or a compiler, or a combination of both Fernández [2014].

An interpreter corresponds to a program that reads and analyses a program of a high-level language and, if no error is detected, directly executes the program's instructions. Otherwise it outputs an error message Fernández [2014]. The first interpreter to be conceived for a high-level language was created in 1950, for a language called Short Code Kangas [2023].

A compiler, on the other hand, is a program that reads and translates a program in a certain language, called the source language, into another language, which corresponds to the target language, while reporting errors that may be found in the source program while translating it Aho et al. [2007]. The source language is commonly a high-level language, whereas the target language is usually machine language. Thus a compiler can transform a high-level program into an equivalent executable program Fernández [2014].

Implementing an interpreter instead of a compiler has both advantages and disadvantages. For example, the implementation of interpreters is not as difficult, and the error messages they provide are easier to understand Fernández [2014]. However, executable target machine code resulting from compilation is usually faster to execute than interpreted code Aho et al. [2007]. Since in the context of this project fastness is not a primary concern in executing programs of **CQL**, implementing an interpreter instead of a compiler serves our purposes adequately.

The lexical analyser of a language reads a program and identifies the corresponding sequence of tokens, which are the lexical ingredients of the programming language (identifiers and separators, such

as the semicolon, are examples of tokens). If the program contains characters that do not belong to the tokens of the language, the lexical analyser produces an error message. The identified sequence of tokens may then be sent to the parser Fernández [2014].

The syntax of a programming language defines how programs are written. The grammar of a language specifies, through a set of rules, the syntax of that language. The parser (also known as syntax analyser) of a language reads a sequence of tokens and, if that sequence corresponds to a syntactically correct program, elaborates the possible corresponding parse trees, which are representations of the syntactic structure of the sequence of tokens Fasold and Connor-Linton [2006]. Otherwise the parser outputs an error message. Thus parsing involves checking whether a given input is part of a certain language, *i.e.* whether it is in accordance with the underlying syntax, a verification that is usually done with the support of the grammar of the language Fernández [2014].

As stated in Section 1.2, our main contribution is the implementation of an interpreter for **CQL**, *i.e.* an interpreter for a simple concurrent quantum language. In order to achieve that, it is necessary to build a parser for evaluating whether a given input (*i.e.* program) is part of **CQL**, and, if so, to obtain its syntactic structure. A lexical analyser is also necessary for identifying the tokens received by the parser.

In the following section we present Parsec. Note that parsers built using Parsec can also act as lexical analysers, besides being syntax analysers as well O'Sullivan et al. [2008]. Indeed the parsers we develop using Parsec, including that of **CQL**, are also responsible for the lexical analysis.

2.2 The Tool Parsec

The description of Parsec provided in this section is based on Leijen et al. [2022], O'Sullivan et al. [2008].

Parsec is a monadic parser combinators Haskell library Leijen et al. [2022]. More concretely, it provides parser combinators based on monads. Here, the term *combinator* refers to the combinator pattern existing in Parsec, in which there are *combinators* that combine simpler parsers in order to form a more complex one [HaskellWiki, 2021, 2007]. An important example of such a combinator is <|>, which, intuitively, takes two parsers as argument and returns a new parser such that, when it receives an input, tries to consume it with the first parser and if the latter does not consume any then applies the second parser.

This combinator pattern offers an advantage when using Parsec: we can implement parsing systems in a modular way, recycling different more primitive parsers. For example, in the implementation of a parser of commands (which we describe in Section 5.1) one can implement a parser for each type of

command and then use <|> to generate a parser that handles all commands. It is important to mention as well that parsers built using Parsec are not only able to check whether a given input agrees with a specified syntax, but also, if they succeed, to return that input in the form of a value of a Haskell type (pre-determined by the programmer), which can be used in subsequent steps.

In this subsection's remainder we describe those structures of Parsec that are most relevant to our work.

GenParser

The parsers we implemented using Parsec have a type of the form `GenParser Char st a`, where `a` is their return type. Here, type `Char` as second argument means that the input type of these parsers is `String`.

ParseError

`ParseError` is a data type for outputting parse errors. A value of this type contains the source position of the error and a list of error messages. What follows is an example of a `Left ParseError` value:

```
Left "(unknown)" (line 1, column 5):
unexpected ';'
expecting end of input
```

The above value is returned by a parser of commands (function `parseInputC`, which is described in Subsection 5.1), when applied to the input `"a:=0;"`: an error is returned because the input is not a valid command, as it as an extra `';'` character. The position presented in the example above is that of said character. The error message indicates that the latter character is not expected by the parser.

Function `parse`

In a nutshell, `parse` is a function such that `parse p filePath input` is the result of applying parser `p` to `input` (`filePath` is only used in the error messages corresponding to `ParseError` values, and can even be the empty string) – if `p` fails in parsing the input (*i.e.* there is a parse error while `p` is trying to parse the input), the output will be `Left e`, where `e` is a parse error of type `ParseError`. If parser `p` succeeds, it will be `Right r`, where `r` is the result of the parsing of `input`. Thus the type of `parse p filePath input` is `Either ParseError a`, where `a` is the type of the value returned by `p` (for more information about Haskell's Prelude module's type `Either`, see its documentation).

In the above description of type `ParseError`, the example of a `Left ParseError` value we presented is equal to `parse parseC "(unknown)" "a:=0;"`, where `parseC` is a parser for commands that will be described in Subsection 5.1.

Function <|>

<|> is a function that receives two arguments, `p` and `q`, creating a parser `p <|> q` that first applies `p`. If it succeeds, `p <|> q` returns the value returned by `p`. If it fails, parser `p <|> q` will apply parser `q` and, similarly, if `q` succeeds, `p <|> q` returns the value returned by `q`. If none of them succeeds, parser `p <|> q` will fail. It is relevant to note that parser `q` will only be applied if parser `p` did not consume any of the input's content.

Function `try`

`try` is a function such that, given a parser `p`, the new parser `try p` essentially acts as `p`. The only difference is that, if `try p` fails while consuming an input, function `try` will revert this consumption. Therefore note that the new parser `try(p) <|> q`, where `p` and `q` are arbitrary parsers, will always try parser `q` if `try(p)` fails after consuming some input (this contrasts to `p <|> q` which only applies `q` in cases where no consumption of the input occurred). This is useful for obtaining a parser whose underlying grammar intuitively corresponds to the union of the underlying grammars of parsers `p` and `q`.

Chapter 3

Basics of Probabilistic Concurrency

It was not until the middle of the 1960s that concurrent programming started to be studied on a deeper level by computer scientists Hansen [2013]. The initial motivation for the development of concurrent programming was the goal of building reliable operating systems Hansen [2013]. Besides operating systems, there are other applications where concurrency is useful, such as database systems and industrial automation Schneider [2012]. User interfaces and distributed systems are also examples of such applications Sottile et al. [2009]. In fact, concurrency is an important topic across all areas of computation Sottile et al. [2009].

A program is qualified as concurrent when it contains subprograms that run in parallel – in other words the execution of these subprograms is not determined a priori Sakr and Gaber [2014]. This naturally leads to the notion of non-determinism in programming. When a program has non-determinism, its execution may follow possible different paths, given identical initial conditions Bustard [1990]. Typically the way of resolving non-determinism (*i.e.* of fixing an execution order) is via the notion of a scheduler: a piece of software that chooses which program to execute next based on a history of previous choices and the current state of the computer Segala [1995], Bustard [1990].

In this chapter we will present two languages that allow to write concurrent programs – one in Section 3.1 that does not include probabilistic choice, and another in Subsection 3.2.1 that does. The latter subsection discusses the interaction of concurrency with probabilistic behaviour.

3.1 A Basic Parallel Language and its Semantics

In this subsection we present the syntax and the operational semantics of the parallel language introduced in Brookes [1996].

This language involves four different sets: Ide, which is a set of identifiers; Exp, the set of integer expressions; $BExp$, the set of Boolean expressions; and Com, the set of commands which are seen as

the programs of the language. Let us also fix some notation: I ranges over Ide, E ranges over Exp, B ranges over $BExp$ and C ranges over Com. Identifiers can be seen as memory locations. These and integer expressions are interpreted as integers, while Boolean expressions are interpreted as one of the truth values.

Brookes [1996] proposed the following grammars for $BExp$ and Exp (the symbol ":=" is used to indicate that the symbol on its left can take any of the forms which appear separated by the symbol "|" on its right, since "|" represents alternative Winskel [1993]):

$$B ::= \texttt{true} \mid \texttt{false} \mid \neg B \mid B_1 \ \& \ B_2 \mid E_1 \leq E_2 \tag{3.1}$$

$$E ::= 0 \mid 1 \mid I \mid E_1 + E_2 \mid \texttt{if } B \texttt{ then } E_1 \texttt{ else } E_2 \tag{3.2}$$

The Boolean expressions with values `true` and `false` correspond to the respective truth values, value $\neg B$ corresponds to the negation of B, value B_1 & B_2 corresponds to a conjunction of B_1 and B_2 and value $E_1 \leq E_2$ corresponds to an inequality between expressions. Integer expressions with values 0 and 1 correspond to the respective integer values, value $E_1 + E_2$ corresponds to a sum of expressions, and value $\texttt{if } B \texttt{ then } E_1 \texttt{ else } E_2$ corresponds to a conditional expression.

Commands are then built according to the following syntactic rules:

$$C ::= \texttt{skip} \mid I := E \mid C_1; C_2 \mid \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \mid \texttt{while } B \texttt{ do } C \mid C_1 \parallel C_2 \tag{3.3}$$

Thus a command is either a `skip` (do-nothing command), an assignment $I := E$, a sequence of commands $C_1; C_2$, a conditional $\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2$, a while-loop $\texttt{while } B \texttt{ do } C$ or a parallel composition of two commands $C_1 \parallel C_2$. We explain how these commands are executed when detailing the semantics of the language.

The semantics of a programming language focuses on the meaning of its programs, instead of its form, as is the case with syntax. In this way the semantics of a language describes the effect of its programs when executed Fernández [2014].

The semantics of a programming language is usually considered to have three different facets: operational, denotational and axiomatic. Operational semantics describes the way that programs are executed, in order to transmit their meaning Winskel [1993]. It expresses the meaning of programs through computational steps Fernández [2014], Brookes [1996]. Denotational semantics, on the other hand, uses abstract mathematical concepts for defining the meaning of programs Winskel [1993]. Lastly, axiomatic semantics conveys the meaning of a program by establishing its properties, namely the constraints on its

variables, before and after its execution Fernández [2014]. While denotational semantics and axiomatic semantics present an advantage in terms of proving properties of programs, operational semantics is more useful regarding the implementation of programming languages, due to the fact that it defines which computational steps correspond to the execution of a program Fernández [2014]. Thus in this dissertation our focus is on operational semantics.

Operational semantics - preliminary concepts

In order to better understand the operational semantics of the language in Brookes [1996], it is useful to first present preliminary definitions and notation:

- n ranges over the set of non-negative integers, which is represented by N, while v ranges over the set of truth values, denoted by $V = \{tt, ff\}$.

- A state is a finite partial function that attributes integer values to identifiers. s ranges over the set S of states, and corresponds to $S = Ide \to_p N$. dom(s) is the domain of s. $[\, s \mid I = n\,]$ is the state s except that identifier I has stored the integer value n. The expression $[I_1 = n_1, \; \ldots, \; I_k = n_k]$ represents the state s such that $s\,(I_i) = n_i$ and s is undefined everywhere else.

- free$[\![E]\!]$ corresponds to the set of identifiers which occur free in E (and analogously for Boolean expressions and commands). The set of free identifiers in a command are defined in the following manner:

$$
\begin{aligned}
&\text{free}[\![\texttt{skip}]\!] = \{\} \\
&\text{free}[\![I := E]\!] = \{I\} \cup \text{free}[\![E]\!] \\
&\text{free}[\![C_1; C_2]\!] = \text{free}[\![C_1]\!] \cup \text{free}[\![C_2]\!] \\
&\text{free}[\![\texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2]\!] = \text{free}[\![B]\!] \cup \text{free}[\![C_1]\!] \cup \text{free}[\![C_2]\!] \\
&\text{free}[\![\texttt{while } B \texttt{ do } C]\!] = \text{free}[\![B]\!] \cup \text{free}[\![C]\!] \\
&\text{free}[\![C_1 \parallel C_2]\!] = \text{free}[\![C_1]\!] \cup \text{free}[\![C_2]\!].
\end{aligned}
\tag{3.4}
$$

- Brookes [1996] specifies three ingredients for defining the operational semantics of commands: a set of configurations $Conf$, given by $Conf = \{\langle C,\; s\rangle \in Com \times S \mid \text{free}[\![C]\!] \subseteq \text{dom}(s)\}$, which represent the computer's internal state; a transition relation $\to \subseteq Conf \times Conf$ that describes the possible internal state transitions; and a subset of successfully terminated configurations, which tell us which executions have terminated. Specifically a configuration of the form $\langle C,\; s\rangle$ corresponds

to a stage in a computation where the next command to be executed is C and s is the current state. A transition of the form $\langle C, s\rangle \rightarrow \langle C', s'\rangle$ represents a computational step that results in a new state s' and a new command C' to be executed.

- A transition $\langle C, s\rangle \rightarrow \langle C', s'\rangle$ is possible if and only if it follows from the application of the transition rules for commands presented in Figure 2.

- A configuration $\langle C, s\rangle$ is considered to be successfully terminated if such can be derived from those rules, with $\langle C, s\rangle term$ meaning that $\langle C, s\rangle$ is successfully terminated.

- A core concept in operational semantics is the distinction between big-step semantics and small-step semantics Hüttel [2010]. Transitions associated with a big-step semantics are from an initial configuration to a terminal configuration Hüttel [2010]. On the other hand, transitions associated with a small-step semantics correspond to only one computational step, and do not lead necessarily to a terminal configuration Hüttel [2010]. In our case the transition relation $\rightarrow$ corresponds to the small-step operational semantics, since it attributes to a certain configuration $\langle C, s\rangle$ a configuration $\langle C', s'\rangle$ obtained after one computational step. Such style of semantics fits very naturally in the concurrent paradigm Hüttel [2010] because of the need to account for external interferences that a command can be subjected to at every computational step.

- $\langle E, s\rangle \Downarrow n$ means that E evaluates to n in state s, and similarly $\langle B, s\rangle \Downarrow v$ means that B evaluates to v in state s.

Transition rules

Now we discuss the transition rules associated with the small-step operational semantics of the language presented in Brookes [1996]. We start with the transition rules for commands shown in Figure 2.

$$\frac{}{\langle \texttt{skip},\ s\rangle \mathit{term}} \qquad \frac{\langle E,\ s\rangle \Downarrow n}{\langle I := E,\ s\rangle \rightarrow \langle \texttt{skip},\ [\, s \mid I = n]\rangle}$$

$$\frac{\langle C_1,\ s\rangle \rightarrow \langle C_1',\ s'\rangle}{\langle C_1; C_2,\ s\rangle \rightarrow \langle C_1'; C_2,\ s'\rangle} \qquad \frac{\langle C_1,\ s\rangle \mathit{term}}{\langle C_1; C_2,\ s\rangle \rightarrow \langle C_2,\ s\rangle}$$

$$\frac{\langle B,\ s\rangle \Downarrow tt}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ s\rangle \rightarrow \langle C_1,\ s\rangle} \qquad \frac{\langle B,\ s\rangle \Downarrow ff}{\langle \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2,\ s\rangle \rightarrow \langle C_2,\ s\rangle}$$

$$\frac{}{\langle \texttt{while}\ B\ \texttt{do}\ C,\ s\rangle \rightarrow \langle \texttt{if}\ B\ \texttt{then}\ (C; \texttt{while}\ B\ \texttt{do}\ C)\ \texttt{else skip},\ s\rangle}$$

$$\frac{\langle C_1,\ s\rangle \rightarrow \langle C_1',\ s'\rangle}{\langle C_1 \| C_2,\ s\rangle \rightarrow \langle C_1' \| C_2,\ s'\rangle} \qquad \frac{\langle C_2,\ s\rangle \rightarrow \langle C_2',\ s'\rangle}{\langle C_1 \| C_2,\ s\rangle \rightarrow \langle C_1 \| C_2',\ s'\rangle} \qquad \frac{\langle C_1,\ s\rangle \mathit{term} \quad \langle C_2,\ s\rangle \mathit{term}}{\langle C_1 \| C_2,\ s\rangle \mathit{term}}$$

Figure 2: Transition rules for commands relative to the parallel language introduced in Brookes [1996].

$\langle \texttt{skip},\ s\rangle \mathit{term}$ simply shows that a configuration where `skip` is the next command to be executed is always successfully terminated.

The second transition rule expresses that if, in state s, E evaluates to n and $I := E$ is the next command to be executed, then the current state will become s, with the exception that it attributes n to identifier I, and the next command to be executed will become `skip`. Hence the computation terminates after this transition.

The third and fourth transition rules show that, in a given state s, when the next command to be executed is $C_1; C_2$, then C_2 will only be executed when $\langle C_1,\ s\rangle$ is a successfully terminated configuration.

The fifth and sixth transition rules illustrate that, in a given state s, if `if` B `then` C_1 `else` C_2 is the next command to be executed, then the next command to be executed will become C_1 if B evaluates to true in state s, or it will become C_2 if B evaluates to false in that state.

The seventh transition rule shows that if, in a given state s, the next command to be executed is `while` B `do` C, then the next command to be executed is `if` B `then` $(C;$ `while` B `do` $C)$ `else skip`.

The eighth and ninth rules illustrate the possibility of interleaving the execution steps of C_1 with those of C_2 when executing $C_1 \| C_2$ Brookes [1996]. On the other hand, the tenth transition rule constrains $\langle C_1 \| C_2,\ s\rangle$ to only be successfully terminated if $\langle C_1,\ s\rangle$ and $\langle C_2,\ s\rangle$ are so as well. Analysing the eighth and ninth rules, one can conclude that non-determinism is a property of programs corresponding to a

parallel composition, *i.e.*, commands of the form $C_1 \| C_2$, when neither one of their components (not C_1 nor C_2) have terminated. Note that commands $C_1 \parallel C_2$ and $C_2 \parallel C_1$ are equivalent Brookes [1996].

Now that the transition rules associated with the language have been discussed, we present an intuitive example of them at work. This will give the reader a general idea of how they describe a program's execution.

Example 3.1.1. Let us consider the initial configuration

$$\langle \texttt{if}\ (\neg(\texttt{true}\,\&\,\texttt{false}))\ \texttt{then}\ (x := 1; x := 0)\ \texttt{else skip},\ [x = 0]\rangle.$$

Following the rules from Figure 2, we have the following sequence of transitions:

$$\begin{aligned}
&\langle \texttt{if}\ (\neg(\texttt{true}\,\&\,\texttt{false}))\ \texttt{then}\ (x := 1; x := 0)\ \texttt{else skip},\ [x = 0]\rangle \\
&\quad \to \langle x := 1; x := 0,\ [x = 0]\rangle \\
&\quad \to \langle \texttt{skip}; x := 0,\ [x = 1]\rangle \\
&\quad \to \langle x := 0,\ [x = 1]\rangle \\
&\quad \to \langle \texttt{skip},\ [x = 0]\rangle term.
\end{aligned}$$

In words, there is a transition from the initial configuration to $\langle x := 1; x := 0,\ [x = 0]\rangle$ after evaluating $(\neg(\texttt{true}\,\&\,\texttt{false}))$ to true. Then, we assign the value 1 to x leading to $\langle \texttt{skip}\,; x := 0,\ [x = 1]\rangle$, which is reduced to $\langle x := 0,\ [x = 1]\rangle$ by the next transition, while the current state remains the same. Finally, the value 0 is assigned to x leading to $\langle \texttt{skip},\ [x = 0]\rangle term$, which is a successfully terminated configuration. Hence the computation finished successfully.

The reason for choosing this language and its semantics as a basis for **CQL** lies on the fact that it has been well studied, and it includes concurrent programs, just as expected for **CQL**. Clearly we will also need to bring probabilistic behaviour into the picture.

3.2 Adding Probabilistic Choice operations into the mix

As detailed in Chapter 4, some programs in **CQL**, namely those corresponding to the measurement of a qubit's state, have probabilistic behaviour, in the sense that their execution may yield different outputs, each with a certain probability. Before introducing the syntax and semantics of **CQL**, in the next subsection we discuss a parallel language that includes a probabilistic choice operator, and consequently gives rise to probabilistic behaviour. Our justification for not yet introducing **CQL** is that probabilistic behaviour is

interesting by itself and is completely independent of quantum theory. In other words we are progressively adding up conceptual ingredients to our programming language until naturally obtaining a language for quantum concurrency.

Another example of a language with probabilistic choice can be found in López and Núñez [2004], which presents the syntax and operational semantics of a basic language that allows to express the probabilistic choice between two probabilistic processes. The study of probabilistic models of computation is decades-old López and Núñez [2004], Baier and Hermanns [1999], Kozen [1981], and has been motivated by the goal of formalising probabilistic behaviour of both software and hardware systems Baier and Hermanns [1999]. According to López and Núñez [2004], the first publication on probabilistic automata is Rabin [1963]. Reference Kozen [1981] refers to probabilistic Turing machines as a model already being used in Gill [1974].

3.2.1 A Basic Parallel Language with Probabilistic Choice

We now discuss a basic parallel language with probabilistic choice Jones and Plotkin [1989], whose syntax and semantics were defined by the supervising team and are an extension of those proposed in Brookes [1996] (presented in Section 3.1). The former language corresponds to an extension of the latter by adding the possibility of writing programs that represent a probabilistic choice. This language involves the same four sets of elements: Ide, Exp, $BExp$ and Com. The syntax for Ide, Exp and $BExp$ remains the same, while the syntax for Com is now given by the following grammar:

$$C ::= \texttt{skip} \mid I := E \mid C_1; C_2 \mid \texttt{if}\ B\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{while}\ B\ \texttt{do}\ C \mid C_1 \parallel C_2 \mid C_1 \oplus_p C_2 \tag{3.5}$$

Note the addition of the program construct $C_1 \oplus_p C_2$ that runs C_1 with probability p or C_2 with probability $1-p$. We maintain the same notation as in the previous section, unless otherwise stated.

Small-step semantics

Before presenting the transition rules associated with the small-step semantics of this language, let us first introduce some relevant concepts. The set of all discrete probability distributions on a set X is given by

$$\mathsf{D}\,(X) = \left\{ \varphi : X \to [0,1] \mid \mathsf{supp}\,(\varphi)\ \text{ finite or countably infinite, } \sum_{x \in X} \varphi(x) = 1 \right\}, \tag{3.6}$$

where $\mathsf{supp}(\varphi) = \{x \in X \mid \varphi(x) > 0\}$ corresponds to the support of φ Sokolova and de Vink [2004]. A discrete probability distribution is also simply called a distribution Sokolova and de Vink [2004]. We will use a sum $\sum_{x \in X} \varphi(x)\, x$ for representing a distribution φ on X.

We now consider that the transition relation associated with the execution of commands is given by $\rightarrow \subseteq Conf \times \mathsf{D}(Conf)$. In words, the transitions associated with the small-step semantics are now of the form $\langle C,\ s\rangle \rightarrow \varphi$, with $\varphi \in \mathsf{D}(Conf)$, which dictates that a transition between configurations is now labeled by a probability.

The transition rules for commands, associated with the small-step semantics, can be found in Figure 3.

$$\langle \mathtt{skip},\ s\rangle term \qquad \frac{\langle E,\ s\rangle \Downarrow n}{\langle I := E,\ s\rangle \rightarrow 1 \cdot \langle \mathtt{skip},\ [s \,|\, I = n]\rangle}$$

$$\frac{\langle C_1,\ s\rangle \rightarrow \sum_i p_i \cdot \langle C_i,\ s_i\rangle}{\langle C_1; C_2,\ s\rangle \rightarrow \sum_i p_i \cdot \langle C_i; C_2,\ s_i\rangle} \qquad \frac{\langle C_1,\ s\rangle term}{\langle C_1; C_2,\ s\rangle \rightarrow 1 \cdot \langle C_2,\ s\rangle}$$

$$\langle C_1 \oplus_p C_2,\ s\rangle \rightarrow p \cdot \langle C_1,\ s\rangle + (1-p) \cdot \langle C_2,\ s\rangle$$

$$\frac{\langle B,\ s\rangle \Downarrow tt}{\langle \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2,\ s\rangle \rightarrow 1 \cdot \langle C_1,\ s\rangle} \qquad \frac{\langle B,\ s\rangle \Downarrow ff}{\langle \mathtt{if}\ B\ \mathtt{then}\ C_1\ \mathtt{else}\ C_2,\ s\rangle \rightarrow 1 \cdot \langle C_2,\ s\rangle}$$

$$\langle \mathtt{while}\ B\ \mathtt{do}\ C,\ s\rangle \rightarrow 1 \cdot \langle \mathtt{if}\ B\ \mathtt{then}\ (C; \mathtt{while}\ B\ \mathtt{do}\ C)\ \mathtt{else}\ \mathtt{skip},\ s\rangle$$

$$\frac{\langle C_1,\ s\rangle \rightarrow \sum_i p_i \cdot \langle C_i,\ s_i\rangle}{\langle C_1 \| C_2,\ s\rangle \rightarrow \sum_i p_i \cdot \langle C_i \| C_2,\ s_i\rangle} \qquad \frac{\langle C_2,\ s\rangle \rightarrow \sum_j p_j \cdot \langle C_j,\ s_j\rangle}{\langle C_1 \| C_2,\ s\rangle \rightarrow \sum_j p_j \cdot \langle C_1 \| C_j,\ s_j\rangle}$$

$$\frac{\langle C_1,\ s\rangle term \quad \langle C_2,\ s\rangle term}{\langle C_1 \| C_2,\ s\rangle term}$$

Figure 3: Transition rules for commands in the basic parallel language with probabilistic choice.

From the rules above, one can conclude that, when the new command $C_1 \oplus_p C_2$ is executed, there is a probability p of C_1 being executed and a probability $1-p$ of C_2 being executed. The behaviour of the other commands remains essentially the same as before, except for the fact that now each configuration leads to a distribution on configurations, after a computational step, rather than to a single configuration. Notice that these rules show that if $\langle C,\ s\rangle \rightarrow \varphi$ then φ has finite support.

It is important to notice that, since concurrent programs have non-determinism, there may be different

probability distributions that a given initial configuration can lead to on completion of one computational step. The following example illustrates the latter point.

Example 3.2.1. Consider the initial configuration $\langle a := 0 \parallel (a := 1 \oplus_{0.5} a := 1+1), [a=3]\rangle$. From the rules of Figure 3 the following two transitions can occur from this configuration – the first one is a consequence of first executing an atomic step of command $a := 0$, and the second one results from first executing an atomic step of command $(a := 1 \oplus_{0.5} a := 1+1)$:

$$\langle a := 0 \parallel (a := 1 \oplus_{0.5} a := 1+1), [a=3]\rangle \rightarrow$$
$$1 \cdot \langle \texttt{skip} \parallel (a := 1 \oplus_{0.5} a := 1+1), [a=0]\rangle$$
$$\langle a := 0 \parallel (a := 1 \oplus_{0.5} a := 1+1), [a=3]\rangle \rightarrow\ 0.5 \cdot \langle a := 0 \parallel a := 1, [a=3]\rangle$$
$$+\ 0.5 \cdot \langle a := 0 \parallel a := 1+1, [a=3]\rangle.$$

It is possible to represent more intuitively the computational steps that can be taken when executing a command of this language, in such a way that non-determinism and probabilistic choice are both represented. This is based on the definition of Segala probabilistic automata Sokolova and de Vink [2004], which we present next. According to this reference, this kind of automata was introduced by Segala and Lynch in references Segala and Lynch [1994], Segala [1995] and can be defined in the following manner (the following definition corresponds to a simplification of the original one from Segala and Lynch):

Definition 3.2.1. A Segala probabilistic automaton is a triple (S, A, α) where S is a set of states, A a set of actions, and $\alpha : S \rightarrow \mathsf{P}(\mathsf{D}(A \times S))$ a transition function and $\mathsf{P}(X)$ representing the powerset of set X Sokolova and de Vink [2004].

The transition function α of a Segala probabilistic automaton when receiving an initial state s can be represented by using a straight arrow from s for representing a transition to a distribution φ, with $\varphi \in \alpha(s)$. This is then sequenced with a squiggly arrow towards a state s' labeled by $a\,[p]$ for representing a transition to state s' with $\varphi(a, s') = p$ and $p > 0$. In this way, straight arrows represent the choice of a distribution while the squiggly ones represent probabilistic choice. Thus, multiple straight arrows can represent non-determinism, as Example 3.2.2 illustrates.

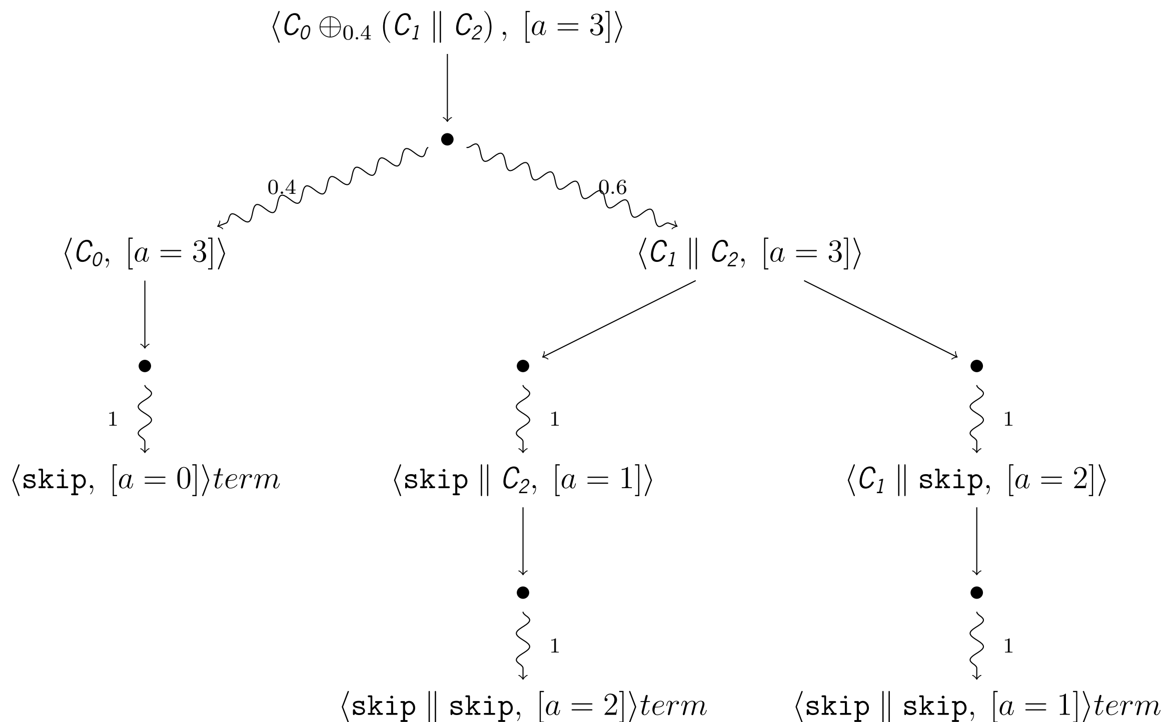
In Varacca and Winskel [2006], the authors adapt the definition of probabilistic automata presented in Segala [1995] and consider that a probabilistic automaton on a set S of states corresponds to the combination of an initial state $s_0 \in S$ with a function $k : S \rightarrow \mathsf{P}_f(\mathsf{D}(S))$, where $\mathsf{P}_f(X)$ is the finite powerset of X, which includes the empty set. This finite powerset is given by the set that contains only the finite subsets of the powerset of X Jacobs [2017]. Varacca and Winskel [2006] shows how these

probabilistic automata can be represented by using alternating trees where states are represented by black nodes, probability distributions are represented by hollow nodes and the edges from hollow to black nodes are only labeled by probabilities.

For representing the execution of a command *C* of the language given an initial state *s*, with $\langle C,\ s\rangle \in Conf$, we use a probabilistic automaton as defined by Varacca and Winskel [2006]. However, in order to display the transitions that may exist between configurations, we consider function k typed as $k : Conf \rightarrow \mathsf{P}_f(\mathsf{D}(Conf))$. In this way, the connection between the automaton that represents the execution of a command and the rules from Figure 3 becomes clearer.

In order to represent transitions in our probabilistic automata, we adapt the way suggested in Sokolova and de Vink [2004] for representing transitions in a Segala probabilistic automaton. Specifically we use straight and squiggly arrows for representing transitions to distributions and configurations, respectively. However, similarly to what happens in the alternating trees presented by Varacca and Winskel [2006], our squiggly arrows are only labeled by the probability of the corresponding transition, and we use nodes for representing distributions depicted in black. The following example shows how we represent transitions in our probabilistic automata.

Example 3.2.2. Below is the probabilistic automaton we use for representing the computation resulting from the initial configuration $\langle a := 0 \oplus_{0.4} (a := 1 \parallel a := 1+1)\ ,\ [a=3]\rangle$. Command C_0 corresponds to $a := 0$, command C_1 corresponds to $a := 1$ and command C_2 corresponds to $a := 1+1$.



In this example, the upper black node corresponds to distribution $0.4\langle C_0,\ [a=3]\rangle + 0.6\langle C_1 \| C_2,\ [a=3]\rangle$,

and there is a probability of 0.4 of C_0 being executed and a probability of 0.6 of $C_1 || C_2$ being executed, which is conveyed by the probabilities next to the squiggly arrows. The two straight arrows starting in configuration $\langle C_1 || C_2,\ [a = 3] \rangle$ illustrate the existence of non-determinism and the fact that $C_1 || C_2$ is a concurrent program. This probabilistic automaton shows that, if the initial configuration leads to $\langle C_0,\ [a = 3] \rangle$, which happens with a probability of 0.4, the computation ends up in configuration $\langle \texttt{skip},\ [a = 0] \rangle term$. Otherwise command $C_1 || C_2$ is executed. In this case, if C_1 is the first command to be executed, the computation ends up in configuration $\langle \texttt{skip} \mid\mid \texttt{skip},\ [a = 2] \rangle term$; on the other hand, if C_2 is executed first, the final configuration will be $\langle \texttt{skip} \mid\mid \texttt{skip},\ [a = 1] \rangle term$.

Big-step semantics

The big-step semantics developed by the supervising team is inspired by Segala [1995], Varacca [2003]. When explaining the implementation of the big-step semantics of the language in Section 6.2, we will make use of the following concepts presented by Varacca and Winskel [2006].

A finite path of a probabilistic automaton with a set of states S is a sequence $((s_0\ \varphi_1\ s_1\ \cdots\ \varphi_n)\ s_n)$ from set $(S \times \mathsf{D}(S))^* \times S$ with $\varphi_i(s_i) > 0$. We use $s_0\ \varphi_1\ s_1\ \cdots\ \varphi_n\ s_n$ as a simplified notation for representing this sequence. Notice that, since the set of states of the probabilistic automata associated with this language is $Conf$, a finite path of these automata is an element of set $(Conf \times \mathsf{D}(Conf))^* \times Conf$, *i.e.* it is a sequence that alternates configurations with distributions on configurations. The probability of a path r, denoted as $\Pi(r)$, corresponding to $s_0\ \varphi_1\ s_1\ \cdots\ \varphi_n\ s_n$ is defined by:

$$\Pi(r) = \prod_{1 \le i \le n} \varphi_i(s_i)\,. \tag{3.7}$$

Let $l(r)$ denote the last state of a path r, which in the context of this language represents a terminal configuration. Varacca and Winskel [2006] presents the notion of scheduler as a means for settling the uncertainty corresponding to non-determinism. A probabilistic scheduler for a probabilistic automaton with function $k : S \to \mathsf{P}_f(\mathsf{D}(S))$ can be defined as a partial function

$$\mathcal{S} : (S \times \mathsf{D}(S))^* \times S \to \mathsf{D}(\mathsf{D}(S))\,, \tag{3.8}$$

with $\mathrm{supp}(\mathcal{S}(r)) \subseteq k(l(r))$. In words, in the context of this language, $\mathcal{S}$ attributes to a path r a distribution on a set of distributions on configurations, in such a way that the support of that distribution is a subset of the set of distributions that $l(r)$ can transition to. Given the transition rules of Figure 3, each configuration cannot lead to more than two different distributions. Thus the support of a distribution attributed by a scheduler will only contain a maximum of two distributions on configurations.

The set of maximal paths of a probabilistic automaton, determined by a scheduler $\mathcal{S}$, is the set of paths $s_0\,\varphi_1\,s_1\,\cdots\,\varphi_n\,s_n$ for which $k\,(s_n) = \varnothing$ and φ_{i+1} is determined by applying $\mathcal{S}$ to path $s_0\,\varphi_1\,\cdots\,s_i$, for every $i < n$, with φ_{i+1} being included in the support of $\mathcal{S}(s_0\,\varphi_1\,\cdots\,s_i)$. We will denote this set by MP $(k, \mathcal{S})$. Paths in this set are, thus, those defined by a scheduler $\mathcal{S}$ that ends in a terminal configuration.

Based on the notation of Varacca and Winskel [2006], we denote by MP $(C, s, \mathcal{S})$ the set of maximal paths determined by scheduler $\mathcal{S}$ of the automaton associated with the initial configuration given by command C and state s, with $\mathcal{S}$ being a scheduler for that automaton.

Based on Varacca [2003], we define the probability $\Pi_{\mathcal{S}}\,(r)$ of a finite path r defined by a probabilistic scheduler $\mathcal{S}$ in the following manner:

$$\Pi_{\mathcal{S}}(s_0) = 1 \tag{3.9}$$

$$\Pi_{\mathcal{S}}(r\varphi_n s_n) = \Pi_{\mathcal{S}}(r) \cdot \mathcal{S}(r)(\varphi_n) \cdot \varphi_n(s_n). \tag{3.10}$$

The following example illustrates the meaning of some of these concepts.

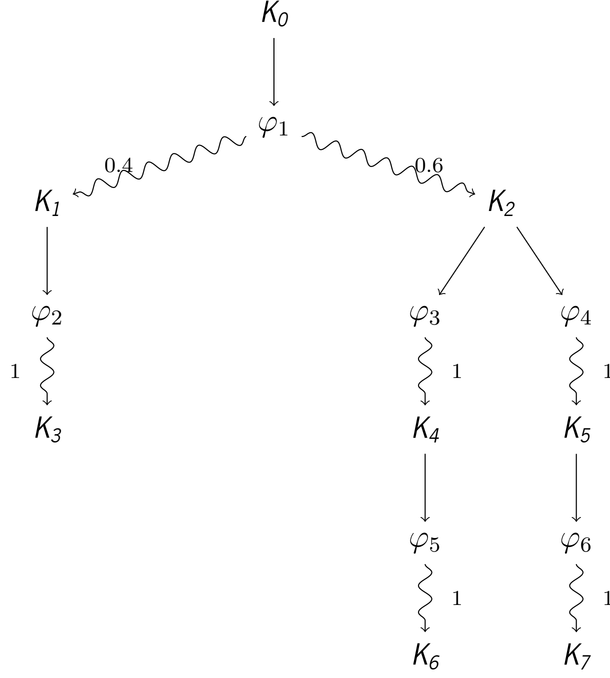
Example 3.2.3. Consider the initial configuration, $\langle a := 0 \oplus_{0.4} (a := 1 \parallel a := 1 + 1)\,,\ [a = 3] \rangle$, and the corresponding probabilistic automaton from Example 3.2.2. Let us represent this automaton replacing the black nodes by labels representing the corresponding distributions, as well as replacing configurations with the following corresponding labels, with C_0, C_1 and C_2 corresponding to commands $a := 0$, $a := 1$ and $a := 1 + 1$, respectively:

$K_0 = \langle C_0 \oplus_{0.4} (C_1 \parallel C_2)\,,\ [a = 3] \rangle$ $\qquad$ $K_1 = \langle C_0,\ [a = 3] \rangle$

$K_2 = \langle C_1 \parallel C_2,\ [a = 3] \rangle$ $\qquad$ $K_3 = \langle \texttt{skip},\ [a = 0] \rangle term$

$K_4 = \langle \texttt{skip} \parallel C_2,\ [a = 1] \rangle$ $\qquad$ $K_5 = \langle C_1 \parallel \texttt{skip},\ [a = 2] \rangle$

$K_6 = \langle \texttt{skip} \parallel \texttt{skip},\ [a = 2] \rangle term$ $\qquad$ $K_7 = \langle \texttt{skip} \parallel \texttt{skip},\ [a = 1] \rangle term$



The set of maximal paths of this automaton determined by a probabilistic scheduler $\mathcal{S}$ that allows the transition to every distribution in the automaton is:

$$\text{MP}\left(a := 0 \oplus_{0.4} \left(a := 1 \parallel a := 1+1\right), [a = 3], \mathcal{S}\right) = \{r_1, r_2, r_3\},$$

with

$$r_1 = K_0\, \varphi_1\, K_1\, \varphi_2\, K_3, \qquad r_2 = K_0\, \varphi_1\, K_2\, \varphi_3\, K_4\, \varphi_5\, K_6, \qquad r_3 = K_0\, \varphi_1\, K_2\, \varphi_4\, K_5\, \varphi_6\, K_7.$$

Therefore from Equations 3.9 and 3.10 we obtain the following probabilities for these maximal paths, with $\Pi_{\mathcal{S}}(K_0) = 1$, $\mathcal{S}(K_0)(\varphi_1) = 1$, $\varphi_1(K_1) = 0.4$, $\mathcal{S}(K_0\, \varphi_1\, K_1)(\varphi_2) = 1$ and $\varphi_2(K_3) = 1$. Notice that from the considerations above $\mathcal{S}(K_0\, \varphi_1\, K_2)(\varphi_3) = \mathcal{S}(K_0\, \varphi_1\, K_2)(\varphi_4) = 0.5$. $\Pi_{\mathcal{S}}(r_2)$ and $\Pi_{\mathcal{S}}(r_3)$ are calculated similarly to how $\Pi_{\mathcal{S}}(r_1)$ is calculated.

$$\begin{aligned}
\Pi_{\mathcal{S}}(r_1) &= \Pi_{\mathcal{S}}(K_0\, \varphi_1\, K_1\, \varphi_2\, K_3) \\
&= \Pi_{\mathcal{S}}(K_0) \cdot \mathcal{S}(K_0)(\varphi_1) \cdot \varphi_1(K_1) \cdot \mathcal{S}(K_0\, \varphi_1\, K_1)(\varphi_2) \cdot \varphi_2(K_3) \\
&= 1 \cdot 1 \cdot 0.4 \cdot 1 \cdot 1 = 0.4, \\
\Pi_{\mathcal{S}}(r_2) &= 1 \cdot 1 \cdot 0.6 \cdot 0.5 \cdot 1 \cdot 1 \cdot 1 = 0.3, \\
\Pi_{\mathcal{S}}(r_3) &= \Pi_{\mathcal{S}}(r_2) = 0.3.
\end{aligned}$$

The terminal configurations corresponding to these maximal paths are $l(r_1) = K_3$, $l(r_2) = K_6$ and $l(r_3) = K_7$.

Chapter 4

Quantum Programming

4.1 Basic Notions of Quantum Computing

In a nutshell, quantum computing is an area of computing science that focuses on the use of concepts of quantum mechanics to perform computations Som and Chakrabarti [2011], Karmakar et al. [2017]. One of the original contributors to the idea of using quantum mechanics in computing was Richard Feynman, who, in the early 1980's, questioned the ability of classical computers to simulate quantum systems McIntyre et al. [2012]. Feynman [1982] then proposed using computers based on quantum mechanical concepts for this task Nielsen and Chuang [2010]. Also in the 1980's, David Deutsch proposed a model of a quantum computer that led him to build some evidence suggesting a greater computational capacity of quantum computers, comparing to classical ones Nielsen and Chuang [2010]. In the 1990's, new advances supported this idea, including the formulation of some algorithms that demonstrated this greater computational power. These include the algorithm created by Peter Shor in 1994 for prime factorisation of integers, and the one created by Lov Grover in 1995 for searching in an unstructured space Nielsen and Chuang [2010].

4.1.1 Qubits and States

Quantum systems obey the postulates of quantum mechanics, which provide a mathematical description of them McIntyre et al. [2012]. A core notion of quantum computing is that of a qubit, which is an instance of a quantum system. Qubits, also known as quantum bits, play in quantum computing the role that bits do in classical computing, as they are responsible for storing information in a quantum computer McIntyre et al. [2012]. Although it is possible to treat qubits as mathematical entities, qubits exist in the form of physical systems Nielsen and Chuang [2010].

The first postulate of quantum mechanics establishes that the state of any quantum system is rep-

resented in mathematical terms by a normalised ket, which contains all the information that is possible to know about that system. Kets can be understood as vectors, known as quantum state vectors, whose dimensionality depends on the quantum system whose state they represent. A ket is a symbol belonging to the Dirac notation of quantum mechanics, developed by Paul A. M. Dirac. It has the form $|\cdots\rangle$, where $\cdots$ is not fixed and corresponds to a label. For any ket $|\psi\rangle$ there is another symbol $\langle\psi|$ belonging to the Dirac notation, also corresponding to a vector, called a bra McIntyre et al. [2012].

In matrix notation, each ket can be represented by a column vector, and any bra $\langle\psi|$ is represented by a row vector equal to the Hermitian conjugate of the column vector corresponding to $|\psi\rangle$. The Hermitian conjugate of a matrix A is obtained by complex conjugating its elements and transposing it, and we represent it as $A^\dagger$. Thus, for any bra $\langle\psi|$ represented by a matrix B,

$$B = K^\dagger, \tag{4.1}$$

with K being the matrix representing ket $|\psi\rangle$.

A ket $|\psi\rangle$ is normalised if it satisfies the following condition McIntyre et al. [2012]:

$$\langle\psi|\psi\rangle = 1. \tag{4.2}$$

By using the matrix notation of kets and bras, it is possible to calculate $\langle\psi|\psi\rangle$ using matrix multiplication.

The state of a qubit can be represented by a superposition state $|\psi\rangle$ that is expressed as a combination of states $|0\rangle$ and $|1\rangle$ in the following way, where the symbol $\doteq$ is used to express the matrix representation of the element at its left:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle \doteq \begin{pmatrix} a \\ b \end{pmatrix}, \tag{4.3}$$

with a and b complex numbers such that $|a|^2 + |b|^2 = 1$. The latter equation, $|a|^2 + |b|^2 = 1$, can be obtained from applying to $|\psi\rangle$ in Equation 4.3 the normalisation condition in Equation 4.2. States $|0\rangle$ and $|1\rangle$ are analogous to the two possible values that a bit can have (0 and 1). Notice that their matrix representation is the following:

$$|0\rangle \doteq \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle \doteq \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{4.4}$$

Notice that $|0\rangle$ and $|1\rangle$ form a basis for the vector space of kets representing a qubit, which means that any of these kets can be expressed as a linear combination of $|0\rangle$ and $|1\rangle$, and that these kets are linearly independent. This basis is called the computational basis, and $|0\rangle$ and $|1\rangle$ are called computational basis states.

The postulates of quantum mechanics also dictate that, if a given physical system is composed of x physical systems numbered from 1 to x, and the i-th component system is in state $|\psi_i\rangle$, then the state $|\psi\rangle$ of the composite system is given by:

$$|\psi\rangle = |\psi_1\rangle \otimes ... \otimes |\psi_n\rangle \,, \tag{4.5}$$

where $\otimes$ denotes the tensor product. It is possible to abbreviate the tensor product $|v\rangle \otimes |w\rangle$ as $|vw\rangle$. The tensor product of two vectors can be represented as the Kronecker product of the matrices corresponding to those vectors. For example, what follows is the matrix representation of $|01\rangle$:

$$|01\rangle \doteq \begin{pmatrix} 1 \\ 0 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} . \tag{4.6}$$

The definition of the Kronecker product is presented in Appendix D. The state $|\psi\rangle$ of a n-qubit system corresponds to a linear combination of 2^n states of the form $|x_1 x_2 \cdots x_n\rangle$, with each x_i corresponding to either 0 or 1, *i.e.*:

$$|\psi\rangle = \sum_{i=1}^{2^n} a_i \, |x_1 x_2 \cdots x_n\rangle_i \,. \tag{4.7}$$

Thus, the state of such a system is represented by a column vector with 2^n elements. The probability of measuring the system of n-qubits to be in the state $|x_1 x_2 \cdots x_n\rangle_i$ is given by $|a_i|^2$ Barnett [2009], with a_i being a complex number and $|x_j\rangle$ corresponding to the state of the j-th qubit.

A state of a composite system is considered an entangled state when it cannot be expressed as a tensor product of states of the systems that constitute it. The four Bell states are examples of 2-qubit entangled states. They are given by:

$$|\beta_{00}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}, \tag{4.8}$$

$$|\beta_{01}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}}, \tag{4.9}$$

$$|\beta_{10}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}}, \tag{4.10}$$

$$|\beta_{11}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}}. \tag{4.11}$$

4.1.2 Quantum Gates

Besides qubits, quantum computers also need quantum gates for performing operations on the information stored in qubits. 1-qubit quantum gates operate on the state of a qubit in such a way that the matrix A

representing its initial state and the matrix A' representing its final state after the action of the gate are related in the following manner:

$$A' = UA, \tag{4.12}$$

where U is a unitary 2×2 transformation matrix representing the effect of the gate. A unitary matrix U is one such that $U^{\dagger}U = I$, where I is the identity matrix McIntyre et al. [2012]. Notice that a matrix represents a quantum gate if and only if it is unitary. Some examples of 1-qubit quantum gates include gates *X*, *Y*, and *Z*, which are represented by the Pauli matrices σ_x, σ_y and σ_z, respectively,

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{4.13}$$

Gate X is the *NOT* gate, transforming a ket $|0\rangle$ into a ket $|1\rangle$, and vice versa. The Hadamard gate is another example of a 1-qubit gate, given by the following matrix:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{4.14}$$

Notice that this gate transforms kets $|0\rangle$ and $|1\rangle$ into the superposition states $|+\rangle$ and $|-\rangle$,

$$|+\rangle = \frac{1}{\sqrt{2}} \left(|0\rangle + |1\rangle\right), \tag{4.15}$$

$$|-\rangle = \frac{1}{\sqrt{2}} \left(|0\rangle - |1\rangle\right). \tag{4.16}$$

Lastly, another example of a 1-qubit quantum gate is the identity gate, which is represented by the identity 2×2 matrix:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}. \tag{4.17}$$

This gate is characterised by not modifying the state of the qubit it operates on. We are going to focus on the matrix representation of gates, as such an approach is helpful for the implementation of **CQL**.

A $m \times n$ matrix A is considered a linear operator that acts on vectors in $\mathbb{C}^n$, attributing to each vector $|v\rangle$ in this vector space a vector $|w\rangle$ in $\mathbb{C}^m$ through matrix multiplication $A\,|v\rangle = |w\rangle$. From this point of the chapter on, we consider that kets can be interpreted as the respective column vectors, when required by the context where they occur. Let A be a $2^m \times 2^m$ matrix acting on kets representing the state of a system of m qubits, and let B be a $2^n \times 2^n$ matrix acting on those representing the state of a system of n qubits. Thus we conclude that $(A \otimes B)$ is a linear operator such that

$$(A \otimes B)\left(|v\rangle \otimes |w\rangle\right) = A\,|v\rangle \otimes B\,|w\rangle, \tag{4.18}$$

where $|v\rangle$ is a ket representing the state of a system of m qubits, $|w\rangle$ is a ket representing that of a system of n qubits. Notice that the tensor product of two matrices corresponds to their Kronecker product.

Consider a system with n qubits whose state $|\psi\rangle$ is given by Equation 4.7. What follows is the state $|\psi'\rangle$ resulting from applying a 2×2 matrix U_i to the state of the i-th qubit of that system, with $|x_1 x_2 \cdots x_n\rangle_i = |x_1\rangle_i \otimes |x_2\rangle_i \otimes \cdots \otimes |x_n\rangle_i$:

$$|\psi'\rangle = \sum_{i=1}^{2^n} a_i U_1 |x_1\rangle_i \otimes U_2 |x_2\rangle_i \otimes \cdots \otimes U_n |x_n\rangle_i \,. \tag{4.19}$$

Using Equation 4.18, $|\psi'\rangle$ in the above equation can be rewritten as:

$$\begin{aligned} |\psi'\rangle &= (U_1 \otimes U_2 \otimes \cdots \otimes U_n) \left(\sum_{i=1}^{2^n} a_i |x_1 x_2 \cdots x_n\rangle_i \right) \\ &= (U_1 \otimes U_2 \otimes \cdots \otimes U_n) |\psi\rangle \,. \end{aligned} \tag{4.20}$$

Therefore, in order to obtain the state $|\psi'\rangle$ after applying a 1-qubit gate to specific qubits, the above equation can be used, where U_i is the matrix corresponding to the gate being applied to qubit number i.

Besides 1-qubit quantum gates, quantum computers can also use 2-qubit quantum gates. One example of a 2-qubit gate is the *CNOT* gate. It takes as input the states of two qubits, called the control qubit and the target qubit. The state of the control qubit, which remains unchanged, determines the effect of the *CNOT* gate on the state of the target qubit. If the former qubit is in state $|0\rangle$, the state of the latter does not change. However, if the state of the former is $|1\rangle$, a *NOT* gate is applied to the state of the latter. Thus, if we consider that the input state of a *CNOT* gate is $|ct\rangle$, with $|c\rangle$ being the state of the control qubit and $|t\rangle$ the state of the target one, this gate leaves states $|00\rangle$ and $|01\rangle$ unchanged, while turning state $|10\rangle$ into $|11\rangle$ and vice versa. The *CNOT* gate can be represented by the matrix

$$U_{\mathit{CNOT}} = A_0 \otimes I + A_1 \otimes \sigma_x, \tag{4.21}$$

where A_0 and A_1 are matrices $|0\rangle\langle 0|$ and $|1\rangle\langle 1|$, respectively. Notice that, in the above equation, $|0\rangle\langle 0|$ and $|1\rangle\langle 1|$ are applied to the control qubit and I and σ_x are applied to the target one Barnett [2009]. Remembering Equation 4.20, if the initial state of a given system of n qubits is $|\psi\rangle$, then its final state after applying a *CNOT* gate to the i-th qubit, the control one, and to the j-th qubit, the target one, with $i < j$, is given by:

$$|\psi'\rangle = \left(I^{\otimes(i-1)} \otimes A_0 \otimes I^{\otimes(n-i)} + I^{\otimes(i-1)} \otimes A_1 \otimes I^{\otimes(j-i-1)} \otimes \sigma_x \otimes I^{\otimes n-j} \right) |\psi\rangle \,, \tag{4.22}$$

where $I^{\otimes n}$ represents the tensor product of n elements equal to I. Another example of a 2-qubit gate is the *CZ* gate, which also receives the states of a control qubit and of a target one as input. The only

difference between the *CZ* gate and the *CNOT* gate is that the former applies gate *Z* to the target qubit if the control one is in state $|1\rangle$, instead of applying the *NOT* gate. The *CZ* gate can be represented by the following matrix:

$$U_{CZ} = I \otimes I - 2A_1 \otimes A_1. \tag{4.23}$$

Notice that it is irrelevant whether the first matrix in each of the above tensor products acts on the target or the control qubits, as long as the second matrix acts on the other qubit Barnett [2009].

Notice that an operator is a mathematical entity that transforms a given ket into another ket, and thus matrices corresponding to gates are considered operators.

4.1.3 Quantum Measurements

When measuring the state of a qubit, it is not possible to determine its superposition state. Instead the only possible results from such measurement are states $|0\rangle$ and $|1\rangle$, when measuring in the computational basis, even though the qubit can be in a superposition of these states. When the state of a qubit is measured its state collapses to the one it was measured in.

The postulates of quantum mechanics also establish what happens when performing a measurement on a quantum system in general form. The description of quantum measurements relies on measurement operators, which operate on the state of the system being measured. Let M_m be a measurement operator relative to the measurement outcome m, which represents the state in which the system is measured. The postulates of quantum mechanics dictate that, when the initial state of the system just before the measurement is $|\psi\rangle$, the probability of obtaining outcome m when measuring the state of the system is given by:

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle . \tag{4.24}$$

The state of said system after performing this measurement is the following:

$$|\psi'\rangle = \frac{M_m |\psi\rangle}{\sqrt{\langle\psi|M_m^\dagger M_m|\psi\rangle}} = \frac{M_m |\psi\rangle}{\sqrt{p(m)}}. \tag{4.25}$$

Notice that, given an operator A, the matrix representing the Hermitian conjugate $A^\dagger$ of A is equal to Hermitian conjugate of the matrix representing A. Given two matrices A and B, $(AB)^\dagger = B^\dagger A^\dagger$. Let A_m be the matrix representing M_m. Thus Equations 4.24 and 4.25 can be rewritten in matrix notation as:

$$p(m) = \langle\psi|A_m^\dagger A_m|\psi\rangle = (A_m |\psi\rangle)^\dagger A_m |\psi\rangle , \tag{4.26}$$

$$|\psi'\rangle = \frac{A_m |\psi\rangle}{\sqrt{p(m)}}, \tag{4.27}$$

where we are interpreting the kets and bras as their respective matrix representations.

Let us consider the case where the measurement of the state $a|0\rangle + b|1\rangle$ of a qubit is performed in the computational basis. The measurement operator associated with obtaining result $|0\rangle$ is $M_0 = |0\rangle\langle 0|$ and the one associated with obtaining result $|1\rangle$ is $M_1 = |1\rangle\langle 1|$. Using Equations 4.26 and 4.27 we verify that the probability of measuring the qubit in state $|0\rangle$ is

$$p(0) = \begin{pmatrix} a^* & b^* \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}^\dagger \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = |a|^2, \tag{4.28}$$

and that the state of the qubit after this measurement is

$$|\psi'\rangle = \frac{1}{|a|} \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \frac{a}{|a|} \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \tag{4.29}$$

which is the matrix representation of $\frac{a}{|a|}|0\rangle$. Notice that multipliers with modulus one, such as $\frac{a}{|a|}$, can be ignored in kets, since they do not alter the properties that can be observed in the corresponding physical system. Therefore $|\psi'\rangle$ in Equation 4.29 can be understood as $|0\rangle$. The probability of measuring the qubit in state $|1\rangle$ and the state that results from this measurement can be obtained in an analogous way.

Suppose we perform a measurement on the i-th qubit of a system of n qubits, in the computational basis, with $|\psi\rangle$ being the initial state of that system. The measurement operator associated with measuring this qubit must leave the states of the other qubits unchanged. Thus the measurement operator associated with measuring the i-th qubit in state $|b\rangle$, with $b \in \{0, 1\}$, is represented by the following matrix:

$$A_{b,i,n} = I^{\otimes(i-1)} \otimes A_b \otimes I^{\otimes(n-i)}, \tag{4.30}$$

where A_b is the matrix representing M_b. Thus considering Equations 4.26 and 4.27, the probability of measuring the i-th qubit in state $|b\rangle$ and the state $|\psi'(b,i,n)\rangle$ of the system after this measurement are, respectively:

$$p(b,i,n) = (A_{b,i,n}|\psi\rangle)^\dagger A_{b,i,n}|\psi\rangle, \tag{4.31}$$

$$|\psi'(b,i,n)\rangle = \frac{A_{b,i,n}|\psi\rangle}{\sqrt{p(b,i,n)}}. \tag{4.32}$$

4.1.4 Quantum Teleportation

We now present a summarised description of a technique that employs quantum computing, which is called quantum teleportation. This description is based on the more detailed one presented in Nielsen and Chuang [2010].

Suppose two entities, called Alice and Bob, share a previously prepared pair of qubits in a Bell state; *i.e.* each of them possesses one of the qubits. Suppose also that Alice and Bob are separated, and Alice has the mission of sending Bob an unknown state $|\psi\rangle$ of another qubit she possesses, while only being able to send bits to Bob. We consider that $|\psi\rangle$ has the following generic form:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle\,. \tag{4.33}$$

Let us consider the following state as the initial state of the system of three qubits in question – two from Alice and one from Bob:

$$|\psi\rangle_{in} = |\psi\rangle \otimes |\beta_{00}\rangle\,, \tag{4.34}$$

with $|\beta_{00}\rangle$ being the Bell state defined in Equation 4.8. Let us establish that the first qubit is the one whose state she wants to communicate to Bob, the second qubit is Alice's and is the one belonging to the pair shared with Bob; the third qubit is owned by Bob.

Alice starts by applying a *CNOT* gate to her qubits, with the control qubit being the first one and the target qubit being the second one. Alice then proceeds to apply an Hadamard gate to the first qubit. At this point, the state of the system of three qubits is:

$$\begin{aligned} |\varphi\rangle = \frac{1}{2}\,[&|00\rangle \otimes (a\,|0\rangle + b\,|1\rangle) + |01\rangle \otimes (a\,|1\rangle + b\,|0\rangle) \\ &+ |10\rangle \otimes (a\,|0\rangle - b\,|1\rangle) + |11\rangle \otimes (a\,|1\rangle - b\,|0\rangle)]\,. \end{aligned} \tag{4.35}$$

Next, Alice measures the state of her two qubits, and the corresponding result can either be $|00\rangle$, $|01\rangle$, $|10\rangle$ or $|11\rangle$; she then sends to Bob two bits representing the result. Depending on this result, Bob will proceed in a specific manner. If Alice has measured the second qubit in state $|1\rangle$, Bob applies an *X* gate to his qubit. After that, if Alice has measured the first qubit in state $|1\rangle$, Bob applies a *Z* gate to his qubit. In this way, for example, if Alice obtains result $|11\rangle$, then the state of Bob's qubit becomes $a\,|1\rangle - b\,|0\rangle$. If an *X* gate is applied to this qubit, followed by a *Z* gate, its state becomes $|\psi\rangle$.

By the end of this procedure, the state of Bob's qubit finally becomes $|\psi\rangle$, and the goal of quantum teleportation is achieved.

4.2 A Concurrent Quantum Language

We now present the language that this project focuses on: **CQL**. The name **CQL** stands for Concurrent Quantum Language. Its syntax and semantics were defined by the supervising team Fernandes [2024]

and are based on those of the language proposed in Brookes [1996], which was presented in Section 3.1, and on Ying [2016]. As previously mentioned, the former corresponds to an extension of the latter by adding the basic quantum features. Namely the states of programs become states of quantum systems, and the language allows to apply quantum gates and perform quantum measurements on those systems.

We consider that the execution of our language is based on the QRAM architecture, where a classical processor and a quantum one establish a collaboration in which the former is the master and the latter is the slave Lanzagorta and Uhlmann [2022]. More specifically, the classical processor sends quantum instructions to the quantum processor, so that the latter can execute them. We are only considering one quantum processor for executing quantum instructions.

This language involves two different sets: the set of commands, Com, and that of quantum variables, $QVar$. The syntax of commands is defined by the following grammar:

$$C ::= \texttt{skip} \mid U(\widetilde{q}) \mid C_1; C_2 \mid C_1 \parallel C_2 \mid \texttt{Meas}(q) \rightarrow (C_1, C_2) \mid \texttt{while Meas}\,(q) \rightarrow C \tag{4.36}$$

The commands $\texttt{skip}$, $C_1; C_2$ and $C_1 \parallel C_2$ have the same meaning as in the languages previously presented. U stands for a quantum gate. In the context of this project, we establish it can be one of the following: the Hadamard gate, the identity gate, gates X, Y and Z, the *CNOT* gate and the *CZ* gate. Thus $U(\widetilde{q})$ represents the application of gate U to the list of qubits $\widetilde{q}$. Notice that we only allow list $\widetilde{q}$ to contain two qubits, since the allowed quantum gates are act either on one or two qubits, and that we only allow $\widetilde{q}$ to have two qubits when U is a 2-qubit gate. In order to refer to qubits, we use quantum variables, such as $q1$ or x. We use H and I for representing the Hadamard gate and the identity gate, respectively. The remaining gates are represented by their own name (*e.g.* X represents the X gate). We will term a $U(\widetilde{q})$ command as a gate command.

The $\texttt{Meas}(q) \rightarrow (C_1, C_2)$ command, which we will term as the measurement command, represents the conditional execution of commands, depending on the result of a quantum measurement performed in the computational basis. More specifically, when this command is executed, a measurement of the state of qubit q is performed. If the result of such measurement is $|0\rangle$, command C_1 is executed; otherwise command C_2 is executed.

Lastly, the $\texttt{while Meas}\,(q) \rightarrow C$ command represents a while loop where command C is executed if the result of measuring the state of qubit q is $|1\rangle$. If this result is $|0\rangle$ the loop is interrupted. We will term this command as the while command.

Selinger and Valiron [2008] presents the notion of a linking function as a bijective function that attributes to a variable an integer number in set $\{0, \cdots, n-1\}$, with n being the number of variables. We slightly adapt this concept and consider that a linking function is a bijective function that sends quantum

variables to integers in set $\{1, \cdots, n\}$, with n being the number of quantum variables. This is useful in order to assign a qubit variable to a location in the memory.

We now consider that configurations $\langle C,\ s\rangle$ are composed of a command C and the state s of a n-qubit system, which can be represented by a column vector with 2^n complex numbers.

4.2.1 Operational Semantics

We now present the small-step and big-step operational semantics of the language. The operational semantics of **CQL** is similar to that of the parallel language with probabilistic choice presented in Subsection 3.2.1, with both languages involving probabilistic behaviour and non-determinism. As such, many of the concepts introduced in Subsection 3.2.1 are useful for describing the operational semantics of **CQL**. We use the same notation as in Subsection 3.2.1 unless stated otherwise.

Small-step semantics

Just like previously done for the language in Subsection 3.2.1, we consider that the transition relation associated with the execution of commands is given by $\rightarrow \subseteq Conf \times \mathsf{D}\left(Conf\right)$ and the transitions associated with the small-step semantics have the form $\langle C,\ s\rangle \rightarrow \varphi$, with $\varphi \in \mathsf{D}\left(Conf\right)$, *i.e.* an initial configuration always leads to a probability distribution on a set of configurations. The transition rules for commands associated with this semantics are presented in Figure 4.

The transition rules for `skip` commands, sequences of commands and parallel compositions remain the same as in Figure 3.

In the transition rule for gate commands, $U\left(\tilde{q}\right)\left(s\right)$ corresponds to the state that results from applying gate U to qubits $\tilde{q}$, when the initial state is s. Thus $U\left(\tilde{q}\right)\left(s\right)$ can be calculated using Equation 4.20, when U is a 1-qubit gate, or using Equation 4.22, or an analogous one, when U is the *CNOT* or the *CZ* gate. In order to know the locations of the qubits in $\tilde{q}$, we use the aforementioned linking function.

In the rule corresponding to this command, $p^s_{b,q}$, with $b \in \{0, 1\}$, is the probability of measuring qubit q in state $|b\rangle$ when the initial state is s. Notice that the measurement command is in some sense analogous to the probabilistic choice command $C_1 \oplus_p C_2$, of the language of Subsection 3.2.1.

Regarding the while command, notice that its transition rule is analogous to that of the while command of the language presented in Subsection 3.2.1. In the case of **CQL**, the measurement command plays the role of the if command of that language.

We represent schematically the computational steps that can take place when executing a command of the language in the same manner that was described for doing so in the context of the previous language

$$\langle \texttt{skip},\ s\rangle term \qquad \frac{\langle C_1,\ s\rangle \to \sum_i p_i \cdot \langle C_i,\ s_i\rangle}{\langle C_1; C_2,\ s\rangle \to \sum_i p_i \cdot \langle C_i; C_2,\ s_i\rangle} \qquad \frac{\langle C_1,\ s\rangle term}{\langle C_1; C_2,\ s\rangle \to 1 \cdot \langle C_2,\ s\rangle}$$

$$\frac{\langle C_1,\ s\rangle \to \sum_i p_i \cdot \langle C_i,\ s_i\rangle}{\langle C_1 \| C_2,\ s\rangle \to \sum_i p_i \cdot \langle C_i \| C_2,\ s_i\rangle} \qquad \frac{\langle C_2,\ s\rangle \to \sum_j p_j \cdot \langle C_j,\ s_j\rangle}{\langle C_1 \| C_2,\ s\rangle \to \sum_j p_j \cdot \langle C_1 \| C_j,\ s_j\rangle}$$

$$\frac{\langle C_1,\ s\rangle term \quad \langle C_2,\ s\rangle term}{\langle C_1 \| C_2,\ s\rangle term}$$

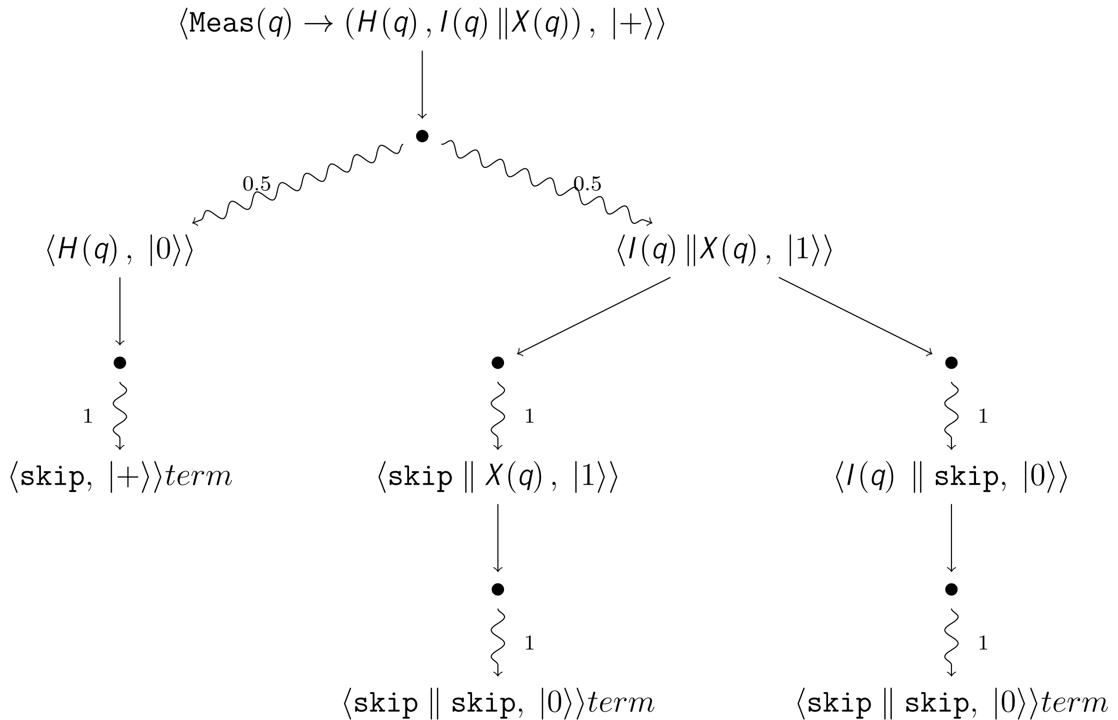
$$\langle U(\tilde{q}),\ s\rangle \to 1 \cdot \langle \texttt{skip},\ U(\tilde{q})(s)\rangle \qquad \langle \texttt{Meas}(q) \to (C_1, C_2),\ s\rangle \to p^s_{0,q} \cdot \langle C_1,\ s_{0,q}\rangle + p^s_{1,q} \cdot \langle C_2,\ s_{1,q}\rangle$$

$$\langle \texttt{while Meas}(q) \to C,\ s\rangle \to 1 \cdot \langle \texttt{Meas}(q) \to (\texttt{skip}, C; \texttt{while Meas}(q) \to C),\ s\rangle$$

Figure 4: Transition rules for commands relative to **CQL**.

(Subsection 3.2.1). The following example illustrates how to do so for a command of **CQL**.

Example 4.2.1. Below is the probabilistic automaton we use for representing the computation that results from the initial configuration $\langle \texttt{Meas}(q) \to (H(q), I(q) \| X(q)),\ |+\rangle\rangle$, where $|+\rangle$ is the state of a one-qubit system.



The above example can be summed up in words as follows. There is a probability of 0.5 of measuring

a qubit q in state $|+\rangle$ and obtaining $|0\rangle$ as a result, equal to the probability of obtaining $|1\rangle$ as a result. In the first case the resulting state $|0\rangle$ is fed to an Hadamard gate and we obtain $|+\rangle$ as final result. In the second case the resulting state $|1\rangle$ is fed to $I(q) \| X(q)$, which will always return $|0\rangle$.

Note that, just as in the case of the language presented in Subsection 3.2.1, the big-step semantics of **CQL** developed by the supervising team is inspired by Segala [1995], Varacca [2003]. The same concepts related to the big-step semantics that are explained in that subsection will be useful for reasoning about the implementation of the big-step semantics of **CQL**.

Part II

Implementation and Case Study

Chapter 5

Parser

5.1 Basic Parallel Language

Let us now describe the implementation of the parser for the basic parallel language introduced in Brookes [1996], which is presented in Section 3.1. Reference O'Sullivan et al. [2008] was very useful to implement this parser and to make this description, as it gives examples on how to implement parsers using Parsec, and also provides explanations about the tool itself. Lipovača [2011] was also useful for this implementation (and for understanding certain aspects of Haskell as well), just like the Haskell's community's central package archive of open source software `https://hackage.haskell.org/`. The further description of Parsec (including its functions) provided in this section is based on Leijen et al. [2022], O'Sullivan et al. [2008].

The interested reader may consult the GitHub repository associated to this project in Dias [2024]. It contains the implementation corresponding to this chapter and Chapter 6. Appendix A presents a brief user manual of our interpreter of **CQL**, and also some guidelines about the necessary Haskell modules for its use.

For this implementation we defined five data types: `E`, `B`, `BAux`, `C` and `CAux`, which we present next.

```
1 data C = Skip | Asg String E | Seq C C | Paral C C | IfTE_C B C C | WhDo B C
2        deriving (Show, Eq)
```

Listing 5.1: Definition of data type `C`, from file `GrammarBrookes.hs`.

```
1 data CAux = SkipAux | AsgAux String E | SeqAux CAux CAux | ParalAux CAux CAux
2             | IfTE_CAux BAux CAux CAux | WhDoAux BAux CAux | StrC String
3           deriving Show
```

Listing 5.2: Definition of data type `CAux`, from file `GrammarBrookes.hs`.

```
1 data B = BTrue | BFalse | Not B | And B B | Leq E E
```

```
        deriving (Show, Eq)
```

Listing 5.3: Definition of data type B, from file GrammarBrookes.hs.

```
data BAux = BTrueAux | BFalseAux | NotAux BAux | AndAux BAux BAux
            | LeqAux E E | StrB String
          deriving Show
```

Listing 5.4: Definition of data type BAux, from file GrammarBrookes.hs.

```
data E = Zero | One | Id String | PlusE E E | IfTE_E B E E
        deriving (Show, Eq)
```

Listing 5.5: Definition of data type E, from file GrammarBrookes.hs.

Data type C corresponds to *C*, *i.e.* the commands that the language allows (see Rule 3.3). In other words, the allowed values for this data type represent those corresponding to valid commands (*e.g.* `Seq C C` represents a sequence of commands). We have established that identifiers are represented by the `String` data type.

The values allowed for data type `CAux` correspond to those allowed for type C, with the exception that `CAux` has an extra allowed value, `StrC String`. This value does not exist in data type C, otherwise it would disagree with the syntax of Com, described by Rule 3.3. The value constructor `StrC String` is an auxiliary one. Its role is to facilitate the implementation of the parser, which is noticeable in the definition of function `someSemicolons` (see Listing 5.13), for example. It does not represent any type of command. Data type B corresponds to *B*, *i.e.* Boolean expressions (see Rule 3.1). `BAux` and B are related to each other in the same way as `CAux` and C. `StrB String` plays an analogous role to that of `StrC String`. Lastly, data type E corresponds to *E*, *i.e* integer expressions (see Rule 3.2).

For example, the command `a:=0 ; (b:=a || skip)` is represented as the value of type C: `Seq (Asg "a" Zero) (Paral (Asg "b" (Id "a")) Skip)`. We use left-associativity by convention to represent sequences of commands and parallel compositions as C values. For example, the command `a:=0 || b:=1 || c:=0` is represented as the value of type C: `Paral (Paral (Asg "a" Zero) (Asg "b" One)) (Asg "c" Zero)`. We also use left-associativity for Boolean and integer expressions (*e.g.* the Boolean expression `true` & `false` & ¬`true` is represented as `And (And Btrue BFalse) (Not BTrue)`).

Function `parseInputC` represents the parser of the language. Its definition is:

```
parseInputC :: String -> Either ParseError C
parseInputC input = parse parseC "(unknown)" input
```

Listing 5.6: Function parseInputC, from file ParserBrookes.hs.

`parseInputC` receives the input for the parser as an argument, and the value it returns represents the result of its application to the input. Specifically, if the input corresponds to a valid command, the parser succeeds in parsing it and the function will return `Right c`, where `c` is a value of type `C` corresponding to the input. On the other hand, if the input is not considered to be a command, the parser fails to parse it, and the function will return `Left b`, where `b` represents a parse error. Below is an example of the output of this function for two different inputs – one represents success and the other represents failure in parsing.

Example 5.1.1. In the first case `parseInputC` receives a valid command, and therefore parsing is successful. On the other hand the input in the second case is not a valid command (it has an extra `';'`), and so the parser fails:

```
parseInputC "a:=0 ; b:=1" = Right (Seq (Asg "a" Zero) (Asg "b" One))
```

```
parseInputC "a:=0 ;; b:=1" =
Left "(unknown)" (line 1, column 6):
unexpected ';'
expecting end of input
```

`parseC` is the actual parser of commands of the language. We present its source code below:

```
1 parseC :: GenParser Char st C
2 parseC = do
3     x <- parseCAux
4     return (cAuxToC x)
```

Listing 5.7: Function `parseC`, from file `ParserBrookes.hs`.

In words, it relies on an auxiliary parser, `parseCAux`, to do the parsing. `parseC` will start by applying `parseCAux` to its input and `x` will acquire the value returned by `parseCAux`, as the first line of the do block in Listing 5.7 indicates. `parseCAux` returns a `CAux` value corresponding to its input. In the last line of the do block, `parseC` returns `cAuxToC x`. Note that `cAuxToC` is a function that converts a `CAux` value into the corresponding `C` value (its definition can be found in Subsection B.1.1). Thus `parseC` returns the `C` value corresponding to its input. The above definition contains an example of how the type of a parser can be defined. Since `parseC` returns a `C` value, its type is `GenParser Char st C`.

Before presenting the definition of `parseCAux`, it is useful to describe parser `parseCSelect`, which works as an auxiliary parser of `parseCAux`. `parseCSelect` is a parser for commands. It will try to apply different parsers (`pCSeq`, `pCParal`, `pCSkip`, `pCAsg`, `pCIf`, `pCWhile` and `pCParen`, which will be discussed later on) to the input, until finding one that succeeds. Its definition is:

```
parseCSelect = try(pCSeq) <|> try(pCParal) <|> try(pCSkip) <|> try(pCAsg) <|>
               try(pCIf) <|> try(pCWhile) <|> pCParen
```

Listing 5.8: Function `parseCSelect`, from file `ParserBrookes.hs`.

This definition is an example of how functions `<|>` and `try` can be combined in order to establish an alternative between parsers. In essence, `parseCSelect` is obtained from a collection of parsers – one for each type of command plus a parser for handling parentheses. In detail, `pCSeq` parses commands of the form $C_1; C_2$ (*i.e.* `pCSeq` is a parser for a language of sequences of commands), parser `pCParal` parses those of the form $C_1 \| C_2$, parser `pCSkip` parses the `skip`, parser `pCAsg` parses assignments $I := E$, parser `pCIf` parses conditions `if` B `then` C_1 `else` C_2, parser `pCWhile` parses while-loops `while` B `do` C and parser `pCParen` parses commands between parentheses. It is as though `parseCSelect` ends up selecting the suitable parser for a command (with or without parentheses around it) given as input. `parseCSelect` returns a `CAux` value representing the parsed command. Note that this parser still succeeds if only the beginning of its input corresponds to a command. For example, it returns `AsgAux "i" Zero` for input `"i:=0 ..."`. In case of `parseCSelect` having parsed a command inside parentheses, the returned value excludes the parentheses (*e.g.* `parseCSelect` returns `SkipAux` for input `"(skip)"`).

Going back to parser `parseCAux`, its definition is the following:

```
parseCAux = do
    entersOnly
    c <- parseCSelect
    spacesAndEnters
    eof
    return c
```

Listing 5.9: Function `parseCAux`, from file `ParserBrookes.hs`.

We have established that the inputs for which the parser of the language succeeds start with zero or more newline characters, followed by a command of the language, which in turn can only be followed by zero or more characters that are either a space or a newline character. In the definition of `parceCAux` presented in Listing 5.9 , besides using `parseCSelect`, we use `entersOnly`, which parses zero or more newline characters, `spacesAndEnters`, which parses zero or more characters that are either a space (*i.e.* a `' '` character) or a newline character (for more information about these two parsers see Subsection B.1.1), and `eof`, which only succeeds if the current parsing position is the end of input. In this way `parseCAux`, unlike `parseCSelect`, only succeeds for inputs that consist just of a command and the allowed white space around it, For example, it fails for input `"i:=0 ..."`. Thus `parseCAux` is indeed a parser of

allowed programs of the language. It returns the command corresponding to its input in the form of a `CAux` value, as the last line of the do block in Listing 5.9 indicates.

The order in which the auxiliary parsers appear in the definition of Listing 5.8 is not arbitrary. Suppose `parseCAux` is given as input a command such as `skip`; *C* or `skip`$||$*C*. If `pCSkip` is the first parser to appear in that definition, `parseCAux` will fail in parsing the input, as *C* will not be consumed. Thus, `pCSeq` and `pCParal` must be tried before `pCSkip` and, following an analogous reasoning, `pCAsg`. The remaining explanation for that order is based on having considered that:

- the parallel composition has priority over the sequence of commands, *i.e.* $C_1; C_2||C_3; C_4$ is interpreted as $C_1; (C_2||C_3); C_4$, *i.e.* as a sequence of commands;
- `pCSkip` fails when trying to parse an identifier (identifiers cannot have value `"skip"`);
- valid `if` and `while` commands contain curly brackets surrounding the commands that constitute them (*e.g.* `"if i<=j then {skip} else {i:=0}"` is considered a valid `if` command).

We now discuss the implementation of parser `pCSeq`. It has the following definition:

```
1 pCSeq = do
2     (SeqAux (StrC c1) (StrC c2)) <- pCSeqAux
3     return (SeqAux (stringToC (c1)) (stringToC (c2)) )
```

Listing 5.10: Function `pCSeq`, from file `ParserBrookes.hs`

It has an auxiliary parser, `pCSeqAux`, which also parses sequences of commands:

```
1 pCSeqAux = try(lastSemicolon) <|> someSemicolons
```

Listing 5.11: Function `pCSeqAux`, from file `ParserBrookes.hs`

`pCSeqAux` returns `SeqAux (StrC c1) (StrC c2)`, which corresponds to the parsed sequence of commands. Here, `c1` corresponds to the sequence in the input without its last command, while `c2` corresponds to its last command. For example, if the input of `pCSeqAux` is `"skip; a:=0; b:=1"`, `c1` will be `"skip;a:=0"` and `c2` will be `"b:=1"`. In this way the value that `pCSeqAux` returns is represented using left-associativity, and so is the value returned by `pCSeq`. Parser `pCSeq` returns the value that `pCSeqAux` returns – `SeqAux (StrC c1) (StrC c2)` – with the only difference being that `StrC c1` and `StrC c2` are converted into the `CAux` values corresponding to `c1` and `c2`, respectively, using function `stringToC`, which turns a `String` value with a command into the corresponding `CAux` value, and whose definition can be found in Subsection B.1.1. In this way, if `pCSeq` has a sequence

of the form $C_1; C_2$ as input, it will return `SeqAux c1 c2`, where `c1` and `c2` are the two `CAux` values corresponding to C_1 and C_2, respectively.

We consider that a term of a sequence of commands is any command (with or without parentheses around it), except for a sequence without parentheses around it. For example, (`skip`; `skip`) is considered as a term of a sequence of commands, while `skip`; `skip` is not. `lastSemicolon` parses a sequence with exactly two terms. For example, it succeeds for `"skip ; a:=0"` and `"(skip; a:=0); b:=1"`, but not for `"skip; a:=0; b:=1"`. On the other hand, `someSemicolons` parses a sequence with more than two terms (*e.g.* `"skip; a:=0; b:=1"`). Both parsers return a `CAux` value representing the parsed sequence. Their definitions are as follows:

```
1 lastSemicolon = do
2     c1 <- comSeq
3     spacesOnly
4     char ';'
5     spacesOnly
6     entersOnly
7     c2 <- comSeq
8     notFollowedBy (semicolAfterSpaces)
9     return (SeqAux (StrC c1) (StrC c2))
```

Listing 5.12: Function `lastSemicolon`, from file `ParserBrookes.hs`.

```
1 someSemicolons = do
2     c <- comSeq
3     spacesOnly
4     char ';'
5     spacesOnly
6     entersOnly
7     (SeqAux (StrC c1) (StrC c2)) <- pCSeqAux
8     return (SeqAux (StrC (c ++ ";" ++ c1)) (StrC c2))
```

Listing 5.13: Function `someSemicolons`, from file `ParserBrookes.hs`.

`comSeq` in the two definitions above parses terms of a sequence of commands. Its definition is the following:

```
1 comSeq = do
2     com <- try(pCParal) <|> try(pCSkip) <|> try(pCAsg) <|> try(pCIf) <|>
3            try(pCWhile) <|> pCParen
```

```
4     return (cToString (com))
```

Listing 5.14: Function `comSeq`, from file `ParserBrookes.hs`.

In this definition `com` is the `CAux` value representing the parsed command, and `cToString` turns a `CAux` value into the corresponding `String` value. Therefore `comSeq` returns the string corresponding to the parsed command.

Going back to the definition of `lastSemicolon`, we have established that, in a sequence of commands, the term at the left of the `';'` character can only be separated from it by zero or more spaces, and the term at the right of this character can only be separated from it by zero or more spaces followed by zero or more newline characters. In this definition `spacesOnly` parses zero or more spaces (for more information about this parser, see Subsection B.1.1). `lastSemicolon` also makes use of `char`, which is a function such that `char c` parses one single character `c` and returns the parsed character. `notFollowedBy p`, on the other hand, is a parser that only succeeds if `p` fails, and does not consume any input. Lastly `semicolAfterSpaces` parses zero or more spaces followed by the `';'` character. Notice that, if this parser only parsed the `';'` character, `lastSemicolon` would succeed when receiving inputs such as `"skip ; a:=0 ; b:=1"` (although it would not parse them completely), which is not intended.

Notice that the beginning of the definitions of parsers `lastSemicolon` and `someSemicolons` is the same: both parsers will start by applying `comSeq` to the input, which returns a string with the first term of the sequence. Then they both apply parser `spacesOnly`, followed by `char ';'`, which parses `';'` after the first term of the sequence. Parser `pCSeqAux` will first apply `try(lastSemicolon)` to the input. In the definition of `lastSemicolon`, `c2` is a string with the second term of the sequence. This parser will only succeed if `notFollowedBy (semicolAfterSpaces)` succeeds. Therefore if there is not the `';'` character after the second term of the sequence, even after some allowed white space, `try(lastSemicolon)` will succeed. On the other hand, if there is the `';'` character after the second term of the sequence, even if there is some spaces separating it from that term, parser `try(lastSemicolon)` will fail, and `pCSeqAux` will apply parser `someSemicolons` to its whole input (including the first term of the sequence).

The importance of the value constructor `StrC String` is visible in function `someSemicolons` – it is useful for `CAux` to have a value constructor involving `String` values, since they can be easily concatenated, which can be seen in the last line of the definition of this function.

We implemented parser `pCParal` using a similar strategy to the one used for implementing parser `pCSeq`. Below is the definition of `lastParal`, an auxiliary parser of `pCParal`, whose definition and role

are analogous to those of `lastSemicolon`.

```
lastParal = do
    c1 <- comParal
    spacesOnly
    string "||"
    spacesOnly
    c2 <- comParal
    notFollowedBy (paralAfterSpaces)
    return (ParalAux (StrC c1) (StrC c2))
```

Listing 5.15: Function `lastParal`, from file `ParserBrookes.hs`

`string` is a function such that `string s` parses string `s`. In the case of parallel compositions, we have established that the first command and the second command can only be separated from the `"||"` string by zero or more spaces. `comParal` parses every command, including commands inside parentheses, except for sequences without parentheses around them and parallel compositions without parentheses around them. For example, it succeeds for input `"(skip || skip)"`, but not for `"skip ; skip"` or `"skip || skip"`. It returns the string corresponding to the parsed command, and its definition is similar to that of `comSeq` (presented in Listing 5.14). The role of parser `paralAfterSpaces` is analogous to that of `semicolAfterSpaces`.

We have considered the following restrictions for identifiers:

- they can only be composed of alphabetic or numeric Unicode characters and underscores (*i.e.* `'_'` characters);
- their first character must be either an alphabetic Unicode character or an underscore – it cannot be a digit;
- they cannot only be composed of underscores.

We have fixed the following words as reserved words of the language: *skip*, *if*, *then*, *else*, *while*, *do*, *true* and *false*. The parser for identifiers is `parseIdeStr`, which has the following definition:

```
parseIdeStr = do
    notFollowedBy (reservedWord)
    i <- (try(startUnderscore) <|> startLetter)
    return i
```

Listing 5.16: Function `parseIdeStr`, from file `ParserBE_Brookes.hs`.

Parser `reservedWord` parses a reserved word as long as neither an alphabetic or numeric Unicode character nor an underscore follows the word. Thus, `reservedWord` fails when it receives valid identifiers that start with reserved words (*e.g.* `"dot"`) as input. `startUnderscore` parses identifiers that start with the underscore character, and `startLetter` parses the ones that start with a letter. Hence, `parseIdeStr` fails when it receives a reserved word and the latter is not the beginning of an identifier. It returns a string with the parsed identifier.

We have established that the three elements of an assignment (an identifier, the $:=$ symbol and an integer expression) can only be separated from each other by zero or more spaces. The definition of `pCAsg` is thus the following:

```
1 pCAsg = do
2     i <- parseIdeStr
3     spacesOnly
4     string ":="
5     spacesOnly
6     e <- parseESelect
7     return (AsgAux i e)
```

Listing 5.17: Function `pCAsg`, from file `ParserBrookes.hs`.

`parseESelect` is a parser for integer expressions, which returns an `E` value representing the parsed expression. Its definition is similar to that of `parseCSelect` (presented in Listing 5.8), and can be found in Subsection B.1.1.

Regarding parser `pCSkip`, its definition is the following:

```
1 pCSkip = do
2     string "skip"
3     notFollowedBy (try(alphaNum) <|> parseUnderscore)
4     return SkipAux
```

Listing 5.18: Function `pCSkip`, from file `ParserBrookes.hs`.

`alphaNum` parses both alphabetic and numeric Unicode characters, returning the parsed character, while `parseUnderscore` succeeds when applied to an underscore character. Thus `pCSkip` fails when it receives as input an identifier that starts with *skip*, such as `"skip3"`.

Let us now consider parser `pCWhile`, which has the following definition:

```
1 pCWhile = do
2     string "while"
3     separateElems
```

```
4       b <- parseBSelect
5       separateElems
6       string "do"
7       separateOrJoined
8       char '{'
9       separateOrJoined
10      c <- parseCSelect
11      separateOrJoined
12      char '}'
13      return (WhDoAux b c)
```

Listing 5.19: Function `pCWhile`, from file `ParserBrookes.hs`.

`separateElems` succeeds when applied to one of the following two alternatives for white space: at least one space followed by any number (including zero) of newline characters; at least one newline character. Thus, using this parser, we have established that the Boolean expression in `while` commands and `if` commands must be separated from other elements by one of these alternatives. On the other hand, `separateOrJoined` succeeds either when applied to one of these two alternatives or when applied to the empty string (for more details about parsers `separateElems` and `separateOrJoined`, see Section B.1.1). In this way, we have established that curly brackets in `while` commands, as well as in `if` commands, do not need to be separated from other elements by white space.

`parseBSelect` parses Boolean expressions, and does not parse any white space that may exist before or after them in the input. The definition of this parser is analogous to those of `parseCSelect` (presented in Listing 5.8) and `parseESelect` (presented in Listing B.4), and can be found in Subsection B.1.1. `parseBSelect` returns a value of type `BAux` representing the parsed Boolean expression. In the case of having parsed a Boolean expression inside parentheses, the returned value excludes the parentheses.

The definition of `pCIf` is analogous to that of `pCWhile`. The one of `pCParen` is presented below:

```
1 pCParen = do
2       char '('
3       spacesOnly
4       cInsideParen <- parseCSelect
5       spacesOnly
6       char ')'
7       return cInsideParen
```

Listing 5.20: Function `pCParen`, from file `ParserBrookes.hs`.

Parser `pCParen` accepts zero or more spaces between the parentheses characters and the command inside them, since we have established that the parentheses characters can only be separated by zero or more spaces from that command. It returns the `CAux` value corresponding to the command inside parentheses.

5.2 Concurrent Quantum Language

We now describe the implementation of the parser of **CQL**. This implementation is based on the one described in the previous section. The same sources that have been useful for implementing the parser of the previous section, which are mentioned in the first paragraph of Section 5.1, have also been useful for this implementation.

For this implementation we defined five data types: `C`, `CAux`, `G`, `QVar` and `QVarList`, which we present next.

```
1 data C = Skip | Seq C C | U G QVarList | Meas QVar C C | Wh QVar C
2          | Paral C C
3        deriving (Show, Eq)
```

Listing 5.21: Definition of data type `C`, from file `GrammarQ.hs`.

```
1 data CAux = SkipAux | SeqAux CAux CAux | UAux G QVarList
2             | MeasAux QVar CAux CAux | WhAux QVar CAux
3             | ParalAux CAux CAux | Str String
4           deriving Show
```

Listing 5.22: Definition of data type `CAux`, from file `GrammarQ.hs`.

```
1 data G = H | I | X | Y | Z | CNOT | CZ
2           deriving (Show, Eq)
```

Listing 5.23: Definition of data type `G`, from file `GrammarQ.hs`.

```
1 type QVar = String
```

Listing 5.24: Definition of data type `QVar`, from file `GrammarQ.hs`.

```
1 type QVarList = [QVar]
```

Listing 5.25: Definition of data type `QVarList`, from file `GrammarQ.hs`.

Data type `C` corresponds to *C*, *i.e.* the commands that the language allows (see Rule 4.36). `CAux` and `C` are related to each other in the same way as in the previous section. Notice that `CAux` and `C` are now defined differently, in a different file. However data types `CAux` and `C` play an analogous role to that of the same data types in the previous section. Data type `G` represents the gates included in the language, with each of its value constructors representing each of these gates. Lastly, `QVar` represents quantum variables and `QVarList` represents lists of quantum variables. `QVar` and `QVarList` are defined in such a way that `QVar` is equivalent to type `String`, while `QVarList` is equivalent to type `[QVar]`. For example, string `"q1"` can be used for representing a quantum variable, while list `["q1","q2"]` can be used for representing a list of these variables. The restrictions we impose on quantum variables are the same as those imposed on identifiers, which are explained in the previous section. However, we have established different reserved words for **CQL**, namely: *skip*, *H*, *I*, *X*, *Y*, *Z*, *CNOT*, *CZ*, *or*, *Meas* and *while*. Similarly to how we deal with identifiers, quantum variables can start with reserved words (*e.g.* `"CNOTcontrol"` is a valid quantum variable).

Functions `parseInputC`, `parseC` and `parseCAux`, which are described in the previous section, maintain their definition and role in this implementation. In this way for **CQL** we maintain the requirement that the inputs for which the parser of the language succeeds can only start with zero or more newline characters, followed by a command of the language, which in turn can only be followed by zero or more characters that are either a space or a newline character.

However, the auxiliary function `cAuxToC` of `parseC`, which converts values of type `CAux` to the corresponding `C` values, now has a different definition that agrees with the types `C` and `CAux` used in this implementation. Its definition can be found in Subsection B.1.2.

Besides that, the auxiliary function `parseCSelect` of `parseCAux` now has a different definition that agrees with the syntax of **CQL**:

```
1 parseCSelect = try(pCParal) <|> try(pCSeq) <|> try(pCSkip) <|>
2                try(pCGate) <|> try(pCMeas) <|> try(pCWhile) <|> pCParen
```

Listing 5.26: Function `parseCSelect`, from file `ParserQ.hs`.

Each of the auxiliary parsers of `parseCSelect` parses a different type of command, except for `pCParen`, which parses any command inside parentheses. More concretely, `pCParal` parses parallel compositions, `pCOr` parses non-deterministic choice commands, `pCSeq` parses sequences of commands, `pCSkip` parses `skip` commands, `pCGate` parses gate commands, `pCMeas` parses measurement commands and `pCWhile` parses while commands. Thus parseCSelect maintains the same role as in the previous section, but is now adapted to **CQL**. Just like in the implementation described in the previous section,

when `parseCSelect` parses a command inside parentheses, the `CAux` value it returns excludes those parentheses.

The order in which the auxiliary parsers of `parseCSelect` appear in its definition is, once again, not arbitrary. However, for the case of **CQL**'s parser, we now consider that the sequence of commands has priority over the parallel composition, *i.e.* $C_1; C_2 || C_3$ is interpreted as the parallel composition $(C_1; C_2) || C_3$. This choice of priority is explained by a matter of preference concerning the writing of commands of the language, and can be made differently. Notice as well that we have established that the command that is inside a while command is surrounded by curly brackets.

Parser `pCParal` and its auxiliary functions are defined analogously to how they are defined in the implementation discussed in the previous section. The only difference concerns the spacing allowed in parallel compositions. For the implementation of **CQL**'s parser, we consider that the second command in a parallel composition can be in a line below the first command. Thus now we force that, after the $||$ symbol, there can only be zero or more spaces followed by zero or more newline characters before the second command. Therefore, for example, the definition of parser `lastParal`, whose definition for the implementation of the previous section is in Listing 5.15, is now the following:

```
1 lastParal = do
2     c1 <- comParal
3     spacesOnly
4     string "||"
5     spacesOnly
6     entersOnly
7     c2 <- comParal
8     notFollowedBy (paralAfterSpaces)
9     return (ParalAux (Str c1) (Str c2))
```

Listing 5.27: Function `lastParal`, from file `ParserQ.hs`.

Notice as well that, just like the definition of `parseCSelect` has now changed in order to adapt to **CQL**, so has that of `comParal`. The latter now parses any command (with or without parentheses around it), except for parallel compositions with no parentheses around them. For example, `"(skip || skip)"` is parsed by `comParal`, while `"skip || skip"` is not.

Parser `pCSeq` and its auxiliary functions are also defined analogously to how they are defined in the implementation discussed in the previous section. However, the definition of `comSeq` has also changed. This parser now parses any command (with or without parentheses around it), except for sequences with no parentheses around them, parallel compositions with no parentheses around them and non-deterministic

choice commands with no such parentheses. For example, "skip || skip" is not parsed by comSeq, while "(skip || skip)" is. The definition of parser comSeq, whose definition for the implementation of the previous section is in Listing 5.14, is now the following:

```
comSeq = do
    com <- try(pCSkip) <|> try(pCGate) <|> try(pCMeas) <|> try(pCWhile) <|>
           pCParen
    return (cToString (com))
```

Listing 5.28: Function comSeq, from file ParserQ.hs.

Regarding the parser for skip commands, its definition is the following:

```
pCSkip = do
    string "skip"
    return SkipAux
```

Listing 5.29: Function pCSkip, from file ParserQ.hs.

The definition of the parser for gate commands is as follows:

```
pCGate = try(pGate1Q) <|> pGate2Q
```

Listing 5.30: Function pCGate, from file ParserQ.hs.

pGate1Q and pGate2Q are parsers for gate commands; the former corresponds to 1-qubit gates and the latter to 2-qubit gates. The definition of pGate1Q is the following:

```
pGate1Q = do
    g <- gate1Q
    separateOrJoined
    char '('
    separateOrJoined
    q <- parseQVar
    qs <- (try(parseQVars) <|> return [])
    separateOrJoined
    char ')'
    return (UAux g (q:qs))
```

Listing 5.31: Function pGate1Q, from file ParserQ.hs.

gate1Q is a parser for strings representing 1-qubit gates belonging to the language. More specifically, it parses strings "H", "I", "X", "Y" and "Z" and returns the value of type G corresponding to the parsed string. parseQVars parses a comma followed by a list of quantum variables, separated by commas. It returns a list of type QVars with those variables. For example, parseQVars succeeds for input ",

q1, q2", in which case it returns ["q1","q2"]. Its definition can be found in Subsection B.1.2. Note that q:qs is a list whose first element is q and qs is the remainder. The definition of parser pGate2Q is similar to that of pGate1Q. Both parsers are defined in such a way that the different elements of gate commands (*e.g.* the string representing the gate and the parentheses) need not be separated by white space.

As to the parser for commands of the form $\texttt{Meas}(q) \to (C_1, C_2)$, its definition is as follows:

```
pCMeas = do
    string "Meas"
    separateOrJoined
    char '('
    spacesOnly
    q <- parseQVar
    spacesOnly
    char ')'
    separateOrJoined
    string "->"
    separateOrJoined
    char '('
    separateOrJoined
    c1 <- parseCSelect
    spacesOnly
    char ','
    separateOrJoined
    c2 <- parseCSelect
    separateOrJoined
    char ')'
    return (MeasAux q c1 c2)
```

Listing 5.32: Function pCMeas, from file ParserQ.hs.

parseQVar is a parser for quantum variables, which returns a value of type QVar with the parsed quantum variable.

The definition of parseQVar is very similar to that of parseIdeStr, whose definition is in Listing 5.16; the only difference lays on the fact that we have established different reserved words for **CQL**, comparing to those established for the language in which the previous section focuses on. Its definition is the following:

```
parseQVar = do
```

```
2     notFollowedBy (reservedWordQ)
3     i <- (try(startUnderscore) <|> startLetter)
4     return i
```

Listing 5.33: Function `parseQVar`, from file `ParserQ.hs`.

Just like parser `reservedWord` (which is described in the previous section), parser `reservedWordQ` parses a reserved word as long as neither an alphabetic or numeric Unicode character nor an underscore follows said word.

The parser for commands of the form `while` `Meas` $(q) \longrightarrow C$ has the following definition:

```
1 pCWhile = do
2     string "while"
3     separateElems
4     string "Meas"
5     separateOrJoined
6     char '('
7     spacesOnly
8     q <- parseQVar
9     spacesOnly
10    char ')'
11    separateOrJoined
12    char '{'
13    separateOrJoined
14    c <- parseCSelect
15    separateOrJoined
16    char '}'
17    return (WhAux q c)
```

Listing 5.34: Function `pCWhile`, from file `ParserQ.hs`.

This definition is similar to that presented for `pCWhile` in Listing 5.19, in the previous section. Notice that, in order to minimize ambiguity in **CQL**'s programs, we have established that the arrow of a while command (see the grammar in Equation 4.36) is replaced by curly brackets around the command that is inside it, when writing programs of the language. For example, a command of the form `while` `Meas` $(q) \longrightarrow C_1; C_2$ can be interpreted as a while command and as a sequence of commands, if no priority is established regarding these two types of command. Thus, for example, `"while Meas (q1) {skip}"` is parsed by `pCWhile`. In the definition of this parser, we establish that strings `"while"` and `"Meas"` must be separated by white space, which can be either at least one space followed by any number (including zero) of newline characters, or at least one newline character.

Finally, the definition of `pCParen` is the following:

```
pCParen = do
    char '('
    separateOrJoined
    cInsideParen <- parseCSelect
    separateOrJoined
    char ')'
    return cInsideParen
```

Listing 5.35: Function `pCParen`, from file `ParserQ.hs`.

`pCParen` is defined similarly to how it is defined in the implementation discussed in the previous section (see Listing 5.20). The only difference concerns the allowed white space in commands inside parentheses. In the case of **CQL**, the command inside parentheses needs not be in the same line as the parentheses.

Chapter 6

Semantics

The transition rules of the operational semantics of a language can be used to determine, for a given configuration (*i.e.* a program of that language to be executed and a current state), what the next configuration may be – in the case of small-step semantics – or what the final configuration may be – in the case of big-step semantics.

In order to build our interpreter for **CQL** it is necessary to implement programs that output the result of executing commands of the language, for a given initial state. Such programs correspond to an implementation of the transition rules of the big-step operational semantics of **CQL**, which are acquired from the small-step transition rules.

In this chapter, we discuss the implementation of functions that represent the transition rules corresponding to the operational semantics of three languages. Each of chapter's sections corresponds to a different language: the fist one focuses on the language presented in Brookes [1996]; the second one concerns the parallel language with probabilistic choice discussed in Subsection 3.2.1; and the last section corresponds to **CQL**, whose semantics is presented in Section 4.2. The goal of the first two sections is to facilitate the comprehension of the last section, as the languages of each section become progressively more similar to our concurrent quantum language. For developing this implementation, the Haskell's community's central package archive of open source software `https://hackage.haskell.org/` was a useful source, as well as reference Lipovača [2011].

6.1 Basic Parallel Language

We start with the basic parallel language introduced in Brookes [1996], which was discussed in Section 3.1.

In this section, data type C is the same as the one defined in Listing 5.1. Data type S represents states and is defined in the following way:

```
1 type S = [(String, Integer)]
```

Listing 6.1: Definition of data type S, from file GrammarBrookes.hs.

Thus, in our implementation, a state is represented as a list of tuples, each of them composed of a string (corresponding to an identifier) and an integer value. For example, state $[a = 1,\ b = 2]$ is represented as the following value of type S: `[("a",1),("b",2)]`.

6.1.1 Big-step semantics

For this language we focus on using the operational semantics for determining the list of terminal configurations (*i.e.* successfully terminated configurations) that can be derived from a certain initial configuration.

List of configurations

We now discuss the implementation of function `bigStepList` which, for a given configuration (*i.e.* a command to be executed and a current state), returns a list consisting of the terminal configurations that can be achieved from that configuration. In other words, `bigStepList c s` corresponds to the list of the terminal configurations that configuration $\langle$c, s$\rangle$ can lead to, after a certain number of computational steps. This implementation is based on the transition rules presented in Figure 2, associated with the small-step semantics.

```
1  bigStepList :: C -> S -> [(C, S)]
2  bigStepList Skip s = [(Skip,s)]
3  bigStepList (Asg i e) s = [(Skip, (changeSt i n s) )]
4      where n = (bigStepExp e s)
5  bigStepList (Seq c1 c2) s =  if (term c1 s) then (bigStepList c2 s)
6                               else leaves c2 (bigStepList c1 s)
7  bigStepList (IfTE_C b c1 c2) s = if (bigStepBExp b s) then (bigStepList c1 s)
8                                   else bigStepList c2 s
9  bigStepList (WhDo b c) s = if (bigStepBExp b s)
10                            then (bigStepList (Seq c (WhDo b c)) s)
11                            else (bigStepList Skip s)
12 bigStepList (Paral c1 c2) s
13     | term (Paral c1 c2) s = [(Paral c1 c2, s)]
14     | term c1 s = concat (map (paralBigStep c1) (smallStepList c2 s))
15     | term c2 s = concat (map (paralBigStep c2) (smallStepList c1 s))
16     | otherwise = concat (map (paralBigStep c2) (smallStepList c1 s))
```

```
17                     ++ concat (map (paralBigStep c1) (smallStepList c2 s))
```

Listing 6.2: Function bigStepList, from file SemBrookes.hs.

We have established that, if the arguments of function bigStepList correspond to a successfully terminated configuration, then it returns a list with just that configuration. Line 2 of the above definition corresponds to an example of the latter case – since $\langle \texttt{Skip},\ \texttt{s} \rangle$ is a successfully terminated configuration, bigStepList returns [(Skip,s)] when receiving command Skip and state s as arguments.

Let us now focus on the definition of bigStepList for the assignment command Asg i e. bigStepExp is an auxiliary function such that n = (bigStepExp e s) means that $\langle \texttt{e},\ \texttt{s} \rangle \Downarrow \texttt{n}$, with n being a value of type Integer, e a value of type E and s a value of type S. Its definition can be found in Listing B.13 of Subsection B.2.1. changeSt is an auxiliary function such that changeSt i n s is a value of type S representing $[\,\texttt{s}\,|\,\texttt{i} = \texttt{n}\,]$, with i being an identifier, n being an Integer value corresponding to a non-negative integer and s being a value of type S. Thus this definition corresponds to the second rule of Figure 2, which leads to a successfully terminated configuration.

Regarding the definition of bigStepList for the sequential command Seq c1 c2, term is an auxiliary function such that term c s, with c being a command and s being a state, equals True if and only if we can prove that configuration $\langle \texttt{c},\ \texttt{s} \rangle$ is successfully terminated. Its definition can be found in Listing B.14 of Subsection B.2.1. Hence, the case in this definition where (term c1 s) is true corresponds to the fourth rule of Figure 2. In this case, the next configuration that $\langle \texttt{c1};\texttt{c2},\ \texttt{s} \rangle$ leads to is $\langle \texttt{c2},\ \texttt{s} \rangle$, and thus we calculate bigStepList for the latter configuration. On the other hand, the case where (term c1 s) is false corresponds to the third rule of Figure 2. In this case we know, from the rules in Figure 2, that the computation starting in the initial configuration $\langle \texttt{c1};\texttt{c2},\ \texttt{s} \rangle$ will have the following stages, where the $\cdots$ represent the possibility for other configurations in between:

$$\langle \texttt{c1};\texttt{c2},\ \texttt{s} \rangle \to \cdots \to \langle \texttt{c1'};\texttt{c2},\ \texttt{s'} \rangle \to \langle \texttt{c2},\ \texttt{s'} \rangle \to^* \langle \texttt{c2'},\ \texttt{s''} \rangle term \qquad (6.1)$$

with $\langle \texttt{c1},\ \texttt{s} \rangle \to^* \langle \texttt{c1'},\ \texttt{s'} \rangle term$. Here, $\langle \texttt{c},\ \texttt{s} \rangle \to^* \langle \texttt{c'},\ \texttt{s'} \rangle$ means that $\langle \texttt{c},\ \texttt{s} \rangle$ leads, after a certain number of computational steps, to $\langle \texttt{c'},\ \texttt{s'} \rangle$, which is a terminal configuration. In this case, bigStepList (Seq c1 c2) s corresponds to a list with all possible values of $\langle \texttt{c2'},\ \texttt{s''} \rangle$. Auxiliary function leaves allows to obtain this list. Its definition is the following:

```
1 leaves :: C -> [(C,S)] -> [(C,S)]
2 leaves c roots = concat (map (bigStepList c) rootStates)
3     where rootStates = map snd roots
```

Listing 6.3: Function leaves, from file SemBrookes.hs.

When applying function leaves to c2 and (bigStepList c1 s), variable roots becomes a list with all possible values of (c1', s') and variable rootStates becomes a list with all possible values of s'. Thus, leaves c2 (bigStepList c1 s) corresponds to a list with all possible values of ⟨c2', s''⟩ (the description of functions map, snd and concat can be found in Prelude module's documentation).

Let us now focus on the definition of bigStepList for the conditional command IfTE_C b c1 c2. bigStepBExp is an auxiliary function such that v = (bigStepBExp b s) means that $\langle$b, s$\rangle \rightarrow^*$ v, with v being a truth value of type Bool, b a value of type B and s a value of type S. Its definition can be found in Listing B.15 of Subsection B.2.1. Thus, the case where bigStepBExp b s is true corresponds to the fifth rule of Figure 2 and the case where bigStepBExp b s is false corresponds to the sixth rule of Figure 2.

Analysing the definition of bigStepList for the while command WhDo b c, one can conclude that it corresponds to (bigStepList (IfTE_C b (Seq c (WhDo b c)) Skip) s), which is in agreement with the eighth rule of Figure 2.

Let us now consider the definition of bigStepList for the parallel composition command Paral c1 c2. paralBigStep is an auxiliary function such that paralBigStep c1 (c2, s) is equal to bigStepList (Paral c2 c1) s, where c1 and c2 are arbitrary commands and s is an arbitrary state. smallStepList is a function that, for a given configuration, returns a list of the possible configurations that can be achieved from that configuration, through a transition, *i.e.* through one computational step. In other words, smallStepList c s corresponds to the list of configurations that can be achieved from configuration ⟨c, s⟩ through this step. The definition of this function and that of paralBigStep can be found in Subsection B.2.1, in Listings B.17 and B.16, respectively. Hence, if ⟨c1, s⟩ is a successfully terminated configuration and ⟨c2, s⟩ is not, then bigStepList (Paral c1 c2) s will be equal to a list corresponding to all possible values of ⟨c3, s''⟩, with ⟨c1||c2', s'⟩ $\rightarrow^*$ ⟨c3, s''⟩$term$ and with ⟨c1||c2, s⟩ $\rightarrow$ ⟨c1||c2', s'⟩ being a transition obtained from the ninth rule of Figure 2. This agrees with the fact that, in this case, only this transition can be executed from configuration ⟨c1||c2, s⟩, since c1 has terminated. The definition for the cases where ⟨c1, s⟩ is not successfully terminated can be understood in an analogous way (the description of function (++) can be found in Prelude module's documentation).

Notice that function bigStepList is supposed to be used as an argument of applySem, which is responsible for checking if all identifiers present in the command given to bigStepList are declared in the state that this function is given. applySem f c s corresponds to the result of applying function f to command c and state s, if s is defined on all the free identifiers of c. Otherwise, applySem f c s

raises an error indicating that such condition is not fulfilled. Its definition is in Listing B.18 of Subsection B.2.1.

6.2 Basic Parallel Language with Probabilistic Choice

We now focus on the basic parallel language with probabilistic choice discussed in Subsection 3.2.1, which, as previously mentioned, corresponds to an extension of the language in Brookes [1996] (on which the previous section focuses).

Data type CpC represents commands of the language and corresponds to C from the grammar presented in Equation 3.5. It is defined as follows:

```
data CpC = SkipPC | AsgPC String E | SeqPC CpC CpC | PC Prob CpC CpC
           | IfTE_PC B CpC CpC | WhDoPC B CpC | ParalPC CpC CpC
       deriving (Show, Eq)
```

Listing 6.4: Definition of data type CpC, from file GrammarBrookes.hs.

Notice that the allowed values for CpC correspond to those allowed for data type C, defined in Listing 5.1, except for PC Prob CpC CpC, which represents command $C_1 \oplus_p C_2$. We represent probabilities using data type Prob, which we define as a synonym of type Double. The latter is used for representing double-precision floating-point numbers, as indicated in Prelude module's documentation. Such a number corresponds to a 64-bit approximate representation of a real number IBM [2023b]. In Listing 6.4, data types E and B are those whose definition is presented in Listings 5.5 and 5.3, respectively.

6.2.1 Big-step semantics

We present two approaches for the implementation of the transition rules of the language. The first one does not use a scheduler and allows to obtain all the possible final distributions on configurations that can be derived from an initial configuration. The second approach uses a scheduler that, when deciding between two possible distributions, attributes to each of them a probability of 0.5 (this restriction will be explained in this subsection).

Big-step without scheduler (list of distributions)

We now discuss the implementation of function bigStepList, which returns the terminal configurations that can result from a given initial configuration. The function receives a command c of type CpC and a state s of type S, in such a way that (bigStepList c s) is a list that consists of the final probability

distributions on configurations that can be achieved from the initial configuration $\langle$c, s$\rangle$. The support of these probability distributions on set $Conf$ only contains terminal configurations that can be achieved from that initial configuration. We choose to represent (bigStepList c s) as a value of type [[ConfPC]], with data type ConfPC being defined in the following way:

```
1 type ConfPC = (Prob, CpC, S)
```

Listing 6.5: Definition of data type ConfPC, from file GrammarBrookes.hs.

Each value (p,c,s) of type ConfPC represents a configuration $\langle$c, s$\rangle$ with an associated probability of p, and we use type [ConfPC] for representing distributions on configurations. For example, in this context, list [(0.8,SkipPC,[("a",0)]),(0.2,SkipPC,[("a",1)])] of type [ConfPC] represents distribution $0.8 \cdot \langle \texttt{skip},\ [a = 0]\rangle + 0.2 \cdot \langle \texttt{skip},\ [a = 1]\rangle$. The following example shows the result of applying function bigStepList to a given initial configuration.

Example 6.2.1. Consider the initial configuration $\langle a := 0 \oplus_{0.4} (a := 1 \parallel a := 1 + 1)\,,\ [a = 3]\rangle$, from Example 3.2.3. Applying function bigStepList to it, we obtain:

```
    bigStepList (PC 0.4 (AsgPC "a" Zero) (ParalPC (AsgPC "a" One)
(AsgPC "a" (PlusE One One)))) [("a",3)] =

    [[(0.4,SkipPC,[("a",0)]),(0.6,ParalPC SkipPC SkipPC,[("a",2)])],
    [(0.4,SkipPC,[("a",0)]),(0.6,ParalPC SkipPC SkipPC,[("a",1)])]]
```

The output of bigStepList presented above corresponds to a list containing two distributions: $0.4 \cdot \langle \texttt{skip},\ [a = 0]\rangle + 0.6 \cdot \langle \texttt{skip} \parallel \texttt{skip},\ [a = 2]\rangle$ and $0.4 \cdot \langle \texttt{skip},\ [a = 0]\rangle + 0.6 \cdot \langle \texttt{skip} \parallel \texttt{skip},\ [a = 1]\rangle$. Let us consider that there is a scheduler $\mathcal{S}$ determining the path of the computation that starts in our initial configuration. Note that this scheduler just serves the purpose of explaining the output of function bigStepList, whose definition does not need the implementation of a scheduler. Recalling the probabilistic automaton from Example 3.2.3, the output of bigStepList can be interpreted as the fact that, if $\mathcal{S}$ chooses φ_3 instead of φ_4 when determining the path of the computation after configuration c_2, there is a probability of 0.4 of obtaining $\langle \texttt{skip},\ [a = 0]\rangle$ as the terminal configuration, and a probability of 0.6 of obtaining $\langle \texttt{skip} \parallel \texttt{skip},\ [a = 2]\rangle$ instead; however, if the scheduler opts for φ_4, the probability of each possible terminal configuration is given by distribution $0.4 \cdot \langle \texttt{skip},\ [a = 0]\rangle + 0.6 \cdot \langle \texttt{skip} \parallel \texttt{skip},\ [a = 1]\rangle$. Therefore each distribution in the output represents a different choice made by scheduler $\mathcal{S}$, regarding the path after configuration c_2.

In general terms, the terminal configurations returned by bigStepList when applied to a given configuration $\langle C,\ s\rangle$ are the last configurations of the maximal paths in set $\mathrm{MP}\,(C, s, \mathcal{S})$, where $\mathcal{S}$ is a

scheduler determining the path of computations. Each distribution in list `(bigStepList c s)` represents a specific combination of choices made by scheduler $\mathcal{S}$ between two possible distributions which a path can lead to. Notice that, in the above example, the scheduler only makes such a choice once, between φ_3 and φ_4.

We now present the definition of function `bigStepList`, which is based on the transition rules presented in Figure 3, associated with the small-step semantics.

```
1  bigStepList :: CpC -> S -> [[ConfPC]]
2  bigStepList SkipPC s = [[(1, SkipPC, s)]]
3  bigStepList (AsgPC i e) s = [[(1, SkipPC, changeSt i n s)]]
4      where n= (bigStepExp e s)
5  bigStepList (SeqPC c1 c2) s
6      | term c1 s = bigStepList c2 s
7      | otherwise = concat $ map bigStepD (beforeC2 c1 c2 s)
8  bigStepList (PC p c1 c2) s = [ (mult p a) ++ (mult (1-p) b)
9                                | a <- (bigStepList c1 s), b <- (bigStepList c2 s)]
10 bigStepList (IfTE_PC b c1 c2) s = if (bigStepBExp b s) then bigStepList c1 s
11                                   else bigStepList c2 s
12 bigStepList (WhDoPC b c) s = if (bigStepBExp b s)
13                              then (bigStepList (SeqPC c (WhDoPC b c)) s)
14                              else (bigStepList SkipPC s)
15 bigStepList (ParalPC c1 c2) s
16     | term (ParalPC c1 c2) s = [[(1, ParalPC c1 c2, s)]]
17     | otherwise = concat $ map bigStepD (smallStepList (ParalPC c1 c2) s)
```

Listing 6.6: Function `bigStepList`, from file `SemProbConc.hs`.

Just like in the implementation of the previous section, we have considered that, if the arguments of function `bigStepList` correspond to a successfully terminated configuration, its output corresponds to that same configuration. In this case, it returns a distribution in which that configuration has probability 1, as represented in line 2 of the above definition.

From the transition rules in Figure 3, we know that a configuration of the form $\langle I := E,\ s\rangle$ leads to only one possible final probability distribution on configurations, which is $1 \cdot \langle \texttt{skip},\ [\,s\,|\,I = n\,]\rangle$. This distribution is represented in line 3 of the above definition.

Regarding the definition of `bigStepList` for command `SeqPC c1 c2`, line 6 from the above definition follows from the fourth rule in Figure 3, and is analogous to line 5 of the definition of the function in Listing 6.2, which was explained in the previous section. Notice that function `term` maintains the role mentioned in the previous section, but now has a different definition. On the other hand, line 7 makes

use of functions bigStepD and beforeC2, which are described below.

The definition of function beforeC2 is the following:

```
1 beforeC2 :: CpC -> CpC -> S -> [[ConfPC]]
2 beforeC2 c1 c2 s = let afterC1 = bigStepList c1 s
3                    in (map (replaceBy c2) afterC1)
```

Listing 6.7: Function beforeC2, from file SemProbConc.hs.

After some transitions, configuration $\langle \texttt{c1},\ \texttt{s}\rangle$ leads to one or more distributions on terminal configurations, *i.e.* distributions of the form $\sum_i p_i \cdot \langle C_i,\ s_i\rangle term$. afterC1 in the above definition represents a list with these distributions. replaceBy is an auxiliary function such that, for a given command c2 and a list l of type [ConfPC], (replaceBy c2 l) is a list of this type resulting from replacing by c2 each c in all elements (p,c,s) of l. Thus (beforeC2 c1 c2 s) in the above definition corresponds to a list of all possible distributions of the form $\sum_i p_i \cdot \langle C_2,\ s_i\rangle$, resulting from replacing C_i by C_2 in the distributions corresponding to afterC1. Remembering the third and fourth transition rule of Figure 3, (beforeC2 c1 c2 s) is the list of distributions that $\langle \texttt{c1}; \texttt{c2},\ \texttt{s}\rangle$ can lead to, after the execution of c1 and before the execution of c2.

We now discuss function bigStepD, which is defined as follows:

```
1 bigStepD :: [ConfPC] -> [[ConfPC]]
2 bigStepD [] = [[]]
3 bigStepD ((p,c,s):t) = [ (mult p a) ++ b | a <- (bigStepList c s),
4                                             b <- (bigStepD t) ]
```

Listing 6.8: Function bigStepD, from file SemProbConc.hs.

Given an initial distribution on configurations d of type [ConfPC], (bigStepD d) is a list of the final probability distributions of configurations that distribution d can lead to. For example, remembering Example 3.2.3, let d1 represent distribution $0.4 \cdot K_1 + 0.6 \cdot K_2$, and let d2 and d3 represent distributions $0.4 \cdot K_3 + 0.6 \cdot K_6$ and $0.4 \cdot K_3 + 0.6 \cdot K_7$, respectively. Then, bigStepD d1 = [d2,d3]. In general terms, bigStepD applied to distribution $\sum_i p_i\langle c_i,\ s_i\rangle$ outputs a list with all possible values of $\sum_{i,j} p_i p_j\langle c_j,\ s_j\rangle$, with $\sum_j p_j\langle c_j,\ s_j\rangle$ being the final probability distribution that $\langle c_i,\ s_i\rangle$ leads to (which can have different possible values). The list in line 3 of the above equation is a comprehension list, whose elements corresponds to a different combination of a and b, where a ranges over the elements of list (bigStepList c s) and b ranges over those of list (bigStepD t). More information about list comprehensions can be found in Lipovača [2011]. mult is a function such that (mult p a) is a [ConfPC] value resulting from multiplying by p all probabilities in distribution a. Thus, going back to

line 7 of the definition of `bigStepList` in Listing 6.6, the output of `(bigStepList (SeqPC c1 c2) s)` when ⟨`c1, s`⟩ is not successfully terminated corresponds to a list of all the final distributions that can be achieved from those in list `(beforeC2 c1 c2 s)`.

Moving on to the definition of `bigStepList` for command `(PC p c1 c2)`, the output of this function is equivalent to `bigStepD [(p,c1,s),(1-p,c2,s)]`. The reason for this is that $\langle \texttt{c1} \oplus_{\texttt{p}} \texttt{c2},\ \texttt{s} \rangle$ leads to distribution `[(p,c1,s),(1-p,c2,s)]`.

The definiton of `bigStepList` for commands `(IfTE_PC b c1 c2)` and `(WhDoPC b c)` follows the same reasoning as in the definition of the function in Listing 6.2, for the analogous commands. The same happens with the definition of `bigStepList` for command `(ParalPC c1 c2)`, when the latter has terminated.

Regarding the definition of `bigStepList` for command `(ParalPC c1 c2)` in the case where this command has not terminated, we use function `smallStepList`. This is a function such that, for a command `c` and a state `s`, `smallStepList c s` is a list with the distributions on configurations that ⟨`c, s`⟩ can lead to after a computational step, and is represented by a value of type `[[ConfPC]]`. Its definition can be found in Listing B.22 of Appendix B.2.2. Line 17 of the definition of `bigStepList` follows a similar reasoning to that used for line 7. The output of this function in this line corresponds to a list with all the final distributions that can be obtained from those in list `(smallStepList (ParalPC c1 c2) s)`.

Notice that we have implemented a function `applyBigStepList`, which is responsible for checking if all identifiers present in the command given to `bigStepList` are declared in the state that this function is given, and for checking if all probability values in the command are valid (*i.e.* belong to range $[0, 1]$). `applyBigStepList c s` is a simplification of the result of applying function `bigStepList` to command `c` and state `s`, if `s` is defined on all the free identifiers of `c` and there are no invalid probability values in this command. Otherwise, `applyBigStepList c s` raises an error indicating that such a condition is not fulfilled. For more details about this simplification and the definition of `applyBigStepList`, see Appendix B.2.2 and Listing B.23 in particular.

Big-step with scheduler (one configuration)

We now present function `bigStep`, which given a command `c` of type CpC and a state `s` of type S outputs a value of type `IO (CpC, S)` that returns `(c',s')`, with ⟨`c', s'`⟩ being a terminal configuration obtained from ⟨`c, s`⟩. In order to obtain ⟨`c', s'`⟩ given ⟨`c, s`⟩, we use a scheduler for resolving non-determinism. More specifically, if there are two distributions to which a configuration can lead to, this

scheduler will attribute to each distribution a probability of 0.5. This restriction follows from the fact that, for this implementation, we abstract from the implementation of a more complex scheduler and focus on the implementation of the semantics. Then, given a distribution selected by the scheduler, the probability of a certain configuration being selected is the one attributed to it by the distribution.

The definition of `bigStep` is then as follows:

```
1  bigStep :: CpC -> S -> IO (CpC,S)
2  bigStep SkipPC s = return (SkipPC,s)
3  bigStep (AsgPC i e) s = return (SkipPC, changeSt i n s)
4      where n= (bigStepExp e s)
5  bigStep (SeqPC c1 c2) s = if (term c1 s) then (bigStep c2 s) else do
6      (c1',s') <- smallStep c1 s
7      bigStep (SeqPC c1' c2) s'
8  bigStep (PC p c1 c2) s =
9      let dist = [(1, p),(2, 1-p)]
10         event =  makeEventProb dist
11     in do
12        n <- enact event
13        if (n==1) then (bigStep c1 s) else (bigStep c2 s)
14 bigStep (IfTE_PC b c1 c2) s = if (bigStepBExp b s) then (bigStep c1 s)
15                               else (bigStep c2 s)
16 bigStep (WhDoPC b c) s = if (bigStepBExp b s)
17                          then (bigStep (SeqPC c (WhDoPC b c)) s)
18                          else (bigStep SkipPC s)
19 bigStep (ParalPC c1 c2) s
20     | term (ParalPC c1 c2) s = return (ParalPC c1 c2, s)
21     | term c1 s = bigStep2nd c1 c2 s
22     | term c2 s = bigStep1st c1 c2 s
23     | otherwise = do
24         x <- sched
25         if (x==0) then (bigStep1st c1 c2 s) else (bigStep2nd c1 c2 s)
```

Listing 6.9: Function `bigStep`, from file `SemProbConc.hs`.

The definition of this function is based on that of function `bigStepList` (see Listing 6.6) and also on the rules of Figure 3. Notice that `return x` of type `IO a` will perform an I/O (*i.e.* input or output) action and then return value `x` of type `a`, as indicated by Lipovača [2011] and Prelude module's documentation. The use of the `IO` monad in the type definition of `bigStep` is explained by the fact that it facilitates the implementation of probabilistic events and randomness, whose usefulness is described below.

When `bigStep`'s input corresponds to a configuration $\langle$`c, s`$\rangle$ that can only lead to one terminal configuration, it returns the latter. When its input corresponds to a terminal configuration, it also returns the latter. This explains the definition of `bigStepList` for commands `SkipPC`, `(AsgPC i e)` and `(ParalPC c1 c2)` when this command has terminated.

In the above definition, in cases where a certain configuration $\langle$`c, s`$\rangle$ leads to another known configuration $\langle$`c', s'`$\rangle$, we attribute to `bigStep c s` the value of `bigStep c' s'`. Such happens, for example, in line 5 of the above definition (remembering the fourth rule in Figure 3).

`smallStep` is a function such that, given a command `c` and a state `s`, `(smallStep c s)` is a value of type `IO (CpC, S)` that returns `(c',s')`, with $\langle$`c, s`$\rangle \rightarrow \varphi$ and with $\langle$`c', s'`$\rangle$ being a configuration in the support of φ. We obtain $\langle$`c', s'`$\rangle$ following the same reasoning used for obtaining the return value of function `bigStep`, *i.e.* we use a scheduler for selecting a distribution φ, in whose support is configuration $\langle$`c', s'`$\rangle$. The definition of `smallStep` can be found in Listing B.24 of Appendix B.2.2. In line 6, `(c1',s')` acquires the value returned by `(smallStep c1 s)` and in line 7 `(bigStep (SeqPC c1 c2) s)` returns the value returned by `(bigStep (SeqPC c1' c2) s')`, which is in agreement with the third rule in Figure 3.

The definition of `bigStep` for command `(PC p c1 c2)` makes use of functions `makeEventProb` and `enact` from module Numeric.Probability.Game.Event. The former is a function such that `event` in line 10 corresponds to a probabilistic event where 1 and 2 are two possible outcomes and their probabilities are `p` and `1-p`, respectively, as indicated by Numeric.Probability.Game.Event module's documentation. On the other hand, according to this documentation, `enact` is a function such that `enact event` in line 12 returns the outcome of simulating `event`. Thus the value returned by `bigStepList` has a probability `p` of being that returned by `(bigStep c1 s)` and a probability `1-p` of being that returned by `(bigStep c2 s)`, which agrees with the fifth rule of Figure 3.

Regarding the definition of `bigStep` for command `(ParalPC c1 c2)`, function `bigStep1st` is used for returning a configuration resulting from first executing an atomic step of `c1` and `bigStep2nd` is used for returning one resulting from first executing an atomic step of `c2`. The definition of these auxiliary functions can be found in Listings B.27 and B.28, in Subsection B.2.2. In line 24 of the above definition, `sched` is a function that returns a pseudo-random integer that is either 0 or 1. Its definition can be found in Listing B.29 from Subsection B.2.2. Thus, when both components of `(ParalPC c1 c2)` have not terminated, there is a 0.5 probability of first executing an atomic step of `c1` and an equal probability of first executing an atomic step of `c2`, as line 25 of the above definition shows. The following example illustrates the output of `bigStep` for command `(Paral c1 c2)` when neither `c1` nor `c2` have terminated.

Example 6.2.2. Consider the initial configuration $\langle a := 0 \parallel a := 1,\ [a = 2]\rangle$. Figure 5 shows multiple outputs of `bigStep` for this configuration. One can verify that these outputs correspond either to configuration $\langle \texttt{skip} \parallel \texttt{skip},\ [a = 0]\rangle$ or to $\langle \texttt{skip} \parallel \texttt{skip},\ [a = 1]\rangle$, as expected.

```
*Main> bigStep (ParalPC (AsgPC "a" Zero) (AsgPC "a" One)) [("a",2)]
(ParalPC SkipPC SkipPC,[("a",0)])
*Main> bigStep (ParalPC (AsgPC "a" Zero) (AsgPC "a" One)) [("a",2)]
(ParalPC SkipPC SkipPC,[("a",0)])
*Main> bigStep (ParalPC (AsgPC "a" Zero) (AsgPC "a" One)) [("a",2)]
(ParalPC SkipPC SkipPC,[("a",1)])
*Main> bigStep (ParalPC (AsgPC "a" Zero) (AsgPC "a" One)) [("a",2)]
(ParalPC SkipPC SkipPC,[("a",0)])
*Main> bigStep (ParalPC (AsgPC "a" Zero) (AsgPC "a" One)) [("a",2)]
(ParalPC SkipPC SkipPC,[("a",1)])
```

Figure 5: Multiple results of `bigStep` applied to configuration $\langle (a := 0 \parallel a := 1)\ ,\ [a = 2]\rangle$.

Notice that we have implemented a function `applyBigStep`, which is responsible for checking if all identifiers present in the command given to `bigStep` are declared in the state that this function is given, and for checking if all probability values in the command are valid (*i.e.* belong to range $[0, 1]$). `applyBigStep c s` corresponds to the result of applying function `bigStep` to command `c` and state `s`, if `s` is defined on all the free identifiers of `c` and there are no invalid probability values in this command. Otherwise, `applyBigStep c s` raises an error indicating that such condition is not fulfilled. The definition of `applyBigStep` is presented in Listing B.30 of Subsection B.2.2.

6.3 Concurrent Quantum Language

The implementation described in this section has similarities with the one described in the previous section, as **CQL** and the language in which the previous section focuses on have similar operational semantics.

We start by introducing data types `S`, which represents states of a quantum system, and `Op`, which represents operators. We choose to represent states and operators as complex matrices. In order to implement these data types and for manipulating complex matrices, we use modules Data.Matrix, which is used for matrix operations, and Data.Complex, which is used for dealing with complex numbers. Using these modules, we define both `S` and `Op` as synonyms of type `Matrix (Complex Double)`. The definition of type `S` is now as follows:

```
1 type S = Matrix (Complex Double)
```

Listing 6.10: Definition of data type `S`, from file `GrammarQ.hs`.

Type `Matrix a` represents matrices whose elements are of type `a`, as Data.Matrix module's documentation indicates. On the other hand, type `Complex Double` represents complex numbers whose real and

imaginary parts are represented as `Double` values, according to Data.Complex module's documentation.

We also define data type `L`, in such a way that its values are functions of type `QVar -> Int`. Type `L` represents linking functions.

In the implementation discussed in this section, type `C`, which represents commands, corresponds to that defined in Listing 5.21 of Section 5.2.

6.3.1 Big-step semantics

As in the previous section, we present two approaches for the implementation of the transition rules of the language. The first one does not use a scheduler and therefore allows to obtain all the possible final distributions on configurations that can be derived from an initial configuration. The second approach uses a scheduler that, when deciding between two possible distributions, attributes to each of them a probability of 0.5 (this restriction will be explained in this subsection).

Big-step without scheduler (list of distributions)

We now discuss the implementation of function `bigStepList`. This function is analogous to function `bigStepList` presented in the previous section (see Listing 6.6), with the difference that, in the case of **CQL**, `bigStepList` also receives a linking function as an argument, besides a configuration and a state. Thus, given a command `c`, a linking function `l` and a state `s`, `(bigStepList c l s)` is a list that consists of the final probability distributions on configurations that can be achieved from the initial configuration $\langle$`c, s`$\rangle$, with `l` attributing integer n to the variable that represents the n-th qubit of the system in state `s`. The support of these distributions only contains terminal configurations that can be achieved from $\langle$`c, s`$\rangle$.

We choose to represent `(bigStepList c l s)` as a value of type `[[ProbConf]]`, with data type `ProbConf` being defined as a synonym of type `(Prob, C, S)`. The description of type `Prob` is given in the previous section. `ProbConf` is analogous to type `ConfPC`, which is described in the previous section. Thus we now use `[ProbConf]` for representing distributions on configurations.

We now present the definition of `bigStepList`, which is based on the rules presented in Figure 4, associated with the small-step semantics, and on the definition of `bigStepList` from the previous section (see Listing 6.6).

```
1 bigStepList :: C -> L -> S -> [[ProbConf]]
2 bigStepList Skip l s = [[(1,Skip,s)]]
3 bigStepList (Seq c1 c2) l s
```

```
4     | (term c1 s) = bigStepList c2 l s
5     | otherwise = concat $ map (bigStepD l) (beforeC2 c1 c2 l s)
6 bigStepList (U g vars) l s = [[(1,Skip, applyGate g (qNums vars l) s)]]
7 bigStepList (Meas q c1 c2) l s
8     | (p0 == 0) = bigStepList c2 l s1
9     | (p1 == 0) = bigStepList c1 l s0
10    | otherwise = bigStepD l [(p0, c1, s0), (p1, c2, s1)]
11        where p0 = prob 0 (l(q)) s
12              p1 = prob 1 (l(q)) s
13              s0 = state 0 (l(q)) s
14              s1 = state 1 (l(q)) s
15 bigStepList (Wh q c) l s = bigStepList ( Meas q Skip (Seq c (Wh q c)) ) l s
16 bigStepList (Paral c1 c2) l s
17    | term (Paral c1 c2) s = [[(1, Paral c1 c2, s)]]
18    | otherwise = concat $ map (bigStepD l) (smallStepList (Paral c1 c2) l s)
```

Listing 6.11: Function `bigStepList`, from file `SemQC.hs`.

The definition of `bigStepList` for commands `Skip`, `(Seq c1 c2)` and `(Paral c1 c2)` follows the same reasoning as that used for the definition in Listing 6.6, for the analogous commands. Notice that function `term` maintains the role that is mentioned in the previous section, but now is defined differently. The definiton of `bigStepD` is analogous to that in the previous section (see Listing 6.8), and the same happens with `beforeC2` (see Listing 6.7). The definition of these two functions can be found in Listings B.31 and B.32 of Subsection B.2.3.

Regarding function `smallStepList` in line 18 of the above definition, it is analogous to function `smallStepList` from the implementation of the previous section (see Listing B.22) in the same way as `bigStepList` is analogous to the homonymous function of the previous section. Thus, for a command `c`, a linking function `l`, and a state `s`, with `l` attributing integer n to the variable corresponding to the n-th qubit of the system in state `s`, `smallStepList c l s` is a list with the distributions on configurations that $\langle$`c, s`$\rangle$ can lead to after a computational step, and is represented by a value of type `[[ProbConf]]`. The definition of `smallStepList` can be found in Listing B.33 of Appendix B.2.3.

As to the definition of `bigStepList` for command `(U g vars)`, `qNums` in line 6 of the above definition is a function such that `(qNums vars l)` is the list of integers corresponding to the quantum variables in list `vars`, according to linking function `l`. In that same line, `applyGate` is a function such that `(applyGate g nums s)` corresponds to the state that results from applying gate g to the qubits whose number is in `nums`, when the initial state is `s`. Thus line 6 of the above definition follows from the

seventh rule of Figure 4.

Regarding the definition of `bigStepList` for command `(Meas q c1 c2)`, `p0` and `p1` represent the probabilities of measuring qubit `q` in states $|0\rangle$ and $|1\rangle$, respectively, while `s0` and `s1` correspond to the states of the quantum system in state `s`, after measuring qubit `q` in states $|0\rangle$ and $|1\rangle$, respectively. The definition of `bigStepList` for this command corresponds to the eighth rule of Figure 4.

The definition of `bigStepList` for command `(Wh q c)` can be understood considering the last rule of Figure 4.

We now discuss some relevant auxiliary functions of `bigStepList`. Function `applyGate` is defined as follows:

```
1 applyGate :: G -> [Int] -> S -> S
2 applyGate g nums s
3     | g == H = applyH nums s
4     | g == I = s
5     | g == X = applyX nums s
6     | g == Y = applyY nums s
7     | g == Z = applyZ nums s
8     | g == CNOT = applyCNOT nums s
9     | otherwise = applyCZ nums s
```

Listing 6.12: Function `applyGate`, from file `SemQC.hs`.

Given a list `nums` of integers corresponding to qubits, and an initial state `s` of a quantum system, `(applyH nums s)` is the state of the system after applying an Hadamard gate to the qubits whose number is in list `nums`. The definition of `applyH` follows from Equation 4.20 and is presented next:

```
1 applyH :: [Int] -> S -> S
2 applyH nums s = mult matrix s
3     where matrix = applyToSomeQ had nums (numQubits s)
```

Listing 6.13: Function `applyH`, from file `SemQC.hs`.

`mult matrix s` is the result of multiplying matrices `matrix` and `s`. The definition of function `mult` can be found in Listing B.34 from Subsection B.2.3. `had` is a value of type `Op` that represents the matrix corresponding to the Hadamard gate. `numQubits` is a function that, given a state `s`, outputs the number of qubits of the system in state `s`. Its definition can be found in Listing B.35 of Subsection B.2.3. Lastly, `applyToSomeQ` is a function that receives an operator `op`, a list `nums` of integers representing qubits and an integer `nqubits` corresponding to the number of qubits of a quantum system, and outputs the transformation matrix that corresponds to applying `op` to the state of the qubits whose number is in

nums. This transformation matrix is to be applied to the state of the quantum system. The definition of applyToSomeQ is the following:

```
applyToSomeQ :: Op -> [Int] -> Int -> Op
applyToSomeQ op nums nqubits
    | (nqubits == 1) = if (nums == [1]) then op else if (nums == [])
                          then ident else error "The second argument of function
                          applyToSomeQ can only be [] or [1], if its third
                          argument is 1."
    | otherwise = tensorProduct (replaceByGate op nums listId)
       where listId = gateList ident nqubits
```

Listing 6.14: Function applyToSomeQ, from file SemQC.hs.

ident is an operator corresponding to the 2×2 Identity matrix. Thus, if there is only one qubit in the system, and list nums is empty, the output of applyToSomeQ corresponds to this Identity matrix, as line 4 of the above equation indicates. tensorProduct is a function that receives as argument a list of operators and outputs an operator corresponding to their tensor product. Its definition can be found in Listing B.36 of Subsection B.2.3. gateList in line 7 of the above definition is a function that, given a value op of type Op and a value n of type Int, outputs a list with n elements, all equal to op. Thus listId represents a list of nqubits operators, all representing the 2×2 Identity matrix. Lastly, replaceByGate is a function such that (replaceByGate op nums l) in the above definition is a list of operators corresponding to l after replacing by operator op the elements of l whose indexes belong to integer list nums. Its definition can be found in Listing B.40 of Appendix B.2.3.

In the definition of applyGate (see Listing 6.12), functions applyX, applyY and applyZ are analogous to applyH in terms of their role and definition. These three functions are responsible for applying gates *X*, *Y* and *Z*, as their names suggest. In that definition, line 4 is explained by the fact that applying the Identity gate to some qubits of a system does not change the state of this system. As to function applyCNOT used in this definition, given a list nums of two integers, the first one representing the control qubit and the second one representing the target one, and an intial state s of a quantum system, (applyCNOT nums s) is the state of the system after applying a *CNOT* gate to the two qubits. applyCZ is analogous to applyCNOT regarding their role and definition, with applyCZ being responsible for the application of gate *CZ*. The definition of applyCNOT is based on Equation 4.22 and is as follows:

```
applyCNOT :: [Int] -> S -> S
applyCNOT l s
    | (length l /= 2) = error "First argument of function applyCNOT must be
                                 a list with two elements."
```

```
5     | otherwise = if (control /= target) then mult matrix s else error
6                   "The control and target qubits given as argument to
7                    function applyCNOT cannot be the same."
8         where control = head l
9               target = last l
10              nqubits = numQubits s
11              listId = gateList ident nqubits
12              matrix0 = applyToSomeQ m0 [control] nqubits
13              matrix1 = tensorProduct $ replaceByGate x [target]
14                         (replaceByGate m1 [control] listId)
15              matrix = sumMatrices matrix0 matrix1
```

Listing 6.15: Function `applyCNOT`, from file `SemQC.hs`.

Notice that `applyCNOT` raises an error if the list it receives has a number of elements different than 2, or if the list contains two equal elements. Variables `control` and `target` in the above definition correspond, respectively, to the first and last elements of non-empty list `l`, as the description of functions `head` and `last` in Prelude module's documentation indicates. `m0` in line 12 is an operator corresponding to matrix A_0, which represents $|0\rangle\langle 0|$, and analogously for `m1` in line 14. Argument `x`, on the other hand, is an operator representing the σ_x Pauli matrix. Therefore `matrix0` and `matrix1` represent, respectively, the first and second terms of the sum of the transformation matrix in Equation 4.22. `sumMatrices` is a function such that `matrix` in the above definition is the matrix resulting from the sum of `matrix0` and `matrix1`. Its definition can be found in Listing B.41 of Subsection B.2.3. The definition of `applyCZ` can be found in Listing B.42 of Subsection B.2.3.

We now move on to functions `prob` and `state`, used in the definition of `bigStepList` for command `(Meas q c1 c2)` (see Listing 6.11). The definition of `prob` is as follows:

```
1 prob :: Int -> Int -> S -> Prob
2 prob i n s
3     | (i == 0 || i == 1) = realPart $ matrixToElem $ mult mToStateDagger
4                                 mToState
5     | otherwise = error ((show i)++" cannot be the first argument of
6                              function prob.")
7         where nqubits = numQubits s
8               m = if (i==0) then applyToSomeQ m0 [n] nqubits
9                   else applyToSomeQ m1 [n] nqubits
10              mToState = mult m s
```

```
11              mToStateDagger = dagger mToState
```

Listing 6.16: Function `prob`, from file `SemQC.hs`.

`prob` is a function such that `(prob i n s)` is the probability of measuring qubit number `n` in state $|\mathtt{i}\rangle$, with $\mathtt{i} \in \{0, 1\}$, if the initial state of the system of qubits is `s`. `dagger mToState` in the above definition corresponds to the Hermitian conjugate of `mToState`. The definition of `dagger` can be found in Listing B.43 of Subsection B.2.3. Thus `(mult mToStateDagger mToState)` corresponds to $p(b, i, n)$ in Equation 4.31, in the form of a 1×1 matrix. `matrixToElem` is a function such that, when receiving a matrix of type `Matrix a` with only one element, outputs that same element. Regarding `realPart`, it is a function such that, given a value of type `Complex Double`, outputs its real part as a value of type `Double`, as indicated by Data.Complex module's documentation. Thus `(prob i n s)` outputs $p(b, i, n)$ as a `Prob` value, for $\mathtt{i} \in \{0, 1\}$.

The definition of `state` is as follows:

```
1  state :: Int -> Int -> S -> S
2  state i n s
3      | (i == 0 || i == 1) = fromLists (finalState)
4      | otherwise = error ((show i)++" cannot be the first argument of
5                              function state.")
6          where nqubits = numQubits s
7                m = if (i==0) then applyToSomeQ m0 [n] nqubits
8                    else applyToSomeQ m1 [n] nqubits
9                mToState = mult m s
10               mToStateL = toLists mToState
11               p = prob i n s
12               finalState = map ( map ( divideBy (realToComp (sqrt p)) ) )
13                            mToStateL
```

Listing 6.17: Function `state`, from file `SemQC.hs`.

`state` is a function such that `(state i n s)` is the state of the system whose initial state is `s`, after measuring its `n`-th qubit in state $|\mathtt{i}\rangle$, with $\mathtt{i} \in \{0, 1\}$. In the above equation, `finalState` corresponds to $|\psi'(b, i, n)\rangle$ from Equation 4.32, in the form of a value of type `S`. Functions `fromLists` and `toLists` belong to module Data.Matrix and are described in Subsection B.2.3. Notice that `sqrt` is a function from module Prelude such that `sqrt p` is the squared root of `p`. `realToComp` turns a `Double` value into its corresponding `Complex Double` value, and its definition makes use of the `:+` constructor included in Data.Complex module. Lastly, `divideBy` is a function such that `divideBy a b` is a `Complex Double` value resulting from dividing `b` by `a`, with `a` and `b` being `Complex Double` values.

Notice that we have implemented a function `applyBigStepList`, such that `(applyBigStep-List c l s)` is a simplification of the result of applying function `bigStepList` to command `c`, linking function `l` and state `s`. For more details about this simplification and the definition of `applyBigStepList`, see Appendix B.2.3 and Listing B.44 in particular.

We have also implemented a function called `bigStepListFile`, which, for a command `c` of the language written in a file `f`, a linking function `l` and a state `s`, `(bigStepListFile f l s)` prints on the terminal a `String` value corresponding to `(applyBigStepList c l s)`.

Big-step with scheduler (one configuration)

We now discuss the implementation of function `bigStep`. This function is analogous to function `bigStep` presented in the previous section (see Listing 6.9), with the difference that, in the case of **CQL**, `bigStep` also receives a linking function as an argument, besides a configuration and a state. Thus, given a command `c`, a linking function `l` and a state `s`, `bigStep` is a function that outputs a value of type `IO (C, S)` that returns `(c',s')`, with $\langle$`c', s'`$\rangle$ being a terminal configuration obtained from $\langle$`c, s`$\rangle$, and with `l` attributing integer n to the variable that represents the n-th qubit of the system in state `s`. For this project we abstract from the implementation of an appropriate scheduler that aims to minimize noise in quantum computing. As such, we implement a simple scheduler such that, if there are two distributions to which a configuration can lead to, this scheduler will attribute to each distribution a probability of 0.5, just like in the previous section. Then, given a distribution selected by the scheduler, the probability of a certain configuration being selected is the one attributed to it by the distribution. The definition of `bigStep` is then as follows:

```
1  bigStep :: C -> L -> S -> IO (C,S)
2  bigStep Skip l s = return (Skip,s)
3  bigStep (Seq c1 c2) l s = if (term c1 s) then bigStep c2 l s else do
4      (c1',s') <- smallStep c1 l s
5      bigStep (Seq c1' c2) l s'
6  bigStep (U g vars) l s = return (Skip, applyGate g (qNums vars l) s)
7  bigStep (Meas q c1 c2) l s
8      | (p0 == 0) = bigStep c2 l s1
9      | (p1 == 0) = bigStep c1 l s0
10     | otherwise = do
11             n <- enact event
12             if (n==1) then (bigStep c1 l s0) else (bigStep c2 l s1)
13         where p0 = prob 0 (l(q)) s
```

```
14                p1 = prob 1 (l(q)) s
15                s0 = state 0 (l(q)) s
16                s1 = state 1 (l(q)) s
17                dist = [(1, p0),(2, p1)]
18                event = makeEventProb dist
19 bigStep (Wh q c) l s = bigStep ( Meas q Skip (Seq c (Wh q c)) ) l s
20 bigStep (Paral c1 c2) l s
21     | term (Paral c1 c2) s = return (Paral c1 c2, s)
22     | term c1 s = bigStep2nd c1 c2 l s
23     | term c2 s = bigStep1st c1 c2 l s
24     | otherwise = do
25         x <- sched
26         if (x==0) then (bigStep1st c1 c2 l s) else (bigStep2nd c1 c2 l s)
```

Listing 6.18: Function `bigStep`, from file `SemQC.hs`.

The definition of this function is based on that of function `bigStepList` (see Listing 6.11) and on the rules in Figure 4 as well.

Function `smallStep` used in this definition is analogous to the homonymous function described in the previous section, with the difference that, in the case of **CQL**, `smallStep` also receives a linking function as an argument, besides a configuration and a state. Thus, given a command `c`, a linking function `l` and a state `s`, with `l` attributing integer n to the variable that represents the n-th qubit of the system in state `s`, `smallStep` is a function such that `(smallStep c l s)` is a value of type `IO (C, S)` that returns `(c',s')`, with $\langle$`c, s`$\rangle \rightarrow \varphi$ and with $\langle$`c', s'`$\rangle$ being a configuration in the support of φ. We obtain $\langle$`c', s'`$\rangle$ following the same reasoning as that used for obtaining the return value of function `bigStep`, *i.e.* we use a scheduler for selecting a distribution φ, in whose support is configuration $\langle$`c', s'`$\rangle$. The definition of `smallStep` can be found in Listing B.45 of Appendix B.2.3.

Auxiliary functions `bigStep1st` and `bigStep2nd` are analogous to the homonymous functions that are used in the definition of `bigStep` presented in the previous section, and their definition can be found in Listings B.48 and B.49 of Subsection B.2.3.

6.3.2 Histogram

We have also implemented functions for building histograms that, after obtaining multiple results of `bigStep` for a certain initial configuration, show the frequency of each result. `histogramBigStep` is a function such that, for an integer number `n`, a command `c`, a linking function `l`, and a state `s`, plots an histogram whose input data is a list representing `n` results of `(bigStep c l s)`. `histogramBigStep`

also prints on the terminal a histogram caption, which contains the configuration corresponding to each label of the histogram. It has the following definition:

```
histogramBigStep :: Int -> C -> L -> S -> IO ExitCode
histogramBigStep n c l s = do
    input <- listBigStep n c l s
    putStrLn "------------------------------------------\n"
    putStrLn "Histogram Caption:"
    putStrLn ""
    caption 1 (diffResults input)
    putStrLn "------------------------------------------"
    histogramInt (confIntoDouble input) "Results of the big-step semantics"
```

Listing 6.19: Function `histogramBigStep`, from file `SemQC.hs`.

In the above definition, `listBigStep n c l s` returns a list with `n` results of `(bigStep c l s)`. The definition of `listBigStep` can be found in Listing B.50 of Appendix B.2.3. Lines 4 to 8 in the above definition are used to print on the terminal the histogram caption. The description of function `putStrLn` can be found in the Prelude module and in Lipovača [2011]. Given a list `l` of type `[(C,S)]`, `caption 1 l` prints on the terminal a caption with a list that attributes to each element of `l` a label of the histogram. The definition of function `caption` can be found in Listing B.51 of Appendix B.2.3. `diffResults` is a function such that, given a list `l` of type `[(C,S)]`, outputs the list that results from removing from `l` all the repeated elements. Lastly, `histogramInt` is a function such that `(histogramInt dataSet t)` plots an histogram whose input is list `dataSet` of type `[Double]` and whose title is `t`, as long as `dataSet` is non-empty. Otherwise, an error is raised. In the histogram, each different result has a label of the form `<conf` x`>`, where x is an integer. The definition of `histogramInt` can be found in Listing B.52 of Appendix B.2.3. `confIntoDouble` is a function such that `(confIntoDouble input)` corresponds to a list of type `[Double]`, whose i-th element corresponds to an integer representing the i-th element of `input`. For example, given arbitrary configurations `c1`, `c2` and `c3` of type `(C,S)`:

```
confIntoDouble [c1,c2,c3,c1,c3] = [1.0,2.0,3.0,1.0,3.0]
```

Notice that the integer representing each configuration of `input` is determined in such a way that a configuration represented by integer x will have label `<conf` x`>`, in the histogram.

We have also implemented a function called `histBigStepFile` such that, given an integer `n` representing a number of executions, a file `f` with a command corresponding to `c`, a linking function `l` and a state `s`, `(histBigStepFile n f l s)` plots the same histogram as `(histogramBigStep n c l s)`, with `l` attributing integer n to the variable that represents the n-th qubit of the system in state `s`.

In Chapter 7, some examples of histograms output by function `histBigStepFile` are given; they are shown in Figures 8, 11, 13 and 15 of that chapter.

Chapter 7

Examples and Case Study

In this chapter we present some examples of the outputs provided by our interpreter for **CQL**, as well as a case study that focuses on quantum teleportation, where the usefulness of our interpreter is exemplified.

7.1 Examples

In this section we explore applications of our interpreter for **CQL**. These examples serve the goal of showing the functionality of our implementation.

7.1.1 Example 1: A Simple Quantum Program

This example focuses on applying our interpreter to a simple **CQL** program written in file `cql1.txt`, whose content is shown in Figure 6. In the name of the file `cql1.txt`, `1` alludes to this first example. Specifically the command presented in Figure 6 expresses the application of an Hadamard gate to qubit

```
H(q);
Meas(q) -> (skip,skip)
```

Figure 6: Content of file `cql1.txt`.

`q`, followed by the measurement of the state of this qubit. Figure 7 shows the result of applying function `applyBigStepFile` to file `cql1.txt`, to a linking function `l` that attributes integer 1 to qubit `q`, and to state `state0`, which represents state $|0\rangle$. For defining `l`, we use the following line of code:

```
1 l("q") = 1
```

Listing 7.1: Definition of linking function `l` from file `SemQC.hs`.

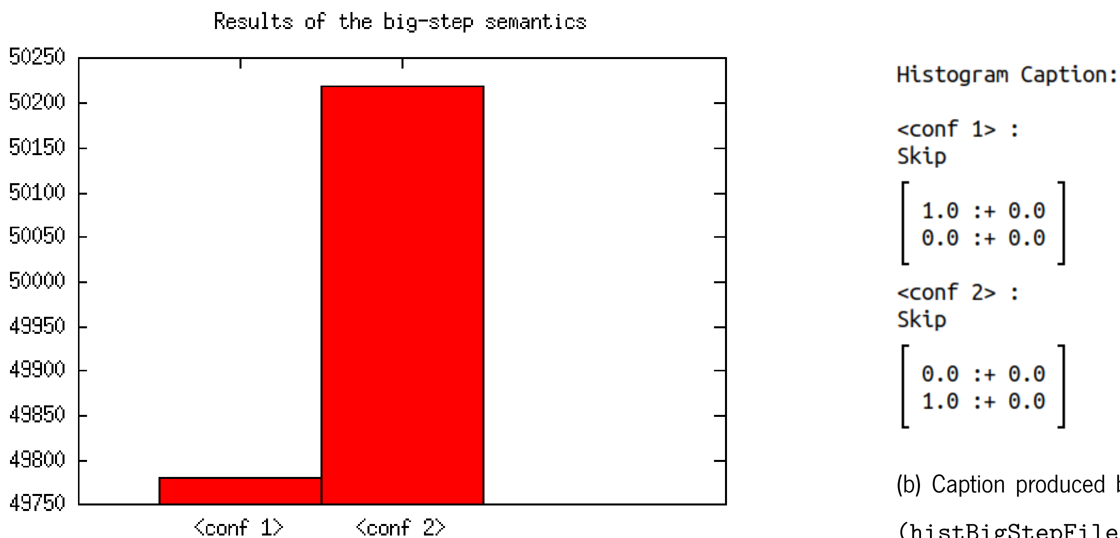
The output in Figure 7 corresponds to a list of distributions consisting of only one distribution, given by $0.5 \cdot \langle \texttt{skip},\ |0\rangle\rangle + 0.5 \cdot \langle \texttt{skip},\ |1\rangle\rangle$, where the probability values are rounded to one decimal place. From this output, one concludes that the command in file `cql1.txt` with initial state $|0\rangle$ yields this

```
*ParserSemQ> bigStepListFile "cql1.txt" l state0
[[(0.4999999999999999,Skip,
 ⎡ 1.0 :+ 0.0 ⎤
 ⎣ 0.0 :+ 0.0 ⎦),
(0.4999999999999999,Skip,
 ⎡ 0.0 :+ 0.0 ⎤
 ⎣ 1.0 :+ 0.0 ⎦)]]
```

Figure 7: Result of `bigStepListFile` applied to file `cql1.txt`, linking function `l` and state `state0`.

final probability distribution. Indeed the Hadamard gate applied to state $|0\rangle$ produces state $|+\rangle$, and the probability of obtaining result $|0\rangle$ from measuring a qubit in state $|+\rangle$ is the same as the probability of obtaining result $|1\rangle$, which is 0.5. Thus the result presented in Figure 7 is in agreement with what is expected.

We now focus on the application of function `histBigStepFile` to the same command, linking function and state. The result is presented in Figure 8. The histogram in Figure 8 represents the results



(a) Histogram plotted by `(histBigStepFile 100000 "cql1.txt" l state0)`. Notice that the labels in the vertical axis of the histogram are in the range 49750 to 50250.

(b) Caption produced by `(histBigStepFile 100000 "cql1.txt" l state0)`.

Figure 8: Result of `(histBigStepFile 100000 "cql1.txt" l state0)`. Each result `<conf x>` in Figure 8a, with `x` being an integer, has a caption in Figure 8b, with the command and state (in matrix form) corresponding to the result.

of executing the command `cql1.txt` 10^5 times. Specifically it shows that, in the 10^5 times that the command is executed, the output of such an execution is terminal configuration $\langle \texttt{skip}, |0\rangle\rangle$ around 49775 of those times, and it is terminal configuration $\langle \texttt{skip}, |1\rangle\rangle$ around 50225 of those 10^5 times.

The frequency of these two outputs is thus close to 50%. Therefore the results shown in the histogram also agree with our expectations based on the theory, and with the results from Figure 7 as well.

7.1.2 Example 2: Introducing Concurrency

The next example focuses on applying our interpreter to a simple **CQL** concurrent program shown in Figure 9.

```
Meas (q) -> (H(q), I(q) || X(q))
```

Figure 9: Content of file `cql2.txt`.

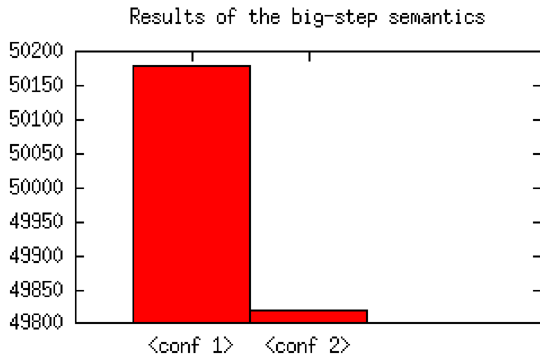
Figure 10 shows the result of applying function `applyBigStepFile` to file `cql2.txt`, to linking function `l` and to state `statePlus`, which represents state $|+\rangle$. Thus, it shows the result of `applyBigStepList` for the inital configuration from Example 4.2.1 and linking function `l`. The result in

```
*ParserSemQ> bigStepListFile "cql2.txt" l statePlus
[[(0.4999999999999999,Skip,
 ⎡ 0.7071067811865475 :+ 0.0 ⎤
 ⎣ 0.7071067811865475 :+ 0.0 ⎦),
(0.4999999999999999,Paral Skip Skip,
 ⎡ 1.0 :+ 0.0 ⎤
 ⎣ 0.0 :+ 0.0 ⎦)],
[(0.4999999999999999,Skip,
 ⎡ 0.7071067811865475 :+ 0.0 ⎤
 ⎣ 0.7071067811865475 :+ 0.0 ⎦),
(0.4999999999999999,Paral Skip Skip,
 ⎡ 1.0 :+ 0.0 ⎤
 ⎣ 0.0 :+ 0.0 ⎦)]]
```

Figure 10: Result of `bigStepListFile` applied to file `cql2.txt`, linking function `l` and state `statePlus`.

Figure 10 expresses that there are two final distributions on configurations that can be obtained from this initial configuration, both equal (in approximate terms) to $0.5 \cdot \langle \texttt{skip}, |+\rangle\rangle + 0.5 \cdot \langle \texttt{skip} || \texttt{skip}, |0\rangle\rangle$. This agrees with the probabilistic automaton from Example 4.2.1. Notice that, although the two distributions are equal in this case, they would be different if the *I* gate in the command was replaced by a *H* gate, for example.

We now focus on the application of function `histBigStepFile` to the same file, linking function and state. The result of such application is presented in Figure 11. The histogram in Figure 11 represents the results of executing the command 10^5 times. Specifically it shows that, in the 10^5 times that the



(a) Histogram plotted by (`histBigStepFile 100000 "cql2.txt" l statePlus`). Notice that the labels in the vertical axis of the histogram are in the range 49800 to 50200.

```
Histogram Caption:

<conf 1> :
Paral Skip Skip
[ 1.0 :+ 0.0 ]
[ 0.0 :+ 0.0 ]

<conf 2> :
Skip
[ 0.7071067811865475 :+ 0.0 ]
[ 0.7071067811865475 :+ 0.0 ]
```

(b) Caption produced by (`histBigStepFile 100000 "cql2.txt" l statePlus`).

Figure 11: Result of (`histBigStepFile 100000 "cql2.txt" l statePlus`). Each result `<conf x>` in Figure 11a, with `x` being an integer, has a caption in Figure 11b, with the command and state (in matrix form) corresponding to the result.

command is executed, the output of such an execution is terminal configuration $\langle \texttt{skip} || \texttt{skip},\ |0\rangle\rangle$ in around 50175 of those times, and it is (in approximate terms) terminal configuration $\langle \texttt{skip},\ |+\rangle\rangle$ in around 49825 of those 10^5 times. Therefore the frequency of these two outputs is close to 50%, and the results agree with the results shown in Figure 10, and with Equations 3.9 and 3.10 as well.

7.2 Case study: Quantum Teleportation

In this case study we apply our tool to a program representing the quantum teleportation technique (described in Subsection 4.1.4). This program is written in file `qTelepSeq.txt`. The name of this file is explained by the fact that this program represents quantum teleportation (which explains the `qTelep`) and corresponds to a sequence of commands (which explains the `Seq`).

```
CNOT(q1,q2) ; H(q1) ;
Meas (q2) -> (skip, X(q3));
Meas (q1) -> (skip, Z(q3))
```

Figure 12: Content of file `qTelepSeq.txt`.

Let `lT` represent a linking function that attributes 1 to `q1`, 2 to `q2`, 3 to `q3` and 4 to `q4` (for now, `q4` will not be necessary). Consider as well a value `qTelepInitState` of type `S` that represents the initial state $|\psi\rangle_{in}$ of the system of three qubits, which is given by Equation 4.34, considering that $a = b = \frac{1}{\sqrt{2}}$,

i.e.:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right). \tag{7.1}$$

Figure 19 in Appendix C shows the result of applying function `applyBigStepFile` to file `qTelepSeq-.txt`, to linking function `lT` and to state `qTelepInitState`. From the result in Figure 19, one concludes that, given an initial configuration corresponding to the command in file `qTelepSeq.txt` and the initial state $|\psi\rangle_{in}$, the final probability distribution obtained from that configuration is, approximately, $0.25 \cdot \langle \texttt{skip},\ |00\psi\rangle\rangle + 0.25 \cdot \langle \texttt{skip},\ |01\psi\rangle\rangle + 0.25 \cdot \langle \texttt{skip},\ |10\psi\rangle\rangle + 0.25 \cdot \langle \texttt{skip},\ |11\psi\rangle\rangle$. We expect that, by the end of the teleportation procedure, the state of Bob's qubit becomes $|\psi\rangle$ and the states of Alice's qubits is either $|00\rangle$, $|01\rangle$, $|10\rangle$ or $|11\rangle$, with the probability of each of these states being 0.25, taking into account Equation 4.35. Thus the result in Figure 19 agrees with what is expected from theory. We now focus on the application of function `histBigStepFile` to the same file, linking function and state as those used as argument of `applyBigStepFile`. The result of such application is presented in Figure 13. In this figure, `<conf 1>`, `<conf 2>`, `<conf 3>` and `<conf 4>` correspond (in approximate

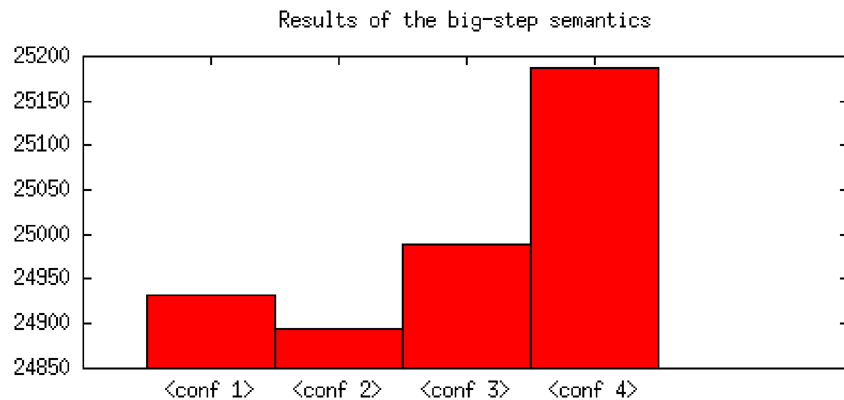


Figure 13: Histogram plotted by `(histBigStepFile 100000 "qTelepSeq.txt" lT qTelepInitState)`. Notice that the labels in the vertical axis of the histogram are in the range 24850 to 25200. Each result `<conf x>` in this histogram, with `x` being an integer, has a caption in Figure 20 of Appendix C, with the command and state (in matrix form) corresponding to the result.

terms) to terminal configurations $\langle \texttt{skip},\ |01\psi\rangle\rangle$, $\langle \texttt{skip},\ |11\psi\rangle\rangle$, $\langle \texttt{skip},\ |10\psi\rangle\rangle$ and $\langle \texttt{skip},\ |00\psi\rangle\rangle$, respectively, as shown by the caption in Figure 20 of Appendix C. The histogram in Figure 13 shows that, in the 10^5 times that the command in file `qTelepSeq.txt` is executed, the frequency in which each of these four configurations is the output is close to 25%. Therefore the results of this histogram agree with the results of the application of function `applyBigStepFile` to file `qTelepSeq.txt`, linking function `lT` and to state `qTelepInitState`, which are shown in Figure 19 from Appendix C.

Now let us consider the case where a user of our interpreter decides to test it is possible to minimize

noise in quantum teleportation by transforming the sequence $H(q1); \texttt{Meas}(q2) \to (\texttt{skip}, X(q3))$ into $H(q1) || \texttt{Meas}(q2) \to (\texttt{skip}, X(q3))$ and letting a scheduler decide what the best order of execution is for the latter command is, in order to minimize noise. We now focus on the program that represents such an attempt of introducing concurrency into the program in qTelepSeq.txt. Through our tool, we will evaluate if the program resulting from such an attempt produces the same results as the original one. The program corresponding to this attempt is written in file qTelepAttempt.txt, whose content is shown in Figure 14.

```
CNOT(q1,q2) ;
(H(q1) || Meas (q2) -> (skip, X(q3)));
Meas (q1) -> (skip, Z(q3))
```

Figure 14: Content of file qTelepAttempt.txt.

Applying function histBigStepFile to file qTelepAttempt.txt, to linking function lT and to state qTelepInitState, we obtain the histogram shown in Figure 15. Comparing the histograms in

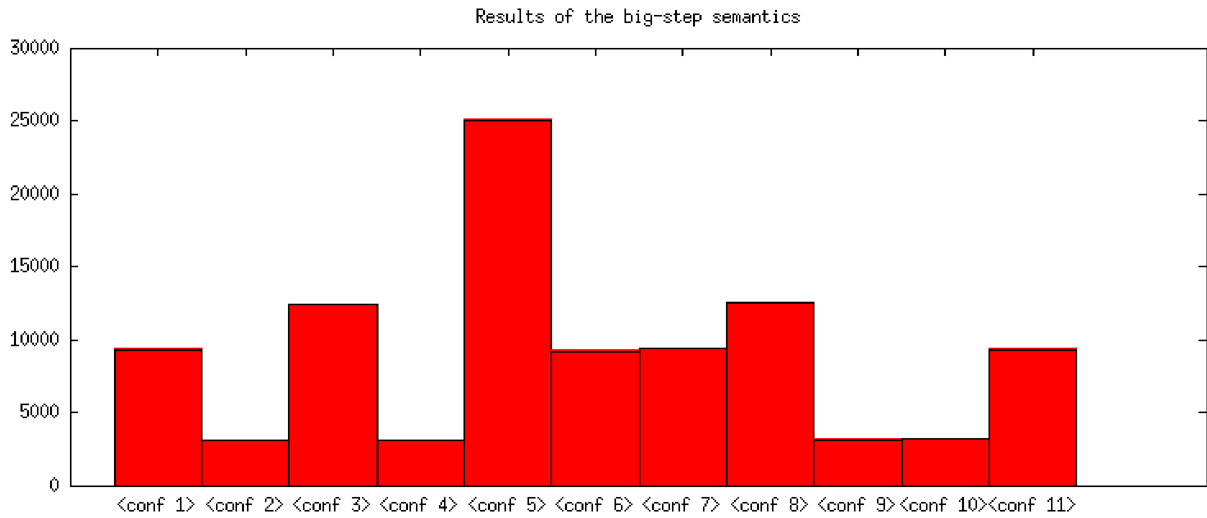


Figure 15: Histogram plotted by (histBigStepFile 100000 "qTelepAttempt.txt" lT qTelepInitState). Notice that the labels in the vertical axis of the histogram are in the range 0 to 30000. Each result <conf x> in this histogram, with x being an integer, has a caption, which is shown in Figures 21, 22 and 23 of Appendix C, with the command and state (in matrix form) corresponding to the result.

Figures 13 and 15, we conclude that there are configurations that can be obtained from the command in file qTelepAttempt.txt that were not obtained from the command in file qTelepSeq.txt, for the same initial state and for 10^5 executions of the latter command. In fact, one of such configurations is <conf 5>, which corresponds, in approximate terms, to (see Figure 22):

$$\langle \texttt{skip}, \frac{1}{\sqrt{2}}(|000\rangle + |010\rangle) \rangle,$$

which is not a terminal configuration expected from theory. Therefore we can conclude that the program in file `qTelepAttempt.txt` does not correctly represent the quantum teleportation technique. Thus, if it is used as program representing this technique, an appropriate scheduler controlling its execution would have to rule out the paths of execution that lead to the undesired results.

Chapter 8

Conclusions and future work

8.1 Conclusions

A main contribution of this dissertation project is an implementation that allows to simulate the execution of **CQL** programs. Specifically it allows to obtain histograms that show the results of multiple executions of a command of the language, and also allows to obtain all the possible final results of executing a command, for a given initial state.

Consequently, the tool we implemented offers a way to test if the introduction of concurrency in a certain quantum program does not change its outcome. For example, in the case study discussed in Section 7.2, we show that our tool allows to verify that the described attempt of incorporating concurrency in the quantum teleportation technique is not correct, as its results do not match the expected ones. Therefore our implementation can be helpful when trying to introduce concurrency to a quantum program with the aim of reducing the probability of noise affecting its results. However, for this latter case, the use of an appropriate scheduler is also necessary. The implementation of such a scheduler is part of the future work.

From this project, it is also possible to understand in a more concrete way the advantage offered by Parsec regarding the implementation of parsers in a modular way. Indeed Parsec allows to combinate more primitive parsers for implementing a more complex one. Also, the fact that parsers built using this tool can also be responsible for the lexical analysis allows our implementation to be more condensed.

This dissertation also allows to conclude that the operational semantics of a language indeed facilitates its implementation. In particular, it allows to better understand how to use the small-step semantics of a language in order to obtain the final result of executing a program. Moreover we conclude that some theoretical concepts related to probabilistic automata can be useful for representing such an execution, and for predicting its results.

All in all, this dissertation project allows to take conclusions on how to implement a concurrent quan-

tum language using Haskell, and how this implementation can be useful for testing if the introduction of concurrency in quantum programs does not change their intended input-output behaviour.

8.2 Future work

An important part of the future work is the implementation of a scheduler for our concurrent programs that aims to minimize the amount of time in which each qubit is needed, while maintaining the expected input-output behaviour of these programs. For example, let us consider a quantum program of the form $H(q); P; X(q)$, where P represents a program that is independent of qubit q. In order to minimize the probability of noise affecting the results of this program, one can convert it into a concurrent one of the form $H(q) \| P \| X(q)$, while using an appropriate scheduler for reordering the execution of the program. This scheduler would determine that the optimal order of execution would correspond to either that of program $H(q); X(q); P$ or that of $P; H(q); X(q)$.

Some aspects of our implementation of **CQL** can also be improved. For example, the results of executing **CQL** programs can be made easier to understand if the states are represented in Dirac notation instead of matrix notation. This improvement would also allow the captions of histograms to be more compact. Besides, it may be possible to improve the way in which the histograms are implemented in Haskell, in such a way that the labels in their vertical axis always start at 0. Moreover, a possible improvement is to remove from quantum states any global phase they may have when being displayed, since such phases can be ignored. For example, remember that state $|\psi'\rangle$ in Equation 4.29 can be understood as $|0\rangle$.

Lastly another possible improvement of the tool is to simulate the effect of noise in quantum computing, in order to better evaluate if the introduction of concurrency together with an appropriate scheduler can indeed minimize noise.

Bibliography

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2007. ISBN 0321486811.

Christel Baier and Holger Hermanns. *Weak Bisimulation for Fully Probabilistic Processes*. Number 99-12 in CTIT technical report series. Centre for Telematics and Information Technology (CTIT), Netherlands, 1999.

Stephen Barnett. *Quantum Information*. Oxford Master Series in Physics. OUP Oxford, 2009. ISBN 9780198527626.

Stephen Brookes. Full abstraction for a shared-variable parallel language. *Information and Computation*, 127(2):145–163, 1996. ISSN 0890-5401. doi: https://doi.org/10.1006/inco.1996.0056. URL https://www.sciencedirect.com/science/article/pii/S0890540196900565.

David W. Bustard. Concepts of Concurrent Programming. 4 1990. doi: 10.1184/R1/6572699. v1. URL https://kilthub.cmu.edu/articles/report/Concepts_of_Concurrent_Programming/6572699.

Inês Dias. AnInterpreterForAConcurrentQuantumLanguage (GitHub Repository), 2024. URL https://github.com/ines-correiadias/AnInterpreterForAConcurrentQuantumLanguage. [Online].

R. Fasold and J. Connor-Linton. *An Introduction to Language and Linguistics*. Cambridge University Press, 2006. ISBN 9780521847681. URL https://books.google.pt/books?id=dlzthEZGkmsC.

Vítor Fernandes. Semantics for quantum concurrency, 2024. [Unpublished PhD work].

M. Fernández. *Programming Languages and Operational Semantics: A Concise Overview*. Undergraduate Topics in Computer Science. Springer London, 2014. ISBN 9781447163688. URL https://books.google.pt/books?id=dzi5BQAAQBAJ.

Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21 (6/7):467–488, 1982.

John T. Gill. Computational complexity of probabilistic turing machines. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, page 91–95, New York, NY, USA, 1974. Association for Computing Machinery. ISBN 9781450374231. doi: 10.1145/800119.803889. URL https://doi.org/10.1145/800119.803889.

Alexander Graham. *Kronecker Products and Matrix Calculus with Applications*. Dover Books on Mathematics. Dover Publications, 2018. ISBN 9780486824178. URL https://books.google.pt/books?id=DMBYDwAAQBAJ.

P.B. Hansen. *The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls*. Springer New York, 2013. ISBN 9781475734720. URL https://books.google.pt/books?id=c__1BwAAQBAJ.

HaskellWiki. Combinator pattern, 2007. URL https://wiki.haskell.org/Combinator_pattern. [Online].

HaskellWiki. Converting numbers, 2016. URL https://wiki.haskell.org/Converting_numbers. [Online].

HaskellWiki. Combinator, 2021. URL https://wiki.haskell.org/Combinator. [Online].

H. Hüttel. *Transitions and Trees: An Introduction to Structural Operational Semantics*. Cambridge University Press, 2010. ISBN 9781139788595. URL https://books.google.pt/books?id=f9zmrShQj3YC.

IBM. IBM Debuts Next-Generation Quantum Processor & IBM Quantum System Two, Extends Roadmap to Advance Era of Quantum Utility, December 2023a. URL https://newsroom.ibm.com/2023-12-04-IBM-Debuts-Next-Generation-Quantum-Processor-IBM-Quantum-System-Two,-Extends-Roadmap-to-Advance-Era-of-Quantum-Utility. [Online].

IBM. Numbers, 2023b. URL https://www.ibm.com/docs/en/idr/11.4.0?topic=types-numbers. [Online].

Mike izbicki. Graphics.histogram, 2012. URL https://hackage.haskell.org/package/Histogram-0.1.0.2/docs/Graphics-Histogram.html. [Online].

Bart Jacobs. *Introduction to Coalgebra: Towards Mathematics of States and Observation*. Cambridge tracts in theoretical computer science. Cambridge University Press, 2017. ISBN 9781107177895.

Claire Jones and Gordon D Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 186–187. IEEE Computer Society, 1989.

Arttu Kangas. Birth of the compiler. 2023.

Swagata Karmakar, Ashmita Dey, and Indrajit Saha. Use of quantum-inspired metaheuristics during last two decades. In *2017 7th International Conference on Communication Systems and Network Technologies (CSNT)*, pages 272–278. IEEE, 2017.

Youngseok Kim, Andrew Eddins, Sajant Anand, Ken Xuan Wei, Ewout van den Berg, Sami Rosenblatt, Hasan Nayfeh, Yantao Wu, Michael Zaletel, Kristan Temme, and Abhinav Kandala. Evidence for the utility of quantum computing before fault tolerance. *Nature*, 618(7965):500–505, Jun 2023. ISSN 1476-4687. doi: 10.1038/s41586-023-06096-3. URL https://doi.org/10.1038/s41586-023-06096-3.

Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3): 328–350, 1981. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(81)90036-2. URL https://www.sciencedirect.com/science/article/pii/0022000081900362.

Marco Lanzagorta and Jeffrey Uhlmann. *Quantum Computer Science*. Synthesis Lectures on Quantum Computing. Springer International Publishing, 2022. ISBN 9783031025129. URL https://books.google.pt/books?id=fYByEAAAQBAJ.

Daan Leijen, Paolo Martini, and Antoine Latter. parsec: Monadic parser combinators, 2022. URL https://hackage.haskell.org/package/parsec-3.1.15.1. [Online].

Miran Lipovača. *Learn You a Haskell For Great Good!* no starch press, 2011. Online available at http://learnyouahaskell.com/.

Natalia López and Manuel Núñez. *An Overview of Probabilistic Process Algebras and Their Equivalences*, pages 89–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24611-4. doi: 10.1007/978-3-540-24611-4_3. URL https://doi.org/10.1007/978-3-540-24611-4_3.

John Martinis and Sergio Boixo. Quantum Supremacy Using a Programmable Superconducting Processor, October 2019. URL https://blog.research.google/2019/10/quantum-supremacy-using-programmable.html. [Online].

David H. McIntyre, Corinne A. Manogue, Janet Tate, and Oregon State University. *Quantum Mechanics: A Paradigms Approach*. Always learning. Pearson, 2012. ISBN 9780321765796.

Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. doi: 10.1017/CBO9780511976667.

Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real world Haskell - code you can believe in*, chapter 16. Using Parsec. O'Reilly Media, 2008. Online available at http://book.realworldhaskell.org/read/using-parsec.html.

John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, August 2018. ISSN 2521-327X. doi: 10.22331/q-2018-08-06-79. URL http://dx.doi.org/10.22331/q-2018-08-06-79.

Michael O. Rabin. Probabilistic automata. *Information and Control*, 6(3):230–245, 1963. ISSN 0019-9958. doi: https://doi.org/10.1016/S0019-9958(63)90290-0. URL https://www.sciencedirect.com/science/article/pii/S0019995863902900.

S. Sakr and M. Gaber. *Large Scale and Big Data: Processing and Management*. An Auerbach book. CRC Press, 2014. ISBN 9781466581517. URL https://books.google.pt/books?id=JiPcBQAAQBAJ.

F.B. Schneider. *On Concurrent Programming*. Texts in Computer Science. Springer New York, 2012. ISBN 9781461218302. URL https://books.google.pt/books?id=Mu7SBwAAQBAJ.

R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. In B. Jonsson and J. Parrow, editors, *Proc. 5th International Conference on Concurrency Theory (CONCUR'94)*, volume 836 of *LNCS*, pages 481–496. Springer, 1994.

Roberto Segala. *Modeling and verification of randomized distributed real-time systems*. PhD thesis, Massachusetts Institute of Technology, 1995.

Peter Selinger and Benoît Valiron. On a fully abstract model for a quantum linear functional language. *Electronic Notes in Theoretical Computer Science*, 210:123–137, 2008.

Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, 1999. doi: 10.1137/S0036144598347011. URL https://doi.org/10.1137/S0036144598347011.

Ana Sokolova and Erik P. de Vink. *Probabilistic Automata: System Types, Parallel Composition and Comparison*, pages 1–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. ISBN 978-3-540-24611-4. doi: 10.1007/978-3-540-24611-4_1. URL https://doi.org/10.1007/978-3-540-24611-4_1.

Amrita Som and Amlan Chakrabarti. A new bsqdd approach for synthesis of quantum circuit. In *2011 International Symposium on Electronic System Design*, pages 212–216. IEEE, 2011.

M.J. Sottile, T.G. Mattson, and C.E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman & Hall/CRC Computational Science. CRC Press, 2009. ISBN 9781420072143. URL https://books.google.pt/books?id=J5-ckoCgc3IC.

Henning Thielemann. Demo.hs, 2022. URL https://hackage.haskell.org/package/gnuplot-0.5.7/src/src/Demo.hs. [Online].

Daniele Varacca. *Probability, nondeterminism and concurrency: two denotational models for probabilistic computation*. BRICS, 2003.

Daniele Varacca and Glynn Winskel. Distributing probability over non-determinism. *Mathematical structures in computer science*, 16(1):87–113, 2006.

Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

Mingsheng Ying. *Foundations of quantum programming*. Morgan Kaufmann, 2016.

Appendices

Appendix A
User Manual

This appendix presents a succinct user manual of our implementation of **CQL**, as well as some guidelines about the necessary Haskell modules for using it. Here, we focus on two functions of our implementation, `bigStepListFile` and `histBigStepFile`, from which it is possible to obtain the results of executing a program in two different ways that are be described below.

Function `bigStepListFile` from file `ParserSemQ.hs` allows to obtain the final probability distributions on configurations that can be achieved from a given initial configuration. More concretely, given a a command `c` of the language written in a file `f`, a linking function `l` and a state `s`, `(bigStepListFile f l s)` prints on the terminal a `String` value corresponding to a list of type `[[ProbConf]]` that consists of the final probability distributions that can be achieved from the initial configuration $\langle$`c, s`$\rangle$, with `l` attributing integer n to the variable that represents the n-th qubit of the system in state `s`. More information about function `bigStepListFile` can be found in Section 6.3. Figure 16 shows an example of use of this function. This example is explained in more detail in Subsection 7.1.1.

```
*ParserSemQ> bigStepListFile "cql1.txt" l state0
[[(0.4999999999999999,Skip,
┌            ┐
│ 1.0 :+ 0.0 │
│ 0.0 :+ 0.0 │
└            ┘),
(0.4999999999999999,Skip,
┌            ┐
│ 0.0 :+ 0.0 │
│ 1.0 :+ 0.0 │
└            ┘)]]
```

Figure 16: Result of `bigStepListFile` applied to file `cql1.txt`, linking function `l` and state `state0`. The content of this file is presented in Figure 17, `l` attributes integer 1 to variable `q` and `state0` corresponds to state $|0\rangle$.

```
1 H(q);
2 Meas(q) -> (skip,skip)
```

Figure 17: Content of file `cql1.txt`.

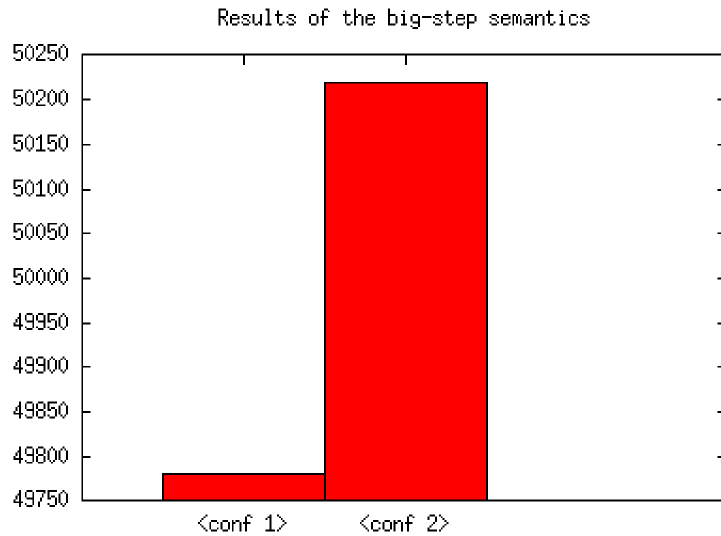
In order to use function `bigStepListFile`, it is necessary to use some modules that can be obtained from https://hackage.haskell.org/:

- Module Text.ParserCombinators.Parsec from package parsec;
- Module Data.Matrix from package matrix;
- Module Data.Complex from package base.

Function `histBigStepFile` from file `ParserSemQ.hs` is used for outputting histograms that represent the results of several executions of a command, for a given initial state. Specifically, given an integer `n` representing a number of executions, a file `f` with a command corresponding to `c`, a linking function `l` and a state `s`, with `l` attributing integer n to the variable that represents the n-th qubit of the system in state `s`, `(histBigStepFile n f l s)` plots an histogram whose input data is a list representing `n` results of executing command `c` when the initial state is `s`, and also prints on the terminal a histogram caption, which contains the configuration corresponding to each label of the histogram. More information about function `histBigStepFile` can be found in Section 6.3. Figure 18 shows an example of use of `histBigStepFile`, with 18b showing the arguments given to the function, which are the same as those given to `bigStepListFile` in the example in Figure 16 (with the exception of argument `100000`). This example is explained in more detail in Subsection 7.1.1.

The same modules needed for using function `bigStepListFile` (presented above in this appendix) are needed for using function `histBigStepFile`, as well as the following modules, which can also be obtained from `https://hackage.haskell.org/`:

- Module System.Random from package random;
- Module Numeric.Probability.Game.Event from package game-probability;
- Modules System.Exit from package base;
- Module Graphics.Histogram from package Histogram;
- Module Graphics.Gnuplot.Frame.OptionSet from package gnuplot.



(a) Histogram plotted by (histBigStepFile 100000 "cql1.txt" l state0). Notice that the labels in the vertical axis of the histogram are in the range 49750 to 50250.

```
*ParserSemQ> histBigStepFile 100000 "cql1.txt" l state0
-------------------------------------------

Histogram Caption:

<conf 1> :
Skip
⎡ 1.0 :+ 0.0 ⎤
⎣ 0.0 :+ 0.0 ⎦
<conf 2> :
Skip
⎡ 0.0 :+ 0.0 ⎤
⎣ 1.0 :+ 0.0 ⎦

-------------------------------------------
ExitSuccess
```

(b) Output of (histBigStepFile 100000 "cql1.txt" l state0) in the terminal. The output between dashed lines corresponds to the histogram caption.

Figure 18: Result of (histBigStepFile 100000 "cql1.txt" l state0). Each result <conf x> in Figure 18a, with x being an integer, has a caption in Figure 18b, with the command and state (in matrix form) corresponding to the result.

Appendix B
Minor implementation details

This appendix discusses some minor details of our implementation.

B.1 Implementation of parsers

B.1.1 Basic Parallel Language

This subsection is relative to the implementation discussed in Section 5.1. The further description of Parsec's functions provided in this section is based on Leijen et al. [2022].

The definition of `cAuxToC` is the following:

```
cAuxToC :: CAux -> C
cAuxToC (SkipAux) = Skip
cAuxToC (AsgAux i e) = Asg i e
cAuxToC (SeqAux c1 c2) = Seq (cAuxToC c1) (cAuxToC c2)
cAuxToC (ParalAux c1 c2) = Paral (cAuxToC c1) (cAuxToC c2)
cAuxToC (IfTE_CAux b c1 c2) = IfTE_C (bAuxToB b) (cAuxToC c1) (cAuxToC c2)
cAuxToC (WhDoAux b c) = WhDo (bAuxToB b) (cAuxToC c)
cAuxToC (StrC s) = error "Unable to convert to C."
```

Listing B.1: Function `cAuxToC`, from file `ParserBrookes.hs`.

`cAuxToC` converts each possible value of type `CAux` into the corresponding value of type `C`, with the exception of `CAux` values that start with `StrC`, since they have no corresponding value of type `C`. In the above definition, `bAuxToB` is a function analogous to `cAuxToC` – it turns a `BAux` value into the corresponding `B` value.

The definition of `stringToC` is as follows:

```
stringToC :: String -> CAux
stringToC input = let right = parse parseCAux "(unknown)" input
                  in eitherToT (right)
```

Listing B.2: Function `stringToC`, from file `ParserBrookes.hs`.

In the definition above, `right` will either start with `Left` or `Right`. In case `input` can be successfully parsed by `parseCAux`, `right` will be `Right a`, where `a` is the `CAux` value corresponding to `input`, and thus the value that `stringToC input` must have. That is precisely the reason why `stringToC` returns `eitherToT(right)`. `eitherToT` has the following definition:

```haskell
eitherToT :: Either ParseError t -> t
eitherToT (Right x) = x
eitherToT (Left x) = error "Parse error."
```

Listing B.3: Function `eitherToT`, from file `ParserBE_Brookes.hs`.

If `input` cannot be successfully parsed by `parseCAux`, `right` will be equal to `Left b`, where b represents a parse error, and function `error` in `eitherToT` will stop execution and display an error message that includes `"Parse error."`, as the description of function `error` in Haskell's Prelude module's documentation indicates.

The definition of `parseESelect` is as follows:

```haskell
parseESelect = try(pEPlus) <|> try(pEZero) <|> try(pEOne) <|> try(pEIf) <|>
               try(pEId) <|> pEParen
```

Listing B.4: Function `parseESelect`, from file `ParserBE_Brookes.hs`.

`pEPlus` parses integer expressions with value $E_1 + E_2$, `pEZero` parses those with value 0, `pEOne` parses those with value 1, `pEIf` parses those with value `if` B `then` E_1 `else` E_2, `pEId` parses those with value I and `pEParen` parses those between parentheses. Thus, `parseESelect` parses integer expressions, with or without parentheses around them. We have considered that, similarly to the case of `if` commands, valid `if` expressions contain curly brackets surrounding the expressions constituting them (*e.g.* `"if i<=j then {0} else {1}"` is considered a valid `if` expression).

The order in which the auxiliary parsers appear in the definition of `parseESelect` is not arbitrary. In order to guarantee that `parseESelect` always parses completely a sum of integer expressions, `pEPlus` must be tried before all the other auxiliary parsers of `parseESelect`. (*e.g.* if the input of `parseESelect` is `"0 + 1"` and `pEZero` is tried first, `" + 1"` will not be consumed by `parseESelect`).

The definition of `parseBSelect` is as follows:

```haskell
parseBSelect = try(pBAnd) <|> try(pBNot) <|> try(pBLeq) <|> try(pBTrue) <|>
               try(pBFalse) <|> pBParen
```

Listing B.5: Function `parseBSelect`, from file `ParserBE_Brookes.hs`.

`pBAnd` parses Boolean expressions with value B_1 & B_2, `pBNot` parses those with value $\neg B$, `pBLeq` parses those with value $E_1 \leq E_2$, `pBTrue` parses those with value `true`, `pBFalse` parses those with value `false` and `pBParen` parses those between parentheses. Thus `parseBSelect` parses Boolean expressions, with or without parentheses around them.

There is a reason for the order in which the auxiliary parsers appear in the definition of `parseBSelect`. In order to guarantee that `parseBSelect` always parses completely a conjunction of Boolean expressions, `pBAnd` must be tried before `pBLeq` (*e.g.* if the input of `parseBSelect` corresponds to $E_1 \leq$

E_2 & B and pBLeq is tried first, B will not be consumed by parseBSelect). For a similar reason pBTrue, pBFalse and pBParen must also be tried after pBAnd. Since we considered that the negation of Boolean expressions has priority over their conjunction (*i.e.* $\neg B_1$ & B_2 is interpreted as $(\neg B_1)$ & B_2), pBAnd must also be tried before pBNot.

The definition of parser entersOnly and those of its auxiliary parsers are the following:

```
1 entersOnly = many parseEnter
```

Listing B.6: Function entersOnly, from file ParserBE_Brookes.hs.

```
1 parseEnter = satisfy isEnter
```

Listing B.7: Function parseEnter, from file ParserBE_Brookes.hs.

```
1 isEnter :: Char -> Bool
2 isEnter c = (c == '\n')
```

Listing B.8: Function isEnter, from file ParserBE_Brookes.hs.

satisfy is a function such that satisfy f is a parser that succeeds when applied to a character for which function f returns True, with f :: Char -> Bool. Thus parseEnter parses one newline character. spacesAndEnters and spacesOnly are defined in an analogous way to that used for defining entersOnly.

The definition of separateElems is the following:

```
1 separateElems = try(atLeastOneSpace >> entersOnly) <|> atLeastOneEnter
```

Listing B.9: Function separateElems, from file ParserBE_Brookes.hs.

Function atLeastOneSpace parses one or more spaces, while atLeastOneEnter parses one or more newline characters. Parser atLeastOneSpace >> entersOnly applies parser atLeastOneSpace followed by parser entersOnly, as the description of function (>>) in the Control.Monad module's documentation indicates.

We finish this Appendix subsection by presenting the definition of parser separateOrJoined.

```
1 separateOrJoined = try(separateElems) <|> string ""
```

Listing B.10: Function separateOrJoined, from file ParserBE_Brookes.hs.

B.1.2 Concurrent Quantum Language

This subsection is relative to the implementation discussed in Section 5.2.

The definition of cAuxToC is the following:

```
1 cAuxToC :: CAux -> C
2 cAuxToC (SkipAux) = Skip
3 cAuxToC (SeqAux c1 c2) = Seq (cAuxToC c1) (cAuxToC c2)
4 cAuxToC (UAux g l) = U g l
5 cAuxToC (MeasAux q c1 c2) = Meas q (cAuxToC c1) (cAuxToC c2)
```

```
6 cAuxToC (WhAux q c) = Wh q (cAuxToC c)
7 cAuxToC (ParalAux c1 c2) = Paral (cAuxToC c1) (cAuxToC c2)
8 cAuxToC (Str s) = error "Unable to convert to C."
```

Listing B.11: Function `cAuxToC`, from file `ParserQ.hs`.

`cAuxToC` converts each value of type `CAux` into the corresponding value of type `C`, with the exception of `CAux` values that start with `Str`, as they have no corresponding value of type `C`.

The definition of `parseQVars` is as follows:

```
1 parseQVars = do
2     spacesOnly
3     char ','
4     separateOrJoined
5     q <- parseQVar
6     qs <- (try (parseQVars) <|> return [])
7     return (q:qs)
```

Listing B.12: Function `parseQVars`, from file `ParserQ.hs`.

B.2 Implementation of the semantics

B.2.1 Basic Parallel Language

This subsection concerns the implementation discussed in Section 6.1.

The definition of `bigStepExp` is presented next. In this definition, `getValue` is a function such that `getValue s i` is equal to the integer value that state `s` attributes to identifier `i`.

```
1 bigStepExp :: E -> S -> Integer
2 bigStepExp Zero s = 0
3 bigStepExp One s = 1
4 bigStepExp (Id i) s = getValue s i
5 bigStepExp (IfTE_E b e1 e2) s
6     | bigStepBExp b s = bigStepExp e1 s
7     | otherwise = bigStepExp e2 s
8 bigStepExp (PlusE e1 e2) s = (bigStepExp e1 s) + (bigStepExp e2 s)
```

Listing B.13: Function `bigStepExp`, from file `SemBE_Brookes.hs`.

The definition of function `term` is the following, according to the transition rules in Figure 2:

```
1 term :: C -> S -> Bool
2 term Skip s = True
3 term (Paral c1 c2) s = (term c1 s) && (term c2 s)
4 term c s = False
```

Listing B.14: Function `term`, from file `SemBrookes.hs`.

In the above definition, && is a function that represents the conjunction of Boolean values, and its description can be found in Prelude module's documentation.

The definition of `bigStepExp` is as follows:

```
1 bigStepBExp :: B -> S -> Bool
2 bigStepBExp (BTrue) s = True
3 bigStepBExp (BFalse) s = False
4 bigStepBExp (Not b) s = not (bigStepBExp b s)
5 bigStepBExp (And b1 b2) s = (bigStepBExp b1 s) && (bigStepBExp b2 s)
6 bigStepBExp (Leq e1 e2) s = (bigStepExp e1 s) <= (bigStepExp e2 s)
```

Listing B.15: Function `bigStepBExp`, from file `SemBE_Brookes.hs`.

Function `paralBigStep` is defined as follows:

```
1 paralBigStep :: C -> (C,S) -> [(C,S)]
2 paralBigStep c (c',s') = bigStepList (Paral c' c) s'
```

Listing B.16: Function `paralBigStep`, from file `SemBrookes.hs`.

We now discuss the implementation of function `smallStepList`, which is based on the transition rules presented in Figure 2, associated with the small-step semantics, and has some similarities with that of `bigStepList` (presented in Listing 6.2).

```
1  smallStepList :: C -> S -> [(C,S)]
2  smallStepList Skip s = [(Skip,s)]
3  smallStepList (Asg i e) s = [(Skip, (changeSt i n s))]
4      where n = (bigStepExp e s)
5  smallStepList (Seq c1 c2) s = if (term c1 s) then [(c2,s)]
6                                else (map (lastInSeq c2) (smallStepList c1 s))
7  smallStepList (IfTE_C b c1 c2) s = if (bigStepBExp b s) then [(c1,s)]
8                                     else [(c2,s)]
9  smallStepList (WhDo b c) s = [(IfTE_C b (Seq c (WhDo b c)) Skip, s)]
10 smallStepList (Paral c1 c2) s
11     | term (Paral c1 c2) s = [(Paral c1 c2, s)]
12     | term c1 s = map (paral c1) (smallStepList c2 s)
13     | term c2 s = map (paral c2) (smallStepList c1 s)
14     | otherwise = (map (paral c2) (smallStepList c1 s))
15                   ++ (map (paral c1) (smallStepList c2 s))
```

Listing B.17: Function `smallStepList`, from file `SemBrookes.hs`.

Just like happened when defining function `bigStepList`, we have considered that, if the arguments of function `smallStepList` correspond to a successfully terminated configuration, then it returns a list with just that same configuration. Regarding the definition of `smallStepList` for the sequential command `Seq c1 c2`, `lastInSeq` is an auxiliary function such that `lastInSeq c2 (c1',s')` corresponds to configuration ⟨`c1'`; `c2`, `s'`⟩, with `c1'` and `c2` being arbitrary commands and `s'` being an arbitrary state. In this way, since `smallStepList c1 s` corresponds to a list with all possible values of ⟨`c1'`, `s'`⟩, with ⟨`c1`, `s`⟩ → ⟨`c1'`, `s'`⟩, then `(map (lastInSeq c2) (smallStepList c1 s))` will be equal to a list representing all possible values of ⟨`c1'`; `c2`, `s'`⟩. For example, if `smallStepList c1 s = [(c11', s1'),(c12', s2')]`, then `smallStepList (Seq c1 c2) s` will be `[(Seq c11' c2, s1'),(Seq c12' c2, s2')]`.

Let us now consider the definition of `smallStepList` for command `Paral c1 c2`. It was written following an analogous logic to the one used for defining `smallStepList` for command `Seq c1`

c2. paral is an auxiliary function such that paral c1 (c2', s') corresponds to configuration $\langle$c1$\|$c2', s'$\rangle$, with c1 and c2' being arbitrary commands and s' being an arbitrary state. Hence, if $\langle$c1, s$\rangle$ is a successfully terminated configuration and $\langle$c2, s$\rangle$ is not, then smallStep (Paral c1 c2) s will be equal to a list corresponding to all possible values of $\langle$c1$\|$c2', s'$\rangle$, with $\langle$c2, s$\rangle \rightarrow \langle$c2', s'$\rangle$. This agrees with the fact that, in this case, only the transition corresponding to the ninth rule of Figure 2 can be executed. Analogously, if $\langle$c2, s$\rangle$ is successfully terminated and $\langle$c1, s$\rangle$ is not, only the transition corresponding to the eighth rule of Figure 2 can occur, and smallStep (Paral c1 c2) s will be equal to a list corresponding to all possible values of $\langle$c1'$\|$c2, s'$\rangle$, with $\langle$c1, s$\rangle \rightarrow \langle$c1', s'$\rangle$. Lastly, focusing now on the case where neither $\langle$c1, s$\rangle$ nor $\langle$c2, s$\rangle$ are successfully terminated configurations, we define smallStepList (Paral c1 c2) s as being a list with all all possible values of $\langle$c1$\|$c2', s'$\rangle$, with $\langle$c2, s$\rangle \rightarrow \langle$c2', s'$\rangle$, as well as all possible values of $\langle$c1'$\|$c2, s'$\rangle$, with $\langle$c1, s$\rangle \rightarrow \langle$c1', s'$\rangle$.

The definition of applySem is the following:

```
1 applySem :: (C -> S -> a) -> C -> S ->  a
2 applySem f c s = if (belong (freeC c) s) then (f c s) else error ("Not all free
3                    identifiers in "++(show c)++" are part of state "++(show s))
```

Listing B.18: Function applySem, from file SemBrookes.hs.

(freeC c) is the set of free identifiers in command c. Function freeC is defined in Listing B.19, in agreement with Equations 3.4. belong is a function such that belong listStr s, with listStr being a list of String values and s being a state, is only True if no string in listStr is missing from state s.

```
1 freeC :: C -> [String]
2 freeC Skip = []
3 freeC (Asg i e) = i : (freeE e)
4 freeC (Seq c1 c2) = (freeC c1) ++ (freeC c2)
5 freeC (IfTE_C b c1 c2) = (freeB b) ++ (freeC c1) ++ (freeC c2)
6 freeC (WhDo b c) = (freeB b) ++ (freeC c)
7 freeC (Paral c1 c2) = (freeC c1) ++ (freeC c2)
```

Listing B.19: Function freeC, from file SemBrookes.hs.

In the above definition, freeE and freeB are functions analogous to freeC – freeE e is the set of free identifiers in integer expression e and freeB b is the set of free identifiers in Boolean expression b. They are defined in the following manner:

```
1 freeE :: E -> [String]
2 freeE Zero = []
3 freeE One = []
4 freeE (Id i) = [i]
5 freeE (PlusE e1 e2) = (freeE e1) ++ (freeE e2)
6 freeE (IfTE_E b e1 e2) = (freeB b) ++ (freeE e1) ++ (freeE e2)
```

Listing B.20: Function freeE, from file SemBE_Brookes.hs.

```
freeB :: B -> [String]
freeB BTrue = []
freeB BFalse = []
freeB (Not b) = freeB b
freeB (And b1 b2) = (freeB b1) ++ (freeB b2)
freeB (Leq e1 e2) = (freeE e1) ++ (freeE e2)
```

Listing B.21: Function `freeB`, from file `SemBE_Brookes.hs`.

B.2.2 Basic Parallel Language with Probabilistic Choice

This subsection concerns the implementation discussed in Section 6.2.

Function `smallStepList` is based on the transition rules presented in Figure 3, associated with the small-step semantics, and has some similarities with that of `bigStepList` (presented in Listing 6.6).

```
smallStepList :: CpC -> S -> [[ConfPC]]
smallStepList SkipPC s = [[(1, SkipPC, s)]]
smallStepList (AsgPC i e) s = [[(1, SkipPC, changeSt i n s)]]
    where n= (bigStepExp e s)
smallStepList (SeqPC c1 c2) s = if (term c1 s) then [[(1, c2, s)]]
                                else map (lastInSeqProb c2) (smallStepList c1 s)
smallStepList (PC p c1 c2) s = [[(p,c1,s), (1-p,c2,s)]]
smallStepList (IfTE_PC b c1 c2) s = if (bigStepBExp b s) then [[(1,c1,s)]]
                                        else [[(1,c2,s)]]
smallStepList (WhDoPC b c) s = [[(1, IfTE_PC b (SeqPC c (WhDoPC b c)) SkipPC,
                                    s)]]
smallStepList (ParalPC c1 c2) s
    | term (ParalPC c1 c2) s = [[(1, ParalPC c1 c2, s)]]
    | term c1 s = map (paral c1) (smallStepList c2 s)
    | term c2 s = map (paral c2) (smallStepList c1 s)
    | otherwise = (map (paral c2) (smallStepList c1 s))
                  ++ (map (paral c1) (smallStepList c2 s))
```

Listing B.22: Function `smallStepList`, from file `SemProbConc.hs`.

For example, $\langle C_1 \oplus_p C_2,\ s\rangle$ can only transition to distribution $p \cdot \langle C_1,\ s\rangle + (1-p) \cdot \langle C_2,\ s\rangle$, which explains line 7 of the above definition. The definition of this function is analogous to that of `smallStepList` in Listing B.17. However, `lastInSeqProb` is now an auxiliary function such that, given a command `c'` and a distribution on configurations `d`, `lastInSeqProb c' d` is a value of type `[ConfPC]` corresponding to `d` after replacing each element `(p,c,s)` by `(p, Seq c c',s)`. Thus line 6 of the above definition is in agreement with the third rule of Figure 3. Besides, `paral` is now a function such that, for a command `c'` and a distribution on configurations `d`, `paral c' d` is a value of type `[ConfPC]` corresponding to `d` after replacing each element `(p,c,s)` by `(p, Paral c' c,s)`. Notice that $C_1 || C_2$ is equivalent to $C_2 || C_1$. Thus lines 14 to 17 of the above definition agree with the last three rules of Figure 3.

We now present the definition of function `applyBigStepList`:

```
applyBigStepList :: CpC -> S ->  [[ConfPC]]
applyBigStepList c s = if (belong (freeC c) s) && (validProb c)
                       then simplify (bigStepList c s)
```

```
4                          else error (errorSem c s)
```

Listing B.23: Function applyBigStepList, from file SemProbConc.hs.

This definition is similar to that of function applySem presented in Listing B.18. Function belong is the same as that used in the definition of function applySem, and freeC's definition is analogous to that presented in Listing B.19, in the previous subsection. validProb is an auxiliary function such that, given a command c, validProb c is a Bool value that is True if and only if c does not contain any invalid probability value (*i.e.* outside the $[0,1]$ range). On the other hand, simplify is a function such that, given a list of distributions l of type [[ConfPC]], simplify l is the list of distributions resulting from eliminating from l all configurations with probability 0 and, for each distribution in l, joining values of type ConfPC with the same command and state into just one value of this type. For example, for arbitrary states s1, s2 and s3:

```
simplify [[(0,SkipPC,s1), (1,AsgPC "a" One,s2)], [(0.2,SkipPC,s3),
           (0.8,SkipPC,s3)]] =
[[(1,AsgPC "a" One,s2)], [(1,SkipPC,s3)]]
```

Lastly, errorSem is a function such that, given a command c and a state s, errorSem c s is a string expressing that the condition in line 2 of the above definition is not fulfilled.

We now present the definition of smallStep. It follows a similar reasoning to that used for defining bigStep (whose definition is presented in Listing 6.9) and is also based on the rules from Figure 3:

```
1  smallStep :: CpC -> S -> IO (CpC,S)
2  smallStep SkipPC s = return (SkipPC,s)
3  smallStep (AsgPC i e) s = return (SkipPC, changeSt i n s)
4      where n = (bigStepExp e s)
5  smallStep (SeqPC c1 c2) s = if (term c1 s) then (return (c2,s)) else do
6      (c1',s') <- smallStep c1 s
7      return (SeqPC c1' c2, s')
8  smallStep (PC p c1 c2) s =
9      let dist = [(1, p),(2, 1-p)]
10         event =  makeEventProb dist
11     in do
12        n <- enact event
13        return ( if (n==1) then (c1,s) else (c2,s) )
14 smallStep (IfTE_PC b c1 c2) s = if (bigStepBExp b s) then (return (c1,s))
15                                 else (return (c2,s))
16 smallStep (WhDoPC b c) s = return (IfTE_PC b (SeqPC c (WhDoPC b c)) SkipPC,
17                                    s)
18 smallStep (ParalPC c1 c2) s
19     | term (ParalPC c1 c2) s = return (ParalPC c1 c2, s)
20     | term c1 s = smallStep2nd c1 c2 s
21     | term c2 s = smallStep1st c1 c2 s
22     | otherwise = do
23         x <- sched
24         if (x==0) then (smallStep1st c1 c2 s) else (smallStep2nd c1 c2 s)
```

Listing B.24: Function smallStep, from file SemProbConc.hs.

`smallStep1st` and `smallStep2nd` in the above definition are defined as follows:

```
1 smallStep1st :: CpC -> CpC -> S -> IO (CpC,S)
2 smallStep1st c1 c2 s = do
3     (c1', s') <- smallStep c1 s
4     return (ParalPC c1' c2, s')
```

Listing B.25: Function `smallStep1st`, from file `SemProbConc.hs`.

```
1 smallStep2nd :: CpC -> CpC -> S -> IO (CpC,S)
2 smallStep2nd c1 c2 s = do
3     (c2', s') <- smallStep c2 s
4     return (ParalPC c1 c2', s')
```

Listing B.26: Function `smallStep2nd`, from file `SemProbConc.hs`.

Auxiliary functions `bigStep1st` and `bigStep2nd` are defined as follows:

```
1 bigStep1st :: CpC -> CpC -> S -> IO (CpC,S)
2 bigStep1st c1 c2 s = do
3     (c1',s') <- smallStep c1 s
4     bigStep (ParalPC c1' c2) s'
```

Listing B.27: Function `bigStep1st`, from file `SemProbConc.hs`.

```
1 bigStep2nd :: CpC -> CpC -> S -> IO (CpC,S)
2 bigStep2nd c1 c2 s = do
3     (c2',s') <- smallStep c2 s
4     bigStep (ParalPC c1 c2') s'
```

Listing B.28: Function `bigStep2nd`, from file `SemProbConc.hs`.

Lastly, `sched` has the following definition:

```
1 sched :: IO Int
2 sched = do
3     g <- getStdGen
4     newStdGen
5     return (fst (randomR (0,1) g))
```

Listing B.29: Function `sched`, from file `SemProbConc.hs`.

Functions `getStdGen`, `newStdGen` and `randomR` belong to library System.Random. According to this library's documentation and Lipovača [2011], `getStdGen` is a function such that g is the global pseudo-random number generator, `newStdGen` updates the value of this generator and `randomR (0,1) g` outputs a pair whose first element is a pseudo-random value in the range $[0, 1]$, with each value in this range having equal associated probability. `fst` is a function that outputs the first element of a pair given as input, as indicated by Prelude module's documentation. Notice that `sched` returns an `Int` value, and as such it can only return either 0 or 1.

```
1 applyBigStep :: CpC -> S -> IO (CpC,S)
2 applyBigStep c s = if (belong (freeC c) s) && (validProb c) then bigStep c s
3                    else error (errorSem c s)
```

Listing B.30: Function `applyBigStep`, from file `SemProbConc.hs`.

The above definition is similar to that of `applyBigStepList` (see Listing B.23). However, the former does not employ function `simplify`, while the latter definition does.

B.2.3 Concurrent Quantum Language

This subsection concerns the implementation discussed in Section 6.3.

The definition of `bigStepD` is the following:

```
bigStepD :: L -> [ProbConf] -> [[ProbConf]]
bigStepD l [] = [[]]
bigStepD l ((p,c,s):t) = [ (multProb p a) ++ b | a <- (bigStepList c l s),
                                                 b <- (bigStepD l t) ]
```

Listing B.31: Function `bigStepD`, from file `SemQC.hs`.

In the above definition, `multProb` is a function such that `(multProb p a)` is a `[ProbConf]` value resulting from multiplying by `p` all probabilities in distribution `a`.

Function `beforeC2` is defined as follows:

```
beforeC2 :: C -> C -> L -> S -> [[ProbConf]]
beforeC2 c1 c2 l s = let afterC1 = bigStepList c1 l s
                     in (map (replaceBy c2) afterC1)
```

Listing B.32: Function `beforeC2`, from file `SemQC.hs`.

In the above definition, `replaceBy` is a function such that, for a given command `c2` and a list `l` of type `[ProbConf]`, `(replaceBy c2 l)` is a list of this type resulting from replacing by `c2` each `c` in all elements `(p,c,s)` of `l`.

Function `smallStepList` is based on the transition rules from Figure 4, associated with the small-step semantics, and has some similarities with that of `bigStepList` (presented in Listing 6.11).

```
smallStepList :: C -> L -> S -> [[ProbConf]]
smallStepList Skip l s = [[(1,Skip,s)]]
smallStepList (Seq c1 c2) l s = if (term c1 s) then [[(1,c2,s)]]
                                else map (lastInSeqProb c2) (smallStepList c1 l s)
smallStepList (U g vars) l s = [[(1,Skip, applyGate g (qNums vars l) s)]]
smallStepList (Meas q c1 c2) l s
    | (p0 == 0) = [[(p1, c2, s1)]]
    | (p1 == 0) = [[(p0, c1, s0)]]
    | otherwise = [[(p0, c1, s0), (p1, c2, s1)]]
      where p0 = prob 0 (l(q)) s
            p1 = prob 1 (l(q)) s
            s0 = state 0 (l(q)) s
            s1 = state 1 (l(q)) s
smallStepList (Wh q c) l s = [[(1, Meas q Skip (Seq c (Wh q c)), s)]]
smallStepList (Paral c1 c2) l s
    | term (Paral c1 c2) s = [[(1, Paral c1 c2, s)]]
    | term c1 s = map (paral c1) (smallStepList c2 l s)
    | term c2 s = map (paral c2) (smallStepList c1 l s)
    | otherwise = (map (paral c2) (smallStepList c1 l s))
              ++ (map (paral c1) (smallStepList c2 l s))
```

Listing B.33: Function `smallStepList`, from file `SemQC.hs`.

This definition is also analogous to that of `smallStepList` in Listing B.22. Notice that `lastInSeqProb` and `paral` maintain the same role as in the latter function, but now have a different (but analogous)

definition.

The definition of function `mult` is as follows:

```
mult :: Matrix (Complex Double) -> Matrix (Complex Double)
        -> Matrix (Complex Double)
mult a b = multStd2 a b
```

Listing B.34: Function `mult`, from file `SemQC.hs`.

`multStd2` is a function from module Data.Matrix such that, according to this module's documentation, (`multStd2 a b`) is the matrix that results from the product of `a` and `b`.

The definition of function `numQubits` is the following:

```
numQubits :: S -> Int
numQubits s = if log2IntToDouble == log2 then log2Int
              else error "The matrix given as argument to function numQubits
                         is not a valid quantum state."
    where log2IntToDouble = (fromIntegral log2Int) :: Double
          log2Int = round log2 :: Int
          log2 = logBase 2.0 numElemsDouble
          numElemsDouble = (fromIntegral numElems) :: Double
          numElems = length (toList s)
```

Listing B.35: Function `numQubits`, from file `SemQC.hs`.

`length` in the above definition is a function such that `length l` is an `Int` value corresponding to the number of elements of `l`, just like Prelude module's documentation indicates. `toList` is a function such that `numElems` is the the number of elements of the matrix corresponding to `s`, as indicated by Data.Matrix module's documentation. `fromIntegral` is a function such that `numsElemsDouble` and `log2IntToDouble` correspond, respectively, to `numElems` and `log2Int` converted to `Double` values. On the other hand, `round` is a function such that `log2Int` corresponds to the `Int` value that results from rounding `log2` to the nearest integer. For more information about functions `fromIntegral` and `round`, see HaskellWiki [2016] and Prelude module's documentation. `logBase` in line 7 is a function such that `log2` represents the logarithm to the base 2 of `numElemsDouble`, as indicated by said documentation. Thus, in short, `numQubits s` outputs the `Int` value corresponding to the squared root of the number of elements of state `s`, as long as it is a valid state of a quantum system. Otherwise, `numQubits` raises an error.

The definition of `tensorProduct` is as follows:

```
tensorProduct :: [Op] -> Op
tensorProduct [] = error "No matrices given for the calculation of their
                          tensor product."
tensorProduct [a] = error "Not enough matrices given for the calculation of
                           their tensor product."
tensorProduct [a,b] = fromLists (tensorProductLists (toLists a) (toLists b))
tensorProduct (a:b:t) = tensorProduct [a, (tensorProduct (b:t)) ]
```

Listing B.36: Function `tensorProduct`, from file `SemQC.hs`.

In short, `tensorProduct` raises an error if the number of operators received is less than two. Otherwise, if it receives two operators, `tensorProduct` relies on `tensorProductLists`, which is an auxiliary function that, given two lists of type `[[Complex Double]]`, each representing a matrix, outputs the tensor product of the corresponding matrices in the form of a value of type `[[Complex Double]]`. According to Data.Matrix module's documentation, `fromLists` is a function such that, given a list of type `[[a]]` whose lists all have the same number of elements, outputs a matrix of type `Matrix a` corresponding to said list, while `toLists` is a function that converts a matrix of type `Matrix a` into a list of type `[[a]]` representing this matrix. In the last line of the above definition, the tensor product of more than two operators is obtained by calculating the tensor product between the first operator and the one corresponding to the tensor product of the rest of the operators. The definition of `tensorProductLists` is the following:

```
1 tensorProductLists :: [[Complex Double]] -> [[Complex Double]]
2                       -> [[Complex Double]]
3 tensorProductLists a [] = []
4 tensorProductLists [] b = []
5 tensorProductLists (h:t) b = (map (getLineTensor h) b) ++
6                              (tensorProductLists t b)
```

Listing B.37: Function `tensorProductLists`, from file `SemQC.hs`.

`getLineTensor` is defined as follows:

```
1 getLineTensor :: [Complex Double] -> [Complex Double] -> [Complex Double]
2 getLineTensor [] l = []
3 getLineTensor l [] = []
4 getLineTensor (h:t) l = (multElemLine h l) ++ (getLineTensor t l)
```

Listing B.38: Function `getLineTensor`, from file `SemQC.hs`.

Thus `(getLineTensor l b)` is the result of concatenating lists `(multElemLine ai b)`, with `ai` representing each element of list `l`, and `multElemLine` being defined as follows:

```
1 multElemLine :: Complex Double -> [Complex Double] -> [Complex Double]
2 multElemLine x [] = []
3 multElemLine x (h:t) = xh : (multElemLine x t)
4     where (a,theta) = polar x
5           (b,phi) = polar h
6           xh = mkPolar (a*b) (theta + phi)
```

Listing B.39: Function `getLineTensor`, from file `SemQC.hs`.

In the above definition, `xh` is the result of multiplying `x` by `h`. Therefore `(multElemLine x l)` is the list that results from multiplying every element of `l` by `x`. The descriptions of functions `polar` and `mkPolar` ca be found in Data.Complex module's documentation.

Function `replaceByGate` is defined as follows:

```
1 replaceByGate :: Op -> [Int] -> [Op] -> [Op]
2 replaceByGate op [] l = l
```

```
3 replaceByGate op (h:t) l
4     | (n==0) = error ("Empty operators list received as argument by function
5                       replaceByGate.")
6     | (h > n) = error ("List of operators given as argument to function
7                        replaceByGate does not contain " ++ (show h) ++
8                       " elements.")
9     | (h < 1) = error ("The list of indexes received as argument by function
10                       replaceByGate cannot contain integers less than 1.")
11    | otherwise = replaceByGate op t nextl
12        where n = length l
13              first = [op] ++ ( if (n==1) then [] else (elements 2 n l) )
14              last = (elements 1 (n-1) l) ++ [op]
15              middle = (elements 1 (h-1) l) ++ [op] ++ (elements (h+1) n l)
16              nextl = if (h==1) then first else (if (h==n) then last
17                                                  else middle)
```

Listing B.40: Function `replaceByGate`, from file `SemQC.hs`.

`elements` is an auxiliary function that, given two values a and b of type `Int` and a list `l`, `(elements a b l)` is a list consisting of the elements of `l`, from the a-th element to the b-th one, if `(length l)` $\geq$ b $\geq$ a ≥ 1, and `l` is non-empty. Otherwise, `elements` raises an error message.

The definition of function `sumMatrices` is the following:

```
1 sumMatrices :: Op -> Op -> Op
2 sumMatrices a b = elementwise (+) a b
```

Listing B.41: Function `sumMatrices`, from file `SemQC.hs`.

`elementwise` in the above definition is a function such that, given two equally sized matrices `a` and `b`, `(elementwise f a b)` is a matrix in which each element is the result of applying function `f` to the corresponding elements of `a` and `b`, as Data.Matrix module's documentation indicates.

The definition of `applyCZ` is based on Equation 4.23 and on the reasoning behind Equation 4.22, and is as follows:

```
1 applyCZ :: [Int] -> S -> S
2 applyCZ l s
3     | (length l /= 2) = error "First argument of function applyCZ must be a
4                                list with two elements."
5     | otherwise = if (control /= target) then mult matrix s else error "The
6                   control and target qubits given as argument to function
7                   applyCZ cannot be the same."
8         where control = head l
9               target = last l
10              nqubits = numQubits s
11              listId = gateList ident nqubits
12              matrix1 = tensorProduct listId
13              matrix2 = tensorProduct $ replaceByGate m1 [target]
14                        (replaceByGate m2 [control] listId)
15              matrix = subtMatrices matrix1 matrix2
```

Listing B.42: Function `applyCZ`, from file `SemQC.hs`.

`m2` in the above definition is an operator corresponding to matrix $2A_1$, with A_1 representing $|1\rangle\langle 1|$. `subtMatrices` is a function such that `subMatrices matrix1 matrix2` is the matrix resulting

from the substraction of `matrix2` from `matrix1`. It is defined analogously to `sumMatrices` (see Listing B.41).

Function `dagger` is defined as follows:

```
dagger :: Matrix (Complex Double) -> Matrix (Complex Double)
dagger m = transpose $ complexConjugate m
```

Listing B.43: Function `dagger`, from file `SemQC.hs`.

`transpose` is a function such that `transpose m` corresponds to the transpose of matrix `m`, as indicated by Data.Matrix module's documentation. `complexConjugate` is a function such that `complexConjugate m` is the conjugate of matrix `m`. Its definition makes use of function `conjugate`, which provides the conjugate of values of type `(Complex Double)`, as Data.Complex module's documentation indicates.

The definition of `applyBigStepList` is as follows:

```
applyBigStepList :: C -> L -> S -> [[ProbConf]]
applyBigStepList c l s = simplify (bigStepList c l s)
```

Listing B.44: Function `applyBigStepList`, from file `SemQC.hs`.

In the above definition, `simplify` is a function such that, given a list of distributions `l` of type `[[ProbConf]]`, `simplify l` is the list of distributions resulting from, for each distribution in `l`, joining values of type `ProbConf` with the same command and state into just one value of this type. For example, for arbitrary states `s1`, `s2` and `s3`:

```
simplify [[(0,Skip,s1), (1,Skip,s1)], [(1,Skip,s2)]] =
[[(1,Skip,s1)], [(1,Skip,s2)]]
```

The definition of `smallStep` is the following. It follows a similar reasoning to that used for defining `bigStep` (whose definition is presented in Listing 6.18) and is also based on the rules from Figure 4:

```
smallStep :: C -> L -> S -> IO (C,S)
smallStep Skip l s = return (Skip,s)
smallStep (Seq c1 c2) l s = if (term c1 s) then return (c2,s) else do
    (c1',s') <- smallStep c1 l s
    return (Seq c1' c2, s')
smallStep (U g vars) l s = return (Skip, applyGate g (qNums vars l) s)
smallStep (Meas q c1 c2) l s
    | (p0 == 0) = return (c2, s1)
    | (p1 == 0) = return (c1, s0)
    | otherwise = do
            n <- enact event
            return ( if (n==1) then (c1, s0) else (c2, s1) )
        where p0 = prob 0 (l(q)) s
              p1 = prob 1 (l(q)) s
              s0 = state 0 (l(q)) s
              s1 = state 1 (l(q)) s
              dist = [(1, p0),(2, p1)]
              event = makeEventProb dist
```

```
19 smallStep (Wh q c) l s = return (Meas q Skip (Seq c (Wh q c)), s)
20 smallStep (Paral c1 c2) l s
21     | term (Paral c1 c2) s = return (Paral c1 c2, s)
22     | term c1 s = smallStep2nd c1 c2 l s
23     | term c2 s = smallStep1st c1 c2 l s
24     | otherwise = do
25         x <- sched
26         if (x==0) then (smallStep1st c1 c2 l s) else (smallStep2nd c1 c2 l s)
```

Listing B.45: Function `smallStep`, from file `SemQC.hs`.

`smallStep1st` and `smallStep2nd` in the above definition are defined as follows:

```
1 smallStep1st :: C -> C -> L -> S -> IO (C,S)
2 smallStep1st c1 c2 l s = do
3     (c1', s') <- smallStep c1 l s
4     return (Paral c1' c2, s')
```

Listing B.46: Function `smallStep1st`, from file `SemQC.hs`.

```
1 smallStep2nd :: C -> C -> L -> S -> IO (C,S)
2 smallStep2nd c1 c2 l s = do
3     (c2', s') <- smallStep c2 l s
4     return (Paral c1 c2', s')
```

Listing B.47: Function `smallStep2nd`, from file `SemQC.hs`.

Auxiliary functions `bigStep1st` and `bigStep2nd` are defined as follows:

```
1 bigStep1st :: C -> C -> L -> S -> IO (C,S)
2 bigStep1st c1 c2 l s = do
3     (c1',s') <- smallStep c1 l s
4     bigStep (Paral c1' c2) l s'
```

Listing B.48: Function `bigStep1st`, from file `SemQC.hs`.

```
1 bigStep2nd :: C -> C -> L -> S -> IO (C,S)
2 bigStep2nd c1 c2 l s = do
3     (c2',s') <- smallStep c2 l s
4     bigStep (Paral c1 c2') l s'
```

Listing B.49: Function `bigStep2nd`, from file `SemQC.hs`.

The definition of `listBigStep` is the following:

```
1 listBigStep :: Int -> C -> L -> S -> IO [(C,S)]
2 listBigStep 0 c l s = return []
3 listBigStep n c l s = do
4     a <- bigStep c l s
5     as <- listBigStep (n-1) c l s
6     return (a:as)
```

Listing B.50: Function `listBigStep`, from file `SemQC.hs`.

Function `caption` is defined as follows:

```
1 caption :: Int -> [(C,S)] -> IO ()
2 caption n [] = putStrLn ""
3 caption n (h:t) = do
4     putStrLn ("<conf "++(show n)++"> : \n"++(showCS h))
5     caption (n+1) t
```

Listing B.51: Function `caption`, from file `SemQC.hs`.

In the above definition, `showCS` is a function that converts a value `(c,s)` of type `(C,S)` into a `String` value that contains `c` and `s` in the form of `String` values, separated by a newline character.

The definition of `histogramInt` is the following:

```
1 histogramInt :: [Double] -> String -> IO ExitCode
2 histogramInt [] title = error "Empty input."
3 histogramInt dataSet title = plotAdv "" options hist
4     where max = round (maximum dataSet) :: Int
5           hist = histogramBinSize 1 dataSet
6           options = Opt.title title $ Opt.xRange2d (-1,max+1) $ Opt.xTicks2d
7                     (xTicksData max) (defOpts hist)
```

Listing B.52: Function `histogramInt`, from file `HistogramSem.hs`.

The above definition makes use of modules Graphics.Histogram, Graphics.Gnuplot.Frame.OptionSet and System.Exit. For implementing this definition, references izbicki [2012] and Thielemann [2022] have been very useful, as they contain useful examples. The sources mentioned in the third paragraph of Chapter 6 have also been useful.

In the above definition, `max` is a value of type `Int` representing the largest element of `dataSet`. The description of function `maximum` can be found in Prelude module's documentation, while that of function `round` can be found in this documentation and in HaskellWiki [2016]. `hist` corresponds to an histogram with bin size 1 and with `dataSet` as its input, as Graphics.Histogram module's documentation indicates. Since the bin size corresponds to 1, there is a column for each integer in the x-axis. In lines 6 and 7 of the above definition we specify that the title of the histogram is `title`, the range of the x-axis is $[-1, \texttt{max+1}]$ and we specify the labels of the x-axis of the histogram. Notice that functions `Opt.title`, `Opt.xRange2d` and `Opt.xTicks2d` belong to module Graphics.Gnuplot.Frame.OptionSet, while `defOpts` belongs to module Graphics.Histogram. `xTicksData` is a function such that `(xTicksData max)` is a value of type `[(String,Int)]` of the following form:

```
[("<conf 1>",0),("<conf 2>",1),...,("<conf max>",(max-1))]
```

Thus, in the x-axis of the histogram, integer $(x-1)$ has label `<conf` x`>`, and only integers from 0 to `max`-1 are labeled. Lastly, `plotAdv` is a function such that, if `dataSet` is non-empty, `histogramInt dataSet title` is a value of type `IO ExitCode` that plots `hist` according to the options specified by `options`, as Graphics.Histogram module's documentation indicates. Type `ExitCode` belongs to module System.Exit.

Appendix C
Examples and Case-study

This appendix presents figures related to the examples and case-study discussed in Chapter 7.

```
*ParserSemQ> bigStepListFile "qTelepSeq.txt" lT qTelepInitState
[[(0.24999999999999994,Skip,
 ┌                          ┐
 │ 0.7071067811865475 :+ 0.0│
 │ 0.7071067811865475 :+ 0.0│
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 └                          ┘),
(0.24999999999999994,Skip,
 ┌                                          ┐
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 │                 0.7071067811865475 :+ 0.0│
 │ 0.7071067811865475 :+ 1.7319121124709863e-16│
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 └                                          ┘),
(0.24999999999999994,Skip,
 ┌                          ┐
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 │ 0.7071067811865475 :+ 0.0│
 │ 0.7071067811865475 :+ 0.0│
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 │                0.0 :+ 0.0│
 └                          ┘),
(0.24999999999999994,Skip,
 ┌                                          ┐
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 │                                0.0 :+ 0.0│
 │                 0.7071067811865475 :+ 0.0│
 │ 0.7071067811865475 :+ 1.7319121124709863e-16│
 └                                          ┘)]]
```

Figure 19: Result of `bigStepListFile` applied to file `qTelepSeq.txt`, linking function `lT` and state `qTelepInitState`.

```
Histogram Caption:

<conf 1> :
Skip
┌                                        ┐
                   0.0 :+ 0.0
                   0.0 :+ 0.0
    0.7071067811865475 :+ 0.0
    0.7071067811865475 :+ 0.0
                   0.0 :+ 0.0
                   0.0 :+ 0.0
                   0.0 :+ 0.0
                   0.0 :+ 0.0
└                                        ┘
<conf 2> :
Skip
┌                                                      ┐
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                       0.7071067811865475 :+ 0.0
    0.7071067811865475 :+ 1.7319121124709863e-16
└                                                      ┘
<conf 3> :
Skip
┌                                                      ┐
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                       0.7071067811865475 :+ 0.0
    0.7071067811865475 :+ 1.7319121124709863e-16
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
└                                                      ┘
<conf 4> :
Skip
┌                                        ┐
    0.7071067811865475 :+ 0.0
    0.7071067811865475 :+ 0.0
                   0.0 :+ 0.0
                   0.0 :+ 0.0
                   0.0 :+ 0.0
                   0.0 :+ 0.0
                   0.0 :+ 0.0
                   0.0 :+ 0.0
└                                        ┘
```

Figure 20: Caption produced by `(histBigStepFile 100000 "qTelepSeq.txt" lT qTelepInitState)`, relative to the histogram in Figure 13.

```
Histogram Caption:

<conf 1> :
Skip
⎡ 0.7071067811865476 :+ 0.0 ⎤
⎢ 0.7071067811865476 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎣                0.0 :+ 0.0 ⎦
<conf 2> :
Skip
⎡                0.0 :+ 0.0 ⎤
⎢                0.0 :+ 0.0 ⎥
⎢ 0.7071067811865475 :+ 0.0 ⎥
⎢ 0.7071067811865475 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎣                0.0 :+ 0.0 ⎦
<conf 3> :
Skip
⎡                                                0.0 :+ 0.0 ⎤
⎢                                                0.0 :+ 0.0 ⎥
⎢                                                0.0 :+ 0.0 ⎥
⎢                                                0.0 :+ 0.0 ⎥
⎢ (-0.7071067811865476) :+ (-8.659560562354933e-17) ⎥
⎢                                                0.0 :+ 0.0 ⎥
⎢                         0.7071067811865476 :+ 0.0 ⎥
⎣                                                0.0 :+ 0.0 ⎦
<conf 4> :
Skip
⎡                                       0.0 :+ 0.0 ⎤
⎢                                       0.0 :+ 0.0 ⎥
⎢                                       0.0 :+ 0.0 ⎥
⎢                                       0.0 :+ 0.0 ⎥
⎢                0.7071067811865475 :+ 0.0 ⎥
⎢ 0.7071067811865475 :+ 1.7319121124709863e-16 ⎥
⎢                                       0.0 :+ 0.0 ⎥
⎣                                       0.0 :+ 0.0 ⎦
```

Figure 21: Part 1 of the caption produced by (`histBigStepFile 100000 "qTelepAttempt.txt" lT qTelepInitState`), relative to the histogram in Figure 15. Parts 2 and 3 of this caption are in Figures 22 and 23, respectively.

```
<conf 5> :
Skip
┌                                  ┐
  0.7071067811865476 :+ 0.0
                 0.0 :+ 0.0
  0.7071067811865476 :+ 0.0
                 0.0 :+ 0.0
                 0.0 :+ 0.0
                 0.0 :+ 0.0
                 0.0 :+ 0.0
                 0.0 :+ 0.0
└                                  ┘
<conf 6> :
Skip
┌                                                          ┐
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                                      0.0 :+ 0.0
                       0.7071067811865476 :+ 0.0
  0.7071067811865476 :+ 1.7319121124709866e-16
└                                                          ┘
<conf 7> :
Skip
┌                                  ┐
                 0.0 :+ 0.0
                 0.0 :+ 0.0
  0.7071067811865476 :+ 0.0
  0.7071067811865476 :+ 0.0
                 0.0 :+ 0.0
                 0.0 :+ 0.0
                 0.0 :+ 0.0
                 0.0 :+ 0.0
└                                  ┘
<conf 8> :
Skip
┌                                                              ┐
                                                0.0 :+ 0.0
                                                0.0 :+ 0.0
                                                0.0 :+ 0.0
                                                0.0 :+ 0.0
                                 0.7071067811865476 :+ 0.0
                                                0.0 :+ 0.0
  (-0.7071067811865476) :+ (-8.659560562354933e-17)
                                                0.0 :+ 0.0
└                                                              ┘
```

Figure 22: Part 2 of the caption produced by (`histBigStepFile 100000 "qTelepAttempt.txt" lT qTelepInitState`), relative to the histogram in Figure 15. Parts 1 and 3 of this caption are in Figures 21 and 23, respectively.

```
<conf 9> :
Skip
⎡ 0.7071067811865475 :+ 0.0 ⎤
⎢ 0.7071067811865475 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎢                0.0 :+ 0.0 ⎥
⎣                0.0 :+ 0.0 ⎦
<conf 10> :
Skip
⎡                                          0.0 :+ 0.0 ⎤
⎢                                          0.0 :+ 0.0 ⎥
⎢                                          0.0 :+ 0.0 ⎥
⎢                                          0.0 :+ 0.0 ⎥
⎢                                          0.0 :+ 0.0 ⎥
⎢                                          0.0 :+ 0.0 ⎥
⎢                           0.7071067811865475 :+ 0.0 ⎥
⎣ 0.7071067811865475 :+ 1.7319121124709863e-16 ⎦
<conf 11> :
Skip
⎡                                          0.0 :+ 0.0 ⎤
⎢                                          0.0 :+ 0.0 ⎥
⎢                                          0.0 :+ 0.0 ⎥
⎢                                          0.0 :+ 0.0 ⎥
⎢                           0.7071067811865476 :+ 0.0 ⎥
⎢ 0.7071067811865476 :+ 1.7319121124709866e-16 ⎥
⎢                                          0.0 :+ 0.0 ⎥
⎣                                          0.0 :+ 0.0 ⎦
```

Figure 23: Part 3 of the caption produced by `(histBigStepFile 100000 "qTelepAttempt.txt" lT qTelepInitState)`, relative to the histogram in Figure 15. Parts 1 and 2 of this caption are in Figures 21 and 22, respectively.

Appendix D
The Kronecker product

Consider a $m \times n$ matrix A and a $p \times q$ matrix B. Let a_{ij} and b_{ij} represent the elements in line i and column j of matrices A and B, respectively. The Kronecker product of A and B, represented by $A \otimes B$, is a $mp \times nq$ matrix given by Graham [2018]:

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}. \tag{D.1}$$

In the above equation, $a_{ij}B$ corresponds to a matrix of the same order as B, equal to:

$$\begin{pmatrix} a_{ij}b_{11} & \cdots & a_{ij}b_{1q} \\ \vdots & & \vdots \\ a_{ij}b_{p1} & \cdots & a_{ij}b_{pq} \end{pmatrix}. \tag{D.2}$$

For example, let A and B be given by:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}. \tag{D.3}$$

Then, the Kronecker product $A \otimes B$ is obtained as follows:

$$A \otimes B = \begin{pmatrix} 1\times 5 & 1\times 6 & 2\times 5 & 2\times 6 \\ 1\times 7 & 1\times 8 & 2\times 7 & 2\times 8 \\ 3\times 5 & 3\times 6 & 4\times 5 & 4\times 6 \\ 3\times 7 & 3\times 8 & 4\times 7 & 4\times 8 \end{pmatrix} = \begin{pmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{pmatrix}. \tag{D.4}$$

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020, DOI 10.54499/LA/P/0063/2020 | https://doi.org/-10.54499/LA/P/0063/2020.

This work was financed by National Funds through FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference 10.54499/PTDC/CCI-COM/4280/2021.