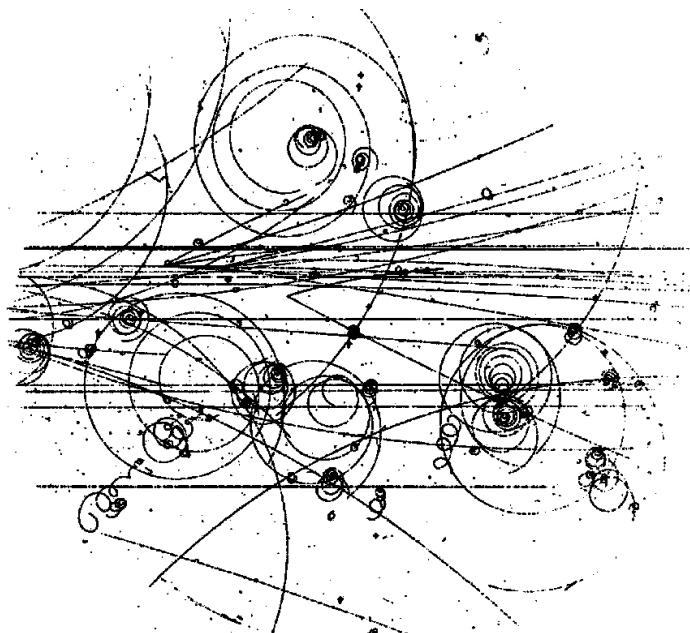


DAVIDE CAMINO

QUANTUM COMPUTING FOR
LOGICAL INFERENCE

Subtitle



CONTENTS

I THEORETICAL BASES	
1 QUANTUM MECHANICS	3
1.1 Experiments	3
1.1.1 Spin	3
1.1.2 Qubit	5
1.1.3 Boolean Logic	5
1.2 Quantum states	6
1.2.1 Vector Spaces	7
1.2.2 Bra and Ket	7
1.2.3 Hidden variables	8
1.2.4 Spin states	9
1.3 Observables	11
1.3.1 Hermitian operator	11
1.3.2 Principles of quantum mechanics	12
1.3.3 Spin Operator	13
1.3.4 Theory and experiments	14
1.3.5 Operator and Measure	16
1.4 Temporal Evolution	16
1.4.1 Unitarity	16
1.4.2 Time-Development Operator	17
1.4.3 The Hamiltonian	18
1.4.4 Commutators	19
1.4.5 Conservation of Energy	20
1.5 Conclusions	20
2 QUANTUM GATE	23
3 QUANTUM ANNEALING	25
4 ONTOLOGY	27
4.1 Knowledge Base	27
4.1.1 Ontology in philosophy	27
4.1.2 Ontology in computer science	27
4.1.3 OWL Language	28
4.1.4 Importance of ontologies	29
4.2 Example ontologies	29
4.2.1 Simple ontology	29
4.2.2 DOLCE ontology	31
4.3 Reasoning on ontology	31
4.3.1 <i>SROIQ</i> DL	32
4.3.2 Interpretation of a knowledge base	32
4.3.3 Complexity of reasoning	33
4.4 Conclusions	34
II TOOLS	
5 ENVIRONMENT SETUP	37

5.1	Python environment	37
5.2	IBM Qiskit	37
5.2.1	Hello World	38
5.2.2	Transpilation	38
5.2.3	Execution	39
5.3	D-Wave Ocean	40
5.3.1	Hello World	40
5.4	PyQUBO and qubovert	41
5.4.1	PyQUBO	42
5.4.2	qubovert	42
5.5	Conclusion	44
6	QA-PROLOG	45
6.1	The project	45
6.1.1	Reason	45
6.1.2	Prolog	46
6.1.3	Feature of QA-Prolog	47
6.1.4	Related works	47
6.2	Pipeline	48
6.2.1	QMASM	48
6.2.2	Yosys and edif2qasm	51
6.2.3	QA-Prolog	53
6.2.4	Overview	55
6.3	Update to the project	55
6.3.1	Restoring QMASM	56
6.3.2	Fixing interaction edif2qasm-QMASM	56
6.3.3	Installation guide	57
6.4	Conclusion	58
III EXPERIMENTS		
7	A QUANTUM ONTOLOGY	61
7.1	Ontology structure	61
7.2	Prolog version	61
7.3	Inference on the ontology	61
7.4	Conclusion	61
8	QAOA	63
8.1	QAOA	63
8.2	From QUBO to Pauli operator	63
8.3	Experiments	63
8.4	Conclusion	63
9	CONCLUSION	65
APPENDIX 1		69

Part I
THEORETICAL BASES

1

QUANTUM MECHANICS

In this chapter we will explore the basics of quantum mechanics in order to understand what we can or cannot do with a quantum computer. The reference architecture for this work is the quantum gate-based quantum computer. In the next pages we will try to justify why the algorithms that run on this hardware need to be reversible.

The chapter starts from the very beginning with the definition of a quantum system and presents the basis to understand the evolution of a quantum system, trying to justify why every evolution needs to be reversible and what exactly reversibility means. At the end of the chapter we will put all of our new knowledge together to derive the famous Schrödinger equation.

With all this work we will be able to imagine the function of a quantum gate and understand the limitations that are imposed when we develop an algorithm for a quantum computer.

1.1 EXPERIMENTS

We start our introduction to quantum mechanics with an experiment. Experiments are not only an excuse to introduce the topic, but the essential key of physics, both classical and modern.

Theory and models need to adapt to the experiments, and when the experimental results are in contradiction with the actual model it means that the model needs to be changed to respect the behavior of the world.

1.1.1 Spin

We analyze the experiment of an electron in a magnetic field. An electron is an electrically charged particle; when some electrons are shot in an electric field all of them are influenced by Coulomb's law; if all electrons have the same initial velocity the beam of electrons remains intact.

What happens to the same beam in a magnetic field? Again electrons are deflected by a force, but this time the beam splits. If the initial velocity was parallel to the x axis and the magnetic field is oriented along the z axis some electrons are deflected upward, some downward, but the intensity of the deflection is the same for all the electrons. This means that no electrons are deflected less or more than the others and the beam splits exactly into two parts.



Figure 1.1: Experiment's schema

38 Starting from this experiment we can make a measuring instrument: this
 39 apparatus \mathcal{A} can be oriented along an arbitrary axis, and in the previous con-
 40 figuration \mathcal{A} displays $+1$ if the electron is deflected upward, -1 otherwise.
 41 We call this number the spin of the electron.

Repeatability of measure
 42 If we measure the spin of an electron and \mathcal{A} displays $+1$, we can confirm
 43 the experiment's results by measuring the spin again and we obtain spin $+1$
 44 every time. This means that the measurements are repeatable (an essential
 45 property to construct models and make predictions). We can think, and it
 46 will be clear later why it is useful, that the first experiment prepares the spin
 47 $+1$ and the others confirm this result.

48 Spin is a quantum property and all the visual representations such as the
 49 rotation of the electron around its axis would lead to misrepresentation. Spin
 50 and rotation, however, have some similarities. Let's analyze what would
 51 happen if we consider a charged sphere in a magnetic field with the laws of
 52 classical physics. We consider a sphere rotating around its axis, and this axis
 53 is parallel to the z axis. The x or y component of the angular momentum is
 54 zero. Measuring the component along a generic axis, oriented like the versor
 55 \hat{n} ¹, we would obtain a result proportional to the projection of \hat{z} on \hat{n} . This
 56 projection can be found with the scalar product $\hat{z} \cdot \hat{n} = \cos \theta^2$, where θ is the
 57 angle between the axes.

58 Now we consider the quantum version of this phenomenon. Let's start by
 59 measuring the z component of the spin and assume that the result is $\sigma_z = +1$;
 60 if we rotate the apparatus \mathcal{A} around, for example, the x axis, we can measure
 61 σ_x . This component would not be zero, and \mathcal{A} keeps displaying only $+1$
 62 or -1 . The single result is not helpful, but we can repeat the experiment,
 63 namely:

- 64 1. orienting \mathcal{A} along the z axis and preparing a spin $\sigma_z = +1$
- 65 2. rotating \mathcal{A} around x
- 66 3. measuring the x component of the spin

67 statistically we would observe the same number of $\sigma_x = +1$ and $\sigma_x = -1$.

68 If we start the experiments with a spin prepared as $\sigma_z = +1$ and then
 69 orient \mathcal{A} along a generic axis \hat{n} each measure would be binary and unpre-
 70 dictable, but the mean of the measures tends to $\hat{z} \cdot \hat{n} = \cos \theta$ where θ is the
 71 angle between \hat{z} and \hat{n} . In the most general case we can start with the ap-
 72 paratus oriented like m and prepare the spin $\sigma_m = +1$, then we rotate \mathcal{A}
 73 around \hat{n} without interfering with the spin and measure again; we would
 74 obtain the statistical result $\langle \sigma_n \rangle = \hat{m} \cdot \hat{n}$ ³.

75 The result of a single measure is non deterministic, but we can make pre-
 76 dictions over the mean values of the measures: the expected values behave
 77 as the single results of the classic experiment.

Invasive experiments
 78 Considering now a sequence of three measures: starting with \mathcal{A} oriented
 79 along z we prepare the spin $\sigma_z = +1$, then we rotate \mathcal{A} to measure σ_x
 80 obtaining, let's say, $+1$ (the reasoning is the same if we obtain -1); lastly
 81 returning with \mathcal{A} parallel to z we cannot make any prediction on the single

¹ A versor is a vector of magnitude 1 (unit vector), it is normally used to specify a direction.

² We can use directly the angle because we are considering versors.

³ The Dirac bracket $\langle \rangle$ denotes the statistical mean of a quantity. We call that expectation value.

82 result, the initial configuration (with $\sigma_z = +1$) is lost forever, the only result
 83 we can predict is that $\langle \sigma_z \rangle = 0$.

84 **1.1.2 Qubit**

85 We have introduced the spin referring to electrons in a magnetic field. How-
 86 ever, we can study the spin without examining the associated electron; we
 87 have isolated a simple physical system, the simplest we can study.

88 Spin belongs to a class of simple physical systems called *qubit*; in all of
 89 these systems the result of a measure is binary. We will see that, even if the
 90 result of a measure is equal to the classical *bit*, the qubit system is described
 91 in a very different way compared to its classical alter ego.

92 **1.1.3 Boolean Logic**

93 In this paragraph we try to understand why we need two different ways
 94 to describe a classical and a quantum state space. To do so we analyze the
 95 results of some logical propositions, both basic and composed via logical
 96 connectives.

97 Starting with the classical case we consider a bag of colored and numbered
 98 balls. We can construct the state space by enumerating all states, namely
 99 taking each ball from the bag and annotating the pair number–color. The
 100 basic propositions we analyze are:

- 101 • The extracted ball is red.
- 102 • The number on the extracted ball is even.

103 If we consider a particular state we can say if a proposition is true or
 104 false; we can also define two subsets of balls, the first with all the red balls
 105 (for this subset the first proposition is true), the second one with the balls
 106 that show an even number (subset that makes the second proposition true).
 107 Considering now disjunction and conjunction:

- 108 • The extracted ball is red *or* even.
- 109 • The extracted ball is red *and* even.

110 Again it is simple to associate a truth value to these propositions if we con-
 111 sider a single state; also we can construct two subsets that satisfy the propo-
 112 sitions from the subsets we defined before: the new subsets are respectively
 113 the union and intersection of the old ones.

114 In the quantum world the situation is very different. Let's start from
 115 propositions that can be verified with a simple experiment:

- 116 • The z component of the spin is $+1$.
- 117 • The x component of the spin is $+1$.

118 If we want to check the first proposition we can orient the apparatus \mathcal{A} along
 119 z and make a measurement; the same procedure can be followed for the
 120 second proposition. The disjunction and conjunction of these propositions
 121 are:

- 122 • The z component of the spin is $+1$ or the x component is $+1$.
 123 • The z component of the spin is $+1$ and the x component is $+1$.

124 Starting with the disjunction. Considering a state prepared, without our
 125 knowledge, with $\sigma_z = +1$. If our first measure is along the z axis, \mathcal{A} will
 126 always display $+1$ and we can immediately conclude that the proposition
 127 is true. If we start measuring the x component, we have a 50% chance that
 128 \mathcal{A} displays $+1$ or -1 ; also this measurement destroys the initial state and
 129 the measure of σ_z becomes non predictable. In this scenario we have a 25%
 130 chance of deducing that the proposition is false; Figure 1.2 shows all the
 The disjunction is not
 commutative^{g2} possible measurement results in this case. The logical value of a proposition
 depends on the order in which we perform the measurements.

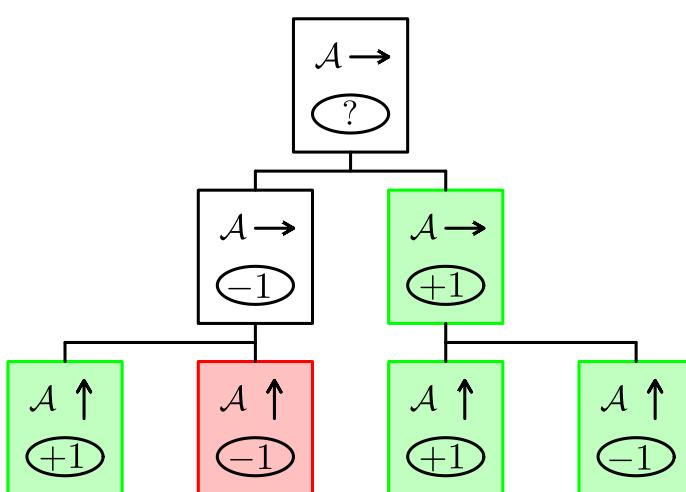


Figure 1.2: The apparatus \mathcal{A} is represented as a box, the arrow represents the direction along which the apparatus is oriented, the display (ellipse) shows the result of measurement. We have highlighted in green the cases in which we can immediately conclude that the disjunction of the propositions is true

133 The conjunction is even worse: no matter the order of the measurements,
 134 the second one destroys the result of the first. The disjunction is true if at
 135 least one of the sub-propositions is true, and if we find a spin component
 136 that is $+1$ we can always confirm this result with another measurement. In
 137 the conjunction the two sub-propositions must be true *at the same time*, but
 The conjunction loses its
 meaning^{g3} with the second measurement we lose all the knowledge of the first one. We
 can never conclude that the conjunction is true.

140 1.2 QUANTUM STATES

141 In the previous section we have understood that a state space of a quantum
 142 system cannot be represented in the same way as a classical state space. Now
 143 we present a formal mathematical model to describe the state space for spin.

144 **Axiom 1.** *The state space for a quantum system is a complex vector space.*

145 This is a physical axiom, which means that it is true because there are a lot
 146 of experiments that confirm this model and none that shows a contradiction.

147 **1.2.1 Vector Spaces**

148 A vector space is a mathematical and abstract construction that can have
 149 multiple dimensions (even infinite) and has, as components, integers, real
 150 or complex numbers, or other elements. An example that shows well how
 151 abstract a vector space can be is the complex-valued continuous function of
 152 variable x ; the set of these functions generates a vector space.

153 In quantum mechanics the state space is described by a vector space hav- *Hilbert space*
 154 ing as element $|A\rangle$ called *ket*. The properties of this space are:

- 155 • the sum of two kets is a ket;
- 156 • addition is commutative;
- 157 • addition is associative;
- 158 • existence of identity element for addition;
- 159 • existence of inverse elements for addition;
- 160 • existence of identity element for scalar multiplication;
- 161 • linearity property.

162 **1.2.2 Bra and Ket**

163 An example of ket that we will find often is the column vector of two dimen-
 164 sions:

$$|A\rangle = \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix}$$

165 where α_1 and α_2 are complex numbers. With this simple example of ket it
 166 is easy to verify the validity of all previously described properties.

167 If, for complex numbers, exists the complex conjugate, for every ket there
 168 exists a *bra*. The set of bra generates a dual conjugate space with respect
 169 to the state space of ket. We denote a bra as $\langle A|$. If $|A\rangle$ is the ket of the
 170 previous example the corresponding bra is a row vector having as elements
 171 the complex conjugate of $|A\rangle$:

$$\langle A| = (\alpha_1^*, \alpha_2^*).$$

172 Name and symbol associated with elements of Hilbert spaces become clear
 173 when we define the product *bra-ket*, this is the corresponding scalar product
 174 of an ordinary vector and is called inner product. Considering bra and ket of
 175 two dimensions we can evaluate the inner product by adding the products
 176 of corresponding components:

$$\langle A | B \rangle = (\alpha_1^*, \alpha_2^*) \cdot \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} = \alpha_1^* \beta_1 + \alpha_2^* \beta_2.$$

177 Having the inner product we can define:

178 **VERSOR** normalized vector $|A\rangle$ in which $\langle A | A \rangle = 1$;

179 **ORTHOGONAL VECTOR** vectors that have a null inner product: $\langle A | B \rangle = 0$.

Inner product

We are familiar with these concepts in two and three dimensions, the first one is a vector of length one, the second is the right angle between two vectors. This representation is misleading in our case, we cannot imagine a ket like an arrow and the state space is completely abstract even if there are properties and operations in common between this space and the 3D space that we are familiar with.

We have lost the geometric interpretation, and it seems that we have defined two completely abstract and useless concepts, we will see next that these are key concepts in the description of quantum systems and have a precise and important physical meaning.

Orthonormal basis

By having a vector space is possible to build a set of orthogonal versors that generates all vectors in the given space. This set is called orthonormal basis and the cardinality of the set is equal to the dimension of the space.

Formally having a basis $\mathcal{B} = \{|i_1\rangle, |i_2\rangle, \dots, |i_N\rangle\}$ of a space with N dimensions, we can write a generic vector in that space as

$$|A\rangle = \sum_{n=1}^N \alpha_n |i_n\rangle = \sum_{n=1}^N |i_n\rangle \langle i_n | A \rangle \quad (1.1)$$

this is the linear combination of the basis versors; where kets $|i_n\rangle$ are the versors in the basis and α_n are the vector components. We can obtain those components with the inner product between the vector $|A\rangle$ and the basis versors:

$$\alpha_i = \langle i | A \rangle. \quad (1.2)$$

1.2.3 Hidden variables

In a classical system we can measure all the variables associated to a physical system and then make a deterministic prediction of the evolution of that system. From the experiments described in the first section we have learned that a quantum system is not completely predictable even if we can make all the measurements that we want⁴. We can ask ourselves if our measurements aren't enough, if there are other variables that can make the prediction completely deterministic. About that topic we don't have any experimental proof, the opinion of physicists is divided in two main visions:

OPINION ONE : there are hidden variables and, if we manage to measure them, the prediction of results become deterministic. These variables can be

- very difficult to measure
- unknowable to us because also we are constituted by quantum material.

OPINION TWO : hidden variables don't exist, we already know all the information about a given system and quantum mechanics is intrinsically non deterministic.

No hidden variables

Probably no experiment could determine which vision is correct, but this doubt doesn't worsen our comprehension of the physical world. We can

⁴ We remember that a measure along one axis destroys our knowledge about the result along another axis.

simply choose one vision and build our model coherently. We choose the simpler one, without hidden variables, all that we have to model are the quantities that we can measure and the measurements allow us to know all the information about a given system.

Even if we have lost complete determinism, knowing the state of a system gives us some information about the system and the successive measurements. In the next section we will see what we can deduce about spin.

1.2.4 Spin states

Let's start enumerating all possible spin states along the coordinate axes. If we rotate the apparatus \mathcal{A} around z , we can obtain $\sigma_z = \pm 1$; we call these states *up* and *down* and label them with kets $|u\rangle$ and $|d\rangle$. Orienting \mathcal{A} along x , we obtain *left* $|l\rangle$ and *right* $|r\rangle$. Lastly, along the y axis, we measure the states *in* $|i\rangle$ and *out* $|o\rangle$.

The hypothesis that there aren't hidden variables allows us to represent the space state in a simple way: each spin state can be represented as a ket in a two-dimensional complex vector space.

To express a vector we need a basis; we choose $\mathcal{B} = \{|u\rangle, |d\rangle\}$ ⁵ and try to obtain all states as a linear combination (*superposition*) of the basis vectors. A generic state $|\mathcal{A}\rangle$ can be expressed as:

$$|\mathcal{A}\rangle = \alpha_u |u\rangle + \alpha_d |d\rangle$$

where α_u and α_d are the components of $|\mathcal{A}\rangle$ along $|u\rangle$ and $|d\rangle$, and can be obtained by projection: $\alpha_u = \langle u | \mathcal{A} \rangle$ and $\alpha_d = \langle d | \mathcal{A} \rangle$ (as in Equation 1.2).

$|\mathcal{A}\rangle$ components are complex numbers and their physical meaning is: having a spin prepared in the state $|\mathcal{A}\rangle = \alpha_u |u\rangle + \alpha_d |d\rangle$ ⁶; $\alpha_u^* \alpha_u$ is the probability of measuring $\sigma_z = +1$, while $\alpha_d^* \alpha_d$ is the probability that a measurement of σ_z will yield -1 . Formally we can denote the probability of measuring $+1$ and -1 as P_u and P_d respectively and write:

$$\begin{aligned} P_u &= \langle \mathcal{A} | u \rangle \langle u | \mathcal{A} \rangle \\ P_d &= \langle \mathcal{A} | d \rangle \langle d | \mathcal{A} \rangle. \end{aligned} \quad (1.3)$$

Spin space states have two dimensions

Probability amplitudes

Components α_u and α_d are called probability amplitudes, and their physical meaning is given by the square of the magnitude. This is the actual probability, and we want the sum of all probabilities to be one. This is equivalent to requiring that $|\mathcal{A}\rangle$ is normalized: $\langle \mathcal{A} | \mathcal{A} \rangle = 1$.

Now we will show why $|u\rangle$ and $|d\rangle$ have to be orthogonal:

$$\begin{aligned} \langle u | d \rangle &= 0 \\ \langle d | u \rangle &= 0. \end{aligned}$$

We try to give an idea with a *reductio ad absurdum*: if $|u\rangle$ and $|d\rangle$ were not orthogonal, the projection of one on the other would not be null. This means

⁵ We will show that these vectors are in fact orthogonal and why they need to be.

⁶ From now on we use "prepared" or "measured" as synonyms: every measurement is invasive and can change the spin state, so no matter what was the previous state, after a measurement the state is the one we have measured.

252 that if we orient \mathcal{A} along z and measure $\sigma_z = +1 = |\mathbf{u}\rangle$, we would have
 253 $\alpha_d = \langle \mathbf{d} | \mathbf{u} \rangle \neq 0$, which is a contradiction to experimental results. If $\alpha_d \neq 0$,
 254 then $\alpha_d^* \alpha_d > 0$; we started with a state prepared as $\sigma_z = +1$ and ended with
 255 a nonzero probability of measuring $\sigma_z = -1$: this is absurd.

Orthogonal states are mutually exclusive
 256 We can extend the reasoning to a general and key concept of quantum mechanics: two orthogonal states are distinct and mutually exclusive. If the system is in the first state, the probability of finding it in the second is zero.

257 Now we are ready to express spin states as linear combinations of the basis
 258 vectors $\mathcal{B} = \{|\mathbf{u}\rangle, |\mathbf{d}\rangle\}$. The representation of the basis vectors themselves is
 259 naturally easy:

$$|\mathbf{u}\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad (1.4)$$

$$|\mathbf{d}\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \quad (1.5)$$

260 To construct vector *right*, let's consider a spin prepared in the state $|\mathbf{r}\rangle$. If
 261 we measure σ_z , we have a 50% chance of obtaining +1 (and 50% for -1); this
 262 means that for $|\mathbf{r}\rangle$ we have $\alpha_u^* \alpha_u = \alpha_d^* \alpha_d = 1/2$. A vector that satisfies this
 263 constraint is:

$$|\mathbf{r}\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}. \quad (1.6)$$

264 The reasoning is the same for state *left*; we also add the constraint that a
 265 state *left* cannot be *right* and vice versa: $\langle \mathbf{r} | \mathbf{l} \rangle = \langle \mathbf{l} | \mathbf{r} \rangle = 0$. We can express
 266 *left* as:

$$|\mathbf{l}\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} \end{pmatrix}. \quad (1.7)$$

267 Lastly, the constraints to find explicit forms for *in* and *out* are:

- 268 • states must be orthogonal: $\langle \mathbf{i} | \mathbf{o} \rangle = \langle \mathbf{o} | \mathbf{i} \rangle = 0$;
- 269 • if we have a spin prepared as *in* or *out*:
 - 270 – equiprobability of measuring $\sigma_z = +1$ and $\sigma_z = -1$;
 - 271 – equiprobability of measuring $\sigma_x = +1$ and $\sigma_x = -1$.

272 Two vectors that satisfy these constraints are:

$$|\mathbf{i}\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{i}{\sqrt{2}} \end{pmatrix} \quad (1.8)$$

$$|\mathbf{o}\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ -\frac{i}{\sqrt{2}} \end{pmatrix}. \quad (1.9)$$

273 This last derivation shows why it is important that the state space is complex:
 274 if we only accepted real components for our vectors, the system of equations
 275 we have implicitly defined would not have any solution⁷.

⁷ To avoid confusion, we point out that $|\mathbf{i}\rangle$ is the ket of state *in*. *i*, instead, is the imaginary unit.

278 1.3 OBSERVABLES

279 We have learned that in classical mechanics we can trust our intuition, and
 280 we can do one or more measurements to know exactly the state of a system:
 281 a measurement does not perturb the state, which is the same before, during,
 282 and after the measurement.

283 In quantum mechanics the situation is more complex; our intuition is mis-
 284 leading, and we need mathematical tools to describe what we can measure:
 285 the observables. These tools are mathematical operators called *machines* (\mathbf{M})
 286 and have as both input and output state vectors.

287 **Axiom 2.** *Machines associated with observables are described by linear operators.*

288 We will show that machines are Hermitian operators, so let's start defining
 289 these operators and describing their properties⁸.

290 1.3.1 Hermitian operator

291 Formally, machines modify a state vector in this way:

$$\mathbf{M}|\mathbf{A}\rangle = |\mathbf{B}\rangle$$

292 The linearity of machines implies that:

$$\mathbf{M}|\mathbf{A}\rangle = |\mathbf{B}\rangle \Rightarrow \mathbf{M}z|\mathbf{A}\rangle = z|\mathbf{B}\rangle$$

293 and:

$$\mathbf{M}(|\mathbf{A}\rangle + |\mathbf{B}\rangle) = \mathbf{M}|\mathbf{A}\rangle + \mathbf{M}|\mathbf{B}\rangle.$$

294 If we choose a basis to represent machines and state vectors, we can write
 295 explicitly the linear operator as an $N \times N$ matrix, where N is the dimension
 296 of the vector space of the state vectors. A generic machine that transforms
 297 spins can be expressed as:

$$\mathbf{M} = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix}.$$

298 When we fix a basis, we are forced to express all state vectors and opera-
 299 tors in that basis, but now we have a set of rules to define the application of
 300 the operator to a state vector, i.e. the matrix multiplication:

$$\mathbf{M}|\mathbf{A}\rangle = \begin{pmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{pmatrix} \times \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \beta_2 \end{pmatrix} = |\mathbf{B}\rangle$$

301 When we consider a linear operator, we can search for eigenvalues and
 302 eigenvectors (if they exist). Eigenvectors are vectors that don't change their
 303 direction when multiplied by the operator; their magnitude is scaled by a
 304 constant factor called the eigenvalue. Formally:

Eigenvalues and eigenvectors

$$\mathbf{M}|\lambda\rangle = \lambda|\lambda\rangle$$

305 where $|\lambda\rangle$ is the eigenvector and λ the eigenvalue.

8 The reason why we need this kind of operator will be clear in Section 1.3.2.

306 Considering the transformation between ket $|A\rangle$ and $|B\rangle$: $\mathbf{M}|A\rangle = |B\rangle$,
 307 taking into account the dual space of bras and searching for a machine that
 308 transforms the bra $\langle A|$ into $\langle B|$, we cannot simply use the matrix having as
 309 elements the complex conjugate of \mathbf{M} ; the correct operator is the *Hermitian*
 310 *conjugate* of \mathbf{M} , which is the transpose of the matrix having as elements
 311 the complex conjugates of \mathbf{M} . We denote the Hermitian conjugate with the
 312 dagger \dagger :

$$\mathbf{M}^\dagger = [\mathbf{M}^*]^T = [\mathbf{M}^T]^*.$$

313 We can now write:

$$\mathbf{M}|A\rangle = |B\rangle \Rightarrow \langle A|\mathbf{M}^\dagger = \langle B|.$$

314 An operator that is equal to its Hermitian conjugate is called a *Hermitian*
 315 *operator*. Formally, \mathbf{M} is Hermitian if and only if

$$\mathbf{M} = \mathbf{M}^\dagger.$$

316 Hermitian operators have some important properties:

- 317 • all eigenvalues are real;
- 318 • eigenvectors form a *complete set*: all vectors obtained with the applica-
 319 tion of the operator can be expressed as a linear combination of eigen-
 320 vectors;
- 321 • if λ_1 and λ_2 are different eigenvalues, the associated eigenvectors are
 322 orthogonal;
- 323 • if two eigenvalues are equal (*degeneracy*), it is always possible to find
 324 two associated eigenvectors that are orthogonal.

Fundamental theorems 325 The last three properties can be summed up in the following way:

326 **Theorem 1.** *The eigenvectors of a Hermitian operator form an orthonormal basis.*

327 1.3.2 Principles of quantum mechanics

328 Let's introduce the first four principles of quantum mechanics, the ones
 329 about observables⁹.

330 **Principles 1.** *Observables in quantum mechanics are described by linear operators*
 331 \mathbf{L} .

332 \mathbf{L} must also be a Hermitian operator: we can consider this proposition an
 333 axiom itself or deduce it from the other principles.

334 **Principles 2.** *The results of a measurement can only be the eigenvalues associated*
 335 *with the observable operator.*

336 Calling λ_i a generic eigenvalue and $|\lambda_i\rangle$ the associated eigenvector, if the
 337 system is in the *eigenstate* $|\lambda_i\rangle$, the measurement always returns λ_i . Since
 338 all λ_i must be physical quantities they must be real, a peculiar property of
 339 Hermitian operators.

⁹ The fifth, and last one, concerns the temporal evolution. It will be discussed later on (Section 1.4).

340 **Principles 3.** *Unambiguously distinguishable states are represented by orthogonal
341 vectors.*

342 Distinguishable states can be separated without ambiguity by a measure-
343 ment. For example, if we want to distinguish between $|u\rangle$ and $|d\rangle$, we mea-
344 sure σ_z : *up* and *down* are distinct. We cannot, instead, say if a certain system
345 is in state *up* or *right*, because even if the system is in the state $|u\rangle$ we can
346 still measure σ_x and find (with 50% chance) that the system is in state $|r\rangle$.

347 The inner product is a measure of how much two states are indistin- *Overlap*
348 guishable; for that reason it is also called overlap. Two states are physically dis-
349 tinct if the overlap is zero.

$$\begin{aligned}\langle u | d \rangle &= 0 \\ \langle u | r \rangle &\neq 0\end{aligned}$$

350 **Principles 4.** *If the system is in state $|A\rangle$ and we measure the observable L , the
351 probability of obtaining λ_i is:*

$$P(\lambda_i) = \langle A | \lambda_i \rangle \langle \lambda_i | A \rangle .$$

352 where λ_i is a generic eigenvalue of L and $\langle \lambda_i |$, $|\lambda_i \rangle$ are the bra and ket asso-
353 ciated with that eigenvalue (eigenvector of λ_i).

354 1.3.3 Spin Operator

355 The principles tell us what properties a machine must have to represent an
356 observable. Let's construct the spin operator σ .

357 Until now, we have measured spins with the apparatus \mathcal{A} , orienting \mathcal{A}
358 along the component of our interest. σ is a mathematical tool that allows
359 us to make predictions about the result of a measurement with \mathcal{A} (fourth
360 principle); as we can rotate \mathcal{A} , we must also rotate σ (mathematically). For
361 this spatial property, σ is called a *3-vector operator*.

362 **OPERATOR σ_z :** Let's start with the simplest operator¹⁰. The second prin-
363 ciple says that all eigenvectors of σ_z are $|u\rangle$ and $|d\rangle$, with associated eigen-
364 values +1 and -1. We can write this assertion as equations:

$$\begin{aligned}\sigma_z |u\rangle &= |u\rangle \\ \sigma_z |d\rangle &= -|d\rangle .\end{aligned}$$

365 In matrix form:

$$\begin{aligned}\begin{pmatrix} (\sigma_z)_{11} & (\sigma_z)_{12} \\ (\sigma_z)_{21} & (\sigma_z)_{22} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} &= \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\ \begin{pmatrix} (\sigma_z)_{11} & (\sigma_z)_{12} \\ (\sigma_z)_{21} & (\sigma_z)_{22} \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} &= -\begin{pmatrix} 0 \\ 1 \end{pmatrix}.\end{aligned}$$

366 The solution of this system is¹¹:

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

¹⁰ This is because we have chosen $\mathcal{B} = \{|u\rangle, |d\rangle\}$ as the basis.

¹¹ It is easy to verify that this operator is also linear.

367 **OPERATOR σ_x :** With the same reasoning, we can construct the operator 368 along the x axis. We have already deduced the representations of *right* and 369 *left* in Equations 1.6 and 1.7. The equations that allow us to construct σ_x are:

$$\begin{pmatrix} (\sigma_x)_{11} & (\sigma_x)_{12} \\ (\sigma_x)_{21} & (\sigma_x)_{22} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix} = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

$$\begin{pmatrix} (\sigma_x)_{11} & (\sigma_x)_{12} \\ (\sigma_x)_{21} & (\sigma_x)_{22} \end{pmatrix} \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{pmatrix} = - \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{pmatrix}.$$

370 The solution of this system is:

$$\sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

371 **OPERATOR σ_y :** The last direction is along the y axis. Considering the 372 expressions for *in* and *out* given in Equations 1.8 and 1.9, and following the 373 second principle, we can write:

$$\begin{aligned} \sigma_y |i\rangle &= |i\rangle \\ \sigma_y |o\rangle &= -|o\rangle. \end{aligned}$$

374 We can rewrite this in matrix form, and the solution we would obtain is:

$$\sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}.$$

375 *Pauli matrices* 376 We have obtained a matrix representation of the three spin operators σ_z , σ_x , and σ_y :

$$\sigma_z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad \sigma_x = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \sigma_y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}. \quad (1.10)$$

377 These famous and important matrices are named after their inventor, Wolfgang 378 Ernst Pauli.

379 1.3.4 Theory and experiments

380 Thanks to the operators σ_z , σ_x , and σ_y , if we know the state vector, we can 381 statistically predict the result of a measurement of the spin along one of 382 the three coordinate axes. What can we say about a measurement taken by 383 orienting the apparatus \mathcal{A} along a generic direction?

384 Considering \mathcal{A} oriented along the unit vector \hat{n} , if σ behaves as a 3-vector, 385 in order to obtain σ_n we only need the inner product:

$$\sigma_n = \vec{\sigma} \cdot \hat{n}$$

386 Expanding the components:

$$\sigma_n = \sigma_x n_x + \sigma_y n_y + \sigma_z n_z.$$

387 If we choose the basis $\mathcal{B} = \{|u\rangle, |d\rangle\}$, we can use the Pauli matrices to 388 express in matrix form the expression for σ_n :

$$\sigma_n = n_x \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} + n_y \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} + n_z \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = \begin{pmatrix} n_z & n_x - i n_y \\ n_x + i n_y & -n_z \end{pmatrix}.$$

Given a direction (expressed by the unit vector \hat{n}), we can construct the matrix we have now made explicit, and then, after finding eigenvalues and eigenvectors, we can know all possible results of a measurement and obtain the probability associated with each result. For example, considering a direction in the $x-z$ plane, the operator σ_n would be:

$$\sigma_n = \begin{pmatrix} \cos \theta & \sin \theta \\ \sin \theta & -\cos \theta \end{pmatrix}$$

where θ is the angle between \hat{n} and z . For this matrix, the eigenvalues and eigenvectors are:

$$\lambda_1 = 1 \quad |\lambda_1\rangle = \begin{pmatrix} \cos \frac{\theta}{2} \\ \sin \frac{\theta}{2} \end{pmatrix}$$

and

$$\lambda_2 = -1 \quad |\lambda_2\rangle = \begin{pmatrix} -\sin \frac{\theta}{2} \\ \cos \frac{\theta}{2} \end{pmatrix}.$$

It should be pointed out that the theory is in agreement with experimental results¹². Eigenvalues are $+1$ and -1 , exactly the only results that the apparatus \mathcal{A} can retrieve. The probability of obtaining a certain result can be evaluated as:

$$\begin{aligned} P(+1) &= |\langle u | \lambda_1 \rangle|^2 = \cos^2 \frac{\theta}{2} \\ P(-1) &= |\langle u | \lambda_2 \rangle|^2 = \sin^2 \frac{\theta}{2} \end{aligned}$$

Lastly, let's calculate the average value for the measurement σ_n . From the first experiment we have seen in Section 1.1.1, we already know that the result of repeated measurements with \mathcal{A} is $\cos \theta$. Let's verify if our model is coherent with the world.

Expectation value

Expected values are obtained as:

$$\langle L \rangle = \sum_i \lambda_i P(\lambda_i)$$

Specifically:

$$\langle \sigma_n \rangle = (+1) \cos^2 \frac{\theta}{2} + (-1) \sin^2 \frac{\theta}{2} = \cos \theta.$$

This is in complete agreement with the experimental results.

Before going on, we present, without proof, a useful theorem about expectation values:

Theorem 2. *To know the expectation value of an observable, we can simply place the operator associated with the observable between the bra and ket of the state vector:*

$$\langle L \rangle = \langle A | L | A \rangle \tag{1.11}$$

where L is an observable, $|A\rangle$ is a state vector, and $\langle A|$ is the corresponding bra.

¹² If not, we must abandon this model and build another one.

⁴¹⁴ **1.3.5 Operator and Measure**

⁴¹⁵ Operators allow us to know the probability of measuring a certain spin given
⁴¹⁶ the direction of the measurement and the state vector. This probability is
⁴¹⁷ expressed by the state vector that we obtain when we apply the operator σ
⁴¹⁸ to the initial state.

⁴¹⁹ It is important not to confuse the measurement act with the application
⁴²⁰ of a machine that represents the observables. The spin state after the mea-
⁴²¹ surement is not the same as the one we obtain after the application of the
⁴²² operator. The operator is only an abstract mathematical construct that allows
⁴²³ us to make statistical predictions about results, but doesn't have physical im-
⁴²⁴ plications.

⁴²⁵ Let's consider an example to clarify the previous assertion. Having a spin
⁴²⁶ prepared in the *up* state, its state vector is $|u\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. If we apply the operator
⁴²⁷ σ_z , we would obtain again $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$, and if we measure the spin with \mathcal{A} oriented
⁴²⁸ along z , it will always display +1, and we conclude that the state after the
⁴²⁹ measurement is $|u\rangle$.

⁴³⁰ Consider now a spin prepared *right*, i.e. $|r\rangle = 1/\sqrt{2}|u\rangle + 1/\sqrt{2}|d\rangle$. Applying
⁴³¹ again the operator σ_z , the new state vector is $1/\sqrt{2}|u\rangle - 1/\sqrt{2}|d\rangle$. This
⁴³² vector tells us the probability of measuring $\sigma_z = +1$ (50%), but it is not the
⁴³³ spin state after the measurement. Using the apparatus \mathcal{A} , we could measure:

- ⁴³⁴ • +1: the final state will be *up*;
- ⁴³⁵ • -1: the final state will be *down*.

⁴³⁶ No matter the result of the measurement, the final state will be different
⁴³⁷ from the one we obtained by applying the operator.

⁴³⁸ **1.4 TEMPORAL EVOLUTION**

⁴³⁹ Let's explore the laws that describe the temporal evolution of a quantum
⁴⁴⁰ system. In particular, we will see how the state vector can evolve over time.

⁴⁴¹ **1.4.1 Unitarity**

⁴⁴² In classical mechanics we are used to having a motion law that links differ-
⁴⁴³ ent states of our system deterministically; this means being able to know
⁴⁴⁴ precisely the following state given the previous one. A good law, however,
⁴⁴⁵ doesn't allow us only to know the future, but also the past states that brought
⁴⁴⁶ the system to the current state¹³.

⁴⁴⁷ *Reversibility* In other words, we want physical transformations to be reversible. This
⁴⁴⁸ requirement is so important that we call this property the *minus first law*,
⁴⁴⁹ because it underlies everything else. If we think about the system states as
⁴⁵⁰ nodes in an oriented graph, reversibility imposes that each node has exactly
⁴⁵¹ one input edge and one output edge. This fundamental law is also true

¹³ For example, if we observe a ball in free fall touching the floor with a certain speed and at a certain time, we can know exactly when and from what height the ball started its fall.

452 in quantum mechanics and is called *unitarity*¹⁴, and it assures us that no
453 information is lost. The unitarity law can be expressed as:

454 **Axiom 3.** *If two identical isolated systems are in different states, they stay in differ-
455 ent states, and they were in different states in the past.*

456 **1.4.2 Time-Development Operator**

457 Considering a system in the state $|\Psi(t)\rangle$, where the t indicates that the state
458 vector evolves over time, quantum motion equations allow us to obtain the
459 state at time t given the initial state:

$$|\Psi(t)\rangle = \mathbf{U}(t) |\Psi(0)\rangle. \quad (1.12)$$

460 Thanks to the operator $\mathbf{U}(t)$ we can know exactly the state vector $|\Psi(t)\rangle$ at *Determinism*
461 time t , given $|\Psi(0)\rangle$. This assertion can be rephrased as:

462 **Axiom 4.** *The temporal evolution of the state vector is deterministic.*

463 Quantum mechanics is still non-deterministic, because knowing the state
464 vector doesn't mean knowing the result of a measurement.

465 In order for $\mathbf{U}(t)$ to behave as we want, it has to:

- 466 • be a linear operator;
- 467 • respect reversibility.

468 The second constraint allows us to define the mathematical properties of
469 $\mathbf{U}(t)$. Considering two initially different states $|\Psi(0)\rangle$ and $|\Phi(0)\rangle$, since there
470 exists an experiment capable of certainly distinguishing the states, $|\Psi(0)\rangle$
471 and $|\Phi(0)\rangle$ must be orthogonal:

$$\langle \Psi(0) | \Phi(0) \rangle = 0.$$

472 The minus first law assures that during the entire temporal evolution of the
473 two systems, the state vectors $|\Psi(t)\rangle$ and $|\Phi(t)\rangle$ will continue to be distin-
474 guishable (orthogonal):

$$\langle \Psi(t) | \Phi(t) \rangle = 0 \quad \forall t \geq 0.$$

475 If we rewrite this equation using Formula 1.12, we obtain:

$$\langle \Psi(0) | \mathbf{U}^\dagger(t) \mathbf{U}(t) | \Phi(0) \rangle = 0.$$

476 From this we can see that $\mathbf{U}^\dagger(t) \mathbf{U}(t)$ must behave as the identity operator,
477 that is:

$$\mathbf{U}^\dagger(t) \mathbf{U}(t) = \mathbf{I}. \quad (1.13)$$

478 An operator that behaves as \mathbf{U} is *unitary*.

479 **Principles 5.** *The temporal evolution of state vectors is unitary.*

480 From the unitarity of \mathbf{U} descends the *conservation of overlaps*: the overlap
481 between two states (their inner product), subjected to the same temporal-
482 development operator, is preserved over time.

¹⁴ We will see in the next paragraph the reason for this name

⁴⁸³ **1.4.3 The Hamiltonian**

⁴⁸⁴ Often, in classical physics, a motion law is the result of a differential equation
⁴⁸⁵ where we have exchanged a finite time interval with an infinite number of
⁴⁸⁶ infinitesimal intervals.

⁴⁸⁷ In quantum mechanics we can follow the same path and consider time
⁴⁸⁸ intervals ϵ close to zero. In this scenario, after an ϵ amount of time, the state
⁴⁸⁹ vector will change slightly and “smoothly”, and the operator $\mathbf{U}(\epsilon)$ will be
⁴⁹⁰ very similar to the identity. We can rewrite $\mathbf{U}(\epsilon)$ in order to highlight the
⁴⁹¹ difference with the identity \mathbf{I} as:

$$\mathbf{U}(\epsilon) = \mathbf{I} - i\epsilon\mathbf{H}. \quad (1.14)$$

⁴⁹² For now, i is a mere scale factor that later will help us recognize in \mathbf{H} the
⁴⁹³ quantum version of the classical Hamiltonian.

⁴⁹⁴ We can now express the infinitesimal evolution of a quantum system by
⁴⁹⁵ combining Equations 1.12 and 1.14:

$$|\Psi(\epsilon)\rangle = |\Psi(0)\rangle - i\epsilon\mathbf{H}|\Psi(0)\rangle.$$

⁴⁹⁶ Bringing to the left the time interval:

$$\frac{|\Psi(\epsilon)\rangle - |\Psi(0)\rangle}{\epsilon} = -i\mathbf{H}|\Psi(0)\rangle.$$

⁴⁹⁷ Now considering the limit for $\epsilon \rightarrow 0$, we can see in the left member the time
⁴⁹⁸ derivative of the state vector:

$$\frac{\partial |\Psi(t)\rangle}{\partial t} = -i\mathbf{H}|\Psi(0)\rangle.$$

⁴⁹⁹ Before using \mathbf{H} as the quantum Hamiltonian, we have to verify the di-
⁵⁰⁰ mensional correctness. As in classical mechanics, the Hamiltonian is the
⁵⁰¹ mathematical construct that represents the energy. In our formula, however,
⁵⁰² ignoring the state vector, we have the inverse of time on the left and the
⁵⁰³ energy on the right. To resolve this problem, let's introduce an important
⁵⁰⁴ physical constant: the reduced Planck constant, \hbar .

⁵⁰⁵ The equation becomes:

Time-dependent Schrödinger equation

$$\hbar \frac{\partial |\Psi\rangle}{\partial t} = -i\mathbf{H}|\Psi\rangle \quad \text{or} \quad \frac{\partial |\Psi\rangle}{\partial t} = \frac{-i\mathbf{H}|\Psi\rangle}{\hbar}. \quad (1.15)$$

⁵⁰⁶ The constant \hbar has units of $\text{kg} \cdot \text{m}^2/\text{s}$ and resolves the incompatibility between
⁵⁰⁷ the two members. This equation is fundamental and is called the *generalized*
⁵⁰⁸ *Schrödinger equation*, or time-dependent Schrödinger equation. If we know
⁵⁰⁹ the Hamiltonian of an undisturbed system, we can know the evolution of
⁵¹⁰ the state vector.

⁵¹¹ If \mathbf{H} represents the energy of the system, we should be able to measure it,
⁵¹² so \mathbf{H} has to be an observable. If \mathbf{H} is an observable, it must be a Hermitian
⁵¹³ operator; let's verify it. Starting from Equation 1.13 and substituting \mathbf{U} with
⁵¹⁴ Expression 1.14, we obtain:

$$(\mathbf{I} + i\epsilon\mathbf{H}^\dagger)(\mathbf{I} - i\epsilon\mathbf{H}) = \mathbf{I}.$$

515 Expanding to first order in ϵ , we find:

$$\mathbf{H}^\dagger - \mathbf{H} = 0 \Rightarrow \mathbf{H}^\dagger = \mathbf{H}.$$

516 We have concluded that \mathbf{H} is an Hermitian operator that represents an
517 observable: the energy of the system. Eigenvalues of \mathbf{H} are the results of all
518 possible direct measurements of the energy of the system.

Quantum Hamiltonian

519 1.4.4 Commutators

520 In a system that evolves with time, we expect that the expectation values for
521 a certain observable \mathbf{L} will also change. Thanks to equation 1.11 on page 15,
522 we can write explicitly the time dependence of expectation values:

$$\langle \mathbf{L} \rangle = \langle \Psi(t) | \mathbf{L} | \Psi(t) \rangle.$$

523 The time derivative¹⁵ is:

$$\frac{d}{dt} \langle \Psi(t) | \mathbf{L} | \Psi(t) \rangle = \langle \dot{\Psi}(t) | \mathbf{L} | \Psi(t) \rangle + \langle \Psi(t) | \mathbf{L} | \dot{\Psi}(t) \rangle.$$

524 Substituting bra and ket with the time-dependent Schrödinger Equation 1.15
525 (namely $|\dot{\Psi}(t)\rangle = \frac{-i}{\hbar} \mathbf{H} |\Psi(t)\rangle$), we obtain:

$$\frac{d}{dt} \langle \Psi(t) | \mathbf{L} | \Psi(t) \rangle = \frac{i}{\hbar} \langle \Psi(t) | \mathbf{H} \mathbf{L} | \Psi(t) \rangle - \frac{i}{\hbar} \langle \Psi(t) | \mathbf{L} \mathbf{H} | \Psi(t) \rangle.$$

526 That can be rewritten as:

$$\frac{d}{dt} \langle \Psi(t) | \mathbf{L} | \Psi(t) \rangle = \frac{i}{\hbar} \langle \Psi(t) | [\mathbf{H}, \mathbf{L}] | \Psi(t) \rangle.$$

527 The quantity $\mathbf{H}\mathbf{L} - \mathbf{L}\mathbf{H}$ is called the *commutator*, and since, in general, the
528 product between operators (matrices) is not commutative, the commutator
529 is not zero (when it is zero, we say that \mathbf{H} and \mathbf{L} commute). Commutators
530 are important in physics, and the commutator between two operators, in this
531 case \mathbf{H} and \mathbf{L} , is denoted by:

$$\mathbf{H}\mathbf{L} - \mathbf{L}\mathbf{H} = [\mathbf{H}, \mathbf{L}].$$

532 With the commutator we can express concisely the derivative of the expec-
533 tation value for the observable \mathbf{L} :

$$\frac{d}{dt} \langle \mathbf{L} \rangle = \frac{i}{\hbar} \langle [\mathbf{H}, \mathbf{L}] \rangle \quad (1.16)$$

534 or equivalently:

$$\frac{d}{dt} \langle \mathbf{L} \rangle = -\frac{i}{\hbar} \langle [\mathbf{L}, \mathbf{H}] \rangle. \quad (1.17)$$

535 This equation links variations of the expectation values of an observable
536 (\mathbf{L}) to the expectation values of another physical observable ($-\frac{i}{\hbar} [\mathbf{L}, \mathbf{H}]$)¹⁶.

¹⁵ Derivative of a product: \mathbf{L} doesn't depend on time and the dot denotes the time derivative (Newton notation).

¹⁶ It is possible to demonstrate that if \mathbf{L} and \mathbf{H} are Hermitian, then $[\mathbf{L}, \mathbf{H}]$ is also Hermitian.

537 **1.4.5 Conservation of Energy**

538 In quantum mechanics, when we say that a quantity is conserved, we mean
 539 that the expectation value of that quantity doesn't change. If we look at
 540 Equation 1.17, the condition for the expectation value not to change is that
 541 the commutator between this quantity and the Hamiltonian is zero. It is
 542 possible to demonstrate that:

543 **Theorem 3.** *Having an observable \mathbf{Q} , if $[\mathbf{Q}, \mathbf{H}] = 0$, then every power satisfies*
 544 *$[\mathbf{Q}^n, \mathbf{H}] = 0$. This means that the expectation value $\langle \mathbf{Q} \rangle$ is conserved, and any*
 545 *power of the expectation value $\langle \mathbf{Q}^n \rangle$ does not change with time.*

546 The most obvious quantity that is conserved is the Hamiltonian \mathbf{H} and,
 547 since every operator commutes with itself, we always have:

$$[\mathbf{H}, \mathbf{H}] = 0.$$

548 We can conclude that, under very general conditions, energy is conserved
 549 in quantum mechanics.

550 **1.5 CONCLUSIONS**

551 We conclude this chapter with a recap of what we have discovered in these
 552 pages, trying to put everything together to answer the question that opened
 553 this chapter: what are the physical limits of quantum computing, and why
 554 must our algorithms be reversible?

555 We started the chapter with an experiment that shows that quantum me-
 556 chanics is not deterministic. We can, however, make some predictions if we
 557 consider the expectation value of a measurement instead of a single result.

558 We have built state vectors and understood their mathematical meaning,
 559 focusing on the fact that knowing the state vector doesn't allow us to know
 560 the result of a measurement. We have defined the inner product between
 561 state vectors, observed that it is a measure of the overlap between states,
 562 and concluded that two distinguishable states must be orthogonal.

563 We have linked a state vector to the result of a measurement –to be precise,
 564 to the average of the results of multiple measurements– with machines, Her-
 565 mitian operators that represent observables. We have built the spin operator
 566 and used it to predict the result of a simple experiment, showing how the
 567 theory we have built so far is in accordance with experimental results.

568 Our introduction continues with the analysis of the temporal evolution of
 569 a quantum system. We have described the evolution of a state vector with an
 570 unitary operator; the application of this operator to a state vector produces
 571 the new state in which the system will be. We understood that the tempo-
 572 ral evolution of the state vector is deterministic and that indeterminacy is
 573 caused only by the act of measuring.

574 Considering infinitesimal time intervals, we have deduced the time-dependent
 575 Schrödinger equation and, thanks to this equation, we have shown how to
 576 describe the temporal evolution of expectation values for a certain observ-
 577 able. During this analysis, we also introduced the Hamiltonian of the system,
 578 a Hermitian operator that describes the energy of the system.

579 The discussion ends with a comforting result: as in classical physics, the
 580 energy of a closed system is conserved. We have obtained this result by pre-
 581 senting the commutator and linking the temporal evolution of an observable
 582 with the commutator between the observable and the Hamiltonian (energy)
 583 of the system. The commutator of the Hamiltonian with itself is trivially
 584 zero, so the expectation value for the energy doesn't change.

585 All the information that we have learned allows us to understand the con-
 586 straint of writing only reversible algorithms for quantum-gate-based quan-
 587 tum computers. Quantum gates operate on qubits through physical transfor-
 588 mations¹⁷. These transformations, like all transformations in quantum me-
 589 chanics, are described by unitary operators that are intrinsically reversible.
 590 This means that all quantum gates are reversible.

591 In other words, we can build only quantum gates that, having as input
 592 different (distinguishable) states, return orthogonal states; also, due to the
 593 conservation of overlaps, the inner product between input states is conserved
 594 during the quantum gate transformation.

595 Reversibility doesn't mean that we can go forward and backward in time
 596 as we please, but that all quantum gates express injective functions: if we
 597 know the output, we can know the input, or in more physical terms, if
 598 we know the final state of qubits¹⁸ and the transformations applied to this
 599 system (i.e., those implemented by the quantum gates), we can determine
 600 the initial state.

601 Since every quantum algorithm has to be implemented as a path through
 602 quantum gates, and every quantum gate is reversible, the algorithms as a
 603 whole must also be reversible.

¹⁷ How depend strongly on the particular physical implementation.

¹⁸ This is a complex system (composed of more than one qubit); to fully understand these systems, we should take into account entanglement. Since our discussion is already quite long, and the temporal evolution of an entangled system is still unitary (reversible), we exclude entanglement from our introduction.

604 2 | QUANTUM GATE

605 3 | QUANTUM ANNEALING

606 4 | ONTOLOGY

607 In this chapter we explain what kind of knowledge base (KB) an ontology
608 is, how to build an ontology, and why this knowledge representation is
609 important. To clarify and demonstrate why ontologies are useful, we present
610 an example of a foundational ontology¹, briefly discussing its utility.

611 The rest of the chapter is about reasoning on ontologies; we discuss the
612 semantics of the formal language used to represent knowledge, what we
613 mean when saying interpretation of a KB, and the complexity of finding an
614 interpretation.

615 4.1 KNOWLEDGE BASE

616 In the field of information technologies, an ontology is a structured represen-
617 tation of knowledge about a certain domain of interest; however, the study
618 of knowledge began much before informatics. To better understand what an
619 ontology is, let's start with the philosophical definition and then point out
620 the differences between this vision and the IT one.

621 4.1.1 Ontology in philosophy

622 Ontology was born as a branch of philosophy. In this context it is the sci-
623 ence of what is, of the kinds and structures of objects, properties, events,
624 processes, and relations in every area of reality [1].

625 The goal of an ontology is to give a definitive and exhaustive classification
626 of entities in all spheres of being. With the term "definitive" we mean that
627 an ontology should answer questions such as: "What classes of entities are
628 needed for a complete description and explanation of all the goings-on in the
629 universe?" With the term "exhaustive", instead, we mean that all types of
630 entities and relations between these entities are included in our ontology [1].

631 4.1.2 Ontology in computer science

632 Thanks to the advent of the internet and the development of bigger and
633 bigger software used by bigger and bigger groups of users, what we might
634 call the Tower of Babel problem emerged. Each research group develops
635 its KB with terms and concepts shared and accepted only inside the group.
636 For example, different databases may use identical labels but with different
637 meanings, and the same meaning may be expressed with different names [1].

¹ a very general template that can be used as base (foundation) to build an ontology about a specific domain (more about that in Section 4.2.2).

638 To address the incompatibility problem between software, databases, and
 639 research groups, ontologies have become an important research topic in com-
 640 puter science where the goal is to define standards for data exchange, infor-
 641 mation integration, and interoperability [2].

642 In this field the term ontology gains a new meaning:

643 **Definition 1.** *Ontologies represent a formal and explicit specification of a shared*
 644 *conceptualization [3].*

645 In this definition the keywords are:

646 **CONCEPTUALIZATION:** an ontology creates an abstract model identifying
 647 and defining only the relevant concepts;

648 **EXPLICIT:** the types of concepts and constraints on their use are explicitly
 649 defined;

650 **FORMAL:** an ontology should be machine-readable;

651 **SHARED:** the knowledge represented by the ontology has to be accepted
 652 by a group of people, ideally by everyone.

653 When we use an ontology to represent knowledge we are describing a
 654 graph where entities are bound together through relationships, and classi-
 655 fied according to a formal description of the world [4]. Knowledge bases
 656 expressed with this formalism are divided into two components [5]:

657 **T-BOX:** stores a set of universally quantified assertions (inclusion asser-
 658 tions) stating general properties of concepts and roles;

659 **A-BOX:** contains assertions on individual objects (instance assertions).

660 The T-Box is the conceptualization of the world, while the A-Box is a certain
 661 instance of the world we have modelled in the T-Box.

662 We can see some similarities between an ontology and a database: the T-
 663 Box can be seen as the Entity-Relation schema and the A-Box as the set of all
 664 entries of the database. There is, however, a logical difference between the
 665 world represented by an ontology and the world represented by a database.

666 Databases make the *closed world assumption*: everything that is not present
 667 in the database is automatically false; for example, if a person does not
 668 appear in a bank registry it means that that person is not a client of the
 669 bank.

670 Ontologies, on the other hand, make the *open world assumption* [6], which
 671 means, for example, that we can assert that a certain person is a parent even
 672 if we have not specified any son or daughter.

673 4.1.3 OWL Language

674 OWL 2 Web Ontology Language is an ontology language for the Semantic
 675 Web with a formally defined meaning [7]. Thanks to OWL we can model
 676 classes and relations between classes (T-Box) and individuals with their spe-
 677 cific properties and relations between individuals (A-Box).

678 OWL is a declarative language and defines the state of the world in a
 679 logical way. In particular, we are interested in OWL DL where the meaning
 680 of ontologies expressed with this language is assigned in a Description Logic
 681 style. OWL DL is, therefore, decidable and an appropriate tool (so-called
 682 reasoner) can then be used to infer further information about that state of
 683 the world [7].

684 OWL per se does not specify any syntax, it states only what can or cannot
 685 be expressed in an ontology. The World Wide Web Consortium (W3C) stan-
 686 dardizes various syntaxes, some inspired by functional languages, others
 687 more suitable for storing on web pages. The only syntax that must be imple-
 688 mented by all tools to be compliant with the OWL standard is the RDF/XML
 689 syntax [7] (examples of this syntax are provided in Section 4.2.1).

690 4.1.4 Importance of ontologies

691 Ontologies are important in various fields, from interoperability to machine
 692 learning.

693 In the Semantic Web context, ontologies are a main vehicle for data integra-
 694 tion, sharing, and discovery [8]. Different research groups can use the same
 695 ontology to share a unified vocabulary that helps build common knowledge
 696 and helps to better integrate the results obtained by each group.

697 In a more commercial scenario, an ontology can be used as a transla-
 698 tion layer between different databases or software that are built by different
 699 teams and use different vocabularies.

700 In the machine learning field an ontology could be used to support the
 701 sharing and reuse of formally represented knowledge among neuro-symbolic
 702 AI systems [3].

703 4.2 EXAMPLE ONTOLOGIES

704 To help understanding the structure of ontologies and to show a practical
 705 example of ontology, we present two ontologies: a simple ontology about
 706 family relationships and DOLCE, a foundational ontology.

707 4.2.1 Simple ontology

708 This simple ontology about parental relationships shows the basic structure
 709 of an ontology, helping to understand the graph structure of these KBs and
 710 the relations between the T-Box and A-Box.

711 In Figure 4.1, we can see the T-Box of the ontology: this structure specifies
 712 what our domain of interest is, and what entities could possibly populate
 713 our world. This ontology is about people, so the main class/concept is
 714 People: this class has several subclasses that represent parents, children, and
 715 married people. We can assert that a person belongs to the married class
 716 without specifying the partner (open world assumption) but we can also
 717 infer that a person belongs to the parents class because we have created a
 718 relationship of type parent_of between that person and another person.

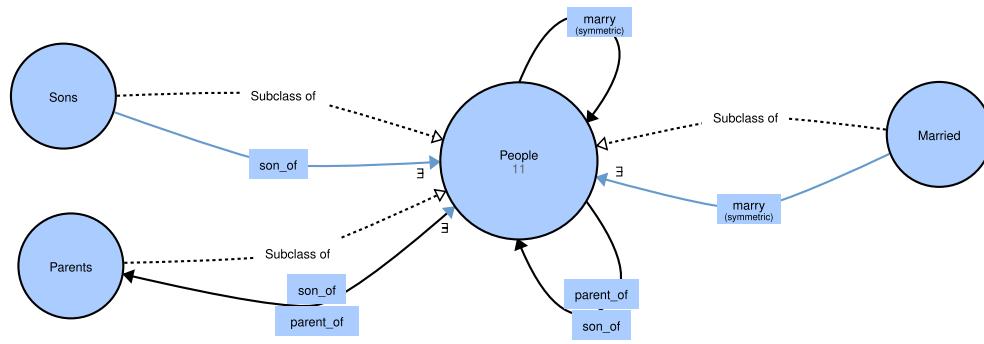


Figure 4.1: Graph for T-Box

719 OWL allows us to express rules to infer when a member of a class belongs
 720 also to another class. The following code shows (in the RDF/XML syntax)
 721 the definition of the class Parent²:

```

1 <owl:Class rdf:about="http://people#Parent">
2   <owl:equivalentClass>
3     <owl:Restriction>
4       <owl:onProperty rdf:resource="http://people#parent_of"/>
5       <owl:someValuesFrom rdf:resource="http://people#Person"/>
6     </owl:Restriction>
7   </owl:equivalentClass>
8   <rdfs:subClassOf rdf:resource="http://people#Person"/>
9 </owl:Class>
```

Listing 4.1: Definition of parents

722 At line 8 we can see that Parent is a subclass of People, and at lines 4 and
 723 5 it is specified that a parent is a person that is parent_of another person.

724 From Figure 4.1 we can also see some properties of the relations:

- 725 • relation marry is symmetric;
- 726 • relation parent_of is the inverse of son_of;
- 727 • we can specify a domain and a range for relations.

728 OWL gives us constructs for all of these specifications (and other more complex ones).

729 Now we can populate the ontology by adding individuals and relations between individuals. For this
 730 small example we take inspiration
 731 from the Simpson family, and in the
 732 family tree (Figure 4.2 on the right)
 733 we can see the small portion of the
 734 family represented. To show what
 735 we mean by open world assumption
 736 we have asserted that Jackie is a mar-
 737 ried person even if in our representation there is no husband.

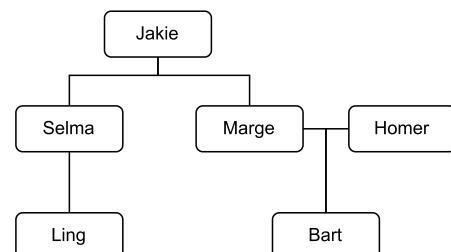


Figure 4.2: Simpson family tree

² The complete code of the ontology can be seen at [url](#).

741 Our ontology covers a small domain, the types of entities that populate
 742 our model are very limited; the next example shows the commitment of
 743 engineering an ontology to represent virtually anything in the universe.

744 4.2.2 DOLCE ontology

745 DOLCE (Descriptive Ontology for Linguistic and Cognitive Engineering) is a
 746 top-level (foundational) ontology [9]; this means that this ontology describes
 747 fundamental aspects of reality and should be used as a base for constructing
 748 an ontology about a particular domain of interest. For this reason DOLCE
 749 defines only the T-Box; the user will then expand the T-Box with specific
 750 classes and relations of interest, and lastly will populate the A-Box.

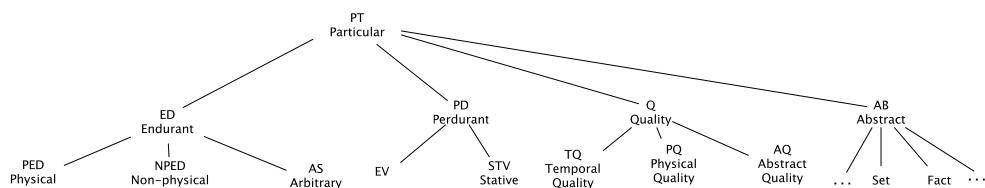


Figure 4.3: First layer of DOLCE taxonomy

751 **STRUCTURE OF DOLCE:** in DOLCE we can model the modification of ob-
 752 jects during time; for this reason DOLCE distinguishes between endurants
 753 and perdurants. Endurants may acquire and lose properties and parts through
 754 time, perdurants are fixed in time [9]. With a simplification we can see en-
 755 durants as the physical entities that are modified by the passing of time (like
 756 objects, animals, and people) and perdurants as events that, once they have
 757 passed, cannot be changed anymore (like a tennis match or a conference).

758 The relation connecting endurants and perdurants is called participation.
 759 A physical entity can be in time by participating in a perdurant, and perdu-
 760 rants happen in time by having endurants as participants [9].

761 Another important aspect of DOLCE is the way we attribute a property to
 762 an entity; this is done by using qualities, which are what can be perceived
 763 and measured. To do so we can assert that a certain entity has a specific
 764 quality and then, when it is possible, quantify that quality.

765 **IMPORTANCE OF DOLCE:** foundational ontologies can be useful in several
 766 fields, from conceptual modeling to natural language processing. DOLCE,
 767 today, is used in a variety of domains where it provides the general cate-
 768 gories and relations needed to give a coherent view of reality [9].

769 4.3 REASONING ON ONTOLOGY

770 In Section 4.1.3 we have introduced the standard language to encode an
 771 ontology; in order to infer new information, starting from the one we already
 772 have, we need to better specify the semantics of OWL DL.

773 4.3.1 *SROIQ* DL

774 The semantics of OWL DL extends the semantics of the description logic
 775 (*DL*) *SROIQ* to provide support for datatypes and punning [10]. For con-
 776 structs available both in OWL DL and in *SROIQ* the semantics correspond
 777 exactly.

778 Description logics allow the modeling of the domain of interest with three
 779 kinds of entity: concepts, roles, and individual names. These entities cor-
 780 respond to unary predicates, binary predicates, and constants in first-order
 781 logic [6]. From the point of view of ontology and OWL, concepts are classes,
 782 roles are relationships, and individual names are the individuals that can
 783 belong to one or more classes.

784 *SROIQ* is one of the most expressive description logics where we have
 785 constructors for:

- 786 • transitive roles: \mathcal{S}
- 787 • role inclusions, local reflexivity, universal role, symmetry, asymmetry,
 role disjointness, reflexivity, and irreflexivity: \mathcal{R} ;
- 788 • nominals: \mathcal{O} ;
- 789 • inverse roles: \mathcal{I} ;
- 790 • qualified number restrictions: \mathcal{Q} .

792 For example, we can construct the ontology shown in Figures 4.1 and 4.2
 793 with a set of assertions like:

794 person(selma) married(jackie) parent_of(marge, bart)

795 Each of these statements is called an axiom and the set of all axioms consti-
 796 tutes our KB.

797 4.3.2 Interpretation of a knowledge base

798 An interpretation I consists of a domain Δ^I and an interpretation function $.^I$
 799 that maps:

$$\begin{aligned} \text{concept } A &\rightarrow A^I \subseteq \Delta^I \\ \text{role } R &\rightarrow R^I \subseteq \Delta^I \times \Delta^I \\ \text{named individual } a &\rightarrow a^I \in \Delta^I \end{aligned}$$

800 In other words I assigns a fixed meaning to all entities in the KB [6]. By
 801 having a fixed meaning, we can say if an axiom α holds in I or not; in the
 802 first case we say that I satisfies α and we write $I \models \alpha$.

803 If all axioms in an ontology are satisfied by I we say that I is a *model* of the
 804 ontology. An ontology is consistent if it accepts at least one model.

805 A reasoner should at least be capable of saying if an ontology is consistent,
 806 but we are also interested in querying knowledge to retrieve new informa-
 807 tion.

808 **QUERY INTERPRETATION:** Considering a KB K , a query q consists of ax-
 809 iom templates where \mathcal{SROIQ} axioms are composed of concept names, role
 810 names, and individual names, but also of concept variables, role variables,
 811 and individual variables. A solution for the query is an interpretation μ that
 812 allows us to rewrite all variables in q with names; we denote with $\mu(q)$ the
 813 result of the substitution.

814 The evaluation of q over K is a set of solutions μ with: [11]

$$815 \quad \{ \mu | K \cup \mu(q) \text{ is a } \mathcal{SROIQ} \text{ knowledge base and } K \models \mu(q) \}$$

816 In other words μ binds all free variables of q to names present in K [11].

817 A naive approach to find the solution to a query is to simply test for each
 818 possible solution mapping μ , if $K \models \mu(q)$; however, in the worst case, the
 819 number of mappings that have to be tested is exponential in the number of
 820 variables in the query [11].

821 4.3.3 Complexity of reasoning

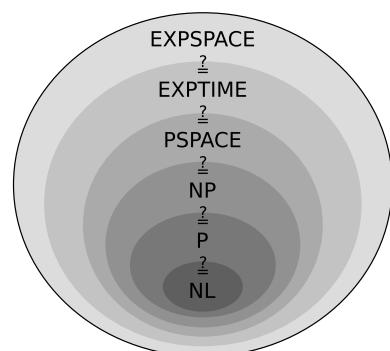
822 Since presenting an actual algorithm for reasoning on ontologies is out of
 823 the scope of this work, we only give some hints about the reasons for the
 824 complexity and then present the theoretical results that prove the problem
 825 of reasoning on ontology is at least PSpace-hard.

826 It is easy to convince oneself that the more axioms there are in an ontology,
 827 the fewer interpretations exist that satisfy all axioms. On the other hand, if
 828 an ontology has fewer models, the more axioms hold in all of them and the
 829 more logical consequences follow from the ontology.

830 In other words the semantics of description logics are *monotonic*: the more
 831 knowledge we embed in an ontology, the more results it returns [6]. A more
 832 formal view is given in [12], where two sources of complexity are identified:

- 833 • OR-branching: the presence of disjunctive constructors;
- 834 • AND-branching: the presence of qualified existential and universal
 835 quantifiers.

836 The AND-branching is responsible for the exponential size of a single inter-
 837 pretation, and the OR-branching is responsible for the exponential number
 838 of different interpretations.



849 **Figure 4.4:** Complexity classes

839 To discuss the complexity of reasoning we take into account the description
 840 logic \mathcal{ALC} ; this DL is a restriction of
 841 \mathcal{SROIQ} [6], so its complexity is a lower bound for \mathcal{SROIQ} . It is possible to prove
 842 the PSpace-hardness of satisfiability in
 843 \mathcal{ALC} [12], therefore also \mathcal{SROIQ} DL is
 844 at least PSpace-hard.

845 This result shows that, unless PSpace =
 846 PTime, the exponential time complexity
 847 of any algorithm that makes inference on
 848 an ontology cannot be improved .

851 For those interested in some numerical examples to better understand
852 what this class of complexity means in a real context, [13] presents the rea-
853 soner Hermit and evaluates its performance on some real ontologies.

854 **4.4 CONCLUSIONS**

855 In this chapter we have explained what an ontology is and we have moti-
856 vated the interest in this field.

857 We showed that an ontology can be useful both in academic research and
858 in industry. Moreover, being a formal and machine-readable structure, it
859 can be queried and used to perform logically demonstrable reasoning whose
860 subject is precisely the knowledge represented within the ontology.

861 We have shown both theoretically and with examples what can be ex-
862 pressed in an ontology and what cannot. We have formally defined what
863 the interpretation of a KB is and showed what a query and its results are.

864 Lastly, we have characterized the complexity of reasoning on ontologies.
865 This complexity is what motivated us to search for other paradigms to infer
866 new knowledge starting from an ontology. In the next chapters we will build
867 the tools necessary to achieve this goal.

868

Part II

869

TOOLS

5 ENVIRONMENT SETUP

871 In this chapter we describe the environment, libraries and tools we use to
872 execute our tests.

873 In the following sections we install the SDKs to develop and interact with
874 quantum computers from IBM and D-Wave. We also present two other use-
875 ful tools to easily write optimization problems.

876 5.1 PYTHON ENVIRONMENT

877 The language used to interface with quantum computers is usually Python.
878 In this section we create a virtual environment in Python in order to commu-
879 nicate with the IBM quantum computer and the D-Wave quantum computer.

880 For our tests we manage Python environments with `conda`. Let's start by
881 creating the virtual environment named `quantum` and activating it with:

```
882 $ conda create --name quantum python=3.12 pip  
883 $ conda activate quantum
```

883 For our tests and to follow the various examples presented both by IBM and
884 D-Wave, it is also useful to be able to run a Jupyter notebook. We can install
885 Jupyter with:

```
886 $ pip install jupyter
```

887 5.2 IBM QISKIT

888 To program a gate-based architecture and to access IBM quantum computers
889 we use the *Qiskit* software stack. The name Qiskit is a general term referring
to a collection of software for executing programs on quantum computers.

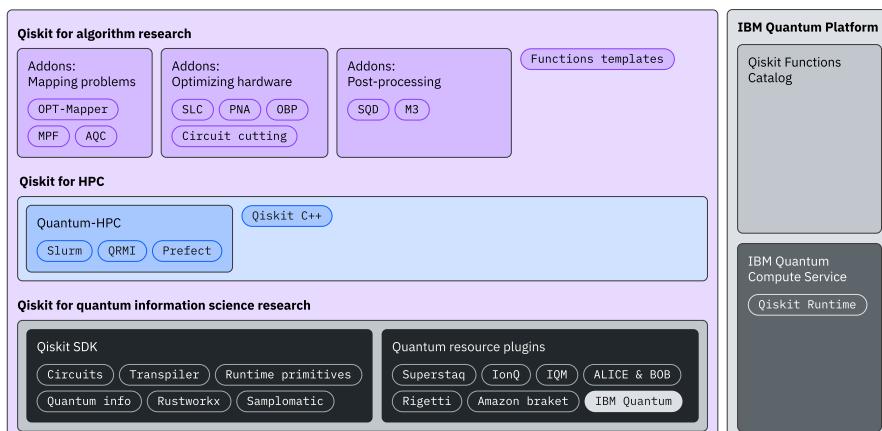


Figure 5.1: Qiskit software stack

890
 891 The core components are *Qiskit SDK* and *Qiskit Runtime*. The first one is
 892 completely open source and allows the developer to define his circuit; the
 893 second one is a cloud-based service for executing quantum computations on
 894 IBM quantum computers.

895 **5.2.1 Hello World**

896 Following the IBM documentation¹ we can install the SDK and the Runtime
 897 with:

```
$ pip install qiskit matplotlib qiskit[visualization]
$ pip install qiskit-ibm-runtime
$ pip install qiskit-aer
```

899 Last command installs Aer, which is a high-performance simulator for
 900 quantum circuits written in Qiskit. Aer includes realistic noise models, and
 901 we will use it later to test our circuit.

902 Sometimes the Qiskit stack suffers from incompatibilities between the
 903 various software components that compose the environment. At the mo-
 904 ment of writing, the latest packages seem to work without any problem.
 905 For our tests we will use `qiskit: 2.2.3`, `qiskit-ibm-runtime: 0.43.1` and
 906 `qiskit-aer: 0.17.2`.

907 If the setup is successful we are now able to run a small test to build a Bell
 908 state (two entangled qubits). The following code assembles the gates, shows
 909 the final circuit and uses a sampler to simulate on the CPU the result of 1024
 910 runs of the program.

```
1  from qiskit import QuantumCircuit
2  from qiskit.primitives import StatevectorSampler
3
4  qc = QuantumCircuit(2)
5  qc.h(0)
6  qc.cx(0, 1)
7  qc.measure_all()
8
9  sampler = StatevectorSampler()
10 result = sampler.run([qc], shots=1024).result()
11 print(result[0].data.meas.get_counts())
12 qc.draw("mpl")
```

Listing 5.1: Building Bell state

911 **5.2.2 Transpilation**

912 Each Quantum Processing Unit (QPU) has a specific topology. We need to
 913 rewrite our quantum circuit in order to match the topology of the selected
 914 device on which we want to run our program. This phase of rewriting,
 915 followed by an optimization, is called transpilation.

916 Considering, for now, a fake hardware (so we do not need an API key)
 917 we can transpile the quantum circuit `qc`, from the code above, to match the

¹ <https://quantum.cloud.ibm.com/docs/en/guides/install-qiskit>

918 topology of a specific QPU. Listing 5.2 implement the transpilation, and
 919 Figure 5.2 shows the circuits generated: (a) is the circuit we have defined
 920 in Listing 5.1 and (b) is the transpiled version where the Hadamard gate is
 921 replaced to match the actual topology of the QPU.

```

1  from qiskit_ibm_runtime.fake_provider import FakeWashingtonV2
2  from qiskit.transpiler import generate_preset_pass_manager
3
4  backend = FakeWashingtonV2()
5  pass_manager = generate_preset_pass_manager(backend=backend)
6
7  transpiled = pass_manager.run(qc)
8  transpiled.draw("mpl")

```

Listing 5.2: Transpilation

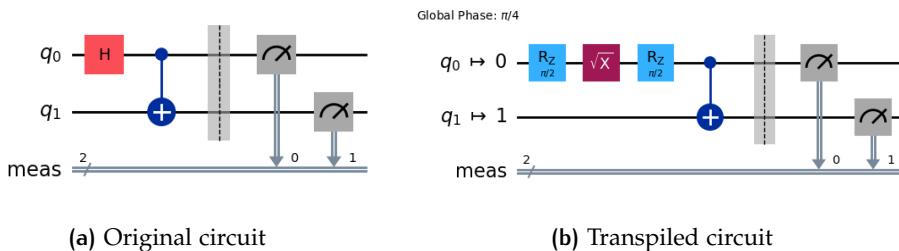


Figure 5.2: Transpilation example

922 5.2.3 Execution

923 To test our transpiled circuit we use Aer, which allows us to simulate also
 924 the noise of real quantum hardware. Listing 5.3 shows how to simulate the
 execution.

```

1  from qiskit_aer.primitives import SamplerV2
2
3  sampler = SamplerV2.from_backend(backend)
4  job = sampler.run([transpiled], shots=1024)
5  result = job.result()
6  print(f"counts Bell circuit: {result[0].data.meas.get_counts()}")

```

Listing 5.3: Simulated execution

925
 926 If we look at the results of the execution we can observe that some answers
 927 present non-entangled qubits; this is caused by the (simulated) noise of the
 928 quantum device. A typical output of the execution could be:

```
929 > counts Bell circuit: {'00': 504, '11': 503, '01': 10, '10': 7}
```

930 Where states 01 and 10 should not be present in an ideal execution with
 931 no errors.

932 5.3 D-WAVE OCEAN

933 To define an optimization problem that can be solved on a D-Wave quantum
 934 computer we use the Ocean software stack. Ocean also allows us to interact
 935 with D-Wave hardware, submit a problem, and simulate the execution on a
 classical CPU.

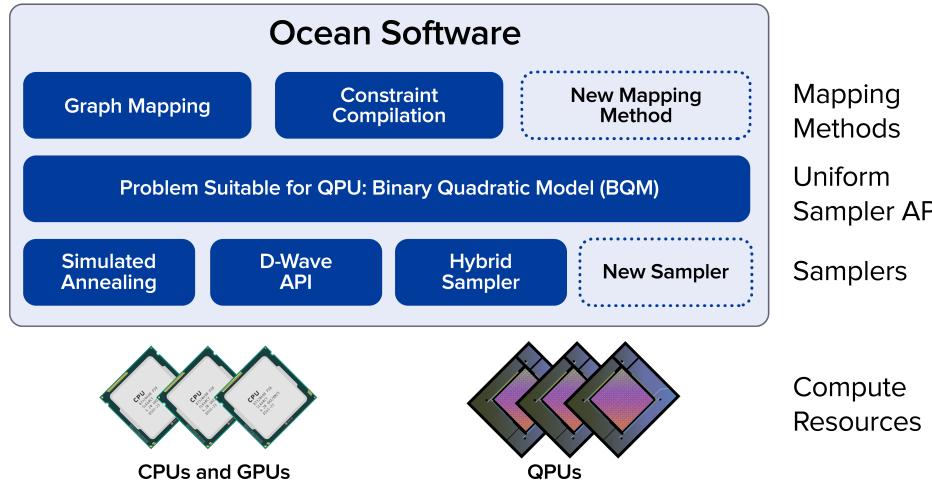


Figure 5.3: Ocean software stack

936
 937 All tools that implement the steps needed to solve your problem on a CPU,
 938 a D-Wave quantum computer, or a quantum-classical hybrid solver can be
 939 installed with:

```
940 $ pip install dwave-ocean-sdk
```

941 After the installation, running the command `dwave setup` will start an in-
 942 teractive prompt that guides us through a full setup. During the setup it
 943 is also possible to add an API token or connect to the D-Wave account to
 944 import a key directly to use the quantum hardware.

945 5.3.1 Hello World

946 To present a simple optimization program we consider the minimum vertex
 947 cover (MVC) problem. Given a graph $G = (V, E)$, the problem asks to find
 948 a subset $V' \subseteq V$ such that, for each edge $\{u, v\} \in E$, at least one of u or v
 949 belongs to V' , and the number of nodes in V' ($|V'|$) is the lowest possible.

950 The reduction from MVC to an Ising formulation is well known. The cost
 951 function that we want to minimize can be expressed by:

$$\text{cost} = \sum_{i=1}^{|V|} v_i + 2 \cdot \sum_{\{i,j\} \in E} (1 - v_i - v_j + v_i v_j)$$

952 where $v_i \in \{-1, 1\}$ and $v_i = 1$ means that $v_i \in V'$, otherwise $v_i = -1$.

953 Like all problems in Ising form we can express the cost as a symmetric
 954 matrix, so our function becomes

$$\text{cost} = v^T \times \mathbf{M} \times v$$

955 where v is the vector containing the binary variables v_i .

956 The figure shows an example graph (5.4a) and the corresponding matrix
957 (5.4b) expressing the cost function.

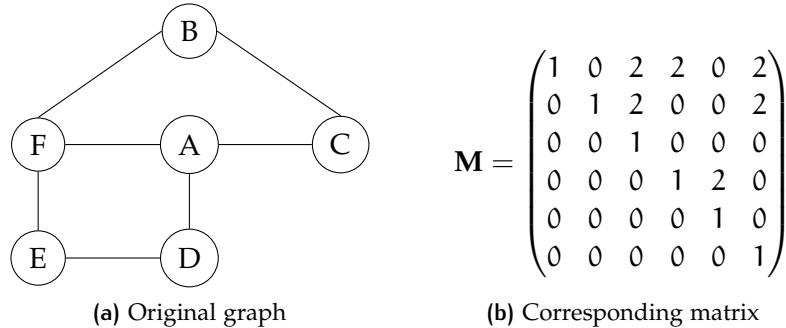


Figure 5.4: Ising formulation

958 The following code presents a possible implementation of the Ising model
959 described above. We have defined two dictionaries to store the matrix coeffi-
960 cients. The last line of code finds ten possible answers to the problem using
961 the simulated annealing function implemented by D-Wave.

```
1 from dwave.samplers import SimulatedAnnealingSampler
2 linear = {'A': 1, 'B': 1, 'C': 1, 'D': 1, 'E': 1, 'F': 1}
3 quadratic = {('B', 'C'): 2, ('B', 'F'): 2, ('C', 'A'): 2, ('D',
4   ↪ 'A'): 2, ('E', 'D'): 2, ('E', 'F'): 2, ('F', 'A'): 2}
5 sampler = SimulatedAnnealingSampler()
6 result = sampler.sample_ising(linear, quadratic, num_reads=10)
```

Listing 5.4: Ising example

962 If we print the results with `print(result.aggregate())` we can observe
963 something similar to this:

A	B	C	D	E	F	energy	num_oc.	
0	-1	-1	+1	+1	-1	+1	-14.0	6
1	+1	+1	-1	-1	+1	-1	-14.0	4
['SPIN', 2 rows, 10 samples, 6 variables]								

964
965 The two different results represent the two correct answers to our particular
966 instance of the MVC problem.

967 5.4 PYQUBO AND QUBOVERT

968 In Listing 5.4 we have manually built the matrix representing the function
969 that we want to minimize. It can be useful to have some tools that allow us
970 to work at a higher level, defining cost functions like Equation ?? that we
971 defined in the section about quantum annealing (Section ??).

972 Considering again the MVC problem, the objective function tends to min-
973 imize the number of nodes in our subset, while the penalty increases the
974 cost if we leave out some edges. This interpretation allows us to transform
975 the Ising model into the more familiar —from the point of view of a com-
976 puter scientist— QUBO model, where all variables $x_i \in \{0, 1\}$. Let's see how
977 PyQUBO and qubovert help us in this task.

978 5.4.1 PyQUBO

979 Reading from the documentation on the PyQUBO site², PyQUBO allows
 980 us to create QUBOs or Ising models from flexible mathematical expressions
 981 easily. Some of the features of PyQUBO are:

- 982 • Python based (C++ backend);
- 983 • Fully integrated with Ocean SDK;
- 984 • Automatic validation of constraints;
- 985 • Placeholder for parameter tuning.

986 We can install PyQUBO with `$ pip install pyqubo` and rewrite our MVC
 problem by defining the Hamiltonian that we want to minimize.

```

1  from pyqubo import Binary, Placeholder, Constraint
2  from dwave.samplers import SimulatedAnnealingSampler
3
4  A, B, C, D, E, F = Binary('A'), Binary('B'), Binary('C'),
5    ↳ Binary('D'), Binary('E'), Binary('F')
6
7  H_objective = (A + B + C + D + E + F)
8  H_penalty = Constraint(((1 - A - C + A*C) +\
9    (1 - A - D + A*D) +\
10   (1 - A - F + A*F) +\
11   (1 - B - C + B*C) +\
12   (1 - B - F + B*F) +\
13   (1 - D - E + D*E) +\
14   (1 - E - F + E*F)) ,label='cnstr0')
15
16  L = Placeholder('L')
17  H = H_objective + L*H_penalty
18  H_internal = H.compile()
19  bqm = H_internal.to_bqm(feed_dict={'L': 2})
20
21  sampler = SimulatedAnnealingSampler()
22  result = sampler.sample(bqm, num_reads=10)
```

Listing 5.5: Rewriting MVC with pyQUBO

987
 988 Listing 5.5 presents a possible re-implementation of Listing 5.4, where we
 989 also see how PyQUBO interfaces with the Ocean SDK (line 17), and how to
 990 create (lines 14–16) and instantiate (line 17) a parametric Hamiltonian.

991 5.4.2 qubovert

992 As written in the documentation³, qubovert is the one-stop package for for-
 993 mulating, simulating, and solving problems in boolean and spin form. Using
 994 our nomenclature, boolean and spin form are respectively QUBO and Ising
 995 form.

² <https://pyqubo.readthedocs.io/en/latest/>

³ <https://qubovert.readthedocs.io/en/latest/index.html>

996 Qubovert allows us to define various types of optimization problems that
 997 can be solved by brute force, with qubovert's simulated annealing, or with
 998 D-Wave's solver. Models defined in qubovert are:

999 **QUBO**: Quadratic Unconstrained Boolean Optimization;
 1000 **QUSO**: Quadratic Unconstrained Spin Optimization (Ising model);
 1001 **PUBO**: Polynomial Unconstrained Boolean Optimization;
 1002 **PUSO**: Polynomial Unconstrained Spin Optimization;
 1003 **PCBO**: Polynomial Constrained Boolean Optimization;
 1004 **PCSO**: Polynomial Constrained Spin Optimization.

1005 In addition to generic models, qubovert has a library of famous NP-complete
 1006 problems mapped to QUBO and Ising forms.

```

1  from qubovert import boolean_var
2  from dwave.samplers import SimulatedAnnealingSampler
3
4  A, B, C, D, E, F = boolean_var('A'), boolean_var('B'),
5    ↵ boolean_var('C'), boolean_var('D'), boolean_var('E'),
6    ↵ boolean_var('F')
7
8  model = A + B + C + D + E + F
9  model.add_constraint_OR(A, C, lam=2)
10 model.add_constraint_OR(A, D, lam=2)
11 model.add_constraint_OR(A, F, lam=2)
12 model.add_constraint_OR(B, C, lam=2)
13 model.add_constraint_OR(B, F, lam=2)
14 model.add_constraint_OR(D, E, lam=2)
15 model.add_constraint_OR(E, F, lam=2)
16
17 qubo = model.to_qubo()
18 dwave_qubo = qubo.Q
19 sampler = SimulatedAnnealingSampler()
  result = sampler.sample_qubo(dwave_qubo, num_reads=10)
```

Listing 5.6: Rewriting MVC with qubovert

1007 Listing 5.6 shows a possible implementation of the MVC problem using
 1008 the tools provided by qubovert. Qubovert allows us to express our problem
 1009 as a PCBO; we use this formulation to express constraints in a more natural
 1010 way. In our example we ensure that each edge is covered simply by enforcing
 1011 that at least one of the nodes linked by the edge is present in the solution.
 1012 This constraint is repeated for each edge in the graph (lines 7–13). To specify
 1013 the Lagrange multiplier (Equation ??) we use the keyword `lam`.

1014 Qubovert, like PyQUBO, can interface with the Ocean SDK, transforming
 1015 a PCBO problem into a QUBO problem (line 15) and then rewriting it in the
 1016 format accepted by the D-Wave solver (or sampler).

1017 5.5 CONCLUSION

1018 In this chapter we have set up an environment to run our future experiments
1019 and tests. We have also shown some small examples to present the main
1020 characteristics and test the tools we will use in our work.

1021 Following this setup allows anyone to recreate exactly the same configura-
1022 tion we use, avoiding (for what we know and test) incompatibilities between
1023 Python packages.

1024 QA-Prolog is a tool that allows one to write a program in a logic programming language and execute it on a quantum annealer. QA-Prolog also retrieves the results returned by the quantum annealer and presents them in a natural and comprehensible way.

1029 In this chapter we introduce the project and give some pointers to other
1030 related works. We describe extensively the pipeline of transformations start-
1031 ing from a Prolog code and ending with a Hamiltonian H_f , as we have
1032 described in Section ???. Next we discuss the changes we have made to the
1033 original QA-Prolog code to restore compatibility with the modern frame-
1034 work used to interface with the D-Wave quantum annealer and to support
1035 the latest version of the library used in the project.

1036 The chapter ends with an installation guide that ensures a working and
1037 reproducible environment to run experiments, both for this chapter and for
1038 the following ones.

1039 6.1 THE PROJECT

1040 QA-Prolog is a project developed by Scott Pakin¹ in 2017 – 2019. It starts
1041 from the question: “Can one express constraint logic programming in the
1042 form accepted by quantum-annealing hardware?” [14].

1043 The hope is that even if today we live in the NISQ² era of quantum comput-
1044 ing, quantum annealers are more easily scalable than quantum gate-based
1045 computers [15], and QA-Prolog could improve Prolog program execution by
1046 replacing backtracking with fully parallel annealing into a solution state [16].

1047 6.1.1 Reason

1048 As we have shown in Chapters ?? and ??, programming a quantum computer
1049 is not an easy task. We express our algorithm in a very low-level way.

1050 On a quantum annealer we have to define a cost function, without con-
1051 straints (which must be transformed into a penalty function). Even if there
1052 are libraries that allow us to express these functions in an easier way, we
1053 need at least to find a QUBO representation of our problem.

1054 Even worse is the situation on quantum gate-based computers. The pro-
1055 grammer has to build a quantum circuit gate by gate, an approach similar to
1056 what is done with FPGAs (Field Programmable Gate Arrays) [17]. We can
1057 indeed see a strong analogy:

¹ Los Alamos National Laboratory: pakin@lanl.gov.

² Noisy intermediate-scale quantum computing.

- 1058 • programmable logic blocks which implement logic functions → quantum gates;
- 1060 • programmable routing that connects these logic functions → the possibility to define the order of gates;
- 1062 • I/O blocks connected to logic → input *qubits* and output *qubits* that we can measure.

1064 FPGAs are components that the majority of computer scientists are not used
 1065 to and are probably out of the interest of a programmer. In the same way,
 1066 the hardness of programming a quantum computer could be a significant
 1067 deterrent to attracting new researchers in the field.

1068 Today are available some “high-level gates” that wrap multiple low-level
 1069 gates into useful patterns. There also exist, both for quantum gate-based
 1070 computers and quantum annealers, some templates of well-known problems
 1071 that need only fine-tuning to be useful for a specific problem.

1072 Despite these sorts of abstractions, programming a quantum computer is
 1073 difficult and very close to machine language.

1074 The goal of QA-Prolog is to have a tool that allows programming in a high-
 1075 level style, with a powerful *logic programming language*, quantum computers.

1076 6.1.2 Prolog

1077 We can see QA-Prolog as a compiler from Prolog to H_f , where the ground
 1078 state of H_f is the solution of our Prolog program. Before starting with the
 1079 compilation process, it is useful to understand the main characteristics of
 1080 Prolog, because it is not an imperative programming language like C or
 1081 Java, but a *declarative* one. For more information about Prolog, the lectures
 1082 of [18] and [19] are recommended.

1083 In Prolog we do not specify step by step an algorithm that resolves our
 1084 problem, we describe instead the formal relationship between the objects in
 1085 our problem and which relations have to be true in our solution [18].

1086 Programming in Prolog consists of:

- 1087 • listing *facts* about objects and relationships between objects;
- 1088 • specifying *rules* to derive new facts from the ones already asserted;
- 1089 • asking questions (*queries*) about objects and their relationships.

1090 From these characteristics we can understand what “declarative” means:
 1091 the program is a list of statements about our problem (our domain of interest);
 1092 the relation between a Prolog program and an ontology is very strict.
 1093 In Prolog we encode a KB made of facts and rules; in Chapter ?? we can
 1094 see a complete example of rewriting from an OWL ontology into a Prolog
 1095 KB. Moreover, Prolog is a logic programming language. This means that
 1096 the core of programming is not to tell the computer what to do, but to tell
 1097 it what is true and ask it to try to draw conclusions [18]. The idea behind
 1098 logic programming is very interesting; for more details [20] and [21] are
 1099 recommended.

1100 **EXAMPLE:** In Listin 6.1, adapted from [19], we present a basic Prolog program in order to show the syntax and the usage of queries.

```

1  sings(mia).
2  listens2Music(yolanda).
3  party.
4
5  dance(yolanda) :- listens2Music(yolanda).
6  happy(yolanda) :- dance(yolanda).
7  happy(mia) :- sings(mia).
8
9  smile(X) :- happy(X).
```

Listing 6.1: Basic KB

1101
1102 The first three lines are facts: we are asserting that Yolanda is listening to
1103 music, Mia is singing, and there is a party. The other lines are rules: we can
1104 identify rules by the `:`- sign that divides the *head* of the rule on the left, from
1105 the *body* on the right. The head of a rule is true if the body is true.

1106 For example, the rule at line 5 can be read as: “If Yolanda is listening
1107 to music, then Yolanda is dancing”. Line 9 shows the usage of a variable:
1108 variables start with an uppercase letter and are placeholders for information.
1109 We can read this rule as “If someone is happy, they smile”.

1110 We can query our KB, asking for example if Yolanda is happy. In SWI-
1111 Prolog [22] we can interact with the interpreter, and the query we evaluate is
1112 `?- happy(yolanda).` (the full stop is part of the syntax and tells the interpreter
1113 that the query is complete). Prolog will answer `yes .`; this is because Yolanda
1114 is listening to music, and if she is listening to music she dances, and if she
1115 dances she is happy.

1116 By analogous reasoning it should be clear why the result of `?- smile(X).`
1117 is `X = mia; X = yolanda.`, where ; means logical disjunction: *or*.

1118 6.1.3 Feature of QA-Prolog

1119 QA-Prolog does not support all the features of Prolog, but enough to make
1120 basic logic programming possible [14].

1121 QA-Prolog supports atoms and positive integers but not floating-point
1122 numbers, strings, or lists. It supports arithmetic and relational operations,
1123 and rules can reference other rules but not recursively. QA-Prolog supports
1124 unification, backtracking, and predicates comprising multiple clauses [14].

1125 QA-Prolog also supports some features not present in the basic version
1126 of Prolog. In particular, operations can be performed on variables even be-
1127 fore they are ground [14]; this means that QA-Prolog is more powerful in
1128 manipulating free variables.

1129 6.1.4 Related works

1130 There are some other attempts to develop a high-level programming lan-
1131 guage for quantum computers, both for the quantum gate model and for
1132 quantum annealers.

1133 *Quantum Prolog* [23] demonstrates that one can express the equivalent of
 1134 a pure version of Prolog over finite relations in terms of a model of discrete quantum computing. This work targets quantum gate computers and
 1135 focuses on the mathematical equivalence of relational programming and discrete quantum computing over the field of Booleans [14].

1138 *C to D-Wave* [24] reuses the pipeline we will discuss, replacing the starting
 1139 step. The paper addresses the difficulty of programming quantum annealers by presenting a compilation framework that compiles a subset of C code into
 1140 quantum machine instructions to be executed on a quantum annealer.

1142 6.2 PIPELINE

1143 We are now ready to describe the pipeline of transformations that brings us
 1144 from a Prolog program to a Hamiltonian H_f .

1145 The chain of transformations is shown in Figure 6.1, where the last step (in
 1146 orange) is the quantum annealer capable of finding the ground state of H_f .
 1147 In purple we can see the various file formats throughout the pipeline, and
 1148 in yellow the software that performs the rewriting. Some of this software is made ad hoc for the QA-Prolog pipeline; others are also used in very
 1149 different fields.

1151 From a high-level point of view, the pipeline rewrites
 1152 the initial KB expressed in
 1153 Prolog into different objects.
 1154 The logical meaning of the
 1155 entities we build during the
 1156 pipeline is:

- 1158 1. Prolog program (KB);
- 1159 2. High-level digital circuit in Verilog;
- 1160 3. Low-level digital circuit in EDIF;
- 1161 4. Symbolic Hamiltonian;
- 1162 5. Physical Hamiltonian.

1165 Let us start analyzing the
 1166 pipeline from the last step,
 1167 the one nearest to the QPU.

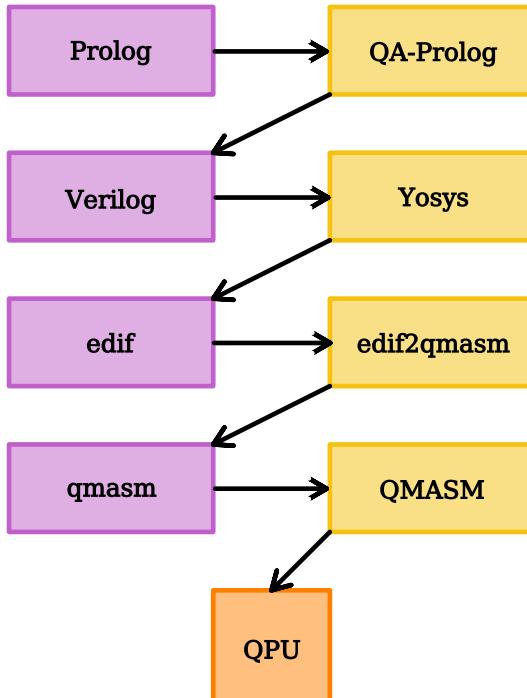


Figure 6.1: QA-Prolog pipeline

1168 6.2.1 QMASM

1169 QMASM is a *quantum macro assembler* [25]; it processes a symbolic Hamiltonian
 1170 and assembles a physical Hamiltonian that can be embedded on a D-
 1171 Wave quantum annealer. It was developed in Python by Scott Pakin with the
 1172 goal of filling a gap in tools for the creation of D-Wave programs. QMASM

1173 is an abstraction layer that allows the programmer not to care about man-
 1174 ually setting specific point weights and coupler strengths on the physical
 1175 topology.

1176 This software can be considered an assembler in the sense that it maps
 1177 a symbolic representation of the operations (assembly language) to the ma-
 1178 chine language. QMASM is not only an assembler, but extends its functional-
 1179 ity by including macros: named blocks of assembly language, parameterized,
 1180 that a program can instantiate multiple times [25].

1181 **FEATURE:** QMASM provides a number of features to simplify low-level
 1182 D-Wave programming [25]. Moreover, we can also use QMASM to assemble
 1183 a physical Hamiltonian that we can later manipulate or solve using classical
 1184 methods or other quantum systems.

1185 Some of the most useful and interesting features of QMASM are:

- 1186 • *qubits* are referenced symbolically, not numerically, both in the source
 code and when QMASM reports execution results;
- 1188 • *qubits* can be pinned to `true` or `false`;
- 1189 • *qubit* patterns can be encapsulated into macros and instantiated repeat-
 edly;
- 1191 • groups of macros can be encapsulated into libraries and reused across
 multiple programs;
- 1193 • QMASM can automatically exclude from the results the solutions known
 to be incorrect and shows only “interesting” *qubits*.

1195 Thanks to this set of features, QMASM is already a useful abstraction layer
 1196 that simplifies the development of programs that target an annealer, classical
 1197 or quantum.

1198 **EXAMPLE:** Let us consider an example that shows the potential of QMASM
 1199 and that is also useful for the following steps of the pipeline. The satisfia-
 1200 bility problem is well known to be an NP-complete problem [26], so it is a
 1201 good candidate for our example. We take into account the simple formula:

$$y = x_1 \wedge \neg(x_2 \vee x_3) \quad (6.1)$$

1202 QMASM allows us to define a macro for every logical operator needed to
 1203 represent the formula and then assemble the final formula by calling these
 1204 macros. In Listings 6.2 we call `A` and `B` the input *qubits* and `Y` the output *qubit*.
 1205 Weights are specified as an Ising problem (Section ??), which is the default
 1206 format for QMASM source files³.

1207 There are multiple interesting details about these macros:

- 1208 • the sign `#` tells QMASM that the following line is a comment and
 should not be processed;

³ as specified in <https://github.com/lanl/qmasm/wiki/File-format>.

- 1210 • the sign \$ is used to tag a *qubit* as ancillary: unless explicitly requested
1211 by the programmer, the intermediate results are not reported in the
1212 solutions;
- 1213 • compared with what we have done in Listing 5.4, defining weights is
1214 much easier and the result is more readable;
- 1215 • thanks to the directive `!assert` we can inform QMASM about con-
1216 straints on the solution. This directive does not change the weights,
1217 but allows the programmer to exclude from the solutions those that
1218 are surely incorrect.

```
# Y = A AND B
!begin_macro AND
  !assert $Y=$A&$B
  $A -0.5
  $B -0.5
  $Y 1
  $A $B 0.5
  $A $Y -1
  $B $Y -1
!end_macro AND
```

(a) *and* gate

```
# Y = NOT A
!begin_macro NOT
  !assert $Y!=!$A
  $A $Y 1.0
!end_macro NOT
```

(b) *not* gate

```
# Y = A OR B
!begin_macro OR
  !assert $Y=$A|!$B
  $A 0.5
  $B 0.5
  $Y -1
  $A $B 0.5
  $A $Y -1
  $B $Y -1
!end_macro OR
```

(c) *or* gate

Listing 6.2: Logical operators

1219 It is possible to verify, for each macro in Listings 6.2, that given a configura-
1220 tion of the input *qubits*, the value of *Y* that minimizes the energy corresponds
1221 to the output of the logic gate we are modeling.

1222 We can save these macros in a file named `gates.qasm` and use it to solve
1223 our problem. To compute the formula $y = x_1 \wedge \neg(x_2 \vee x_3)$ we will use
1224 some intermediate results: we start with $x_4 = (x_2 \vee x_3)$, then we apply the
1225 negation $x_5 = \neg x_4$, and lastly we compute the result as $y = x_1 \wedge x_5$. The
1226 QMASM code implementing this procedure is reported in Listing 6.3.

1227 This example shows how to use macros: we instantiate a macro and give
1228 it a name with `!use_macro <macro_name> <instance_name>` (e.g. line 5), and
1229 then instantiate the *qubits* defined in the macro with our actual *qubits*. For
1230 example, considering the `!use_macro OR` at line 5, we can see that we use x_2
1231 and x_3 as input and an ancillary *qubit* $\$x_4$ as output. Another detail to point
1232 out is the directive `!include` (line 3), which imports all gates used in this
1233 source file.

1234 Finally, we can query the quantum annealer (or in this case a classical
1235 solver) to find a solution for our satisfiability problem. To do so we pin the
1236 output variable *y* to be sure that in the solution its value will be `true`. We
1237 can query the solver with:

```
1238 qasm --run --pin="y := true" --solver="sim_anneal" circ_sat.qasm
```

1239 where `circ_sat.qasm` is the file name of our source code.

```

1 # Solve circuit-satisfiability problem.
2
3 !include <gates>
4
5 !use_macro OR x2_or_x3
6   x2_or_x3.$A = x2
7   x2_or_x3.$B = x3
8   x2_or_x3.$Y = $x4
9
10 !use_macro NOT not_x4
11   not_x4.$A = $x4
12   not_x4.$Y = $x5
13
14 !use_macro AND x1_and_x5
15   x1_and_x5.$A = x1
16   x1_and_x5.$B = $x5
17   x1_and_x5.$Y = y

```

Listing 6.3: Circuit satisfiability

1240 Results are reported in Listing 6.4. Here we can see that the solver has cor-
 1241 rectly found the solution, but only 642 times out of the default 1000 samples.
 1242 This is caused by the stochastic nature of annealing, simulated or quantum.
 1243 QMASM automatically removed the solutions that do not respect the !assert
 1244 directive or do not have the minimum energy.

```

# x1 --> 12
# x2 --> 3
# x3 --> 4
# y --> [True]
Solution #1 (energy = -20.0000, tally = 642):

Variable  Value
-----  -----
x1       True
x2       False
x3      False
y        True

```

Listing 6.4: Circuit satisfiability results

1245 QMASM already offers some powerful abstractions to work with quantum
 1246 annealers. Now we add two additional layers that allow a programming
 1247 style more similar to the paradigms computer scientists are used to, while
 1248 preserving strong control over variable dimensions and therefore over the
 1249 number of *qubits* used.

1250 6.2.2 Yosys and edif2qasm

1251 These steps of the pipeline take as input a *Verilog HDL* (Hardware Descrip-
 1252 tion Language) program and transform it into a symbolic Hamiltonian.

1253 We aggregate two steps because Yosys is not a tool developed for this
 1254 pipeline and is used mostly to optimize the intermediate results of the trans-

1255 formations. Also, Yosys and edif2qasm work on the same logical entity: a
 1256 digital circuit that, in these steps of the pipeline, is converted into a symbolic
 1257 Hamiltonian.

1258 Verilog is a hardware description language that offers different levels of
 1259 design abstraction. The highest level is *behavioural* and uses programming
 1260 constructs such as assignments, conditionals, and while-loops. The lowest
 1261 level is a connection of gates (*netlists*) [27].

1262 Hardware description languages are not something that the majority of
 1263 programmers are familiar with, but they offer some advantages when targeting
 1264 a quantum annealer [28]:

- 1265 • they provide precise control over bit widths in order not to waste any
 1266 *qubit*;
- 1267 • they can be compiled to a small set of primitives: the gates we can
 1268 define with QMASM.

1269 In these steps of the pipeline we start using behavioural constructs, then
 1270 we automatically generate a netlist as the output of *synthesisers*.

1271 The synthesiser used is Yosys [29], a free and open-source software for
 1272 Verilog HDL synthesis.

1273 The two most useful features of Yosys are the possibility of specifying a
 1274 *cell library*, the set of gates used to synthesize the netlist, and the use of the
 1275 external tool *Berkeley ABC* [30] (incorporated in Yosys), providing additional
 1276 code optimizations.

1277 Yosys lowers a Verilog program to an EDIF netlist that uses only a specified
 1278 set of gates; then edif2qasm lowers the netlist into a symbolic Hamiltonian.
 1279 This Hamiltonian uses a set of macros defining the energy for each
 1280 type of gate Yosys can use⁴. What we end up with is a symbolic Hamiltonian
 1281 that can be used as input for QMASM.

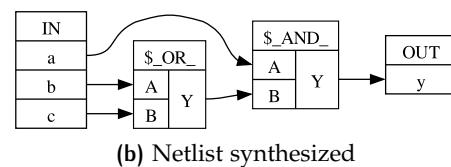
1282 **EXAMPLE:** Considering again the simple formula $y = x_1 \wedge \neg(x_2 \vee x_3)$
 1283 (Equation 6.1).

```

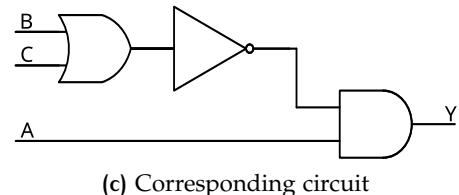
1  module sat (A, B, C, Y);
2    input A, B, C;
3    output Y;
4
5    assign Y = A & ~ (B | C);
6  endmodule

```

(a) Verilog Code



(b) Netlist synthesized



(c) Corresponding circuit

Figure 6.2: Yosys processing

4 The actual set of gates extends the logical ports *and*, *or*, and *not* and is available at: <https://github.com/lanl/edif2qasm/blob/master/stdcell.qasm>.

1284 Figure 6.2 shows a possible Verilog implementation of the formula (a), the
 1285 netlist Yosys has produced from the Verilog code (b), and, only for clarifi-
 1286 cation purposes, the corresponding digital circuit implemented with logic
 1287 gates (c).

1288 Now we can feed the obtained netlist to edif2qasm to obtain the code
 reported in Listing 6.5.

```

1  !include <stdcell>
2
3  !begin_macro sat
4      !use_macro AND $id00006
5      !use_macro NOT $id00004
6      !use_macro OR $id00005
7      $id00004.A = $id00005.Y # $or$sat.v:5$1_Y
8      $id00005.A = b
9      $id00005.B = c
10     $id00006.A = a
11     $id00006.B = $id00004.Y # $not$sat.v:5$2_Y
12     $id00006.Y = y
13  !end_macro sat
14
15  !use_macro sat sat

```

Listing 6.5: Symbolic Hamiltonian

1289 Comparing the automatically generated Listing 6.5 and the manually writ-
 1290 ten 6.3, we can observe some differences:

- in Listing 6.5 we define the new macro `sat`: each module in the Prolog program is converted into a macro;
- the generated code is a little less readable but more compact. We do not need to generate new ancillary variables, but instead use directly as input for macros the output of other macros.

1297 Even if it is less readable, we can comment on the code in Listing 6.5. At
 1298 lines 4, 5, and 6 we declare that we need the macros `AND`, `NOT`, and `OR`, giving
 1299 to each macro a name like `$id<number>`⁵. At line 7 we assign as input `A` of
 1300 the macro `NOT` the output `Y` of the macro `OR`. Given these hints, the rest of the
 1301 code should be clear.

1302 6.2.3 QA-Prolog

1303 This is the last step of the pipeline, the one that provides the maximum level
 1304 of abstraction from the hardware that finds the solution to our problem.

1305 QA-Prolog, like edif2qasm, is written in GO [14, 28] and compiles a
 1306 Prolog program to a Verilog one; each fact or rule is converted into a module.
 1307 Once QA-Prolog has compiled the Prolog source code, including the query
 1308 that can be specified on the command line, into Verilog, QA-Prolog invokes
 1309 Yosys, edif2qasm, and QMASM, as illustrated in Figure 6.1 [14].

1310 QMASM executes the user's program on a D-Wave system or on a simu-
 1311 lator and reports the value of each symbol appearing in the QMASM source

⁵ The symbol \$ tells QMASM that we are not interested in the value of *qubits* inside the macro.

1312 file. QA-Prolog maps these lists of Booleans back to integers and named
 1313 atoms, associates those values with variables named in the user's query, and
 1314 reports all variables and their values to the user just as a typical Prolog en-
 1315 vironment would [14].

1316 **EXAMPLES:** The first example we present shows a small KB that can be
 1317 summarized as "the enemy of my enemy is my friend".

```

1  hates(alice, bob).
2  hates(bob, charlie).

3
4  enemy(X, Y) :- hates(X, Y).
5  enemy(X, Y) :- hates(Y, X).

6
7  friend(X, Y) :-
8    enemy(X, Z),
9    enemy(Z, Y).
```

Listing 6.6: KB enemy of my enemy is my friend

1318 The code is reported in Listing 6.6. Thanks to the declarative nature of
 1319 Prolog we can immediately understand the meaning of our KB: we describe
 1320 relations between three people, assert that if a person hates another one they
 1321 are enemies (lines 4 and 5), and that if two people share an enemy they are
 1322 friends (lines 7, 8, and 9).

1323 We can now query the KB to find if there are two people who are friends
 1324 with:

```
QA-Prolog --qmasm-args="--solver=tabu" --query='friends(P1, P2).'
→ --work-dir="~/work" friends.pl
```

1325 In this command we can see that QA-Prolog allows us to query the KB
 1326 with the exact same syntax we would use to interact with a Prolog interpreter.
 1327 As expected, QA-Prolog *binds* the variables P1 and P2 to Alice and Charlie.

1328 Another detail to point out is the parameter `--work-dir=<path>`, which
 1329 specifies the folder in which QA-Prolog outputs all files and the solver: we
 1330 use a *tabu search algorithm* because the simulated annealing implemented in
 1331 Ocean is not capable of finding a solution even in this small scenario (more
 1332 considerations about that in Chapter ??).

1333 If we inspect the working directory we can see the output of every step
 1334 of the pipeline; in particular, to see how exactly the KB is rewritten into a
 1335 Verilog program, in Appendix .1 there is the result of compilation, and it is
 1336 possible to observe in detail how each fact or rule has been transformed.

1337 The second example concludes our transformation of the satisfiability
 1338 problem (Equation 6.1); unfortunately, this is not very didactic because the
 1339 Prolog program suitable for QA-Prolog processing introduces a consistent
 1340 amount of overhead.

1341 Indeed, we have to implement logical operators by defining their truth
 1342 tables, then assemble the operators in our logical formula. Listing 6.7 shows
 1343 the implementation we use in our test.

1344 We can now run QA-Prolog with the query `sat(A, B, C, true)` to find
 1345 the correct assignment of Boolean variables that satisfy our logic formula.

```

1 and(false, false, false).
2 and(false, true, false).
3 and(true, false, false).
4 and(true, true, true).

5
6 not(false, true).
7 not(true, false).

8
9 or(false, false, false).
10 or(false, true, true).
11 or(true, false, true).
12 or(true, true, true).

13
14 sat(A, B, C, Y) :-
15     or(B, C, X),
16     not(X, Z),
17     and(A, Z, Y).

```

Listing 6.7: Satisfiability in Prolog

1346 Again, we are forced to use the tabu search algorithm. If we now explore
 1347 the working directory of QA-Prolog we can see that the symbolic matrix
 1348 generated by the pipeline is a lot more complex than the one reported in
 1349 Listing 6.5, we can interpret that as the cost of abstraction.

1350 6.2.4 Overview

1351 We have described a pipeline of rewritings that, layer after layer, makes it
 1352 possible to abstract away from the hardware that actually solves our prob-
 1353 lem, moving toward a language, and thus a way of reasoning, that is at a
 1354 higher level. We speak of rewritings because the object we are manipulating
 1355 remains logically unchanged, just as it is essential that the solutions we are
 1356 searching for remain unchanged; the problem and its solutions are simply
 1357 expressed in different languages as one moves down the pipeline.

1358 The advantages of abstraction are essentially the same as those obtained
 1359 when moving from assembly language to a high-level language: greater ease
 1360 of expression, improved readability, and the possibility of working with al-
 1361 ready familiar paradigms. The drawbacks are the overheads that naturally
 1362 arise during the translation process. QA-Prolog is still a prototype, and if to-
 1363 day compilers produce better machine code than a programmer could write
 1364 manually, it is because significant effort and many years of research have
 1365 been invested to achieve excellent results. The hope is that QA-Prolog and
 1366 related works represent a first step in the same direction within the field of
 1367 quantum computing.

1368 6.3 UPDATE TO THE PROJECT

1369 In this section we briefly describe the updates we have made to the project in
 1370 order to restore compatibility with the Ocean framework and with the other

1371 libraries used. We also provide a small guide to install the tools to ensure
 1372 that all packages work properly and our experiments are reproducible.

1373 **6.3.1 Restoring QMASM**

1374 The last commit to QA-Prolog was made in 2019, and the last one to QMASM
 1375 in 2021. Since then, packages and libraries have changed and the compati-
 1376 bility with the QA-Prolog pipeline has broken.

1377 The most fragile component of the pipeline is QMASM, because it is the
 1378 one that interacts with external frameworks that are likely to change. In par-
 1379 ticular, the development of Ocean is very active, and some functions used in
 1380 QMASM have now been removed, deprecated, or moved to other packages.

1381 The incompatibilities encountered between imports used in QMASM and
 1382 external libraries are:

- 1383 • in the `dwave.cloud` and `hibrid` libraries;
- 1384 • in the libraries defining D-Wave samplers (classical solvers);
- 1385 • in the `scipy.stats` library.

1386 The documentation available for Ocean⁶ and for `scipy`⁷ helped us resolve
 1387 these incompatibilities: most of the time the solution consisted simply in
 1388 correcting the name of the library.

1389 Another small bug, caused by a different name of a solver in the command
 1390 line helper and in the actual code, was fixed by restoring the correspondence.

1391 **6.3.2 Fixing interaction edif2qmasm-QMASM**

1392 During the execution of the complete pipeline we encountered an error
 caused by undefined macros in the QMASM source file.

<pre> 1 // Define hates(atom, atom). 2 module \hates/2 (A, B, Valid); 3 ... 4 endmodule </pre>	<pre> 1 # hates/2 2 !begin_macro id00011 3 ... 4 !end_macro id00011 </pre>
(a) Verilog Code	(b) Macro definitions
<pre> 1 # enemies/2 2 !begin_macro id00010 3 !use_macro hates/2 \$id00032 # hates_PWYwG/2 4 ... 5 !end_macro id00010 </pre>	
(c) Macro usage	

Listing 6.8: Undefined macros error

1393 The error can be seen in Listings 6.8: `edif2qmasm` rewrites the netlist
 1394 generated from (a) into the macro (b); here `hates/2` is just a comment, the

⁶ Available at <https://docs.dwavequantum.com/en/latest/index.html>.

⁷ Available at <https://docs.scipy.org/doc/scipy/index.html>.

1396 actual name of the macro is `id00011` (specified at line 2). In (c), however,
 1397 we can see that the macro previously defined is called symbolically and not
 1398 with its actual name (lines 3). This obviously produces an error: the name
 1399 `hates/2` in the QMASM file has no meaning, it is only a comment.

```

1  with open(file, 'r') as input:
2      first = input.readline()
3      second = input.readline()
4      while(second_row != ""):
5          if first.startswith("#") and second.startswith("!begin_macro"):
6              self.name[first[2:-1]] = second[len("!begin_macro "):-1]
7              first_row = second_row
8              second_row = input.readline()
9
10     with open(file, 'r') as input:
11         doc = input.read()
12         lines = doc.splitlines()
13         for line in lines:
14             for word in line.split():
15                 if word in self.name.keys():
16                     line = line.replace(word, self.name[word])
17                     self.new_lines.append(line)

```

Listing 6.9: Preprocessor's core

1400 To solve the problem we added a preprocessing step before parsing with
 1401 QMASM. During the preprocessing all symbolic names are substituted with
 1402 the actual name of the corresponding macro.

1403 The core code of the preprocessor is reported in Listing 6.9. The program
 1404 scans the source file two times: during the first reading a Python dictio-
 1405 nary *symbolic_name-actual_name* is built, then, during the second reading, all
 1406 instances of the symbolic name are replaced with the actual name.

1407 Now all components should work properly, and we can install all the
 1408 software needed and run some experiments.

1409 6.3.3 Installation guide

1410 In order to have a working environment where we can run our experiments,
 1411 we need to install all the software required by the QA-Prolog pipeline. In
 1412 Chapter 5 we have set up a Python environment with all the essential tools;
 1413 to start the installation of the QA-Prolog pipeline we need at least the Ocean
 1414 SDK installed as described in Section 5.3.

1415 The first component we need is QMASM: we can install the updated and
 1416 fixed version from <https://github.com/DavideCamino/qmasm.git>. The in-
 1417 stallation procedure consists of the following three commands:

```

$ git clone https://github.com/DavideCamino/qmasm.git
$ cd qmasm
$ python setup.py install

```

1419 Next we need Yosys and GO: Yosys is part of the pipeline, and GO allows
 1420 us to compile edif2qmasm and QA-Prolog. Both Yosys and GO are available

1421 as binary packages for the majority of GNU/Linux distributions; on Arch⁸
 1422 they can be installed with:

```
1423      # pacman -S yosys
          # pacman -S go
```

1424 The `go install` command installs Go executables in the default directory
 1425 `$HOME/go/bin`. It is useful to add the `bin` subdirectory to `PATH`. This can be
 1426 done by editing the `.bashrc` file (or the corresponding one for the specific
 1427 shell), adding:

```
1428      PATH=$PATH:$HOME/go/bin
          export PATH
```

1429 Finally we can install `edif2qasm` and QA-Prolog with:

```
1430      $ go install github.com/lanl/edif2qasm@latest
          $ go install github.com/lanl/QA-Prolog@latest
```

1431 6.4 CONCLUSION

1432 In this chapter, we presented QA-Prolog, a rewriting pipeline that enables
 1433 the transformation of a Verilog program into a Hamiltonian whose ground
 1434 state represents the solution of the original program. We discussed the vari-
 1435 ous steps of the pipeline in detail, also showing how some parts were mod-
 1436 ified to make them compatible again with current frameworks. Finally, we
 1437 described how to install a working version of the pipeline.

1438 From our discussion, it emerges that QA-Prolog is a modular software
 1439 stack in which it is possible to replace a component of the pipeline with
 1440 another to modify the result; for example, we presented work that starts
 1441 from C instead of Prolog.

1442 This work has the potential to provide a foundation for many other exper-
 1443 iments, that can rely on an already implemented and functioning infrastruc-
 1444 ture, in order to develop new extensions, both upstream and downstream,
 1445 of the pipeline.

8 <https://archlinux.org/>

1446

Part III

1447

EXPERIMENTS

1448 7 | A QUANTUM ONTOLOGY

1449 7.1 ONTOLOGY STRUCTURE

1450 7.2 PROLOG VERSION

1451 7.3 INFERENCE ON THE ONTOLOGY

1452 7.4 CONCLUSION

1453 **8** | QAOA

1454 **8.1 QAOA**

1455 **8.2 FROM QUBO TO PAULI OPERATOR**

1456 **8.3 EXPERIMENTS**

1457 **8.4 CONCLUSION**

9 | CONCLUSION

1459 BIBLIOGRAPHY

- 1460 [1] Barry Smith. "Ontology". In: (2012).
- 1461 [2] Gian Piero Zarri. "Ontologies and Their Practical Implementation". In:
1462 *Encyclopedia of Database Technologies and Applications*. IGI Global Scien-
1463 tific Publishing, 2005, pp. 438–449.
- 1464 [3] Thomas R Gruber. "A translation approach to portable ontology spec-
1465 ifications". In: *Knowledge acquisition* 5.2 (1993), pp. 199–220.
- 1466 [4] Marco Fossati, Emilio Dorigatti, and Claudio Giuliano. "N-ary relation
1467 extraction for simultaneous T-Box and A-Box knowledge base augmen-
1468 tation". In: *Semantic Web* 9.4 (2018), pp. 413–439.
- 1469 [5] Giuseppe De Giacomo, Maurizio Lenzerini, et al. "TBox and ABox
1470 reasoning in expressive description logics." In: *KR* 96.316–327 (1996),
1471 p. 10.
- 1472 [6] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. "A descrip-
1473 tion logic primer". In: *arXiv preprint arXiv:1201.4089* (2012).
- 1474 [7] Pascal Hitzler et al. *OWL 2 Web Ontology Language Primer (Second Edi-
1475 tion)*. W3C Recommendation REC-owl2-primer-20121211. World Wide
1476 Web Consortium (W3C), Dec. 2012. URL: <https://www.w3.org/TR/2012/REC-owl2-primer-20121211/>.
- 1477 [8] Pascal Hitzler. "A review of the semantic web field". In: *Communica-
1478 tions of the ACM* 64.2 (2021), pp. 76–83.
- 1480 [9] Stefano Borgo et al. "DOLCE: A descriptive ontology for linguistic and
1481 cognitive engineering". In: *Applied ontology* 17.1 (2022), pp. 45–69.
- 1482 [10] Boris Motik, Peter F. Patel-Schneider, and Bernardo Cuenca Grau. *OWL
1483 2 Web Ontology Language: Direct Semantics (Second Edition)*. W3C Rec-
1484 ommendation REC-owl2-direct-semantics-20121211. World Wide Web
1485 Consortium (W3C), Dec. 2012. URL: <https://www.w3.org/TR/2012/REC-owl2-direct-semantics-20121211/>.
- 1487 [11] Ilianna Kollia, Birte Glimm, and Ian Horrocks. "Query answering over
1488 SROIQ knowledge bases with SPARQL". In: *Proceedings of the 24th Inter-
1489 national Workshop on Description Logics, Barcelona, Spain*. 2011, pp. 13–
1490 16.
- 1491 [12] Franz Baader. *The description logic handbook: Theory, implementation and
1492 applications*. Cambridge university press, 2003.
- 1493 [13] Birte Glimm et al. "HermiT: an OWL 2 reasoner". In: *Journal of auto-
1494 mated reasoning* 53.3 (2014), pp. 245–269. URL: <http://www.hermi-trs.com/>.
- 1496 [14] Scott Pakin. "Performing fully parallel constraint logic programming
1497 on a quantum annealer". In: *Theory and Practice of Logic Programming*
1498 18.5-6 (2018), pp. 928–949.

- [1499] [15] William M Kaminsky, Seth Lloyd, and Terry P Orlando. "Scalable superconducting architecture for adiabatic quantum computation". In: *arXiv preprint quant-ph/0403090* (2004).
- [1500]
- [1501]
- [1502] [16] Los Alamos National Laboratory / Scott Pakin. *QA-Prolog: Quantum Annealing Prolog*. Accessed: 2026-02-13, GitHub repository. URL: <https://github.com/lanl/QA-Prolog>.
- [1503]
- [1504]
- [1505] [17] Umer Farooq, Zied Marrakchi, and Habib Mehrez. "FPGA architectures: An overview". In: *Tree-Based Heterogeneous FPGA Architectures: Application Specific Exploration and Optimization* (2012), pp. 7–48.
- [1506]
- [1507]
- [1508] [18] William F Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003.
- [1509]
- [1510] [19] Patrick Blackburn, Johan Bos, and Kristina Striegnitz. *Learn Prolog Now! – Free Online Version*. Accessed: 2026-02-13. 2012. URL: <https://lpn.swi-prolog.org/lpnpage.php?pageid=online>.
- [1511]
- [1512]
- [1513] [20] Robert Kowalski and Steve Smoliar. "Logic for problem solving". In: *ACM SIGSOFT Software Engineering Notes* 7.2 (1982), pp. 61–62.
- [1514]
- [1515] [21] Christopher John Hogger. *Introduction to logic programming*. Academic Press Professional, Inc., 1984.
- [1516]
- [1517] [22] Jan Wielemaker et al. "SWI-Prolog". In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96. ISSN: 1471-0684.
- [1518]
- [1519] [23] Roshan P James, Gerardo Ortiz, and Amr Sabry. "Quantum computing over finite fields". In: *arXiv preprint arXiv:1101.3764* (2011).
- [1520]
- [1521] [24] Mohamed W Hassan, Scott Pakin, and Wu-chun Feng. "C to D-wave: a high-level C compilation framework for quantum Annealers". In: *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2019, pp. 1–8.
- [1522]
- [1523]
- [1524]
- [1525] [25] Scott Pakin. "A quantum macro assembler". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2016, pp. 1–8.
- [1526]
- [1527] [26] Michael R Garey and David S Johnson. *Computers and intractability*. Vol. 29. wh freeman New York, 2002.
- [1528]
- [1529] [27] Mike Gordon. "The semantic challenge of Verilog HDL". In: *Proceedings of tenth annual IEEE symposium on logic in computer science*. IEEE. 1995, pp. 136–145.
- [1530]
- [1531]
- [1532] [28] Scott Pakin. "Targeting classical code to a quantum annealer". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 529–543.
- [1533]
- [1534]
- [1535]
- [1536] [29] Clifford Wolf, Johann Glaser, and Johannes Kepler. "Yosys-a free Verilog synthesis suite". In: *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. Vol. 97. 2013.
- [1537]
- [1538]
- [1539] [30] Robert Brayton and Alan Mishchenko. "ABC: An academic industrial-strength verification tool". In: *International Conference on Computer Aided Verification*. Springer. 2010, pp. 24–40.
- [1540]
- [1541]

```

1 // Verilog version of Prolog program friends.pl
2 // Conversion by QA-Prolog, written by Scott Pakin <pakin@lanl.gov>
3 //
4 // This program is intended to be passed to edif2qasm, then to qasm,
5 // and
6 // finally run on a quantum annealer.
7 //
8 // Note: This program uses 3 bit(s) for atoms and 1 bit(s) for
9 // (unsigned)
10 // integers.
11
12 // Define all of the symbols used in this program.
13 `define alice 3'd0
14 `define bob   3'd1
15 `define charlie 3'd2
16 `define enemies 3'd3
17 `define friends 3'd4
18 `define hates   3'd5
19
20 // Define hates(atom, atom).
21 module \hates/2 (A, B, Valid);
22     input [2:0] A;
23     input [2:0] B;
24     output Valid;
25     wire [1:0] $v1;
26     assign $v1[0] = A == `alice;
27     assign $v1[1] = B == `bob;
28     wire [1:0] $v2;
29     assign $v2[0] = A == `bob;
30     assign $v2[1] = B == `charlie;
31     assign Valid = &$v1 | &$v2;
32 endmodule
33
34 // Define enemies(atom, atom).
35 module \enemies/2 (A, B, Valid);
36     input [2:0] A;
37     input [2:0] B;
38     output Valid;
39     wire $v1;
40     \hates/2 \hates_pujkx/2 (A, B, $v1);
41     wire $v2;
42     \hates/2 \hates_DoUbH/2 (B, A, $v2);
43     assign Valid = &$v1 | &$v2;
44 endmodule
45
46 // Define friends(atom, atom).
47 module \friends/2 (A, B, Valid);
48     input [2:0] A;
49     input [2:0] B;

```

```
48  output Valid;
49  (* keep *) wire [2:0] C;
50  wire [2:0] $v1;
51  \enemies/2 \enemies_GcbiL/2 (A, C, $v1[0]);
52  \enemies/2 \enemies_GNLnA/2 (C, B, $v1[1]);
53  assign $v1[2] = A != B;
54  assign Valid = &$v1;
55 endmodule
56
57 // Define Query(atom, atom).
58 module Query (P1, P2, Valid);
59   input [2:0] P1;
60   input [2:0] P2;
61   output Valid;
62   wire $v1;
63   \friends/2 \friends_KMGjP/2 (P1, P2, $v1);
64   assign Valid = &$v1;
65 endmodule
```

Listing .1: QA-Prolog compilation result