A functional programming language for quantum computation with classical control

By
Benoît Valiron
University of Ottawa
September 2004

A Thesis
submitted to the School of Graduate Studies and Research
in partial fulfillment of the requirements
for the degree of
Master of Science in Mathematics[1]

[1]The M.Sc. Program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

Abstract

The objective of this thesis is to develop a functional programming language for quantum computers based on the *QRAM* model, following the work of P. Selinger (2004) on quantum flow-charts. We construct a lambda-calculus without side-effects to deal with quantum bits. We equip this calculus with a probabilistic call-by-value operational semantics. Since quantum information cannot be duplicated due to the *no-cloning property*, we need a resource-sensitive type system. We develop it based on affine intuitionistic linear logic. Unlike the quantum lambda-calculus proposed by Van Tonder (2003, 2004), the resulting lambda-calculus has only one lambda-abstraction, linear and non-linear abstractions being encoded in the type system. We also integrate classical and quantum data types within our language. The main results of this work are the subject-reduction of the language and the construction of a type inference algorithm.

Acknowledgments

This research was supported by graduate scholarships from the faculty of graduate studies and from the department of mathematics.

The results were presented at the FMCS 2004 conference in Calgary. A paper is scheduled to appear in the proceedings of QPL 2004.

A number of people gave me the opportunity to write this thesis. I hope the one I forget will forgive me. I express my thanks to the following people:

To my supervisor Dr. Selinger for his supervision and his advice.

To my committee, Drs. Blute and Howe.

To Dr. Scott who gave me the opportunity to do a master in Ottawa.

To the administrative staff from the department and from the international office, especially Line Bissonette.

To Bob, who helps me to check some of the spelling of this thesis.

To Eric, for having supported me these two years, and to all my Preston- Sweetland's roommates for pushing me to work each morning.

Contents

List of Tables

Chapter 1

Introduction

13 And God said unto Noah, The end of all flesh is come before me; for the earth is filled with violence through them; and, behold, I will destroy them with the earth. 14 Make thee an ark of gopher wood; rooms shalt thou make in the ark, and shalt pitch it within and without with pitch. 15 And this is the fashion which thou shalt make it of: The length of the ark shall be three hundred cubits, the breadth of it fifty cubits, and the height of it thirty cubits.
[6 Gen 13-15, King James version]

Background on quantum computation. Quantum computing has become a fast growing research area in recent years, since Shor [22] has shown in 1994 that quantum computers can factor an integer in polynomial time upon its number of digits. It is not known whether any classical algorithm can solve the problem in polynomial time. The factoring problem has numerous implications in cryptography. In particular the most commonly used algorithm to encode data with a public key is the RSA algorithm, based on the present difficulty to factorize very large numbers [15, p.232]. Quantum computers would stir up the field of cryptography. This discovery has focused attention on quantum computing, which is able to bring change in other domains, such as database manipulation [15, p.248], with algorithms to query elements in databases, and such as numerical methods, with the ability to perform efficiently Fourier transform [15, p.216].

The basic idea behind quantum computation is to encode data using objects governed by the laws of quantum physics. In a classical computer, the smallest unit of data is the *bit*. On the other hand, the smaller unit of data in a quantum computer is a *quantum bit*, or *qubit*. The laws of quantum physics give the constraints applying on qubits. Bits and qubits behave in a complete different manner. For instance, a classical bit can be copied as many times as needed. On the other hand, a quantum bit cannot be duplicated, due to the well-known *no cloning property* of quantum states [17, 15]. However, quantum data types are computationally very powerful, due to the phenomena of quantum superposition and entanglement. A qubit can be modeled as a normalized vector in a two-dimensional Hilbert space. To understand it as a piece of information, one has to choose an orthonormal basis, which we denote as $(|0\rangle, |1\rangle)$. The qubit is then written as $\alpha|0\rangle + \beta|1\rangle$, with $|\alpha|^2 + |\beta|^2 = 1$, and one can understand it as the superposition of the bit 0 and the bit 1. A state of two qubits is a vector of the tensor product of the two Hilbert spaces. There are states of the form $|\phi_1\rangle \otimes |\phi_2\rangle$, but one can also write a state of the form $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Such a state is called an *entangled* state. The operations that one can perform on a quantum state are only of two classes, namely *unitary transformations* and *measurements*. The measurement of a qubit acts as a projection on one of the basis elements. For a good general introduction to quantum computing, see e.g. [17, 15].

Background on functional programming. A functional programming language is a language where programs are seen as functions: a program is usually a piece of code that take arguments and return a value. In a higher-order functional programming language, every function is regarded as a value. In that sense one can speak of a program returning another program. This is a powerful way of understanding programming. A model of this kind of computation is the lambda-calculus, designed in the 1930's by Church [6] and Kleene [12]. It provides an operational semantics for describing computable functions and evaluation. A complete reference on the subject is [2].

The problem. At the moment, computation using quantum computers is mostly understood as a physical process. Very few programming languages exist for dealing with this kind of computer. Trying to understand the process of quantum computation from the point of view of programming languages can help to the discovery of new applications and to have a better understanding of the semantics of such a computation.

Review. Recall that a quantum system can evolve by unitary transformations and measurements. Many existing models of quantum computation put an emphasis on the former, i.e., computation is understood as the evolution of a quantum state by means of unitary gates. In these models, a quantum computer is considered as a purely quantum system, i.e., without any classical parts: Measurements are done at the end of the experiment, often outside of the formal system. One example of such a model is the quantum Turing machine [3, 8], where the entire machine state, including the tape, the finite control, and the position of the head, is assumed to be in quantum superposition. Another example is the quantum lambda calculus of Van Tonder [26, 27], which is a higher-order, purely quantum language without an explicit measurement operation.

One might also imagine a perhaps more realistic model of a quantum computer where unitary operations and measurements can be interleaved. As an example, consider the so-called quantum random access machine model, or *QRAM model* of Knill [13], also described by Bettelli, Calarco and Serafini [5]. Here, a quantum computer consists of a classical computer with a quantum device attached to it. In this configuration, the operation of the machine is controlled by a classical program which emits a sequence of instructions to the quantum device for performing measurements and unitary operations. This situation is summarized by the slogan "quantum data, classical control" [21]. Several programming languages have been proposed to deal with such a model [5, 19], but the one on which this paper is based is the work of Selinger [21].

Van Tonder has built an operational semantics based on linear logic, a resource-sensitive logic formalized by Girard [9]. The idea to build an operational semantics for

linear logic has already been explored [1, 28, 4]. As a matter of fact Van Tonder [26] uses the lambda-calculus described by Wadler [28]. In this calculus, the distinction between linear and non-linear functions is explicit in the terms. Benton [4] has built a different model verifying subject reduction. These two languages, however, do not allow variables to be discarded: constant functions cannot be built. To allow variable to be discarded, one needs a variant of linear logic, the affine linear logic. An interesting work on affine linear logic is the work of Propylov [18], who showed the decidability of affine linear logic. A linear decoration of intuitionistic proofs as we intend to do was done in 1995 in [7].

The solution proposed. This thesis addresses the issue of building up a higher-order functional quantum programming language for quantum computation *with classical control*. In our language, a program is a lambda term, possibly with some quantum data embedded inside. The basic idea is that lambda terms encode the control structure of a program, and thus, they would be implemented classically, i.e., on the classical device of the *QRAM* machine. However, some of the data on which the lambda terms act are possibly qubits, and are stored on the QRAM quantum device. Because our language combines classical and quantum features, it is natural to consider two distinct basic data types: a type of *classical bits* and a type of *quantum bits*. Higher types, such as integers or lists, can be added as necessary.

The challenge has several aspects. One part is that we want the probabilistic reduction to be the only side effect. Due to the measurement operation, the reduction rules are then probabilistic, and one problem we solve is to describe the behavior of the program with respect to this probabilistic reduction. A second part of the challenge is that due to entanglement, quantum bits cannot be directly encoded in the lambda-term. We need a way to encode the qubits of the *QRAM* in the lambda-term. Another part of the challenge is that there are two kind of functions: linear and on-linear ones. In particular, a lambda-term can be duplicable or non-duplicable. Depending on this ability, it can or cannot be applied to a non-linear function, which use its argument more than once. Unlike Van Tonder's lambda-calculus, we want to let the compiler decide whether or not an argument can be applied to a function or not. Finally, we

want to be able to discard variables. This requires an affine type system. Neither the one of van Tonder [26] nor the one of Benton [4], which is linear, can be used. One may ask whether it is possible to fulfill these requirements.

We give a positive answer to this challenge. We build an expressive programming language which embeds quantum operations as functions. Using the well-known technique of *type system* [16], we are able to decide of the validity of a program in our language. The type system is based on affine linear logic. The work of Propylov [18] shows that the problem of the typability of a term is decidable, but, since our proposed type system is only a fragment of the full affine linear logic, we find a simpler algorithm. We use a similar method to [7].

Plan. The plan of the thesis is the following. Chapters 2 to 4 are background, and Chapters 5 to 9 are original work. In Chapter 2, we describe more in depth the basics of quantum computing, and review what is already done. Then, for self-containedness, in Chapter 3 we develop some results on intuitionistic typed lambda-calculus, expose the subject reduction, and develop a type inference algorithm for this language. In Chapter 4 we develop an introduction on linear logic, and explain how this is linked to our model. The next chapters expose the results found during this master thesis: In Chapter 5 a discussion on the lambda-calculus and the reduction rules, where we show how the validity of a program is linked to the choice of reduction procedure. In Chapter 6 we give a type system for the developed language, and in Chapter 7 we prove that it verifies subject reduction. Finally in Chapter 8 and 9, we extend the language and we build a type inference algorithm, based on the intuitionistic skeleton of the type system.

Chapter 2

Quantum programming

In classical computation, we use classical physics to encode the data. The basic unit of data is the *bit* which can take only two values, either 0 or 1. In quantum computation, we use objects governed by quantum physics laws instead of classical physics, in order to encode data. The unit of data is called the *quantum bit*, or *qubit*. A quantum bit can be understood as a normalized vector in a two-dimensional Hilbert space. To understand it as a piece of information, it is customary to choose an arbitrary orthonormal basis denoted $(|0\rangle, |1\rangle)$, and called the *computational basis*. A qubit is then a vector of the form

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \qquad |\alpha|^2 + |\beta|^2 = 1$$

where α and β are complex numbers.

There are many possible physical realization of a qubit. It can be encoded in the polarization of a photon, see Figure 1. If we choose two orthogonal directions as basis, one can encode superposition by setting the plane of polarization with some chosen angle of cosine $|\alpha|^2$ and sine $|\beta|^2$.

The juxtaposition of two qubits in states $|\psi_1\rangle$ and $|\psi_2\rangle$ is represented by the tensor $|\psi_1\rangle \otimes |\psi_2\rangle$, also denoted $|\psi_1\psi_2\rangle$, which is an element of the 4-dimensional Hilbert space with basis $|00\rangle, |01\rangle, |10\rangle, |11\rangle$. More generally, if $\ulcorner n \urcorner_N$ is the binary representation of

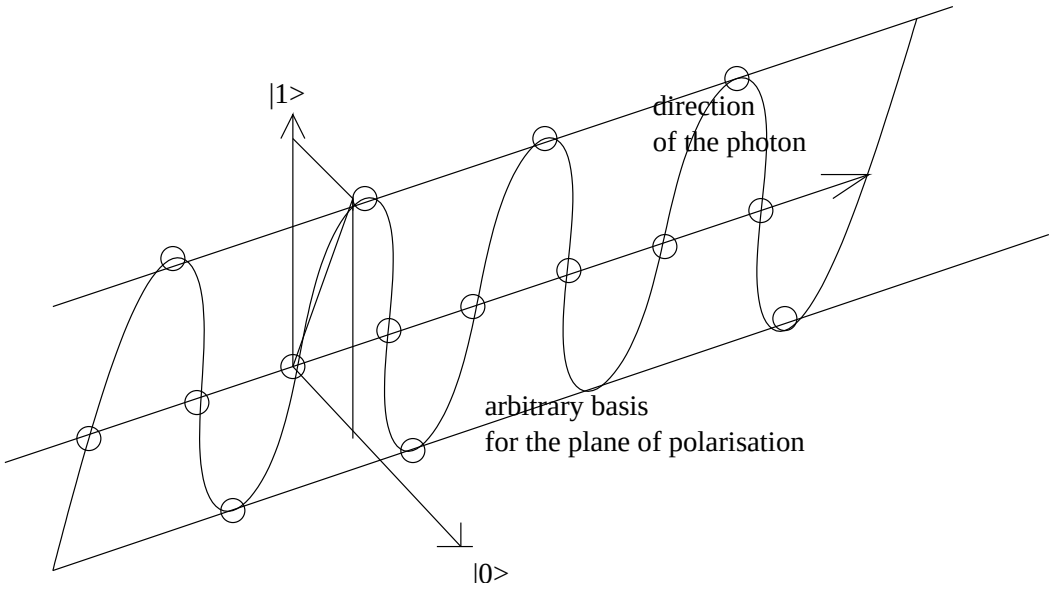


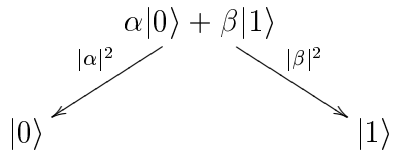
Figure 1: Photon polarization

n with N digits, a vector of N qubits can be expressed as a sum:

$$\sum_{i=0}^{2^N-1} \alpha_i |\ulcorner i \urcorner^N\rangle, \qquad \text{with} \qquad \sum_{i=0}^{2^N-1} |\alpha_i|^2 = 1.$$

2.1 Measurements

To retrieve the information stored in a qubit, one has to *measure* the object that encodes the qubit. The measurement operation is a map that will project the vector $\alpha|0\rangle + \beta|1\rangle$ onto $|0\rangle$ or $|1\rangle$. The measurement yields an observable result which is 0 if the vector was projected onto $|0\rangle$, or 1 if it was projected onto $|1\rangle$. The process can be summarized as follows:



$\alpha|0\rangle + \beta|1\rangle$ projects onto $|0\rangle$ with probability $|\alpha|^2$ and onto $|1\rangle$ with probability $|\beta|^2$. For example, in the case of the photon, measurement is done using a polarized glass

and a light detector. This process is *probabilistic* and *collapses* the superposition of data stored in the qubit. When measuring several qubits, the result is similar. In a two-qubit system $\alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$, assume that we want to measure the first qubit. To understand what will happen, we can factorize the system as follows:

$$|0\rangle \otimes (\alpha_{00}|0\rangle + \alpha_{01}|1\rangle) + |1\rangle \otimes (\alpha_{10}|0\rangle + \alpha_{11}|1\rangle)$$

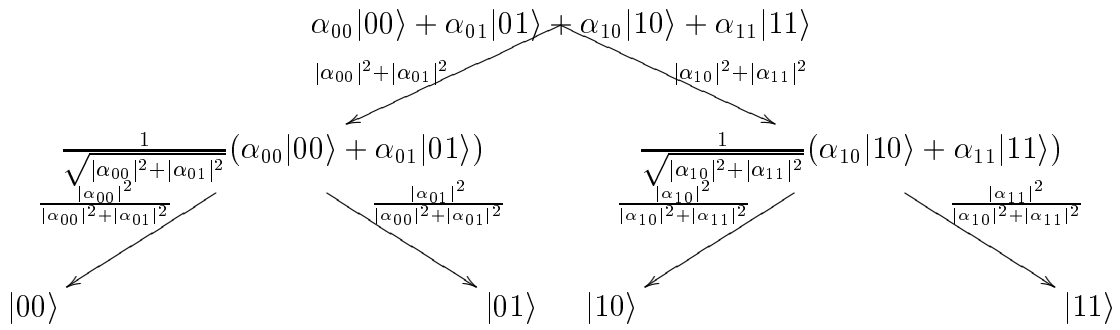
The measurement will collapse the state on one of the two subspaces of basis $|00\rangle, |01\rangle$ and $|10\rangle, |11\rangle$. It will outcome with probability $|\alpha_{00}|^2 + |\alpha_{01}|^2$ the state

$$\frac{1}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}|0\rangle \otimes (\alpha_{00}|0\rangle + \alpha_{01}|1\rangle)$$

and with probability $|\alpha_{10}|^2 + |\alpha_{11}|^2$ the state

$$\frac{1}{\sqrt{|\alpha_{10}|^2 + |\alpha_{11}|^2}}|1\rangle \otimes (\alpha_{10}|0\rangle + \alpha_{11}|1\rangle)$$

Measuring the second qubit is similar, and we can summarize the process with the diagram



and then each sequence of $|xy\rangle$ is reached with probability $|\alpha_{xy}|^2$.

More generally, in the quantum system

$$\sum_{i=0}^{2^N-1} \alpha_i |\ulcorner i \urcorner^N\rangle, \qquad \text{with} \qquad \sum_{i=0}^{2^N-1} |\alpha_i|^2 = 1.$$

the probability to get $|\ulcorner i \urcorner^N\rangle$ when measuring the system is $|\alpha_i|^2$.

2.2 Unitary operations and quantum circuits

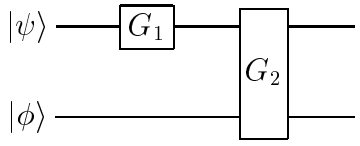
The other kind of operations we can apply on qubits are *unitary matrices*, or *quantum gates*. A unitary matrix A is such that $A^H = A^{-1}$ where A^H is the complex transpose of A: $A^H = \overline{A^t}$

Some important gates are the 3-qubit gate *Toffoli*, the 2-qubit gate *CNOT* and the 1-qubit gates $V_{\pi/8}$, the Hadamard gate H and the phase flip P. They are defined as follows, with bases always written in the lexicographic order. For a 2-qubit system for example, the basis is $(|00\rangle, |01\rangle, |10\rangle, |11\rangle)$.

$$H = \frac{\sqrt{2}}{2}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad P = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} \quad V_{\frac{\pi}{8}} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$$

$$NOT = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad Toffoli = \left(\begin{array}{c|c} id_4 & 0 \\ \hline 0 & CNOT \end{array}\right)$$

A typical quantum computation would be first the creation of an array $|000\ldots0\rangle$, then the application of some chosen quantum gates, and finally the measurement of the result. Note carrying on the same computation twice usually gives two different results: the results are *probabilistic*. Inspired by boolean circuits, a good way to represent the list of unitary gates to apply is to write a *quantum circuit*. Each qubit is represented as a wire and gates are boxes that overlap on wires. The diagram is read from left to right and from top to bottom. For example,



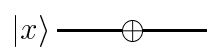
computes the state

$$G_2 \circ (G_1 \otimes id)(|\psi\rangle \otimes |\phi\rangle).$$

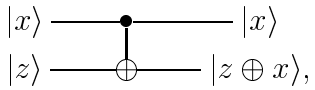
Note that if x, y and z are bits,

$$NOT|x\rangle = |1 \oplus x\rangle,$$
$$CNOT|xy\rangle = |x\rangle \otimes |z \oplus x\rangle \text{ and}$$
$$Toffoli|xyz\rangle = |x\rangle \otimes |y\rangle \otimes |z \oplus xy\rangle$$

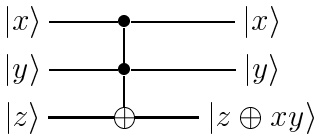
We write NOT as

$NOT =$ 

the $CNOT$ as

$CNOT =$ 

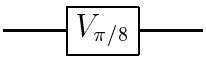
the Toffoli gate as

$Toffoli =$ 

and the other gates with boxes:

$H =$ 

$P =$ 

$V_{\pi/8} =$ 

This set of gates is said to be a *universal set of gates*, in the following sense:

Definition 2.2.1 A set B of quantum gates is said to be *universal* if, for any unitary operator U' on n qubits and any $\epsilon > 0$, there exists a finite circuit U in those gates, and some $\lambda \in \mathbb{C}$ with $|\lambda| = 1$, satisfying:

$$||U - \lambda U'|| < \epsilon \qquad \text{with the norm to be} \qquad ||A|| = \min_{|\phi|=1}(|A\phi|).$$

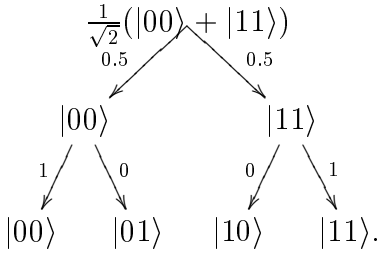
There exist a lot of universal sets of gates. It is not necessary to stick to a specific universal set since each universal set can simulate any other one. Moreover, one set of gates can be suited better for a given physical implementation than another universal set.

2.3 Entanglement

It is interesting to note that the information is *non-local* in an array of qubits. Indeed a 2-qubit system cannot always be written as $|\psi_1\rangle \otimes |\psi_2\rangle$ with ψ_1 and ψ_2 qubits. In particular the notion of a *pair* of qubits cannot be thought of as in classical computation, where each element of the pair can be reached.

Definition 2.3.1 A 2-qubit state $|\psi\rangle$ is said to be *entangled* if it cannot be written as $|\psi_1\rangle \otimes |\psi_2\rangle$. If we can write it under this form, then it is said to be *unentangled*.

For example, the state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ is entangled. One cannot separately determine the values of the first and of the second qubit. The first and the second qubits are completely linked one to the other: measuring the first will immediately allow us to say what will be the result of the measurement of the second one:



If the first qubit was measured to be $|1\rangle$, then the second one is $|1\rangle$ with probability 1. Similarly, if the first qubit was measured to be $|0\rangle$, then the second one is $|0\rangle$ with probability 1.

Entangled states are easy to construct: Consider the state

$$|\phi\rangle = CNOT \circ (H \otimes id)(|00\rangle).$$

This is equal to

$$|\phi\rangle = CNOT \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle).$$

$CNOT$ maps $|00\rangle$ to $|00\rangle$ and $|10\rangle$ to $|11\rangle$. Since it is linear, we have

$$|\phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle),$$

and we reach the previously seen entangled state.

2.4 Simulation of a classical computer on a quantum computer

One may ask whether it is possible to simulate a classical boolean circuit on a quantum computer. One could think that the fact that we can only apply unitary gates would be a constraint. Indeed, a quantum computation is always reversible. If we want to compute any arbitrary function f from $\{0,1\}^n$ to $\{0,1\}^m$, there might be a problem since the function might not be reversible. We can circumvent this problem by replacing f by a reversible function f':

$$\begin{array}{rccc} f': & \{0,1\}^n \times \{0,1\}^m & \longrightarrow & \{0,1\}^n \times \{0,1\}^m \\ & (x,y) & \longmapsto & (x, f(x) \oplus y). \end{array}$$

The reversible function f' can then be implemented on qubits as a unitary transformation. However we need auxiliary qubits. First we need some to store the unwanted information that keeps the computation reversible, and then we need more qubits for scratch space.

Theorem 2.4.1 *Any boolean function can be modeled using a quantum circuit.*

Proof. Any boolean function can be written in terms of *AND* and *NOT* gates. It is sufficient to be able to simulate a *AND* boolean gate, a *NOT* boolean gate, and to be able to duplicate a bit. We only need the *Toffoli* gate:

$$\mathit{Toffoli}(|x\rangle \otimes |y\rangle \otimes |0\rangle) = |x\rangle \otimes |y\rangle \otimes |x \ \mathit{AND} \ y\rangle,$$

and then computing $x \ AND \ y$ is equivalent to computing $\mathit{Toffoli}|xy0\rangle$ and to consider the third qubit. Similarly the *NOT* gate can be simulated as follows:

$$\mathit{Toffoli}(|x\rangle \otimes |1\rangle \otimes |1\rangle) = |x\rangle \otimes |1\rangle \otimes |1 \oplus x \times 1\rangle = |x\rangle \otimes |1\rangle \otimes |\mathit{NOT} \ x\rangle,$$

and then computing $NOT \ x$ is equivalent to computing $\mathit{Toffoli}|x11\rangle$ and to consider the third qubit. To duplicate a bit x, one can compute

$$\mathit{Toffoli}(|x\rangle \otimes |1\rangle \otimes |0\rangle) = |x\rangle \otimes |1\rangle \otimes |0 \oplus x \times 1\rangle = |x\rangle \otimes |1\rangle \otimes |x\rangle$$

and we duplicated the bit and placed it in the first and last qubit. Note that in each computation, we need scratch space. Provided that we are able to initialize a given array of qubits (i.e. to set each of them in some given state $|0\rangle$ or $|1\rangle$), we are able to compute any boolean function. □

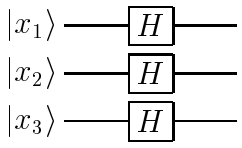
Remark 2.4.2 The last quantum circuit only duplicates bits, not qubits: if $|x\rangle$ is some superposition $\alpha|0\rangle + \beta|1\rangle$, since the previous computation is linear it answers

$$\alpha|010\rangle + \beta|111\rangle$$

which is an entangled state.

2.5 Issues specific to quantum computers

Superposition of states. Some issues are specific to quantum computers. In particular, the power of quantum computation over classical computation is in the *superposition of states*. Given an array of n qubits, one can superimpose the binary representations of all the numbers from 0 to $2^n - 1$. Since the action on this state by a unitary transformation will apply it on each one of the pure states that are superimposed in the qubit by linearity, we are able to do strong parallelism in one operation. For example, consider the following quantum computation:



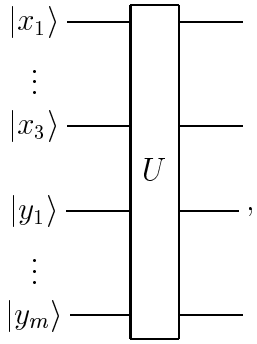
Given $|000\rangle$ it computes the state

$$\frac{1}{\sqrt{2}^3}(|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle) \otimes (|0\rangle + |1\rangle).$$

If we develop, it becomes

$$\tfrac{1}{2\sqrt{2}}(\quad |000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle)$$

which is the superposition of the binary representations of all numbers from 0 to $2^3 - 1 = 7$. Let U be a unitary operator



computing some boolean operation $f : \{0,1\}^3 \to \{0,1\}^3$, on the first five qubits for some fixed $y_1, \ldots y_m$. Then if we compose it with the first quantum circuit, by linearity it computes the function f on each $\ulcorner i \urcorner^3$ for $i = 0 \ldots 7$ in the state superposition:

$$U(\frac{1}{2\sqrt{2}} \sum_{i=0}^{7} |\ulcorner i \urcorner^3\rangle \otimes |y_1 \ldots y_m\rangle) = \frac{1}{2\sqrt{2}} \sum_{i=0}^{7} |f(\ulcorner i \urcorner^3)\rangle \otimes |y_1 \ldots y_m\rangle.$$

This is done in a simple step and would have required 8 computations of f for each $\ulcorner i \urcorner^3$ in a classical computation.

Indeed, using an algorithm based on the strong parallelism occurring during quantum computation, Shor [22, 23] proved that using a quantum computer, factorization of an integer n is of complexity $\mathcal{O}(\log n)$, far better than any pre-existing algorithm using classical methods.

Another characteristic is that qubits cannot be duplicated, due to the *no-cloning property*. Specifically, there is no operation which inputs an unknown state $|\phi\rangle$ and returns $|\phi\rangle \otimes |\phi\rangle$. Indeed, such an operation would map $\alpha|0\rangle + \beta|1\rangle$ to

$$(\alpha|0\rangle + \beta|1\rangle) \otimes (\alpha|0\rangle + \beta|1\rangle) = \alpha^2|00\rangle + \alpha\beta|01\rangle + \alpha\beta|10\rangle + \beta^2|11\rangle,$$

which is not a linear operation (much less unitary).

2.6 Practical considerations

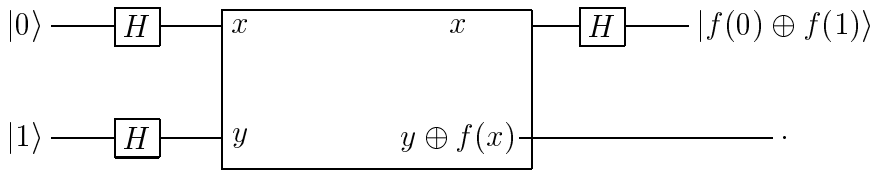
For quantum computation, a strong drawback is the *decoherence phenomenon*. A quantum particle is never alone in its world. There is always interaction with other particles, coming from the box where the particle is stored or from outer space. All these interactions act like measurements and modify the state of the particle. This limits the precision of the computation. Moreover, the decoherence process gets stronger as the number of considered qubits increases. Quantum error-correction [15] can be used to compensate this problem, provided the initial decoherence is not to severe. For the purpose of this thesis, we will ignore this issue, and assume that computations take place in a perfect quantum world.

2.7 Examples

The Deutsch Algorithm. This is an algorithm to find out whether a boolean function is balanced or constant. In classical computation, two calls to the function are needed. In quantum computation, one can find it out in only one call. The algorithm takes as input a two-qubit unitary operator U_f:

$$U_f(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus f(x)\rangle.$$

The quantum circuit for the algorithm is the following:



To find the answer, we have to measure the first qubit: if it is 0 then the function is balanced, if it is 1 it is not.

Note that the input of this algorithm is a "black-box", in other terms a function from two qubits to two qubits.

Proof that the procedure is correct. This will compute the following thing:

$$\begin{aligned}
& (H \otimes id) U_f (H \otimes H)(|0\rangle \otimes |1\rangle) \\
= \; & (H \otimes id) U_f (\tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle)) \\
= \; & (H \otimes id) U_f \tfrac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle) \\
= \; & (H \otimes id) \tfrac{1}{2}(|0\rangle \otimes |0 + f(0)\rangle + |1\rangle \otimes |0 + f(1)\rangle \\
& -|0\rangle \otimes |1 + f(0)\rangle - |1\rangle \otimes |1 + f(1)\rangle) \\
= \; & \tfrac{1}{2\sqrt{2}}((|0\rangle + |1\rangle) \otimes |f(0)\rangle + (|0\rangle - |1\rangle) \otimes |f(1)\rangle \\
& -(|0\rangle + |1\rangle) \otimes |1 + f(0)\rangle - (|0\rangle - |1\rangle) \otimes |1 + f(1)\rangle) \\
= \; & \tfrac{1}{2\sqrt{2}}(|0\rangle \otimes (|f(0)\rangle + |f(1)\rangle - |1 + f(0)\rangle - |1 + f(1)\rangle) \\
& +|1\rangle \otimes (|f(0)\rangle - |f(1)\rangle - |1 + f(0)\rangle + |1 + f(1)\rangle).
\end{aligned}$$

If $f(0) = f(1)$, then $|f(0)\rangle - |f(1)\rangle - |1 + f(0)\rangle + |1 + f(1)\rangle = 0$ and the result is

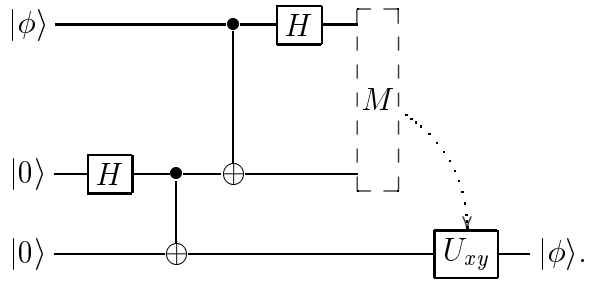
$$\frac{1}{2\sqrt{2}}|0\rangle \otimes (|f(0)\rangle + |f(1)\rangle - |1 + f(0)\rangle - |1 + f(1)\rangle).$$

If $f(0) = 1 + f(1)$, then $|f(0)\rangle + |f(1)\rangle - |1 + f(0)\rangle - |1 + f(1)\rangle = 0$ and the result is

$$\frac{1}{2\sqrt{2}}|1\rangle \otimes (|f(0)\rangle - |f(1)\rangle - |1 + f(0)\rangle + |1 + f(1)\rangle).$$

So the value of the measurement of the first qubit is 0 if the function f is balanced, and 1 in the other case. □

The teleportation algorithm. It is a good example of algorithm that is hardly written in term of quantum circuits: A measurement needs to be done as a part of the formalism. The procedure can be written as follows:



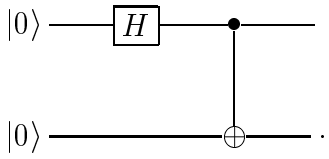
The procedure "teleports" the state of the first qubit to the third one. The dashed-box M represents the measurement of the two first qubits. The gate U_{xy} depends on

two classical bits x and y, which are the result of this measurement:

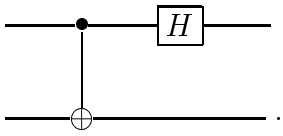
$$\begin{aligned}
&\text{If } M \text{ outputs } 00, \quad U_{00} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \\
&\text{If } M \text{ outputs } 01, \quad U_{01} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\
&\text{If } M \text{ outputs } 10, \quad U_{10} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \\
&\text{If } M \text{ outputs } 11, \quad U_{11} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}.
\end{aligned}$$

The whole procedure is summarized in four steps:

1. Create an entangled state $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ with the two last qubits using the circuit



2. Rotate the two first qubits, using the circuit



3. Then measure the resulting two qubits.

4. Finally, upon the result, apply the right transformation U to the third qubit.

Proof that the procedure is correct. The rotation processes the following computation

$$\begin{array}{cccccc}
 & CNOT & & H \otimes id & & \\
|00\rangle & \mapsto & |00\rangle & \mapsto & \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle), \\
|01\rangle & \mapsto & |01\rangle & \mapsto & \frac{1}{\sqrt{2}}(|01\rangle + |11\rangle), \\
|10\rangle & \mapsto & |11\rangle & \mapsto & \frac{1}{\sqrt{2}}(|01\rangle - |11\rangle), \\
|11\rangle & \mapsto & |10\rangle & \mapsto & \frac{1}{\sqrt{2}}(|00\rangle - |10\rangle).
\end{array}$$

If we apply it to the two first qubits of

$$\begin{aligned} & (\alpha|0\rangle + \beta|1\rangle) \otimes \tfrac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \\ = \; & \tfrac{1}{\sqrt{2}}(\alpha|000\rangle + \alpha|011\rangle + \beta|100\rangle + \beta|111\rangle) \end{aligned}$$

we get

$$\begin{aligned} & \tfrac{1}{2}(\alpha(|000\rangle + |100\rangle) + \alpha(|011\rangle + |111\rangle) + \beta(|010\rangle - |110\rangle) + \beta(|001\rangle - |101\rangle)) \\ = \; & \tfrac{1}{2}(|00\rangle \otimes (\alpha|0\rangle + \beta|1\rangle) + |01\rangle \otimes (\beta|0\rangle + \alpha|1\rangle) \\ & + |10\rangle \otimes (\alpha|0\rangle - \beta|1\rangle) + |11\rangle \otimes (\alpha|1\rangle - \beta|0\rangle)) \end{aligned}$$

If we measure the two first qubits, the third qubit becomes

$$\begin{aligned} \alpha|0\rangle + \beta|1\rangle & \quad \text{if 00 was measured,} \\ \beta|0\rangle + \alpha|1\rangle & \quad \text{if 01 was measured,} \\ \alpha|0\rangle - \beta|1\rangle & \quad \text{if 10 was measured,} \\ \alpha|1\rangle - \beta|0\rangle & \quad \text{if 11 was measured.} \end{aligned}$$

Finally note that if U_{xy} is applied in the case where x, y was measured, the the state of the last qubit is $\alpha|0\rangle + \beta|1\rangle$. □

2.8 Models for quantum computation

Models of quantum computations essentially fall into two classes. In some models, there is a quantum device whose operations is controlled by a classical computer. We refer to such models as having *classical control*. In some other models, there is no classical device. The measurement takes place at the end, it is not part of the formalism. We refer to those models as having *quantum control*.

2.8.1 Quantum control

One can consider that all the parts of the computation occur in an array of quantum bits: an algorithm may be written only in terms of quantum circuits. There is no classical computer to interact with; the whole process is modeled by quantum gates. The canonical example is the algorithm written in terms of quantum circuits.

The quantum Turing machine. A way to understand classical computation is the *universal Turing machine*. Described by Turing [25], it is an automaton together with an infinite tape divided into cells and a cursor. Each cell is either blank or contains a symbol from a finite alphabet. The tape should contain only finitely many non-blank symbols. The automaton is allowed to read or write what is under the cursor, or to move the cursor to the right or to the left upon testing the content of the cell. This very simple machine can model any computation.

Deutsch and Benioff [8, 3] have described a Quantum Turing machine, where *everything* is encoded in quantum data: the tape, the cursor, and the states of the automaton are encoded as a quantum state.

Van Tonder's lambda-calculus. This model of quantum circuit is a more abstract way for visualizing an algorithm. Van Tonder [26, 27] describes a higher order language for writing quantum algorithms. His language does not have a measurement operation, and is encoded in an array of quantum bits. It can then be implemented in a quantum Turing machine.

The terms are defined as follows:

$$\begin{array}{llll} \mathit{Term} & M, N, P & ::= & x \\ & & | & c \\ & & | & !M \\ & & | & \lambda x.M \\ & & | & \lambda !x.M \\ & & | & (MN), \end{array}$$

where c ranges over a set of constants, including 0, 1 as well as constants from unitary gates such as H, the Hadamard gate. The term $!M$ is decorated with ! to indicate that the term can be duplicated: it is said to be *non-linear*. $\lambda !x.M$ is an function that requires a non-linear term as argument.

An example of reduction could be:

$$|(\lambda x.x)(H0)\rangle \longrightarrow_\beta \frac{\sqrt{2}}{2}(|(\lambda x.x)0\rangle + |(\lambda x.x)1\rangle) \longrightarrow_\beta \frac{\sqrt{2}}{2}(|0\rangle + |1\rangle)$$

Van Tonder defines the notion of well-formed term, constructs a computational model for his language and proves that given a well-formed term M, if $\sum_i \alpha_i M_i$ is a reduction of M, then the M_i may differ only in the constants 0 and 1. Moreover:

Theorem 2.8.1 *The computational model provided by the lambda-calculus described by van Tonder is equivalent to the quantum Turing machine.* □

2.8.2 Classical control

Another way to see the quantum computation process is to imagine that a quantum computation is a combination of classical computation, measurements and unitary operations over quantum bits.

QRAM **model.** The *QRAM* model for a quantum computer was described by Knill [13]. In this model, an array of quantum bits is stored in a special device, and the device is linked to a universal classical computer, see Figure 2. The classical device acts on the quantum device by sending to it a sequence of commands to perform initializations (setting a qubit to $|0\rangle$ or $|1\rangle$), built-in unitary operations and measurements. All the classical operations are allowed, they are contained in the classical computer. One can imagine that there is a special library to talk with the quantum device, with special functions to measure, allocate and free qubits, and apply unitary transformations.

Selinger's flow-charts. A language that is based on the use of a *QRAM* model is the flow-charts language from Selinger [21]. This model uses the flow-chart notation to write programs: it is a super-set of a classical flow-chart language. A program is a graph together with a cursor that follows the wires, with data attached to it.

The graph is constructed from the rules in Table 1. Adding the notion of loops and the notion of recursion make the language powerful enough to describe the set of superoperators.

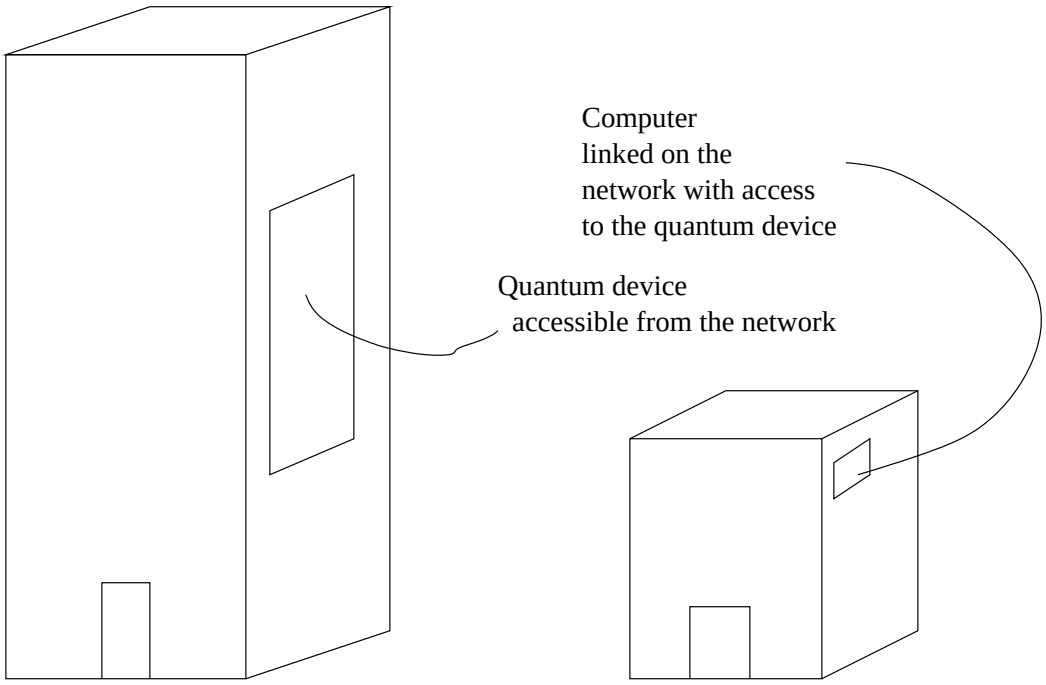


Figure 2: A model of quantum computer

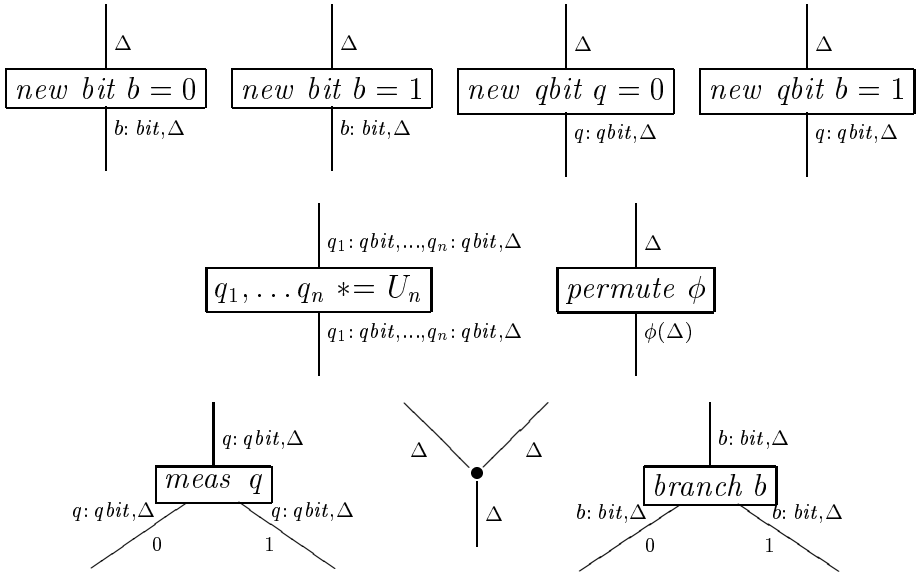


Table 1: Rules for constructing quantum flow-charts

Chapter 3

Lambda-calculus

For self-containedness, we give a brief introduction to the lambda-calculus. For a more detailed description, see e.g. [2]. We describe a lambda-calculus for writing boolean functions, and we present the definitions and results that will be used in later chapters.

3.1 Untyped lambda-calculus

The lambda-calculus is an *expression language*: a program is an expression which evaluate to a value. A lambda-expression, or *lambda-term*, evaluates similarly to $3 + 5$: The addition takes two values as arguments, and reduces to a value:

$$3 + 5 \to 8.$$

This notion of *reduction* is the basis of lambda-calculus.

Also, in lambda-calculus, we have a notation for the notion of function. We write for example

$$\lambda x.x + 3$$

in place of

$$x \mapsto x + 3.$$

We call $\lambda x.M$ an *abstraction*.

The application of 5 to the previous function is written

$$(\lambda x.x + 3)\ 5\ .$$

In general, MN represent the argument N applied to the function M.

A higher-order example is the composition operation. It can be written as

$$\mathbf{C} = \lambda f.\lambda g.\lambda z.g(fz),$$

so that $\mathbf{C}fg = f \circ g$. It takes two arguments g and f and returns $g \circ f$. Note that a function in two arguments is expressed as a function in one argument which returns another function. This notation is called *currying* [16, p. 58]

We add the notion of pair $\langle P, Q\rangle$. To be able to recover the content of a pair $\langle P, Q\rangle$, we use the term *let* $\langle x, y\rangle$=$\langle P, Q\rangle$ *in* N. It evaluates to N with P in place of x and Q is place of y. This operator is linear in x and y.

A special symbol $*$ is provided, called a *unit*. This term does not evaluate to anything.

We formally define a λ-term using an abstract syntax called the Backus-Naur form [14]. Given $\mathcal{V}_{term}$ a countable set of variables and $\mathcal{C}_{term}$ a set of constants,

$$\begin{array}{lllll} \mathit{Term} & M, N, P & ::= & x \\ & & | & c \\ & & | & \lambda x.M \\ & & | & (MN) \\ & & | & \mathit{if}\,(P; M; N) \\ & & | & * \\ & & | & \langle M, N\rangle \\ & & | & \mathit{let}\ \langle x, y\rangle{=}M\ \mathit{in}\ N \end{array}$$

where $x \in \mathcal{V}_{term}$ a set of variable and $c \in \mathcal{C}_{term}$ a set of constants. Since this calculus is for representing boolean functions, we want the constants 0 and 1 to be in $\mathcal{C}_{term}$, for representing the boolean values. The term $\mathit{if}\,(P; M; N)$ is the test operator. The term $\lambda x.M$ is a function of an argument x. It is also called *abstraction*. The term (MN) is the application of N to M. The term $\langle M, N\rangle$ is the pair of first element M and second element N. The term *let* $\langle x, y\rangle$=M *in* N is used to retrieve the content of a product. Finally, $*$ is the unit.

Conventions and notations. Given a pair M, we define two terms $\pi_1(M)$ and $\pi_2(M)$ by

$$\begin{aligned} \pi_1(M) &= \mathit{let}\ \langle x, y\rangle{=}M\ \mathit{in}\ x \text{ and} \\ \pi_2(M) &= \mathit{let}\ \langle x, y\rangle{=}M\ \mathit{in}\ y, \end{aligned}$$

to represent the first and the second projection.

We combine several variable in the same abstraction for clarity:

$$\lambda x_1 x_2 x_3.M = \lambda x_1.\lambda x_2.\lambda x_3.M.$$

The application procedure is associative to the left:

$$M_1 M_2 M_3 M_4 M_5 = (((M_1 M_2)M_3)M_4)M_5.$$

Finally, the λ-abstraction has priority over the application:

$$\lambda x.MN = \lambda x.(MN)\ .$$

Free variables. We can define a boolean *AND* operator by:

$$\lambda x.\mathit{let}\ \langle y, z\rangle{=}x\ \mathit{in}\ \ \mathit{if}(y; \mathit{if}(z; 1; 0); 0)$$

This is a function (an abstraction), with argument x, supposed to be a pair $\langle y, z\rangle$, and returning y *AND* z. We say that x, y and z are *bound* by the abstraction. More generally, a variable occurrence x in a term M is bound if there is an abstraction of variable x that contains it. A variable that is not bound by any abstraction is called a *free* variable. A term that doesn't have free variables is called *closed*. More formally, we will denote $FV(M)$ the set of the free variables of a term M, defined in Table 2.

α-equivalence. Two terms are called α-equivalent, written $M =_\alpha N$, if they differ only in the names of bound variables, e.g.

$$\lambda x.x =_\alpha \lambda y.y.$$

For details on renaming of bound variables, see [2]. From now on, we will identify α-equivalent terms and consider terms to be equal without further mention.

$$
\begin{array}{rcl}
FV(x) & = & \{x\} \\
FV(MN) & = & FV(M) \cup FV(N) \\
FV(\lambda x.M) & = & FV(M) \setminus \{x\} \\
FV(c) & = & \emptyset \\
FV(if(P;M;N)) & = & FV(P) \cup FV(M) \cup FV(N) \\
FV(*) & = & \emptyset \\
FV(\langle M_1, M_2 \rangle) & = & FV(M_1) \cup FV(M_2) \\
FV(let\ \langle x, y \rangle = M\ in\ N) & = & FV(M) \cup (FV(N) \setminus \{x, y\})
\end{array}
$$

Table 2: Definition of the set of free variables

Term substitution To evaluate the programs defined by lambda terms, we need the notion of *term substitution*. A term substitution is a function from $\mathcal{V}_{term}$ to terms such that $\sigma(x) = x$ for all but finitely many variables $x_1 \ldots x_n$. We write $\sigma = \{x_i \mapsto M_i,\ i = 1 \ldots n\}$, and we call $|\sigma|$ the set $\{x_1 \ldots x_n\}$. We extend it to $\bar{\sigma}$ function from terms to terms, defined in Table 3.

Convention. Given a set Y of variables, we write $\sigma|_Y$ the substitution defined by

$$
\sigma|_Y(x) = \begin{cases} \sigma(x) & \text{if } x \in Y, \\ x & \text{else} \end{cases}
$$

For full details on the definition of substitution, see [2].

Conventions. If $\sigma = \{x_1 \mapsto M_1, \ldots x_n \mapsto M_n\}$, we often write $M[M_1/x_1, \ldots M_n/x_n]$ in place of $\bar{\sigma}(M)$.

Fresh variable. Sometimes we need a new variable in a proof. We will call this variable a *fresh variable*. By "fresh", we mean that it has never ever occur anywhere. Whatever term, substitution or set of variables we may have talked about, the fresh variable wasn't there.

$$
\begin{array}{rcl}
\bar{\sigma}(x) & = & \sigma(x) \\
\bar{\sigma}(c) & = & c \\
\bar{\sigma}(MN) & = & \bar{\sigma}(M)\bar{\sigma}(N) \\
\bar{\sigma}(\mathit{if}(P;M;N)) & = & \mathit{if}(\bar{\sigma}(P);\bar{\sigma}(M);\bar{\sigma}(N)) \\
\bar{\sigma}(*) & = & * \\
\bar{\sigma}(\langle M_1, M_2\rangle) & = & \langle\bar{\sigma}(M_1),\bar{\sigma}(M_2)\rangle \\
\bar{\sigma}(\lambda x.M) & = & \lambda x.\overline{\sigma|_{|\sigma|\setminus\{x\}}}(M) \\
\bar{\sigma}(\mathit{let}\ \langle x,y\rangle{=}N\ \mathit{in}\ M) & = & \mathit{let}\ \langle x,y\rangle{=}\bar{\sigma}(N)\ \mathit{in}\ \overline{\sigma|_{|\sigma|\setminus\{x,y\}}}(M)
\end{array}
$$

Table 3: Definition of term-substitution

3.1.1 β-reduction

How can we run a program ?

Intuitively, to run a program, we need to reduce the number of applications that occur, by applying arguments to functions. For example, in arithmetic, to compute

$$(3+5)*7$$

one need to first reduce each side of the multiplication to an integer, then to compute the multiplication. One could write

$$(3+5)*7 \to 8*7 \to 56$$

We say that we *reduce* the term $3+5$, We need to reduce it first, since the multiplication is only defined on *values*.

Formally, we define the single-step β-reduction in Table 4

We extend this relation to $\longrightarrow_\beta{}^*$, transitive and reflexive closure of $\longrightarrow_\beta$, and to $=_\beta$ symmetric, transitive and reflexive closure of $\longrightarrow_\beta$.

The notion of redex is defined as follows:

Definition 3.1.1 A term Q is called a *redex* if it is of one of the following forms:

$$
\begin{array}{ll}
(\lambda x.M)N, & \mathit{let}\ \langle x,y\rangle{=}\langle M,N\rangle\ \mathit{in}\ P, \\
\mathit{if}(0;M;N), & \mathit{if}(1;M;N).
\end{array}
$$

$$
\begin{array}{lrcl}
(\beta) & (\lambda x.M)N & \longrightarrow_\beta & M[N/x] \\
(if_1) & if\,(1; M; N) & \longrightarrow_\beta & M \\
(if_0) & if\,(0; M; N) & \longrightarrow_\beta & N \\
(let) & let\ \langle x_1, x_2\rangle{=}\langle M_1, M_2\rangle\ in\ N & \longrightarrow_\beta & N[M_1/x_1, M_2/x_2]
\end{array}
$$

$$
\frac{M \longrightarrow_\beta M'}{MN \longrightarrow_\beta M'N}\ (cong_1) \qquad \frac{N \longrightarrow_\beta N'}{MN \longrightarrow_\beta MN'}\ (cong_2)
$$

$$
\frac{M \longrightarrow_\beta M'}{\lambda x.M \longrightarrow_\beta \lambda x.M'}\ (\xi_\lambda)
$$

$$
\frac{P \longrightarrow_\beta P'}{if\,(P; M; N) \longrightarrow_\beta if\,(P'; M; N)}\ (\xi^1_{if}) \qquad \frac{M \longrightarrow_\beta M'}{if\,(P; M; N) \longrightarrow_\beta if\,(P; M'; N)}\ (\xi^2_{if})
$$

$$
\frac{N \longrightarrow_\beta N'}{if\,(P; M; N) \longrightarrow_\beta if\,(P; M; N')}\ (\xi^3_{if})
$$

$$
\frac{M \longrightarrow_\beta M'}{\langle M, N\rangle \longrightarrow_\beta \langle M', N\rangle}\ (\xi^1_\times) \qquad \frac{N \longrightarrow_\beta N'}{\langle M, N\rangle \longrightarrow_\beta \langle M, N'\rangle}\ (\xi^2_\times)
$$

$$
\frac{M \longrightarrow_\beta M'}{let\ \langle x, y\rangle{=}M\ in\ N \longrightarrow_\beta let\ \langle x, y\rangle{=}M'\ in\ N}\ (\xi^1_{let})
$$

$$
\frac{N \longrightarrow_\beta N'}{let\ \langle x, y\rangle{=}M\ in\ N \longrightarrow_\beta let\ \langle x, y\rangle{=}M\ in\ N'}\ (\xi^2_{let})
$$

Table 4: β-reduction rules

$(\lambda x.M)N$ is called β-redex. This allows us to speak of different redexes in a given term Q when different subterms of Q are redexes.

Example. Consider the reduction of the following term, where M is any term:

$$(\lambda x.\lambda y.x)M$$

Here we have an application: a function of argument x, fed with M. This application is called a *redex*. We have to substitute all free occurrences of x by M in what's after the '.'. By doing so, we get $\lambda y.M$. We have to be careful: M could contain free occurrences of y. So we have to substitute the bound variable y by a fresh variable z in $\lambda y.x$ before reducing. The result would be: $\lambda z.M$. This is the reason why we need α-equivalence of terms.

An other problem can occur when there is several subterms that are redexes in a term. We have to make a choice.

To illustrate it, consider the following example. Suppose we want to use this function to compose the boolean function **not** and **and** in order to build the boolean **or** function. One can construct them as follows:

$$\begin{aligned}\mathbf{not} &= \lambda x.\, \mathit{if}\,(x;0;1),\\ \mathbf{and} &= \lambda xy.\, \mathit{if}\,(x;\mathit{if}\,(y;1;0);0).\end{aligned}$$

The boolean function **or** can be constructed as follow:

$$\begin{aligned}\mathbf{or} &= \mathbf{not}\lambda xy.\mathbf{and}(\mathbf{not}\ x)(\mathbf{not}\ y)\\ &= (\lambda x.\, \mathit{if}\,(x;0;1))\lambda xy.(\lambda xy.\, \mathit{if}\,(x;\mathit{if}\,(y;1;0);0))\\ &\quad ((\lambda x.\, \mathit{if}\,(x;0;1));x)((\lambda x.\, \mathit{if}\,(x;0;1))\ y)\end{aligned}$$

Consider the computation of $\mathbf{or}(1)(0)$. At each step of the reduction of this term we have to make a choice of which subterm to reduce. Starting by reducing **and** before applying the arguments:

$$\begin{aligned}&\mathbf{not}(\lambda xy.\mathbf{and}(\mathbf{not}\ x)(\mathbf{not}\ y))(1)(0)\\ \longrightarrow_\beta\ &\mathbf{not}(\lambda xy.(\lambda z.\, \mathit{if}\,((\mathbf{not}\ x);\mathit{if}\,(z;1;0);0))((\mathbf{not}\ y)))(1)(0)\\ \longrightarrow_\beta\ &\mathbf{not}(\lambda xy.\, \mathit{if}\,((\mathbf{not}\ x);\mathit{if}\,((\mathbf{not}\ y);1;0);0))(1)(0)\\ \longrightarrow_\beta\ &\mathbf{not}(\lambda y.\, \mathit{if}\,((\mathbf{not}\ 1);\mathit{if}\,((\mathbf{not}\ y);1;0);0))(0)\\ \longrightarrow_\beta\ &\mathbf{not}\ \mathit{if}\,((\mathbf{not}\ 1);\mathit{if}\,((\mathbf{not}\ 0);1;0);0)\end{aligned}$$

If we start by reducing (**not** 1):

$$\begin{aligned} &\longrightarrow_\beta \quad \textbf{not}\ \mathit{if}\,((\mathit{if}\,(1;0;1));\mathit{if}\,((\textbf{not}\ 0);1;0);0) \\ &\longrightarrow_\beta \quad \textbf{not}\ \mathit{if}\,(0;\mathit{if}\,((\textbf{not}\ 0);1;0);0) \end{aligned}$$

One can either continue with (**not** 0), or reduce the first **or**, or reduce directly

$$\mathit{if}\,(0;\mathit{if}\,((\textbf{not}\ 0);1;0);0).$$

If we reduce directly:

$$\begin{aligned} &\longrightarrow_\beta \quad \textbf{not}\ 0 \\ &\longrightarrow_\beta \quad \mathit{if}\,(0;0;1) \\ &\longrightarrow_\beta \quad 1 \end{aligned}$$

With this reduction choice, **or** 1 0 = 1.

Lemma 3.1.2 *If M is closed, and if $M \longrightarrow_\beta M'$, then M' is closed.*

Proof. By easy induction on the derivation of $M \longrightarrow_\beta M'$, it is sufficient to prove that $FV(M) \supseteq FV(M')$. □

A central property of β-reduction is the *confluence*, also known as the *Church Rosser Theorem*:

Theorem 3.1.3 *Given M and N two lambda-terms such that $M =_\beta N$, there exists a term P such that $M \longrightarrow_\beta^* P$ and $N \longrightarrow_\beta^* P$.* □

One can find a complete discussion for this result in [2, p.54].

3.1.2 Reduction strategies

As we have seen, there can be several different redexes in a lambda-term, and the question arises in which order to reduce them. The order of reduction often does matter. For example, consider $(\lambda x.0)M$, with M reducing to a value in 100 steps. We can choose to first reduce the argument M, and the reduction takes 101 steps to

reach a value. On another hand, one can start by reducing the lambda-abstraction. Then we reach in one step

$$(\lambda x.0)M \longrightarrow_\beta 0$$

Therefore, when specifying a programming language, it is customary to fix a *reduction strategy*. A reduction strategy specifies, for any given term, which redex, if any, to reduce in the next step.

A standard reduction strategy, or way to choose which subterm to reduce first, is called call-by-value. For a more complete discussion, see [16]. The idea is to start by reducing arguments before applying them. This is the strategy we applied on $(3+5)*7$: we need to first reduce $3 + 5$ to a value before computing the multiplication. The key-point in call-by-value is an abstraction is consider as being a value: that we never reduce an abstraction. The *values* are defined as follows:

$$\begin{array}{lrcl} \textit{Value} & U, V & ::= & x \\ & & | & c \\ & & | & \lambda x.M \\ & & | & \langle U, V \rangle \\ & & | & *\,. \end{array}$$

Let M, M', N and N' be terms, x a variable and V, V_1 and V_2 values. The call-by-value reduction rules are found in Table 5. These rules implements a call-by-value strategy: A β-redex is reduced only if its argument is already a value. Similarly, in an application MN, the reduction always occurs first in the argument N if it is not already a value. Hence M starts reducing only when N is reduced to a value.

Lemma 3.1.4 *If V is a closed value, there is no term M such that $V \longrightarrow_{CBV} M$.*

Proof. No rules can be applied, so no reduction is possible. □

Lemma 3.1.5 *the call-by-value reduction strategy is deterministic: If $M \longrightarrow_{CBV} M'$ and $M \longrightarrow_{CBV} M''$, then $M' = M''$.*

Proof. By structural induction on M and inspection of the possible rules, only one rule can be applied for each case. □

$$
\begin{array}{lrcl}
(\beta) & (\lambda x.M)V & \longrightarrow_{CBV} & M[V/x] \\
(if_1) & if\,(1; M; N) & \longrightarrow_{CBV} & M \\
(if_0) & if\,(0; M; N) & \longrightarrow_{CBV} & N \\
(let) & let\ \langle x_1, x_2\rangle{=}\langle V_1, V_2\rangle\ in\ N & \longrightarrow_{\beta} & N[V_1/x_1, V_2/x_2]
\end{array}
$$

$$
\frac{M \longrightarrow_{CBV} M'}{MV \longrightarrow_{CBV} M'V}\ (cong_1) \qquad \frac{N \longrightarrow_{CBV} N'}{MN \longrightarrow_{CBV} MN'}\ (cong_2)
$$

$$
\frac{P \longrightarrow_{CBV} P'}{if(P; M; N) \longrightarrow_{CBV} if(P'; M; N)}\ (\xi_{if})
$$

$$
\frac{M \longrightarrow_{CBV} M'}{\langle M, N\rangle \longrightarrow_{CBV} \langle M', N\rangle}\ (\xi^1_{\times}) \qquad \frac{N \longrightarrow_{CBV} N'}{\langle M, N\rangle \longrightarrow_{CBV} \langle M, N'\rangle}\ (\xi^2_{\times})
$$

$$
\frac{M \longrightarrow_{CBV} M'}{let\ \langle x, y\rangle{=}M\ in\ N \longrightarrow_{CBV} let\ \langle x, y\rangle{=}M'\ in\ N}\ (\xi_{let})
$$

Table 5: Intuitionistic call-by-value reduction strategy

3.2 Typed lambda-calculus

The notion of lambda-term is a powerful way of representing functions and programs. But we need a way to prevent run-time errors as much as possible. For example, $if\,(\lambda x.x; 1; 1)$ cannot be reduced, but it is not a value. It is a run-time error. The usual way to prevent them is to use what is called a **type system**. A type is a structure that we associate with a term to define the behavior of this term in a piece of code. For example, in a program, you may want to know if a variable is a *string*, to check if you are allowed to concatenate it with another string. You may also want to know if a variable refers to a function, and what kind of function. This notation makes the program more readable by the programmer to determine exactly how to use an expression. A term which admits a type is called *typable*. A term together with a type is called *well-typed*. A powerful enough type system must verify two things.

It should verify the safety property. This include the *preservation* theorem, also known as *subject reduction*, and the *progress* theorem. A programming language that verifies subject reduction is such that any program keeps the same type while reducing. The progress theorem states that a well-typed term is either a value or can

be reduced.

It should also have a *type inference algorithm*. Given a term, the algorithm has to answer whether or not the term is typable. If it is typable, the algorithm could also give back, if possible, a characterization of the set of all possible types for the term. This algorithm is useful for the programmer since he does not have to specify the type manually.

In this section, we describe a type system for the lambda-calculus defined above, and discuss the safety property and a type inference algorithm.

Type system. Following the mathematical intuition, denotationally a type is a a set of λ-terms. We have a notion of function, a notion of product and some basic terms. We need at least

$$\begin{array}{llll} \mathit{Type} & A, B & ::= & X \\ & & | & \alpha \\ & & | & (A \Rightarrow B) \\ & & | & \top \\ & & | & (A \times B), \end{array}$$

where α spans $\mathcal{C}_{type}$ a set of type constants and X spans $\mathcal{V}_{type}$, a countable set of type variables. $\mathcal{C}_{type}$ needs to contain at least bit, to store the term constants 0 and 1. The notation $(A \Rightarrow B)$ stands for the set of functions of domain A and co-domain B, and $(A \times B)$ for the set of pairs of an element in A and an element in B. $\top$ is the type with a single element $*$.

Typing rules. A term with free variables can only be well typed if its free variables have a well-known type. This is the reason why we define what is called a *typing judgment*. A typing judgment is a tuple

$$\Delta \blacktriangleright M : B$$

where M is a term, B is a type, and Δ is a **set** of variables $|\Delta| = \{x_1, \ldots x_n\}$ together with a function Δ_f from $|\Delta|$ to the set of types. We usually denote Δ by

$$\frac{}{\Delta \blacktriangleright c : A_c}\ (c) \qquad \frac{}{\Delta, x : A \blacktriangleright x : A}\ (x)$$

$$\frac{\Delta, x : A \blacktriangleright M : B}{\Delta \blacktriangleright \lambda x.M : A \Rightarrow B}\ (\lambda) \qquad \frac{\Delta \blacktriangleright M : A \Rightarrow B \quad \Delta \blacktriangleright N : A}{\Delta \blacktriangleright MN : B}\ (app)$$

$$\frac{\Delta \blacktriangleright P : bit \quad \Delta \blacktriangleright M : A \quad \Delta \blacktriangleright N : A}{\Delta \blacktriangleright if\,(P; M; N) : A}\ (if)$$

$$\frac{\Delta \blacktriangleright M_1 : A_1 \quad \Delta \blacktriangleright M_2 : A_2}{\Delta \blacktriangleright \langle M_1, M_2 \rangle : A_1 \times A_k}\ (\times)$$

$$\frac{}{\Delta \blacktriangleright * : \top}\ (\top) \qquad \frac{\Delta \blacktriangleright M : C \times D \quad \Delta, x{:}C, y{:}D \blacktriangleright N : A}{\Delta \blacktriangleright let\ \langle x, y \rangle{=}M\ in\ N : A}\ (let)$$

Table 6: Typing rules for the simply-typed lambda-calculus

$\{x_1 : A_1, \ldots x_n : A_n\}$, with $A_i = \Delta_f(x_i)$. Δ is called a *typing context*. A typing judgment is said to be valid if it can be derived from the rules in Table 6.

We write Δ_1, Δ_2 for $\Delta_1 \cup \Delta_2$ with $|\Delta_1| \cap |\Delta_2| = \emptyset$. We also write $\Delta, x{:}A$ for $\Delta, \{x{:}A\}$.

For each term constant $c \in \mathcal{C}_{term}$, fix a type A_c, such that $A_0 = A_1 = bit$.

3.2.1 Properties of typing judgments

Lemma 3.2.1 (Weakening) *If $x \notin FV(M)$ and $\Delta, x{:}A \blacktriangleright M{:}B$ is valid, then*

$$\Delta \blacktriangleright M{:}B$$

is also valid.

Proof. By structural induction on the typing-tree of $\Delta, x{:}A \blacktriangleright M{:}B$.

(c) $M = c$ and $B = A_c$. Then $\Delta \blacktriangleright M : B$ is an application of the (c) rule.

(x) $M = y$, $y \neq x$ since $x \notin FV(M)$. From the rule, $y \in |\Delta|$. So $\Delta \blacktriangleright M : B$ is valid, applying (x).

(λ) $M = \lambda y.P$. The typing tree starts with

$$\frac{\begin{array}{c}\vdots\\ \Delta, x{:}A, y{:}C \blacktriangleright P{:}D\end{array}}{\Delta, x{:}A \blacktriangleright \lambda y.P{:}C \Rightarrow D.}$$

From the definition of contexts, $y \neq x$. Then since $x \notin FV(M)$, $x \notin FV(P)$. Applying induction hypothesis, one gets that $\Delta, y{:}C \blacktriangleright P{:}D$ is valid. (λ) can be applied: $\Delta \blacktriangleright \lambda y.P{:}C \Rightarrow D$ is valid.

(app) $M = NP$, and the typing tree starts with

$$\frac{\begin{array}{c}\vdots\\ \Delta, x{:}A \blacktriangleright N{:}C \Rightarrow D\end{array} \quad \begin{array}{c}\vdots\\ \Delta, x{:}A \blacktriangleright P : C\end{array}}{\Delta, x{:}A \blacktriangleright NP{:}D}\ (app).$$

Since $FV(NP) = FV(N) \cup FV(P)$, $x \notin FV(N)$ and $x \notin FV(P)$. Then we can apply the induction hypothesis: $\Delta \blacktriangleright N{:}C \Rightarrow D$ and $\Delta, x{:}A \blacktriangleright P{:}C$ are valid. Applying (app), one gets that $\Delta \blacktriangleright NP{:}D$ is valid.

(if) $M = if(N; P; Q)$, and the typing tree starts with

$$\frac{\begin{array}{c}\vdots\\ \Delta, x{:}A \blacktriangleright N{:}bit\end{array} \quad \begin{array}{c}\vdots\\ \Delta, x{:}A \blacktriangleright P{:}A\end{array} \quad \begin{array}{c}\vdots\\ \Delta, x{:}A \blacktriangleright Q{:}A\end{array}}{\Delta, x{:}A \blacktriangleright if(N; P; Q){:}A}\ (if)$$

Since $FV(if(N; P; Q)) = FV(N) \cup FV(P) \cup FV(Q)$, $x \notin FV(N)$, $x \notin FV(P)$ and $x \notin FV(Q)$. Then we can apply the induction hypothesis: $\Delta \blacktriangleright N{:}bit$, $\Delta \blacktriangleright P{:}A$ and $\Delta \blacktriangleright Q{:}A$ are valid typing judgments. With rule (if) we get that $\Delta \blacktriangleright if(N; P; Q){:}A$ is valid.

$(\times)$ $M = \langle M_1, \ldots, M_k \rangle$, and the typing tree starts with

$$\frac{\begin{array}{c}\vdots\\ \Delta, x{:}A \blacktriangleright M_1 : A_1\end{array} \quad \begin{array}{c}\vdots\\ \Delta, x{:}A \blacktriangleright M_2 : A_2\end{array}}{\Delta, x{:}A \blacktriangleright \langle M_1, \ldots, M_k \rangle : A_1 \times \ldots \times A_k}$$

Since $FV(\langle M_1, M_2\rangle) = FV(M_1) \cup FV(M_2)$, the induction hypothesis apply on each branch of the typing tree, and $\Delta \blacktriangleright M_i : A_i$ is valid for all i. One can apply $(\times)$, and we get

$$\Delta \blacktriangleright \langle M_1, M_2\rangle : A_1 \times A_2$$

is valid.

$(\top)$ $M = *$ and the typing tree is

$$\frac{}{\Delta, x{:}A \blacktriangleright * : \top}$$

Applying this rule, one see that $\Delta \blacktriangleright *{:}\top$ is valid.

(let) The typing tree starts with

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Delta, x{:}A \blacktriangleright M{:}C \times D & \Delta, x{:}A, y{:}C, z{:}D \blacktriangleright N{:}A\end{array}}{\Delta, x{:}A \blacktriangleright \mathit{let}\ \langle y, z\rangle{=}M\ \mathit{in}\ N{:}A}.$$

Since $FV(\mathit{let}\ \langle y, z\rangle{=}M\ \mathit{in}\ N) = FV(M) \cup (FV(N) \setminus \{y, z\})$ and from the definition of context, $x \notin FV(N)$ and $x \notin FV(M)$. Then we can apply the induction hypothesis: $\Delta, y{:}C, z{:}D \blacktriangleright N{:}A$ and $\Delta \blacktriangleright M{:}C \times D$ are valid. Applying (let), one gets that $\Delta \blacktriangleright \mathit{let}\ \langle y, z\rangle{=}M\ \mathit{in}\ N{:}A$ is valid.

□

Lemma 3.2.2 (Renaming of variables) *Given a valid typing judgment*

$$\Delta, x{:}C \blacktriangleright M{:}A$$

and z a fresh variable, $\Delta, z{:}C \blacktriangleright M[z/x]{:}A$ is valid with a typing-tree of the same depth as the typing tree of $\Delta, x{:}C \blacktriangleright M{:}A$.

Proof. By induction on the typing tree of $\Delta, x{:}C \blacktriangleright M{:}A$.

(c) $M = c$, $A = A_c$ and the typing tree is

$$\frac{}{\Delta, x{:}C \blacktriangleright c : A_c}$$

Moreover, $M[z/x] = c$ Applying this rule from scratch, $\Delta, z{:}C \blacktriangleright c : A_c$ is valid. Hence the result is true in this case, and the typing tree has a depth of 1.

(x) $M = y$, so there are two cases. First, one can have $y = x$. $A = C$ and the typing tree is

$$\overline{\Delta, x : C \blacktriangleright x : C}$$

$M[z/x] = z$, so directly from (x)

$$\Delta, z : C \blacktriangleright z : C$$

is valid with typing tree of depth 1.

(λ) $M = lambday.N$. The typing tree starts with

$$\frac{\begin{array}{c}\vdots\ \omega \\ \Delta, x{:}C, y{:}A \blacktriangleright N{:}B\end{array}}{\Delta, x{:}C \blacktriangleright \lambda y.N{:}A \Rightarrow B}$$

From the definition of concatenation in typing judgment, $x \neq y$. From induction hypothesis, $\Delta, z{:}C, y{:}A \blacktriangleright N[z/x]{:}B$ is valid with typing tree ω' of depth the depth of ω. Applying (λ), one get

$$\Delta, z{:}C \blacktriangleright \lambda y.(N[z/x]){:}A \Rightarrow B$$

since $y \neq x$, $\lambda y.(N[z/x]) = (\lambda y.N)[z/x]$. So $\Delta, z{:}C \blacktriangleright M[z/x]{:}A \Rightarrow B$ is valid with a typing-tree of the same depth as the typing tree of

$$\Delta, x{:}C \blacktriangleright M{:}A \Rightarrow B.$$

(app) $M = NP$. The typing tree ω starts with

$$\frac{\begin{array}{c}\vdots\ \omega_1 \\ \Delta, x{:}C \blacktriangleright N : B \Rightarrow A\end{array} \quad \begin{array}{c}\vdots\ \omega_2 \\ \Delta, x{:}C \blacktriangleright P : B\end{array}}{\Delta, x{:}C \blacktriangleright NP : A}\ (app)$$

$d(\omega) = 1 + \max(d(\omega_1), d(\omega_2))$. From induction hypothesis

$$\Delta, z{:}C \blacktriangleright N[z/x] : B \Rightarrow A \text{ and } \Delta, z{:}C \blacktriangleright P[z/x] : B$$

are valid and of depth $d(\omega_1)$ and $d(\omega_2)$.
From (app) and the fact that $(NP)[z/x] = N[z/x]P[z/x]$,

$$\Delta, z{:}C \blacktriangleright (NP)[z/x] : A$$

is valid of depth $d(\omega)$.

(if) $M = if(P;Q;N)$. The typing tree ω starts with

$$\frac{\overset{\vdots\ \omega_1}{\Delta, x{:}C \blacktriangleright P : bit} \quad \overset{\vdots\ \omega_2}{\Delta, x{:}C \blacktriangleright Q : A} \quad \overset{\vdots\ \omega_3}{\Delta, x{:}C \blacktriangleright N : A}}{\Delta, x{:}C \blacktriangleright if(P;Q;N) : A}\ (if)$$

$d(\omega) = 1 + \max(d(\omega_1), d(\omega_2), d(\omega_3))$. From induction hypothesis,

$$\Delta, z{:}C \blacktriangleright P[z/x]{:}bit,\ \Delta, z{:}C \blacktriangleright Q[z/x]{:}A \text{ and } \Delta, z{:}C \blacktriangleright N[z/x]{:}A$$

are valid of depth $d(\omega_1)$, $d(\omega_2)$ and $d(\omega_3)$. Applying (if) and from the definition of substitution,

$$\Delta, z{:}C \blacktriangleright if(P;Q;N)[z/x]{:}A$$

is valid of depth $d(\omega)$.

$(\times)$ $M = \langle M_1, M_2 \rangle$, and typing tree ω starts with

$$\frac{\overset{\vdots\ \omega_1}{\Delta, x{:}C \blacktriangleright M_1 : A_1} \quad \overset{\vdots\ \omega_2}{\Delta, x{:}C \blacktriangleright M_2 : A_2}}{\Delta, x{:}C \blacktriangleright \langle M_1, M_2 \rangle : A_1 \times A_k}\ (\times)$$

$d(\omega) = 1 + \max(d(\omega_1), d(\omega_2))$. From induction hypothesis,

$$\Delta, z{:}C \blacktriangleright M_1[z/x]{:}A_1 \text{ and } \Delta, z{:}C \blacktriangleright M_2[z/x]{:}A_2$$

are valid of depth $d(\omega_1)$ and $d(\omega_2)$. Applying $(\times)$ and from the definition of substitution,

$$\Delta, z{:}C \blacktriangleright \langle M_1, M_2 \rangle[z/x]{:}A_1 \times A_2$$

is valid of depth $d(\omega)$.

(*let*) The typing tree starts with

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Delta, x{:}A \blacktriangleright M{:}C \times D & \Delta, x{:}A, y{:}C, t{:}D \blacktriangleright N{:}A\end{array}}{\Delta, x{:}A \blacktriangleright \mathit{let}\ \langle y, t\rangle{=}M\ \mathit{in}\ N{:}A}.$$

By induction hypothesis,

$$\Delta, z{:}A \blacktriangleright M[z/x]{:}C \times D \quad \text{and} \quad \Delta, z{:}A, y{:}C, t{:}D \blacktriangleright N[z/x]{:}A$$

are valid. From the definition of context, x is different from y and t, so $(\mathit{let}\ \langle y, t\rangle{=}M\ \mathit{in}\ N)[z/x] = (\mathit{let}\ \langle y, t\rangle{=}M[z/x]\ \mathit{in}\ N[z/x])$. Then applying the (*let*) rule,

$$\Delta, z{:}A \blacktriangleright (\mathit{let}\ \langle y, t\rangle{=}M\ \mathit{in}\ N)[z/x]{:}A$$

is valid.

($\top$) $M = *$, $A = \top$ and the typing tree is

$$\frac{}{\Delta, x{:}C \blacktriangleright *{:}\top}$$

Moreover, $M[z/x] = *$ Applying this rule from scratch, $\Delta, z{:}C \blacktriangleright *{:}\top$ is valid. Hence the result is true in this case, and the typing tree has a depth of 1.

□

Lemma 3.2.3 (Substitution) *Given*

$$\Delta, x_1{:}C_1, \ldots x_n{:}C_n \blacktriangleright M{:}A,$$

$$\Delta \blacktriangleright N_i{:}C_i \quad \forall i = 1 \ldots n,$$

and

$$\sigma = \{\ x_i \mapsto N_i, \quad i = 1 \ldots n\ \},$$

the typing judgment $\Delta \blacktriangleright \bar{\sigma}(M){:}A$ *is valid.*

Proof.

The typing-tree of $\Delta \blacktriangleright \bar{\sigma}(M){:}A$ is constructed by induction on the structure of the typing tree.

(c) $\bar{\sigma}(c) = c$, then applying the same rule (c), $\Delta \blacktriangleright c{:}A$is valid.

(x) In this case, $M = x$. Since $\bar{\sigma}(x) = \sigma(x)$, there are 2 cases: either $x \notin |\sigma|$, and so $x \in |\Delta|$ and $\sigma(x) = x$: from the rule (x), $\Delta \blacktriangleright x{:}A$ is valid. In the other case, $x = x_i \in |\sigma|$, so $A = C_i$ and $\sigma(x) = M_i$. But from the hypothesis, $\Delta \blacktriangleright M_i : C_i$ is valid. So in both case, the result is valid.

(λ) In this case, $M = \lambda x.N$. The typing tree starts with:

$$\frac{\Delta, x{:}C \blacktriangleright N{:}D}{\Delta \blacktriangleright \lambda x.N{:}C \Rightarrow D}$$

$\bar{\sigma}(\lambda x.N) = \lambda z.\bar{\sigma}'(N) =$, with z a fresh variable and $\sigma' = \sigma \circ \{x \mapsto z\}$. From Lemma 3.2.2, $\Delta, z{:}C \blacktriangleright N[z/x]{:}D$ is valid, and the typing tree have the same depth as the one of $\Delta, x{:}C \blacktriangleright N{:}D$. By induction hypothesis,

$$\Delta, z{:}C \blacktriangleright \bar{\sigma}(N[z/x]){:}D.$$

Since $\bar{\sigma}(N[z/x]) = \bar{\sigma}'(N)$, applying (λ),

$$\Delta \blacktriangleright \bar{\sigma}(\lambda x.N){:}C \Rightarrow D$$

is valid

(app) In this case $M = NP$. The typing tree is:

$$\frac{\Delta \blacktriangleright N{:}C \Rightarrow D \quad \Delta \blacktriangleright P{:}C}{\Delta \blacktriangleright NP : D}$$

From induction hypothesis, $\Delta \blacktriangleright \bar{\sigma}(N){:}C \Rightarrow D$ and $\Delta \blacktriangleright \bar{\sigma}(P){:}C$ Applying (app), $\Delta \blacktriangleright \bar{\sigma}(N)\bar{\sigma}(P) : D$ is valid. Since $\bar{\sigma}(NP) = \bar{\sigma}(N)\bar{\sigma}(P)$.

(if) In this case $M = if(N; P; Q)$. The typing tree is:

$$\frac{\Delta \blacktriangleright N : bit \quad \Delta \blacktriangleright P : C \quad \Delta \blacktriangleright Q : C}{\Delta \blacktriangleright if(N; P; Q) : C}$$

From induction hypothesis,

$$\Delta \blacktriangleright \bar{\sigma}N{:}\, bit,\ \Delta \blacktriangleright \bar{\sigma}P{:}C \text{ and } \Delta \blacktriangleright \bar{\sigma}Q{:}C.$$

Then, applying (if), $\Delta \blacktriangleright if(\bar{\sigma}N; \bar{\sigma}P; \bar{\sigma}Q){:}C$ is valid. Since

$$if(\bar{\sigma}N; \bar{\sigma}P; \bar{\sigma}Q) = \bar{\sigma}(if(N; P; Q)),$$

the result is true in this case.

$(\times)$ $M = \langle M_1, M_2 \rangle$. The typing tree starts with

$$\frac{\Delta \blacktriangleright M_1 : A_1 \quad \Delta \blacktriangleright M_2 : A_2}{\Delta \blacktriangleright \langle M_1, M_2 \rangle : A_1 \times A_2}$$

By induction hypothesis, for all i, $\Delta \blacktriangleright \bar{\sigma}(M_i) : A_i$ is valid. Applying $(\times)$ and using the relation $\bar{\sigma}(\langle M_1, M_2 \rangle) = \langle \bar{\sigma}(M_1), \bar{\sigma}(M_2) \rangle$, the typing judgment

$$\Delta \blacktriangleright \bar{\sigma}(\langle M_1, M_2 \rangle) : A_1 \times A_2$$

is valid

(let) The typing tree starts with

$$\frac{\begin{array}{c}\vdots \\ \Delta, x_1{:}C_1, \ldots x_n{:}C_n \blacktriangleright M{:}C \times D\end{array} \quad \begin{array}{c}\vdots \\ \Delta, x_1{:}C_1, \ldots x_n{:}C_n, y{:}C, t{:}D \blacktriangleright N{:}A\end{array}}{\Delta, x_1{:}C_1, \ldots x_n{:}C_n \blacktriangleright let\ \langle y, t \rangle{=}M\ in\ N{:}A}.$$

By induction hypothesis,

$$\Delta \blacktriangleright \bar{\sigma}M{:}C \times D \quad \text{and} \quad \Delta, y{:}C, t{:}D \blacktriangleright \bar{\sigma}N{:}A$$

are valid. From the definition of context, x is different from all x_i, so

$$\bar{\sigma}(let\ \langle y, t \rangle{=}M\ in\ N) = (let\ \langle y, t \rangle{=}\bar{\sigma}M\ in\ \bar{\sigma}N).$$

Then applying the (let) rule,

$$\Delta \blacktriangleright \bar{\sigma}(let\ \langle y, t \rangle{=}M\ in\ N){:}A$$

is valid.

$(\top)$ $M = *$ and the typing judgment is

$$\frac{}{\Delta \blacktriangleright * : \top}$$

$\bar{\sigma}(*) = *$ so the typing judgment $\Delta \blacktriangleright \bar{\sigma}(*){:}\top$ is valid.

□

Theorem 3.2.4 (Subject Reduction) *If $\Delta \blacktriangleright M{:}A$ is valid and $M \longrightarrow_\beta{}^* N$ then $\Delta \blacktriangleright N{:}A$ is valid.*

Proof.

It is sufficient to prove it for $\longrightarrow_\beta$. We will do that by structural induction on the derivation of $M \longrightarrow_\beta N$: For all valid typing judgments $\Delta \blacktriangleright M : A$, the typing judgment $\Delta \blacktriangleright N : A$ is valid.

(β) In that case, the rule is

$$(\lambda x.P)Q \longrightarrow_\beta P[Q/x]$$

and $(\lambda x.P)Q$ has for unique typing tree

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Delta, x{:}A \blacktriangleright P{:}B\end{array}}{\Delta \blacktriangleright \lambda x.P{:}A \Rightarrow B} \quad \begin{array}{c}\vdots\\ \Delta \blacktriangleright Q{:}A\end{array}}{\Delta \blacktriangleright (\lambda x.P)Q{:}B}\ .$$

From Lemma (3.2.3) one can deduce that $\Delta \blacktriangleright P[Q/x]{:}B$ is well typed.

(if_0) The rule is

$$if\,(0; M; N) \longrightarrow_\beta N$$

and $if\,(0; M; N)$ has for unique typing tree

$$\dfrac{\overline{\Delta \blacktriangleright 0 : bit} \quad \Delta \blacktriangleright M : A \quad \Delta \blacktriangleright N : A}{\Delta \blacktriangleright if\,(1; M; N) : A}\ .$$

There is nothing to do: $\Delta \blacktriangleright N{:}A$

(if_1) For the same reason as above, $\Delta \blacktriangleright M{:}A$

(let) In this case the rule is

$$let\ \langle x, y\rangle{=}\langle M_1, M_2\rangle\ in\ N \longrightarrow_\beta N[M_1/x, M_2/y]$$

and $let\ \langle x, y\rangle{=}\langle M_1, M_2\rangle\ in\ N$ has for unique typing tree

$$\dfrac{\dfrac{\Delta \blacktriangleright M_1{:}C \quad \Delta \blacktriangleright M_2{:}D}{\Delta \blacktriangleright \langle M_1, M_2\rangle : C \times D} \quad \Delta, x{:}C, y{:}D \blacktriangleright N : A}{\Delta \blacktriangleright let\ \langle x, y\rangle{=}\langle M_1, M_2\rangle\ in\ N : A}\ (let).$$

Applying Lemma 3.2.3, one can conclude that

$$\Delta \blacktriangleright N[M_1/x, M_2/y]$$

is valid.

$(cong_1)$ The rule is

$$\dfrac{M \longrightarrow_\beta M'}{MN \longrightarrow_\beta M'N}\ (cong_1)$$

and MN has for unique typing tree

$$\dfrac{\begin{array}{c}\vdots\\ \Delta \blacktriangleright M{:}A \Rightarrow B\end{array} \quad \begin{array}{c}\vdots\\ \Delta \blacktriangleright N{:}A\end{array}}{\Delta \blacktriangleright MN{:}B}.$$

By induction hypothesis, $\Delta \blacktriangleright M'{:}A \Rightarrow B$ is valid. Applying (app), one get

$$\Delta \blacktriangleright M'N{:}B$$

$(cong_2)$ The proof is similar as for $(cong_1)$: the rule is

$$\dfrac{M \longrightarrow_\beta M'}{MN \longrightarrow_\beta MN'}\ (cong_1)$$

and MN has for unique typing tree the same as above. From the induction hypothesis, $\Delta \blacktriangleright N'{:}A$. Applying (app), one get

$$\Delta \blacktriangleright MN'{:}B$$

(if^i) **and** $(\times^i)$ The proof is the same as the one for $(cong_i)$.

(ξ) The (ξ) rules are all on the same model. Here is the proof for (ξ_λ).

$$\frac{P \longrightarrow_\beta P'}{\lambda x.P \longrightarrow_\beta \lambda x.P'} \ (\xi)$$

and $\lambda x.P$ has for unique typing tree

$$\frac{\begin{array}{c}\vdots \\ \Delta, x : A \blacktriangleright P : B\end{array}}{\Delta \blacktriangleright \lambda x.P{:}A \Rightarrow B.}$$

By induction hypothesis, $\Delta, x : A \blacktriangleright P' : B$. Applying ($\lambda$) one get the result:

$$\Delta \blacktriangleright \lambda x.P'{:}A \Rightarrow B$$

□

Corollary 3.2.5 *If $\Delta \blacktriangleright M{:}A$ and $M \longrightarrow_{CBV}{}^* N$ then $\Delta \blacktriangleright N{:}A$.*

Proof. Using Theorem 3.2.4 and the fact that every call-by-value reduction is also a β-reduction, the corollary is true. □

Theorem 3.2.6 (Progress) *If $\blacktriangleright M{:}A$ is valid, either M is a value, or M reduces to some term N by call-by-value.*

Proof. We prove it by induction of the derivation of $\blacktriangleright M{:}A$. If M is a value, there is nothing to prove. If it is not, then there are the following cases

$M = NP$**.** By the (λ) rule, $\blacktriangleright N{:}B \Rightarrow A$ is valid for some type B, and $\Delta \blacktriangleright P{:}B$ is valid. By induction hypothesis, either N or P reduces, or they are both values. If they are both values, then $N = \lambda x.N'$ since this is the only value that can have type $B \Rightarrow A$. And thus call-by-value reduction applies with rule (β). If one of N or P is not a value, then a call-by-value reduction can also be applied, by rule ($cong_1$) or ($cong_2$).

$M = if(N; P; Q)$**.** By the (if) rule, $\blacktriangleright N{:}\, bit$ is valid. Either N is a value, in which case $N = 0$ or $N = 1$ and M reduces by the (if_0) or (if_1) rule, or it is not, and (ξ_{if}) can be applied.

$M = let\ \langle x, y\rangle{=}N\ in\ P$. By the (let) rule, $\blacktriangleright M{:}A$ is valid and comes from

$$\blacktriangleright N{:}C{\times}D \quad \text{and} \quad x{:}C, y{:}D \blacktriangleright P{:}A.$$

First case N could be a value, and then $N = \langle V_1, V_2\rangle$. In this case, the (let) reduction can be applied. In the other case, by induction N can be reduced. Then ξ_{let} can be applied.

□

3.2.2 Type inference algorithm

With the subject reduction and the progress theorems, we are able to certify the well-behavior of a program during reduction using a type system: A well-typed term can never produce a run-time error. In consequence, we are interested to know if given a lambda-term, this term can be well-typed.

Given a typable term, there exist a lot of possible types for this term. Consider the term $\lambda xy.xy$. All these are valid typing judgments:

$$\begin{aligned}
&\blacktriangleright \lambda xy.xy{:}(\alpha \Rightarrow X) \Rightarrow (\alpha \Rightarrow X),\\
&\blacktriangleright \lambda xy.xy{:}((\alpha \times bit) \Rightarrow \alpha) \Rightarrow ((\alpha \times bit) \Rightarrow \alpha),\\
&\blacktriangleright \lambda xy.xy{:}((\alpha \Rightarrow Y) \Rightarrow (\alpha \times A)) \Rightarrow ((\alpha \Rightarrow Y) \Rightarrow (\alpha \times A)),\\
&\blacktriangleright \lambda xy.xy{:}(C \Rightarrow bit) \Rightarrow (C \Rightarrow bit).
\end{aligned}$$

One can see that there is a general form form this typing judgment, namely:

$$\blacktriangleright \lambda xy.xy{:}(A \Rightarrow B) \Rightarrow (A \Rightarrow B).$$

More generally, every term in the simply-typed lambda-calculus has a most general type. We now make this notion precise.

Type substitution We define a *type substitution* to be a function from $\mathcal{V}_{type}$ to types. We extend this notion to a function $\bar{\sigma}$ from types to types as follows:

$$\begin{aligned} \bar{\sigma}(X) &= \sigma(X) \\ \bar{\sigma}(\alpha) &= \alpha \\ \bar{\sigma}(\top) &= \top \\ \bar{\sigma}(A \Rightarrow B) &= \bar{\sigma}(A) \Rightarrow \bar{\sigma}(B) \\ \bar{\sigma}(A \times B) &= \bar{\sigma}(A) \times \bar{\sigma}(B) \end{aligned}$$

Given a typing judgment $\Delta = \{x_1{:}A_1 \ldots x_n{:}A_n\}$, we write

$$\bar{\sigma}\Delta = \{x_1{:}\bar{\sigma}A_1 \ldots x_n{:}\bar{\sigma}A_n\}$$

With the definition of type substitution, we are able to say that a type A is said to be *more general* than a type B, if there exists a type substitution σ such that $\bar{\sigma}(A) = B$. We also say that B is an *instance* of A. We can also define this concept for typing judgments, type derivation and substitutions: σ is more general than τ if there exists ρ such that $\bar{\rho} \circ \sigma = \tau$.

In the previous example, the typing judgment

$$\blacktriangleright \lambda xy.xy{:}(X \Rightarrow Y) \Rightarrow (X \Rightarrow Y)$$

is more general than all the other ones we gave.

Lemma 3.2.7 *Given σ and τ two type substitutions, $\overline{\bar{\sigma} \circ \tau} = \bar{\sigma} \circ \bar{\tau}$.* $\square$

Lemma 3.2.8 *Given a valid typing judgment $\Delta \blacktriangleright M : A$, for any substitution σ, the typing judgment $\bar{\sigma}\Delta \blacktriangleright M : \bar{\sigma}A$ is valid.*

Proof.

by structural induction on the typing tree of $\Delta \blacktriangleright M : A$.

(c) The rule is $\Delta \blacktriangleright c{:}A_c$. For all $c \in \mathcal{V}_{term}$, $\bar{\sigma}(A_c) = A_c$. Hence $\bar{\sigma} \blacktriangleright c{:}\bar{\sigma}A_c$ is valid.

$(\top)$ is similar as the previous case.

(x) The rule is $\Delta, x{:}A \blacktriangleright x{:}A$. the image of this typing judgment by σ is

$$\bar{\sigma}\Delta, x{:}\bar{\sigma}A \blacktriangleright x{:}\bar{\sigma}A$$

which is valid applying (x).

(λ) The rule is

$$\frac{\Delta, x : A \blacktriangleright M : B}{\Delta \blacktriangleright \lambda x.M : A \Rightarrow B}$$

By induction hypothesis

$$\bar{\sigma}\Delta, x{:}\bar{\sigma}A \blacktriangleright M{:}\bar{\sigma}B$$

is valid. Applying (λ) and from the definition of $\bar{\sigma}$,

$$\bar{\sigma}\Delta \blacktriangleright \lambda x.M : \bar{\sigma}(A \Rightarrow B)$$

is valid.

(app) The rule is

$$\frac{\Delta \blacktriangleright M : A \Rightarrow B \quad \Delta \blacktriangleright N : A}{\Delta \blacktriangleright MN : B}$$

By induction hypothesis

$$\bar{\sigma}\Delta \blacktriangleright M : \bar{\sigma}(A \Rightarrow B) \text{ and}$$
$$\bar{\sigma}\Delta \blacktriangleright N : \bar{\sigma}A$$

From the definition of $\bar{\sigma}$

$$\bar{\sigma}\Delta \blacktriangleright M : \bar{\sigma}A \Rightarrow \bar{\sigma}B$$

is valid. Applying (app)

$$\bar{\sigma}\Delta \blacktriangleright MN : \bar{\sigma}B$$

is valid.

(if) The rule is

$$\frac{\Delta \blacktriangleright P : bit \quad \Delta \blacktriangleright M : A \quad \Delta \blacktriangleright N : A}{\Delta \blacktriangleright if(P; M; N) : A}$$

By induction hypothesis

$$\bar{\sigma}\Delta \blacktriangleright P : \bar{\sigma}\ bit, \bar{\sigma}\Delta \blacktriangleright M : \bar{\sigma}A \text{ and}$$
$$\bar{\sigma}\Delta \blacktriangleright N : \bar{\sigma}A$$

are valid. Since $\bar{\sigma}\ bit = bit$, one can apply (if) and

$$\bar{\sigma}\Delta \blacktriangleright if(P; M; N) : \bar{\sigma}A$$

is valid.

$(\times)$ The rule is

$$\frac{\Delta \blacktriangleright M_1 : A_1 \quad \Delta \blacktriangleright M_2 : A_2}{\Delta \blacktriangleright \langle M_1, M_2 \rangle : A_1 \times A_k}$$

By induction hypothesis

$$\bar{\sigma}\Delta \blacktriangleright M_1{:}\bar{\sigma}A_1 \text{ and}$$
$$\bar{\sigma}\Delta \blacktriangleright M_2{:}\bar{\sigma}A_2$$

Since $\bar{\sigma}(A_1 \times A_k) = \bar{\sigma}A_1 \times \bar{\sigma}A_2$,

$$\bar{\sigma}\Delta \blacktriangleright \langle M_1, M_2 \rangle : \bar{\sigma}(A_1 \times A_k)$$

is valid.

(let) The rule is

$$\frac{\Delta \blacktriangleright N{:}C{\times}D \quad \Delta, x{:}C, y{:}D \blacktriangleright P{:}A}{\Delta \blacktriangleright let\ \langle x, y\rangle{=}N\ in\ P{:}A} .$$

By induction hypothesis,

$$\bar{\sigma}\Delta \blacktriangleright N{:}\bar{\sigma}(C{\times}D) \quad \text{and} \quad \bar{\sigma}\Delta, x{:}\bar{\sigma}C, y{:}\bar{\sigma}D \blacktriangleright P{:}\bar{\sigma}A.$$

Since $\bar{\sigma}(C{\times}D) = \bar{\sigma}C{\times}\bar{\sigma}D$, one can apply (let) and

$$\bar{\sigma}\Delta \blacktriangleright let\ \langle x, y\rangle{=}N\ in\ P : \bar{\sigma}A$$

is valid.

□

Unifiers Given two types A and B, we define a *unifier* of A and B to be a type substitution σ such that $\bar{\sigma}(A) = \bar{\sigma}(B)$. We say that σ is *principal*, or *most general*, if any unifier σ' of A and B is less general than σ. For a complete discussion on unifiers, see [16, p.326].

Given two types A and B, we construct a substitution $unify(A, B)$ from the algorithm, provided that a unifier exists for A and B (else the algorithm fails), as follows:

$$\begin{array}{rcll}
unify(X, X) & = & \emptyset, & \\
unify(\alpha, \alpha) & = & \emptyset, & \\
unify(\top, \top) & = & \emptyset, & \\
unify(X, B) & = & \{X \mapsto B\} & \text{if } X \notin FV(B), \\
unify(B, X) & = & \{X \mapsto B\} & \text{if } X \notin FV(B), \\
unify(A \Rightarrow B, C \Rightarrow D) & = & \bar{\tau} \circ \sigma & \sigma = unify(A, C) \text{ and} \\
 & & & \tau = unify(\bar{\sigma}(B), \bar{\sigma}(D)), \\
unify(A \times B, C \times D) & = & \bar{\tau} \circ \sigma & \sigma = unify(A, C) \text{ and} \\
 & & & \tau = unify(\bar{\sigma}(B), \bar{\sigma}(D)), \\
 & \text{else} & & \text{fails.}
\end{array}$$

For example, a unifier for $X \Rightarrow (Y \times bit)$ and $(W \Rightarrow bit) \Rightarrow W$ is

$$\left\{ \begin{array}{rcl} X & \longmapsto & (Y \times bit) \Rightarrow bit, \\ W & \longmapsto & Y \times bit \end{array} \right\}$$

It maps both types to $((Y \times bit) \Rightarrow bit) \Rightarrow (Y \times bit)$.

The unifier is a substitution on types: sometimes it doesn't exist. For example, there is no unifier for $X \Rightarrow Y$ and $W \times Z$. Since we do not have a recursive type system, there is no unifier for Y and $X \times Y$. Such a unifier would give $X \times (X \times (X \times (\ldots)))$ and such an infinite type is not allowed.

Lemma 3.2.9 (Unification) *$unify(A, B)$ gives a most general unifier of A and B, or fails if there is no unifier.*

Proof. A sketch of the proof is given. A complete proof can be found in [16, p.328]. First we show that $unify(A, B)$ is a unifier for A and B. We prove it by induction

on the derivation of *unify*. Then suppose there exists a unifier ρ for A and B. We prove by induction on the derivation of $unify(A, B)$ that $\rho = \bar{\rho} \circ unify(A, B)$. And so $unify(A, B)$ exists and is more general than any other unifier, if any: it is a principal unifier. □

Type inference algorithm. Now, extending the notion of principal unifier to typing judgments, $infer(\Delta \blacktriangleright M : A)$ is define in Table 7. This definition and the following lemma come from [20, p.60]

Lemma 3.2.10 *Given any (valid or non valid) typing judgment* $\Delta \blacktriangleright M{:}B$, $\sigma = infer(\Delta \blacktriangleright M{:}B)$ *returns the principal substitution such that* $\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\sigma}B$ *is valid, or fails if there is no substitution such that* $\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\sigma}B$ *is valid. Such a substitution is called a unifier for the typing judgment.*

Proof.

We prove the lemma in two steps:

1. If $\sigma = infer(\Delta \blacktriangleright M{:}A)$ exists, then $\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\sigma}A$ is valid, proved by induction on the derivation of $infer(\Delta \blacktriangleright M{:}A)$.

   $infer(\Delta, x{:}A \blacktriangleright x{:}B)$ returns $\sigma = unify(A, B)$. So $\bar{\sigma}A = \bar{\sigma}B$, and $\bar{\sigma}\Delta, x{:}\bar{\sigma}A \blacktriangleright x{:}\bar{\sigma}B$ is valid.

   $infer(\Delta \blacktriangleright *{:}B)$ returns $\sigma = unify(B, \top)$. Since $\bar{\sigma}\top = \top$, $\bar{\sigma}B = \top$. Hence $\bar{\sigma}\Delta \blacktriangleright *{:}\bar{\sigma}B$ is valid.

   $infer(\Delta \blacktriangleright c{:}B)$ returns $\sigma = unify(B, A_c)$. For all $c \in \mathcal{V}_{term}$, $\bar{\tau}A_c = A_c$ for any type substitution τ. Thus $\bar{\sigma}B = A_c$, and $\bar{\sigma}\Delta \blacktriangleright c{:}\bar{\sigma}B$ is valid.

   $infer(\Delta \blacktriangleright MN{:}B)$ returns $\bar{\tau} \circ \sigma$, with

   $$\begin{aligned} \sigma &= infer(\Delta \blacktriangleright M : X \Rightarrow B), \\ \tau &= infer(\bar{\sigma}\Delta \blacktriangleright N : \bar{\sigma}X) \text{ and} \end{aligned}$$

   X a fresh variable.

   By induction hypothesis, σ and τ are such that $\bar{\sigma}\Delta \blacktriangleright M : \bar{\sigma}X \Rightarrow \bar{\sigma}B$ and $\bar{\tau}\bar{\sigma}\Delta \blacktriangleright N : \bar{\tau}\bar{\sigma}X$ are valid. $\bar{\sigma}\Delta \blacktriangleright M : \bar{\sigma}X \Rightarrow \bar{\sigma}B)$ is valid then by

$$
\begin{array}{rcl}
\mathit{infer}(\Delta, x{:}A \blacktriangleright x{:}B) & = & \mathit{unify}(A, B) \\
\mathit{infer}(\Delta \blacktriangleright *{:}B) & = & \mathit{unify}(B, \top) \\
\mathit{infer}(\Delta \blacktriangleright c{:}B) & = & \mathit{unify}(B, A_c) \\
\mathit{infer}(\Delta \blacktriangleright MN{:}B) & = & \bar{\tau} \circ \sigma \\
 & & \sigma = \mathit{infer}(\Delta \blacktriangleright M : X \Rightarrow B) \\
 & & \tau = \mathit{infer}(\bar{\sigma}\Delta \blacktriangleright N : \bar{\sigma}X) \\
 & & X \text{ fresh variable} \\
\mathit{infer}(\Delta \blacktriangleright \lambda x.M{:}B) & = & \bar{\tau} \circ \sigma \\
 & & \sigma = \mathit{unify}(B, X \Rightarrow Y) \\
 & & \tau = \mathit{infer}(\bar{\sigma}\Delta, x{:}\bar{\sigma}X \blacktriangleright M{:}\bar{\sigma}Y) \\
 & & X, Y \text{ fresh variables} \\
\mathit{infer}(\Delta \blacktriangleright \langle M, N\rangle{:}B) & = & \bar{\tau} \circ \bar{\rho} \circ \sigma \\
 & & \sigma = \mathit{unify}(B, X \times Y) \\
 & & \rho = \mathit{infer}(\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\sigma}X) \\
 & & \tau = \mathit{infer}(\bar{\rho}\bar{\sigma}\Delta \blacktriangleright N{:}\bar{\rho}\bar{\sigma}Y) \\
 & & X, Y \text{ fresh variables} \\
\mathit{infer}(\Delta \blacktriangleright \mathit{if}\,(P; M; N){:}B) & = & \bar{\eta} \circ \bar{\tau} \circ \bar{\rho} \circ \sigma \\
 & & \sigma = \mathit{infer}(\Delta \blacktriangleright P{:}Y) \\
 & & \rho = \mathit{unify}(\sigma Y, \mathit{bit}) \\
 & & \tau = \mathit{infer}(\bar{\rho}\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\rho}\bar{\sigma}B) \\
 & & \eta = \mathit{infer}(\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright N{:}\bar{\tau}\bar{\rho}\bar{\sigma}B) \\
\mathit{infer}(\Delta \blacktriangleright \mathit{let}\ \langle x, y\rangle{=}M\ \mathit{in}\ N{:}A) & = & \bar{\rho} \circ \sigma \\
 & & \sigma = \mathit{infer}(\Delta \blacktriangleright M{:}X_1 \times X_2) \\
 & & \rho = \mathit{infer}(\bar{\sigma}\Delta, x{:}\bar{\sigma}X_1, y{:}\bar{\sigma}X_2 \blacktriangleright N{:}\bar{\sigma}A) \\
 & & X_1, X_2 \text{ fresh variables}
\end{array}
$$

Table 7: Type inference algorithm for the simply-typed lambda-calculus

Lemma 3.2.8 $\bar{\tau}\bar{\sigma}\Delta \blacktriangleright M : \bar{\tau}\bar{\sigma}X \Rightarrow \bar{\tau}\bar{\sigma}B$ is valid. Applying (app),

$$\bar{\tau}\bar{\sigma}\Delta \blacktriangleright MN : \bar{\tau}\bar{\sigma}B$$

is valid.

$infer(\Delta \blacktriangleright \lambda x.M{:}B)$ returns $\bar{\tau} \circ \sigma$, with $\sigma = unify(B, X \Rightarrow Y)$,

$$\tau = infer(\bar{\sigma}\Delta, x{:}\bar{\sigma}X \blacktriangleright M{:}\bar{\sigma}Y)$$

and X,Y fresh variables. By induction hypothesis, σ and τ are such that $\bar{\sigma}(B) = \bar{\sigma}(X) \Rightarrow \bar{\sigma}(Y)$ and

$$\bar{\tau}\bar{\sigma}\Delta, x{:}\bar{\tau}\bar{\sigma}X \blacktriangleright M{:}\bar{\tau}\bar{\sigma}Y$$

is valid. Applying (λ) and since $\bar{\tau}\bar{\sigma}(B) = \bar{\tau}\bar{\sigma}(X) \Rightarrow \bar{\tau}\bar{\sigma}(Y)$,

$$\bar{\tau}\bar{\sigma}\Delta \blacktriangleright \lambda x.M{:}\bar{\tau}\bar{\sigma}B$$

is valid.

$infer(\Delta \blacktriangleright \langle M, N\rangle{:}B)$ returns $\bar{\tau} \circ \bar{\rho} \circ \sigma$, with

$$\begin{aligned} \sigma &= unify(B, X \times Y), \\ \rho &= infer(\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\sigma}X), \\ \tau &= infer(\bar{\rho}\bar{\sigma}\Delta \blacktriangleright N{:}\bar{\rho}\bar{\sigma}Y) \text{ and} \end{aligned}$$

X, Y fresh variables.

From induction hypothesis,

$$\bar{\rho}\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\rho}\bar{\sigma}X$$

and

$$\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright N{:}\bar{\tau}\bar{\rho}\bar{\sigma}Y$$

are valid, and $\bar{\sigma}B = \bar{\sigma}X \times \bar{\sigma}Y$. Thus

$$\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\tau}\bar{\rho}\bar{\sigma}X$$

is valid using Lemma 3.2.8, and $\bar{\tau}\bar{\rho}\bar{\sigma}B = \bar{\tau}\bar{\rho}\bar{\sigma}X \times \bar{\tau}\bar{\rho}\bar{\sigma}Y$. Thus, applying $(\times)$,

$$\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright \langle M, N\rangle{:}\bar{\tau}\bar{\rho}\bar{\sigma}B$$

is valid.

$infer(\Delta \blacktriangleright if\,(P; M; N){:}B)$ returns $\bar{\eta} \circ \bar{\tau} \circ \bar{\rho} \circ \sigma$, with

$$\begin{aligned}
\sigma &= infer(\Delta \blacktriangleright P{:}Y),\\
\rho &= unify(\sigma Y, bit),\\
\tau &= infer(\bar{\rho}\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\rho}\bar{\sigma}B) \text{ and}\\
\eta &= infer(\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright N{:}\bar{\tau}\bar{\rho}\bar{\sigma}B).
\end{aligned}$$

So by induction hypothesis,

$$\begin{aligned}
&\bar{\sigma}\Delta \blacktriangleright P{:}\bar{\sigma}Y,\\
&\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\tau}\bar{\rho}\bar{\sigma}B \text{ and}\\
&\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright N{:}\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}B
\end{aligned}$$

are valid, with $\bar{\rho}\bar{\sigma}Y = bit$. Applying Lemma 3.2.8,

$$\begin{aligned}
&\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright P{:}\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}Y \text{ and}\\
&\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}B
\end{aligned}$$

are valid, and $\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}Y = bit$. Applying (if),

$$\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}\Delta \blacktriangleright if\,(P; M; N){:}\bar{\eta}\bar{\tau}\bar{\rho}\bar{\sigma}B$$

is valid.

$infer(\Delta \blacktriangleright let\ \langle x, y\rangle{=}M\ in\ N{:}A)$ returns $\bar{\rho} \circ \sigma$, with

$$\begin{aligned}
\sigma &= infer(\Delta \blacktriangleright M{:}X_1 \times X_2)\\
\rho &= infer(\bar{\sigma}\Delta, x{:}\bar{\sigma}X_1, y{:}\bar{\sigma}X_2 \blacktriangleright N{:}\bar{\sigma}A).
\end{aligned}$$

So by induction hypothesis,

$$\begin{aligned}
&\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\sigma}X_1 \times \bar{\sigma}X_2 \text{ and}\\
&\bar{\rho}\bar{\sigma}\Delta, x{:}\bar{\rho}\bar{\sigma}X_1, y{:}\bar{\rho}\bar{\sigma}X_2 \blacktriangleright N{:}\bar{\rho}\bar{\sigma}A
\end{aligned}$$

are valid. From Lemma 3.2.8,

$$\bar{\rho}\bar{\sigma}\Delta \blacktriangleright M{:}\bar{\rho}\bar{\sigma}X_1 \times \bar{\rho}\bar{\sigma}X_2$$

is also valid. Applying (let),

$$\bar{\rho}\bar{\sigma}\Delta \blacktriangleright let\ \langle x, y\rangle{=}M\ in\ N{:}\bar{\rho}\bar{\sigma}A$$

is valid.

2. If there exists a substitution ρ such that

$$\bar{\rho}\Delta \blacktriangleright M{:}\bar{\rho}A$$

is valid, then $\sigma = \mathit{infer}(\Delta \blacktriangleright M{:}A)$ exists and $\bar{\rho} \circ \sigma = \rho$, proved by structural induction on M.

(c) Since A_c does not contain any type variables and since $\bar{\rho}\Delta \blacktriangleright c{:}\bar{\rho}A$ is equal to $\bar{\rho}\Delta, x{:}A_c \blacktriangleright c{:}A_c$, we have the equality $\rho A = A_c$, or $A = A_c$. Since $\mathit{unify}(A_c, A_c) = \mathit{id}$, σ exists and is equal to id. In particular, $\bar{\rho} \circ \sigma = \rho$.

($\top$) This case is done similarly, replacing A_c with $\top$.

(x) The typing judgment $\bar{\rho}\Delta, x{:}\bar{\rho}A \blacktriangleright x{:}\bar{\rho}B$ is valid, then $\bar{\rho}A = \bar{\rho}B$. In particular, ρ is a unifier for A and B. Then a most general unifier can be found from Lemma 3.2.9, and it is given by $\mathit{unify}(A, B)$. From the definition,

$$\mathit{infer}(\Delta, x{:}A \blacktriangleright x{:}B) = \mathit{unify}(A, B).$$

Hence it exists, and from the property of unify, $\bar{\rho} \circ \sigma = \rho$.

(app) If

$$\bar{\rho}\Delta_1, \bar{\rho}\Delta_2, \bar{\rho}!\Gamma \blacktriangleright NP{:}\bar{\rho}A$$

is valid, then from the rule (app),

$$\bar{\rho}\Delta_1, \bar{\rho}!\Gamma \blacktriangleright N{:}B \Rightarrow \bar{\rho}A$$

$$\bar{\rho}\Delta_1, \bar{\rho}!\Gamma \blacktriangleright P{:}B$$

are valid. In particular, given a fresh type variable Z, the substitution $\rho' = \rho \cup \{X \mapsto B\}$ is such that

$$\bar{\rho'}\Delta_1, \bar{\rho'}!\Gamma \blacktriangleright N{:}\bar{\rho'}(Z \Rightarrow A)$$

$$\bar{\rho'}\Delta_1, \bar{\rho'}!\Gamma \blacktriangleright P{:}\bar{\rho'}Z$$

are valid. By induction hypothesis, the inference algorithm succeeds on $\Delta_1, !\Gamma \blacktriangleright N{:}Z \Rightarrow A$, and returns a substitution σ_1 such that $\bar{\rho'} \circ \sigma_1 = \rho'$. Since

$$\bar{\rho'}\Delta_1, \bar{\rho'}!\Gamma \blacktriangleright P{:}\bar{\rho'}Z$$

is equal to

$$\bar{\rho'} \circ \sigma_1 \Delta_1, \bar{\rho'} \circ \sigma_1 !\Gamma \blacktriangleright P{:}\bar{\rho'} \circ \sigma_1 Z,$$

by induction hypothesis, the inference algorithm succeeds also on

$$\bar{\sigma_1} \Delta_1, \bar{\sigma_1} !\Gamma \blacktriangleright P{:}\bar{\sigma_1} Z,$$

and returns a substitution σ_2 such that $\bar{\rho'} \circ \bar{\sigma_1} \circ \sigma_2 = \bar{\rho'} \circ \sigma_1 = \rho'$. The substitution $\mathit{infer}(\Delta_1, \Delta_2, !\Gamma \blacktriangleright NP{:}A)$ is defined to be $\bar{\sigma_1} \circ \sigma_2$.

The proof is similar for the remaining cases

□

Theorem 3.2.11 (Type inference algorithm) *A given term M is typable if and only if*

$$\sigma = \mathit{infer}(x_1{:}X_1, \ldots x_n{:}X_n \blacktriangleright M : Y)$$

with $FV(M) = \{x_1 \ldots x_n\}$ doesn't fail. Moreover, a principal typing judgment for M is

$$x_1{:}\sigma X_1, \ldots x_n{:}\sigma X_n \blacktriangleright M : \sigma Y$$

Proof.

If M is typable and

$$x_1{:}A_1, \ldots x_n{:}A_n \blacktriangleright M : B$$

is a valid typing judgment, then

$$\rho = \{X_1 \mapsto A_1, \ldots X_n \mapsto A_n, Y \mapsto B\}$$

is a unifier for

$$x_1{:}X_1, \ldots x_n{:}X_n \blacktriangleright M : Y$$

hence, *infer* won't fail, and will return a most general unifier, from Lemma 3.2.10. On the converse, if the algorithm does not fail, it gives back a principal unifier, so the term is typable. □

Chapter 4

Linear Logic

The type system used in Chapter 3 is based on intuitionistic logic. Linear logic was introduced by Girard [9] as a resource sensitive logic. One of the basic rules of ordinary logic is the *contraction rule* which states that given a valid proposition A one can deduce $A \wedge A$. This rule can be viewed as a *duplication* of the formula (or of the resource) A. In linear logic, the duplication of resource is in general not allowed, and the contraction rule is dropped. Formulas for which duplication is allowed are explicitly written as $!A$.

In intuitionistic logic, there are two ways for introducing a conjunction:

$$\frac{\Delta \blacktriangleright A \quad \Delta \blacktriangleright B}{\Delta \blacktriangleright A \wedge B} \qquad \frac{\Delta \blacktriangleright A \quad \Gamma \blacktriangleright B}{\Gamma, \Delta \blacktriangleright A \wedge B}$$

They are in fact different: the first one is a superposition, and the second one is a juxtaposition. In intuitionistic linear logic they yield two different conjunctions: the first is called the additive conjunction and is written $\&$, the second is called the multiplicative conjunction and is written $\otimes$. $\oplus$ is the additive disjunction, and $\multimap$ is the linear implication:

$$0 \quad \text{for} \quad \oplus, \qquad \top \quad \text{for} \quad \&, \qquad 1 \quad \text{for} \quad \otimes.$$

For a more complete discussion, see [11].

More formally, a formula in intuitionistic linear logic is defined by the following

abstract syntax:

$$\begin{array}{rcl} A, B, C & ::= & !A \\ & | & (A \otimes B) \\ & | & (A \multimap B) \\ & | & (A \,\&\, B) \\ & | & (A \oplus B) \\ & | & 0 \mid 1 \mid \top. \end{array}$$

A sequent in intuitionistic linear logic is a pair

$$\Delta \rhd A$$

where Δ is a set of intuitionistic linear logic formulas and A is an intuitionistic linear logic formula.

The rules are found in Table 8. Note the absence of structural rules of weakening and contraction.

To be able to manipulate duplicable elements in linear logic, a special unary connective is provided. We denote $!A$ a term on which one can apply weakening and contraction. We say "bang A" for $!A$. The rules we need to add are in Table 9

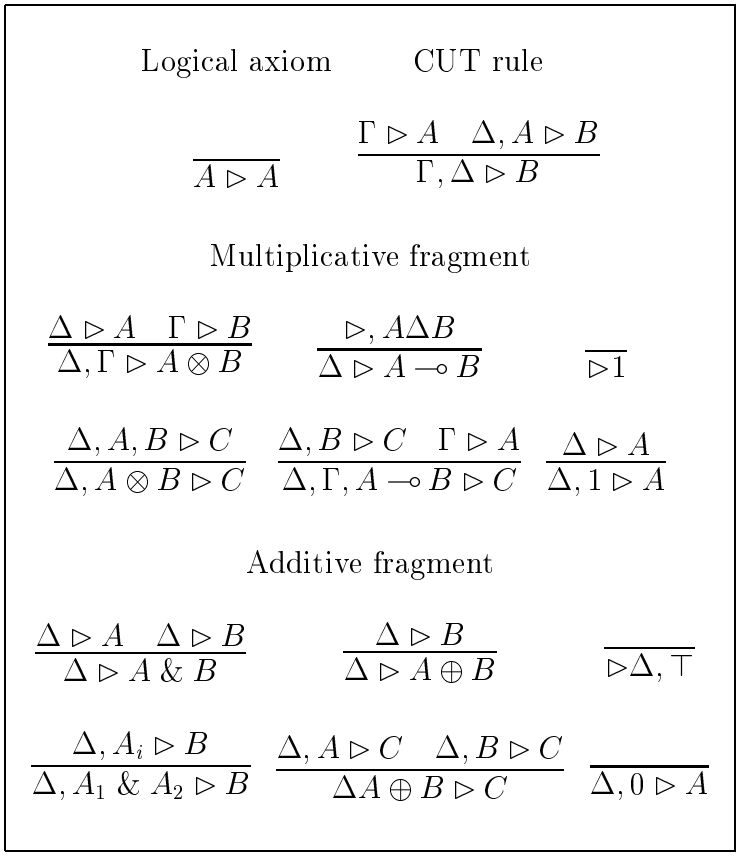
A sequent $A_1, \ldots, A_n \rhd B$ in intuitionistic linear logic can be interpreted as a rule for transforming resources $A_1 \ldots A_n$ into a resource B. The point is that $A_1 \ldots A_n$ are used up in this process, and cannot in general be used more than once.

A good example is the example of the restaurant, inspired by Girard and Lafont [10]. Consider the following menu:

fruit or seafood (in season)
main course
all the chips you can eat
tea or coffee

You have two kinds of choice: the choice between fruit and seafood is made for you, depending on what is available, and the choice of tea or coffee is let to you. Only one main course will be brought to you, but you can eat as many chips as you want. This menu translates into:

$$(\text{fruit} \oplus \text{seafood}) \otimes \text{main} \otimes !\text{chips} \otimes (\text{tea} \,\&\, \text{coffee})$$



$$\frac{}{A \rhd A} \qquad \frac{\Gamma \rhd A \quad \Delta, A \rhd B}{\Gamma, \Delta \rhd B}$$

$$\frac{\Delta \rhd A \quad \Gamma \rhd B}{\Delta, \Gamma \rhd A \otimes B} \qquad \frac{\rhd, A \Delta B}{\Delta \rhd A \multimap B} \qquad \frac{}{\rhd 1}$$

$$\frac{\Delta, A, B \rhd C}{\Delta, A \otimes B \rhd C} \qquad \frac{\Delta, B \rhd C \quad \Gamma \rhd A}{\Delta, \Gamma, A \multimap B \rhd C} \qquad \frac{\Delta \rhd A}{\Delta, 1 \rhd A}$$

$$\frac{\Delta \rhd A \quad \Delta \rhd B}{\Delta \rhd A \,\&\, B} \qquad \frac{\Delta \rhd B}{\Delta \rhd A \oplus B} \qquad \frac{}{\rhd \Delta, \top}$$

$$\frac{\Delta, A_i \rhd B}{\Delta, A_1 \,\&\, A_2 \rhd B} \qquad \frac{\Delta, A \rhd C \quad \Delta, B \rhd C}{\Delta A \oplus B \rhd C} \qquad \frac{}{\Delta, 0 \rhd A}$$

Table 8: Derivation rules for intuitionistic linear logic

dereliction

$$\frac{\Delta, A \rhd B}{\Delta, !A \rhd B}\ (D)$$

weakening

$$\frac{\Delta \rhd B}{\Delta, !A \rhd B}\ (W)$$

contraction

$$\frac{\Delta, !A, !A \rhd B}{\Delta, !A \rhd B}\ (C)$$

$$\frac{\Delta, !A \rhd B}{\Delta, !A \rhd !B}\ (!)$$

Table 9: Derivation rules for exponential

In intuitionistic linear logic, if a sequent $\Delta \rhd A$ is provable then *all* the resources in Δ are used. This might be too strong: If the resources represents variables in a program, one might want not to use all of them. We need a weaker logic: We will replace the axiom rule

$$A \rhd A$$

with

$$\Delta, A \rhd A$$

In other words, a element can be discarded even if it is not of the form $!A$. The logic becomes the *affine intuitionistic linear logic*, or AILL. This fits better our needs as computer scientists. Indeed we want to be able to create a function that will not use its argument. This fragment is therefore the one on which the type system of the language we develop is based: we are able to state whether or not an element can be duplicated, but we may forget any variable we wish to.

Chapter 5

The quantum lambda-calculus: Terms

5.1 Quantum States

We now turn to the question of defining a lambda-calculus for quantum computation with classical control.

We would like to extend the lambda calculus with the ability to manipulate quantum data. We first need a syntax to express quantum states in the lambda calculus. In simple cases, we might simply insert quantum states into a lambda term, such as

$$\lambda x.(\alpha|0\rangle + \beta|1\rangle).$$

However, in the general case, such a syntax is insufficient. Consider for instance the lambda term

$$(\lambda y.\lambda f.fpy)(q),$$

where p and q are quantum bits which are jointly in the entangled state $|pq\rangle = \alpha|00\rangle + \beta|11\rangle$. Such a state cannot be represented locally by replacing p and q with some constant expressions of type qubit. The non-local nature of quantum states thus forces us to introduce a level of indirection into the representation of a state. Thus, to represent a program, we should have a lambda-term M to encode the operations, but also an exterior n-qubit state Q to store the quantum data of the program. Further,

to link both parts, we need a third element, which is a function L from $FV(M)$ to $\{0, \ldots, n-1\}$, such that if $L(x) = i$, the variable x represents the i-th qubit in Q.

We also provide several built-in operations for quantum bits. The operator *new* represents a function that takes a bit (0 or 1) and allocates a new qubit of the corresponding value. We also need to be able to act on qubits via unitary operations; thus, we will assume a given set $\mathcal{U}^1$ of unitary gates. For simplicity we first consider our language without tuples so we will restrict ourselves to unary quantum gates for now; tuples and n-ary gates will be considered in Chapter 8.

In the following examples, we will often use the Hadamard gate H, which we assume to be an element of $\mathcal{U}^1$:

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

Finally, we equip the language with an operation *meas*, which takes a quantum bit, performs a measurement, and returns the classical bit 0 or 1 which is the result of the measurement. Of course, the outcome of this operation is probabilistic. If U ranges over $\mathcal{U}^1$ and x over $\mathcal{V}_{term}$, we define a *term* by the following:

$$\begin{array}{llll} RawTerm & M, N, P & ::= & x \\ & & | & MN \\ & & | & \lambda x.M \\ & & | & if(M; N; P) \\ & & | & 0 \mid 1 \\ & & | & meas \\ & & | & new \\ & & | & U \end{array}$$

Note that compare to the lambda-calculus from Chapter 3, we have removed pairing, unit and the *let* operator. These will be re-introduced in Chapter 8.

As usual, terms are identified up to α-equivalence. In that sense we will write $\lambda x.x = \lambda y.y$.

Definition 5.1.1 A *quantum state* is a triple

$$[Q, L, M]$$

where

- Q is a normalized vector of $\otimes_{i=0}^{n-1}\mathbb{C}^2$, for some $n \geq 0$
- M is a lambda-term,
- L is a function from W to $\{0, \ldots, n-1\}$, where $FV(M) \subseteq W \subseteq \mathcal{V}_{term}$. L is also called the *linking function*.

We denote the set of quantum states by $\mathbb{S}$. If $n = 0$, then we denote the trivial state vector $Q = 1 \in \mathbb{C}$ by $Q = |\rangle$.

A useful subset of $\mathbb{S}$ is the subspace $\mathbb{V}$ of *value states*:

$$\mathbb{V} = \{ \quad [Q, L, V] \in \mathbb{S} \quad | \quad V \text{ is a value} \quad \}$$

Here, a value is defined to be a constant, a variable or a lambda-abstraction as in Chapter 3.

The notion of α-equivalence extends naturally to quantum states, for instance, the states

$$[|1\rangle, \{x \mapsto 0\}, \lambda y.x] \text{ and } [|1\rangle, \{z \mapsto 0\}, \lambda y.z]$$

are equivalent. More formally, the *α-equivalence* on quantum states is the smallest equivalence relation such that if $x \in FV(M)$ and $z \notin FV(M)$, then

$$[Q, L \cup \{x \mapsto i\}, M] =_\alpha [Q, L \cup \{z \mapsto i\}, M[z/x]].$$

We will work under this equivalence when speaking of quantum states.

Convention 5.1.2

In order to simplify the notation, we will often use the following trick: we use p_i to denote the free variable x such that $L(x) = i$. A quantum state is abbreviated by

$$[Q, M']$$

with $M' = M[p_{i_1}/x_1] \ldots [p_{i_n}/x_n]$ if the domain of L is $\{x_1, \ldots, x_n\}$, where $i_k = L(x_k)$.

Reduction of the quantum state. We should now address the question of how a quantum state should be reduced. One restriction is that it is forbidden to duplicate a quantum bit, due to the no-cloning property of quantum physics. Let us illustrate this with an example, using a call-by-value reduction procedure. Let us define a binary **and** operation in our language: $\mathbf{and} = \lambda xy.\, if\,(x; if\,(y; 1; 0); 0)$. Now consider the following term:

$$(\lambda x.\mathbf{and}(meas(x)(meas(H\ x))))\ \ (|0\rangle).$$

Naïvely, we expect this to reduce to

$$\mathbf{and}(meas(|0\rangle))(meas(H\ |0\rangle)),$$

then to measure the right argument $H\ |0\rangle$, then the left argument which reduces to 0 with probability 1, and then apply the *and* function. We expect to obtain the result 0 with probability 1. Using the quantum state notation, let us reduce this term more formally:

$$[|0\rangle, (\lambda x.\mathbf{and}(meas(x))(meas(H\ x))\ \ (p_0)]$$

$$\longrightarrow_{CBV} [|0\rangle, \mathbf{and}(meas(p_0))(meas(H\ p_0))]$$

In the QRAM, applying H to a qubit is modifying the actual state of the qubit. Let us reduce the right argument $(H\ p_0)$:

$$\longrightarrow_{CBV} [\frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right), \mathbf{and}(meas(p_0))(meas(p_0))].$$

Reducing the right argument again, we obtain 1 with probability 0.5, assuming that the measurement is non-destructive. (Indeed, if we used destructive measurement, the program would not even be well-defined, since we would have a p_0 alone):

$$\longrightarrow_{CBV} [|0\rangle, \mathbf{and}(meas(p_0))(0)].$$

and with probability 0.5:

$$\longrightarrow_{CBV} [|1\rangle, \mathbf{and}(meas(p_0))(1)].$$

This reduces to $[|0\rangle, 0]$ with probability 0.5 and to $[|1\rangle, 1]$ with probability 0.5. Clearly, this is not the intended result.

The program is unpredictable due to the duplication of p_0. The problem derives from the fact that a value such as p_0 does not represent a constant, as in the classical lambda calculus, but rather it is a *pointer* into the quantum state. We never act *on* p_0, we act on the value it points to. To ensure the predictability of programs, it is necessary to disallow the duplication of terms that contain p_i's.

We will call an abstraction $\lambda x.M$ *linear* if x appears at most once as a free variable in M. We also say that M is *linear in* x in this case.

Another problem can occur: let us call **plus** the function which acts as the addition modulo 2 on classical bits. We can easily construct such a function in our language:

$$\mathbf{plus} = \lambda xy.\, \mathit{if}(x; \mathit{if}(y; 0; 1); \mathit{if}(y; 1; 0))$$

Consider the state

$$[\,|\rangle, (\lambda x.\mathbf{plus}\ x\ x)(\mathit{meas}(H(\mathit{new}\ \ 0)))]$$

Now reduce this state using call-by-value reduction. Intuitively this shall reduce to:

$$\longrightarrow_{CBV} [|0\rangle, (\lambda x.\mathbf{plus}\ x\ x)(\mathit{meas}(H\ p_0))]$$

$$\longrightarrow_{CBV} [\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), (\lambda x.\mathbf{plus}\ x\ x)(\mathit{meas}\ \ p_0)]$$

and then with probability 0.5:

$$[\,|\rangle, (\lambda x.\mathbf{plus}\ x\ x)(0)] \quad \text{or} \quad [\,|\rangle, (\lambda x.\mathbf{plus}\ x\ x)(1)]$$

$$[\,|\rangle, \mathbf{plus}\ 0\ 0] \quad \text{or} \quad [\,|\rangle, \mathbf{plus}\ 1\ 1]$$

which evaluate both with probability 1 to $[\,|\rangle, 0\,]$

Had we reduced the same term under a call-by-name strategy, we would have obtained in the first step

$$[\,|\rangle, \mathbf{plus}\ (\mathit{meas}(H(\mathit{new}\ \ 0)))\ (\mathit{meas}(H(\mathit{new}\ \ 0))))],$$

and then $[\,|\rangle, 0\,]$ with probability 0.5 and $[\,|\rangle, 1\,]$ with probability 0.5.

Moreover, if we had mixed the call-by-value and call-by-name strategies, the program could have led to an ill-defined result: reducing by call-by-value until

$$[\frac{\sqrt{2}}{2}(|0\rangle + |1\rangle), (\lambda x.\mathbf{plus}\ x\ x)(meas\ \ p_0)]$$

and then changing to call-by-name, we would obtain in one step:

$$[\frac{\sqrt{2}}{2}(|0\rangle + |1\rangle), (\mathbf{plus}\ (meas\ \ p_0)\ (meas\ \ p_0)],$$

which is not a valid program since there are 2 occurrences of p_0.

In other words, it does not make sense to speak of a general β-reduction procedure for the whole quantum state. It is necessary to choose a reduction strategy before writing programs.

5.2 Probabilistic reduction systems

Definition 5.2.1 We define a *probabilistic reduction system* as a tuple $(X, U, R, prob)$ where X is a set of *states*, $U \subseteq X$ is a subset of *value states*, $R \subseteq (X \setminus U) \times X$ is a set of *reductions*, and $prob : R \to [0,1]$ is a *probability function*, where $[0,1]$ is the real unit interval. Moreover, we impose the following conditions:

- For any $x \in X$, $R_x = \{\ x' \mid (x,x') \in R\ \}$ is finite.
- $\sum_{x' \in R_x} prob(x,x') \leq 1$

We call *prob* the one-step reduction, and we use the following notation:

$$x \longrightarrow_p y \quad \text{when} \quad prob(x,y) = p$$

Let us extend *prob* to the n-step reduction:

$$\begin{aligned} prob^0(x,y) &= \begin{cases} 0 & \text{if} \quad x \neq y \\ 1 & \text{if} \quad x = y \end{cases} \\ prob^1(x,y) &= \begin{cases} prob(x,y) & \text{if} \quad (x,y) \in R \\ 0 & \text{else} \end{cases} \\ prob^{n+1}(x,y) &= \sum_{z \in R_x} prob(x,z) prob^n(z,y) \end{aligned}$$

We use the following notation:

$$x \longrightarrow_p^n y \quad \text{when} \quad prob^n(x,y) = p$$

We say that y is *reachable in one step with non-zero probability* from x, denoted $x \longrightarrow_{>0} y$ when $x \longrightarrow_p y$ with $p > 0$. We say that y is *reachable with non-zero probability* from x, denoted $x \longrightarrow^*_{>0} y$ when there exists n such that $x \longrightarrow_p^n y$ with $p > 0$.

We can then compute the probability to reach $u \in U$ from x: It is a function from $X \times U$ to $\mathbb{R}$ defined by:

$$prob_U(x,u) = \sum_{n=0}^{\infty} prob^n(x,u)$$

The total probability for reaching U from x is:

$$prob_U(x) = \sum_{n=0}^{\infty} \sum_{u \in U} prob^n(x,u)$$

On the other hand, there is also the probability to *diverge* from x, or never reaching anything. This value is:

$$prob_\infty(x) = \lim_{n \to \infty} \sum_{y \in X} prob^n(x,y)$$

Lemma 5.2.2 *For all $x \in X$, $prob_U(x) + prob_\infty(x) \leq 1$.*

We define the *error probability of x* to be the number

$$prob_{err}(x) = 1 - prob_U(x) - prob_\infty(x)$$

Definition 5.2.3 We can define a notion of equivalence in X:

$$x \approx y \quad \text{iff} \quad \forall u \in U \begin{cases} prob_U(x,u) = prob_U(y,u) \\ prob_\infty(x) = prob_\infty(y) \end{cases}$$

Definition 5.2.4 In addition to the notion of reachability with non-zero probability, there is also a weaker notion of reachability, given by R: We will say that y is *reachable* from x if xRy. By the properties of *prob*,

$$x \longrightarrow_{>0} y \quad \text{implies} \quad x \rightsquigarrow y$$

with $x \rightsquigarrow y$ for xRy. Let us denote by $\longrightarrow^*$ the relation such that

$$x \rightsquigarrow^* y \quad \text{iff} \quad \exists n \quad xR^n y$$

with R^n defined as the n-th composition of R. Similarly,

$$x \longrightarrow^*_{>0} y \quad \text{implies} \quad x \rightsquigarrow^* y$$

Consistent states and error-states. In a probabilistic reduction system, a state x is called an *error-state* if $x \notin U$ and

$$\sum_{x' \in X} prob(x, x') < 1$$

An element $x \in X$ is *consistent* if there is no error-state e such that $x \rightsquigarrow^* e$

Lemma 5.2.5 *If x is consistent, then $prob_{err}(x) = 0$.*

However, the converse is false: Define

- $X = \{0, 1, 2\}$
- $U = \{2\}$
- *prob* and R are defined by

$$\begin{aligned} &0R0 \quad \text{and} \quad 0 \longrightarrow_{0.5} 0 \\ &0R1 \quad \text{and} \quad 0 \longrightarrow_{0} 1 \\ &0R2 \quad \text{and} \quad 0 \longrightarrow_{0.5} 2 \end{aligned}$$

Here $(X, U, R, prob)$ is a probabilistic reduction system. 1 is an error state, so 0 is not consistent but $prob_{err}(x) = 0$.

Remark 5.2.6 We need the weaker notion of reachability $x \rightsquigarrow^* y$, in addition to reachability with non-zero probability $x \longrightarrow_{>0}{}^* y$, because a null probability of getting a certain result is not an absolute warranty of its impossibility. In the QRAM, suppose we have a qubit in state $|0\rangle$. Measuring it cannot theoretically yield the value 1, but in practice, this might happen with small probability, due to imprecision of the physical operations and decoherence. What will happen if we measure this qubit and get 1? We need to be sure that even in this case the program will not crash. Hence we separate in a sense the null probability of getting a certain result, and the computational impossibility.

5.3 Quantum reduction

We need a deterministic procedure to choose which redex to reduce. Let us analyze a call by value procedure, since this is the most intuitive procedure. Note that the reduction itself is probabilistic, but the choice of redex is deterministic.

Call-by-value reduction. We define a probabilistic call-by-value reduction procedure in Table 10. We write $M \longrightarrow_{CBV\,p} N$ if M reduces to N with probability p, or $M \longrightarrow_p N$ for short. As said before, the reduction in the classical part of the calculus is the usual one. Recall that we write $[Q, M']$ as an abbreviation for a quantum state $[Q, L, M]$ by Convention 5.1 on page 61.

Discussion. In the rule $(meas)$, if $Q = \alpha|Q_0\rangle + \beta|Q_1\rangle$ is normalized with

$$|Q_0\rangle = \textstyle\sum_i \alpha_i |\phi_i^0\rangle \otimes |0\rangle \otimes |\psi_i^0\rangle,$$
$$|Q_1\rangle = \textstyle\sum_i \beta_i |\phi_i^1\rangle \otimes |1\rangle \otimes |\psi_i^1\rangle,$$

and $|0\rangle$ and $|1\rangle$ being the i-th qubit, we write $\mu_0 = |\alpha|^2$ and $\mu_1 = |\beta|^2$. In the rule (new), Q is in a space of dimension 2^n. In the rule (U), if Q is in a space of dimension 2^n, let $Q' = (I_j \otimes U \otimes I_{n-j-1})(Q)$. In any case, V is a value.

A weaker relation. We define a weaker relation $\rightsquigarrow$. This relation models the transformations that can happen due to decoherence and imprecision of physical

$$\frac{}{[Q,(\lambda x.M)V] \longrightarrow_1 [Q,M[V/x]]}\ (\beta)$$

$$\frac{}{[\alpha|Q_0\rangle + \beta|Q_1\rangle, meas\ \ p_i] \longrightarrow_{\mu_0} [|Q_0\rangle, 0]}\ (meas)$$

$$\frac{}{[\alpha|Q_0\rangle + \beta|Q_1\rangle, meas\ \ p_i] \longrightarrow_{\mu_1} [|Q_1\rangle, 1]}\ (meas)$$

$$\frac{}{[Q, new\ \ 0] \longrightarrow_1 [Q \otimes |0\rangle, p_n]}\ (new_0)$$
$$\frac{}{[Q, new\ \ 1] \longrightarrow_1 [Q \otimes |1\rangle, p_n]}\ (new_1)$$

$$\frac{}{[Q, U\ p_j] \longrightarrow_1 [Q', p_j]}\ (U)$$

$$\frac{[Q,N] \longrightarrow_p [Q',N']}{[Q,MN] \longrightarrow_p [Q',MN']}\ (cong_1)$$
$$\frac{[Q,M] \longrightarrow_p [Q',M']}{[Q,MV] \longrightarrow_p [Q',M'V]}\ (cong_2)$$

$$\frac{}{[Q, if\,(0;M;N)] \longrightarrow_1 [Q,N]}\ (if_0)$$
$$\frac{}{[Q, if\,(1;M;N)] \longrightarrow_1 [Q,M]}\ (if_1)$$
$$\frac{[Q,P] \longrightarrow_p [Q',P']}{[Q, if\,(P;M;N)] \longrightarrow_p [Q', if\,(P';M;N)]}\ (\xi_{if})$$

Table 10: Quantum call-by-value reduction

operations. We define $[Q, M] \rightsquigarrow^* [Q', M']$ is $[Q, M] \longrightarrow^* p[Q', M']$, even when $p = 0$, plus the additional rule, if Q and Q' are in the same vector space:

$$[Q, M] \rightsquigarrow [Q', M]$$

Lemma 5.3.1 *Let prob be the function such that* $prob(x, y) = p$ *if* $x \longrightarrow_p y$ *and* 0 *else. If* $x, y \in \mathbb{S}$ $(\mathbb{S}, \mathbb{V}, \rightsquigarrow, prob)$ *is a probabilistic reduction system.* □

Evidently, this probabilistic reduction system has error states, for example,

$$[Q, H(\lambda x.x)].$$

Such error states correspond to run-time errors. In the next chapter, we introduce a type system designed to rule out such error states.

Chapter 6

The quantum lambda-calculus: Types

As we saw in Chapter 3, a type system is a powerful tool to prove the good behavior a program during the reduction. In our language, there are two class of expressions: Those which can be duplicated, such as for example $[|\rangle, \lambda x.x]$, and those who cannot, for example $[|0\rangle, p_0]$. A suitable type system would take this constraint in account. As seen in Chapter 4, the linear logic is a resource sensitive logic. Let us base our type system on this logic. A well-typed term $M{:}!A$ means that M can be duplicated. We need also some type constants. In Chapter 3 there was only one constant type needed, namely bit. In this language, we need bit, but also a type constant $qbit$ to be able to manipulate qubits.

A difference with the simply-typed lambda-calculus of Chapter 3 is the following: A well-typed term M of type $!A$ can be regarded as non-duplicable. In particular, if x is a duplicable variable, it can appear in a term which will use x only once. The notion we need to add is the notion of *subtyping*, noted $<:$. $A <: B$ means that if $M[x]$ is typable when x is of type A, then M is also typable when x is of type B.

$$\frac{}{\alpha <: \alpha}\ (ax) \qquad \frac{A <: B}{!A <: B}\ (D) \qquad \frac{}{X <: X}\ (var)$$

$$\frac{!A <: B}{!A <: !B}\ (!) \qquad \frac{A <: A' \quad B <: B'}{A' \multimap B <: A \multimap B'}\ (\multimap)$$

Table 11: Subtyping relation: First set of rules

6.1 Subtyping

Let us define a type system. We are going to define it together with an subtyping relation $<:$. We need constant types and types for abstractions (the functions). Moreover, we need a notion of duplicability of term. We want to be able to say whether or not a term can be duplicated. For this, we use the notation of linear logic. Let us define:

$$\begin{array}{rlcl} qType & A, B & ::= & \alpha \\ & & | & X \\ & & | & !A \\ & & | & (A \multimap B) \end{array}$$

where α ranges over a set of type constants, X ranges over a countable set of type variables, and $A \multimap B$ stand for "function with argument of type A which returns a result of type B". We want at least two type constants, namely *bit* and *qbit*. The notation "!" is a flag to state that the typed term is duplicable. We will call a type "exponential" if it is written "$!A$".

Notation. If $n \geq 0$, the notation $(n)(A)$ stands for

$$\underbrace{!!!\ldots!!}_{n \text{ times}} A$$

Let us define a subtyping relation $<:$ on this type system.

Lemma 6.1.1 *For any type A and B, if $A <: B$ and $(m = 0) \vee (n \geq 1)$, then $(n)(A) <: (m)(B)$.*

Proof. By induction on m:

- If $m = 0$: let us show by induction that for all n integer, $(n)(A) <: B$
  - If $n = 0$, by hypothesis $A <: B$.
  - If it is true for n, we have:

$$\frac{\begin{array}{c}\vdots\ (ind.hyp.)\\ (n)(A) <: B\end{array}}{(n+1)(A) <: B}\ (D)$$

- $m > 0$: $n \geq 1$ by hypothesis, and so:

$$\frac{\begin{array}{c}\vdots\ (ind.hyp.)\\ (n)(A) <: (m)(B)\end{array}}{(n)(A) <: (m+1)(B)}\ (!)$$

□

Notice that one can rewrite types using the notation:

$$\begin{array}{lrcl} qType & A, B & ::= & (n)(\alpha_i) \\ & & | & (n)(X), (n)(Y) \ldots \\ & & | & (n)(A \multimap B) \end{array}$$

with $n \in \mathbb{N}$.

The rules can be re-written:

The two sets of rules are equivalent.

Proof that rules on Table 12 implies rules on Table 11

(var) Follows directly from Lemma 6.1.1.

(α) Follows directly from Lemma 6.1.1.

$(\multimap_2)$ We know that $A <: A'$ and $B <: B'$. So by $(\multimap)$ we have $A' \multimap B <: A \multimap B'$. And by Lemma 6.1.1 we have obtained the desired result.

□

$$\frac{(m=0)\vee(n\geq 1)}{(n)(X)<:(m)(X)}\ (var_2)$$

$$\frac{\alpha_i\leq\alpha_j \quad (m=0)\vee(n\geq 1)}{(n)(\alpha_i)<:(m)(\alpha_j)}\ (\alpha)$$

$$\frac{A<:A' \quad B<:B' \quad (m=0)\vee(n\geq 1)}{(n)(A'\multimap B)<:(m)(A\multimap B')}\ (\multimap_2)$$

Table 12: Subtyping relation: Second set of rules

Proof that rules on Table 11 implies rules on Table 12 By induction on the proof that $A<:B$:

- If the last rule is (var) or (ax), then use it also in the new proof.
- If the last rule is $(\multimap)$, use $(\multimap_2)$, with $m=n=0$.
- If the last rule is $(!)$ or (D), then the proof will have a sequence of these two rules, up to either (var) or (ax), or $(\multimap)$.

  (var) $A=(n)(X)$ and $B=(m)(X)$ for X some type variable, and $m=0$ or $n\geq 1$. We can concatenate this sequence with the rule (var_2).

  (ax) $A=(n)(\alpha_i)$ and $B=(m)(\alpha_j)$ with $\alpha_i<:\alpha_j$ and $n\geq 1$. We can concatenate this sequence with the rule (α).

  $(\multimap)$ $A=(n)(A_1\multimap A_2)$ and $B=(m)(B_1\multimap B_2)$ with $A_2<:B_2$, $B_1<:A_1$ and $m=0$ or $n\geq 1$. We can concatenate this sequence with the rule $(\multimap_2)$.

□

Lemma 6.1.2 *$A<:B$ has a unique derivation within the rules from Table 12.* □

Lemma 6.1.3 *$(qType,<:)$ is reflexive and transitive.*

Proof. By induction using the rules from Table 12, and the transitivity of the implication in the equivalence:

$$(m = 0) \vee (n \geq 1) \quad \text{iff} \quad (m \geq 1) \Rightarrow (n \geq 1)$$

□

We can define an equivalence relation $\doteqdot$ by

$$A \doteqdot B \quad \text{iff} \quad (A <: B \quad \text{and} \quad B <: A)$$

Lemma 6.1.4 *$(qType/\doteqdot, <:)$ is a poset.* □

Lemma 6.1.5 *If $A<:!B$, then there exists C such that $A =!C$.*

Proof. Using the first set of rules, $A<:!B$ can only come from (D) or $(!)$. In both cases, A is of the form $!C$. □

6.2 Typing rules

We need to define what it means for a quantum state $[Q, L, M]$ to be typable. It turns out that the typing does not depend on Q and L, but only on M. Now, given a term M, we need to be able to say whether or not it is typable. As usual, we introduce typing judgments to deal with terms that may have free variables. Note that the free variables of M which are in the domain of L have to be of type *qbit*.

A *quantum typing judgment* is a tuple

$$\Delta \rhd M : B$$

where M is a term, B is a *qType*, and Δ is a typing context. As usual we denote Δ by $\{x_1 : A_1, \ldots x_n : A_n\}$, with $A_i = \Delta_f(x_i)$. If $\Delta = \{x_1{:}A_1, \ldots x_n{:}A_n\}$, we denote $!\Delta = \{x_1{:}!A_1, \ldots x_n{:}!A_n\}$.

For A and B in $qType$:

The axioms:
$$\frac{A <: B}{\Delta, x : A \rhd x : B}\ (ax_1)$$

For c a constant term,
$$\frac{A_c <: B}{\Delta \rhd c : B}\ (ax_2)$$

For the *if* term,
$$\frac{\Gamma_1, !\Delta \rhd P : bit \quad \Gamma_2, !\Delta \rhd M : A \quad \Gamma_2, !\Delta \rhd N : A}{\Gamma_1, \Gamma_2, !\Delta \rhd if(P; M; N) : A}\ (if)$$

The application:
$$\frac{\Gamma_1, !\Delta \rhd M : A \multimap B \quad \Gamma_2, !\Delta \rhd N : A}{\Gamma_1, \Gamma_2, !\Delta \rhd MN : B}\ (app)$$

The lambda, where $x \notin |\Delta|$:
$$\frac{x : A, \Delta \rhd M : B}{\Delta \rhd \lambda x.M : A \multimap B}\ (\lambda_1)$$

If $FV(M) \cap |\Gamma| = \emptyset$:
$$\frac{\Gamma, !\Delta, x : A \rhd M : B}{\Gamma, !\Delta \rhd \lambda x.M : (n+1)(A \multimap B)}\ (\lambda_2)$$

Table 13: Typing rules for the quantum lambda-calculus

Before we give the typing rules, we give the types for term constants Let us fix a type assignment $c \mapsto A_c$, from the set of constant terms to $qType$:

$$\begin{cases} 0 & \mapsto & !\,bit \\ 1 & \mapsto & !\,bit \\ new & \mapsto & !(bit \multimap qbit) \\ U & \mapsto & !(qbit \multimap qbit) \\ meas & \mapsto & !(qbit \multimap !\,bit) \end{cases}$$

Remark 6.2.1 we set $new :!(bit \multimap qbit)$. We could also have put $!bit$ in place of bit, since we want a bit to be always duplicable. However, this will be a corollary of the typing rules, and we therefore put the most general type for the constant.

The rules for constructing *valid quantum typing judgments* are shown in Table 13. We will say that a quantum state $[Q, L, M]$ is *typable* if there exists a type A such

that

$$x_1 \colon qbit, \ldots x_n \colon qbit \rhd M \colon A$$

is valid, with $\{x_1 \ldots x_n\}$ to be the domain of Q.

6.3 Examples

First let's illustrate the lambda-rules. Consider the following state:

$$[|\rangle, \lambda x.H(new\, x)].$$

This is a function fed with an argument x, supposed to be a bit, which returns a qubit equal to $H|x\rangle$. One can guess a type for the lambda-term:

$$bit \multimap qbit$$

If the term is well-typed, then the following typing judgment is derivable:

$$\rhd \lambda x.H(new\, x) \colon bit \multimap qbit\,.$$

Indeed, a typing derivation is:

$$\dfrac{\dfrac{\dfrac{!(qbit \multimap qbit) <: qbit \multimap qbit}{\rhd H \colon qbit \multimap qbit}\,(c) \qquad \dfrac{\dfrac{!(bit \multimap qbit) <: bit \multimap qbit}{\rhd new : bit \multimap qbit}\,(c) \quad \dfrac{bit <: bit}{x \colon bit \rhd x \colon bit}\,(x)}{x \colon bit \rhd (new\, x) \colon qbit}\,(app)}{x \colon bit \rhd H(new\, x) \colon qbit}\,(app)}{\rhd \lambda x.H(new\, x) \colon (bit \multimap qbit).}\,(\lambda_1)$$

Remark that in this example, the function is *linear* in x. Even if a bit is always duplicable, we don't need this feature in this term. This is expressed by the absence of exponential on the argument bit. Remark that $!(bit \multimap qbit)$ is also a valid type for this term: since the context is empty, one can apply the typing rule (λ_2). Indeed, we can duplicate as needed the function: it is already a value, and there is no reference to any pre-existing qubit.

However sometimes a function can be non-duplicable. Consider the quantum state:

$$[\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle), \{x \mapsto 0\}, \lambda y.x].$$

This is a well-defined quantum state, but the function is non-duplicable. The variable x free in the lambda-term is a pointer to the first qubit in the *QRAM*. This should disallow us to duplicate the term. Indeed, the typing judgment

$$x\colon qbit \rhd \lambda y.x\colon A \multimap qbit$$

is valid with typing derivation

$$\dfrac{\dfrac{}{x\colon qbit, y\colon A \rhd x\colon A \multimap qbit}\ (x)}{x\colon qbit \rhd \lambda y.x\colon A \multimap qbit,}\ (\lambda_1)$$

but

$$x\colon qbit \rhd \lambda y.x\colon !(A \multimap qbit)$$

is not: the variable x is free in the term but appear to be non-duplicable in the context: the rule (λ_2) cannot be applied.

Since the reduction strategy is call-by-value, a term is duplicable if and only if its value is duplicable. A term is always reduced to a value before any possible duplication. As an example, consider the state $[|\rangle, (new\ 1)]$. This state does not contain any non-duplicable element, but it reduces in one step to $[|1\rangle, p_0]$. And as a matter of fact, if it was duplicable, the typing tree would have been:

$$\dfrac{\dfrac{!(bit \multimap qbit) <\colon bit \multimap !\, qbit}{\rhd new : bit \multimap !\, qbit} \quad \dfrac{!\, bit <\colon bit}{\rhd 1\colon bit}}{\rhd new\ 1\colon !\, qbit\,.}$$

But $!(bit \multimap qbit) <\colon bit \multimap !\, qbit$ is not derivable, since $qbit$ is not a subtype of $!\, qbit$.

However, the state $[|0\rangle, \{x \mapsto 0\}, meas\ x]$ is duplicable, even if a qubit appears to be embedded inside the lambda-term. This state reduces in one step to $[|0\rangle, \{x \mapsto 0\}, 0]$, and $x\colon qbit \rhd 1\colon !\, bit$ is perfectly derivable, from the rule for constants.

Let's consider a higher-order term:

$$\lambda xy.x(xy).$$

This is a function of two arguments which is not linear in x. It can be typed in the following way:

$$\rhd \lambda xy.x(xy)\colon !(A \multimap A) \multimap !(A \multimap A).$$

The argument x of the function has to be duplicable. For example the term

$$(\lambda xy.x(xy))H$$

is typable. A valid typing judgment is

$$\triangleright(\lambda xy.x(xy))H :!(qbit \multimap qbit).$$

The typing judgment

$$x\colon qbit \triangleright(\lambda xy.x(xy))\lambda y.x\colon qbit \multimap qbit$$

is not, however, since $\lambda y.x$ is not duplicable.

Chapter 7

Properties of quantum typing judgments

7.1 Preliminary lemmas

Lemma 7.1.1 *If* $x \notin FV(M)$,

$$\Delta, x : A \rhd M : B \quad \textit{implies} \quad \Delta \rhd M : B.$$

Proof. We prove this by structural induction on the proof $\Delta, x{:}A \rhd M : B$, as we did it in Chapter 3, Lemma 3.2.1. □

Lemma 7.1.2 *If* A *is in* $qType$,

$$\Delta \rhd M : A \quad \textit{implies} \quad \Gamma, \Delta \rhd M : A.$$

Proof. By induction on the size on the proof of $\Delta \rhd M : A$. □

Definition 7.1.3 We extend the subtyping relation to contexts by:

$$\Delta <: \Delta' \quad \text{iff } |\Delta'| = |\Delta| \quad \text{and} \quad \forall x \in |\Delta'| \quad \Delta_f(x) <: \Delta'_f(x).$$

Note that this relation is reflexive and transitive.

Lemma 7.1.4 *If the typing judgement $\Delta \rhd N : A$ is valid and if $\Gamma <: \Delta$ and $A <: B$, then*

$$\Gamma \rhd N : B$$

is also valid.

Proof. By induction on the structure of N:

- If N is a constant term, we get the result by the axiom rule.
- If N is a variable x, then $\Delta_f(x) = A'$, with $A' <: A$. If $A <: B$, by transitivity, $A' <: B$ $\Gamma <: \Delta$ so since x belongs to $|\Delta|$, $x \in |\Gamma|$, and $\Gamma_f(x) <: A'$. By transitivity $\Gamma_f(x) <: B$ is true. Hence, by the (ax_1) rule,

  $$\Gamma \rhd x : B$$

  is verified.
- If $N = MP$, $\Delta \rhd N : A$ comes from

  $$\frac{\Delta'_1, !\Phi \rhd M : C \multimap A \quad \Delta'_2, !\Phi \rhd P : C}{\Delta'_1, \Delta'_2, !\Phi \rhd MP : A,} \ (app)$$

  with the split $\Delta = (\Delta'_1, \Delta'_2, !\Phi)$. Since $\Gamma <: \Delta$, Γ splits in $(\Gamma'_1, \Gamma'_2, !\Psi)$ such that

  $$\begin{aligned} \Gamma'_1 &<: \Delta'_1, \\ \Gamma'_2 &<: \Delta'_2, \\ \Psi &<: \Phi. \end{aligned}$$

  Since $A <: B$, $C \multimap A <: C \multimap B$. So by induction hypothesis:

  $$\begin{aligned} \Gamma'_1, !\Psi &\rhd M : C \multimap B \text{ and} \\ \Gamma'_2, !\Psi &\rhd P : C. \end{aligned}$$

  Applying (app) we get

  $$\Gamma'_1, \Gamma'_2, !\Psi \rhd MP : B,$$

  which is exactly

  $$\Gamma \rhd MP : B.$$

  And we get the result.

- If $N = if(M; P; Q)$, the idea is the same as for the product: we have to cut Δ and Γ in pieces and to apply the induction hypothesis. Then apply again the law (if).

- If $N = \lambda x.M$ then only 2 rules can apply: (λ_1) or (λ_2). In both cases, $A = (n)(C \multimap D)$. Since $A <: B$, from the reversibility of the set (2) of subtyping rules, B is of the form $(m)(E \multimap F)$, $m = 0$ or $n \geq 1$, $E <: C$ and $D <: F$. Let us study the 2 cases:

  (λ_1): $n = 0$, so $m = 0$, and the rule says:

  $$\frac{\Delta, x : C \rhd M : D}{\Delta \rhd \lambda x.M : C \multimap D.} \ (\lambda_1)$$

  Then since $\Gamma <: \Delta$ and $E <: C$,

  $$(\Gamma, x : E) <: (\Delta, x : C).$$

  By induction hypothesis we get

  $$\Gamma, x : E \rhd M : F.$$

  Applying (λ_1) we have the result.

  (λ_2): $n \geq 1$. The rule is:

  $$\frac{!\Delta_1, \Delta_2, x : C \rhd M : D}{!\Delta_1, \Delta_2 \rhd \lambda x.M : (n)(C \multimap D),} \ (\lambda_2)$$

  where $\Delta = (!\Delta_1, \Delta_2)$, and $|\Delta_2| \cap FV(M) = \emptyset$. Let us split Γ in (Γ_1, Γ_2), with $|\Gamma_1| = |\Delta_1|$ and $|\Gamma_2| = |\Delta_2|$. For all x in $|\Gamma_1|$, $\Gamma_{1f}(x) <: !\Delta_{1f}(x)$. From Lemma 6.1.5, $\Gamma_{1f}(x)$ is banged. Thus Γ_1 can be re-written as $!\Gamma_1$, and we have

  $$(!\Gamma_1, \Gamma_2, x : E) <: (!\Delta_1, \Delta_2, x : C),$$
  $$!\Delta_1, \Delta_2, x : C \rhd M : D,$$
  $$D <: F.$$

  Applying the induction hypothesis,

  $$!\Gamma_1, \Gamma_2, x : E \rhd M : F.$$

Since $|\Gamma_2| = |\Delta_2|$, $|\Gamma_2| \cap FV(M) = \emptyset$. So either (λ_2) or (λ_1) can apply. In either case,

$$!\Gamma_1, \Gamma_2, \triangleright \lambda x.M : (m)(E \multimap F).$$

□

Lemma 7.1.5 *If V is a value such that*

$$\vdash \Delta \triangleright V :!A$$

Then

$$\forall x \in FV(V) \quad \exists U \in qType \quad \Delta_f(x) =!U.$$

Proof.

- If V is a constant c: The term is closed, hence by vacuity we have the result.
- If $V = \lambda x.M$, the only rule that applies is (λ_2), and Δ splits into $(\Delta_1, !\Delta_2)$ with $FV(M) \cap |\Delta_1| = \emptyset$. So every free variable y except maybe x in M is exponential. Since $FV(\lambda x.M) = (FV(M) \setminus \{x\})$, the Lemma is also true in this case.

□

Lemma 7.1.6 *For A and B qType, and V a value, if $!\Delta, \Gamma_2, x : A \triangleright M : B$ and $!\Delta, \Gamma_1 \triangleright V : A$ are valid, then*

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

is valid.

Proof. Let ω be a proof for

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

We prove it by structural induction on ω. Let $!\Delta, \Gamma_1 \triangleright V : A$ be a valid typing judgment.

- If ω is an axiom, there are three cases.

1) We can have $M = y$, $y \neq x$. Then $y \in |!\Delta, \Gamma_2|$, with $(!\Delta, \Gamma_2)_f(y) = A'$. $A' <: B$ by the axiom rule. $y \in |\Gamma_1, \Gamma_2, !\Delta|$ then

$$\Gamma_1, \Gamma_2, !\Delta \rhd y : B$$

is a result of (ax_1). Since $M[V/x] = y$, the lemma is verified.

2) We can have $M = x$. Then $A <: B$ by the hypothesis of (ax_1). By Lemma 7.1.2, since $!\Delta, \Gamma_1 \rhd V : A$ we get that

$$\Gamma_1, \Gamma_2, !\Delta \rhd V : A.$$

By Lemma 7.1.4,

$$\Gamma_1, \Gamma_2, !\Delta \rhd V : B.$$

And since $M[V/x] = V$, the lemma is verified.

3) Finally, M can be a constant: $M = c$. So $A_c <: B$. (ax_2) says that

$$\Gamma_1, \Gamma_2, !\Delta \rhd c : B$$

is also true. Since $M[V/x] = c = M$. we have also the result.

- Else, if $M = \lambda y.P$. Since M is α-equivalent to $\lambda z.P[z/y]$, z a fresh variable, we can suppose without lost of generality that $y \neq x$, $y \notin |\Gamma_1|$, $y \notin |\Gamma_2|$ and $y \notin |\Delta|$. And so $M[V/x] = \lambda y.P[V/x]$. M is a lambda-abstraction, so the first rule to apply is:

$$\frac{\begin{array}{c}\vdots\ \tau \\ x : A, \Gamma_2, !\Delta, y : C \rhd P : B\end{array}}{x : A, \Gamma_2, !\Delta \rhd \lambda y.P : (n)(C \multimap B).}\ (\lambda_i)$$

for some n integer: if $n = 0$, we apply (λ_1), else we apply (λ_2)

$n = 0$) Then we apply (λ_1). By induction hypothesis, the lemma is true for τ. Then we have:

$$\Gamma_1, \Gamma_2, !\Delta, y : C \rhd P[V/x] : B.$$

And by applying the (λ_1) rule,

$$\Gamma_1, \Gamma_2, !\Delta \rhd \lambda y.P[V/x] : C \multimap B$$

and thus

$$\Gamma_1, \Gamma_2, !\Delta \rhd M[V/x] : C \multimap B,$$

and the lemma is verified

$n > 0$) Then we apply (λ_2): If x is a free variable of P, then A is exponential by the (λ_2) rule, and applying the induction hypothesis,

$$\Gamma_1, \Gamma_2, !\Delta, y : C \rhd P[V/x] : B$$

is valid. Let us write

$$\Phi = (\Gamma_1, \Gamma_2, !\Delta).$$

Since A is exponential, by Lemma 7.1.5, for all z in $FV(V)$, $\Phi_f(z)$ is exponential. By the (λ_2) rule, any free variable z of P is exponential. Since $FV(P[V/x]) = FV(V) \cup (FV(P) \setminus \{x\})$, one can split Φ into $(!\Phi_1, \Phi_2)$, with $|\Phi_1| = FV(P[V/x])$. Then the hypothesis for rule (λ_2) is verified, and we can apply it:

$$\Gamma_1, \Gamma_2, !\Delta, \rhd \lambda y.P[V/x] : (n)(C \multimap B),$$

and the lemma is verified.

If x is not a free variable of P, then the substitution let the term unchanged, and we only add to the context some variables that are not free in P using Lemma 7.1.2: we can still apply (λ_2), and get the result.

- or if $M = PR$.

  $(\Gamma_2, !\Delta, x : A)$ splits in $(\Gamma_{21}, \Gamma_{22}, !\Delta')$ with the rule:

$$\frac{\xi_1 \quad \xi_2}{\Gamma_{21}, \Gamma_{22}, !\Delta' \rhd PR : B,} \ (app)$$

  and

$$\begin{aligned} \xi_1 = & \Gamma_{21}, !\Delta' \rhd \begin{array}{c}\vdots\ \tau_1\end{array} P : C \multimap B, \\ \xi_2 = & \Gamma_{22}, !\Delta' \rhd \begin{array}{c}\vdots\ \tau_2\end{array} R : C. \end{aligned}$$

There are 3 cases:

1) x can be element of $|!\Delta'|$. The $A = !A'$ and $(x : !A')$ is both in ξ_1 and ξ_2. By induction hypothesis, if we split $!\Delta'$ in $(!\Delta'', x : !A')$, we can conclude that

$$\Gamma_1, \Gamma_{21}, !\Delta'' \rhd P[V/x] : C \multimap B \text{ and}$$

$$\Gamma_1, \Gamma_{22}, !\Delta'' \rhd R[V/x] : C.$$

A is exponential, so by Lemma 7.1.5, Γ_1 splits in 2 parts: $(!\Gamma_{11}, \Gamma_{12})$ with $FV(V) \cap |\Gamma_{12}| = \emptyset$. Since no free variable of PR is in $|\Gamma_1|$, and since $FV(P[V/x]) = (FV(P) \setminus \{x\}) \cup FV(V)$, we have

$$\begin{aligned} & FV(P[V/x]) \cap |\Gamma_{12}| \\ = \; & FV(R[V/x]) \cap |\Gamma_{12}| \\ = \; & \emptyset. \end{aligned}$$

By Lemma 7.1.1, one can then find a proof for

$$!\Gamma_{11}, \Gamma_{21}, !\Delta'' \rhd P[V/x] : C \multimap B$$

and

$$!\Gamma_{11}, \Gamma_{22}, !\Delta'' \rhd R[V/x] : C.$$

Then applying (app) we get

$$!\Gamma_{11}, \Gamma_{21}, \Gamma_{22}, !\Delta'' \rhd P[V/x]R[V/x] : B.$$

Applying Lemma 7.1.2, and since $(PR)[V/x] = P[V/x]R[V/x]$, we get:

$$\Gamma_1, \Gamma_{21}, \Gamma_{22}, !\Delta'' \rhd (PR)[V/x] : B.$$

And renaming the context, since

$$(\Gamma_2, !\Delta, x : A) = (\Gamma_{21}, \Gamma_{22}, !\Delta'', x : A),$$

we have what we want:

$$\Gamma_1, \Gamma_2, !\Delta \rhd (PR)[V/x] : B.$$

2) x can be element of $|\Gamma_{21}|$. That means that x is only free in P. In this case, $R[V/x] = R$. In this case, $(x : A)$ occurs only in ξ_1. We apply the induction hypothesis on τ_1 and get

$$\Gamma_1, \Gamma'_{21}, !\Delta' \rhd P[V/x] : C \multimap B,$$

where $\Gamma_{21} = (\Gamma'_{21}, x : A)$. Applying (app) we get the result.

3) If x is element of $|\Gamma_{22}|$, the process is the same as in the previous case: That means that x is only free variable of R. In this case, $P[V/x] = P$. In this case, $x : A$ occur only in (ξ_2). We apply the induction hypothesis on τ_2 and get

$$\Gamma'_{22}, !\Delta' \rhd R[V/x] : C,$$

where $\Gamma_{22} = (\Gamma'_{22}, x : A)$. Applying (app) we get the result.

- at last, if $M = if(P; N; R)$, we apply the same cases as above.

□

Corollary 7.1.7 *If*

- $\Gamma_1, !\Delta, x : A \rhd M : B$,
- $\Gamma_2, !\Delta \rhd V : (\tau)A$,

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

Proof. From Lemma 7.1.6 and Lemma 7.1.4. □

7.2 Subject reduction

Theorem 7.2.1 *If* $\Delta \rhd M{:}U$ *is valid and* $[Q, L, M] \rightsquigarrow^* [Q', L', M']$ *then* $\Delta' \rhd M'{:}U$ *is valid, where* $\Delta' = \Delta, x_1{:}\, qbit, \ldots x_n{:}\, qbit$ *and* $|L'| \setminus |L| = \{x_1, \ldots x_n\}$.

Proof We are going to restrict the study to call-by-value, it extends easily to $\rightsquigarrow$. Since it is a relation defined by induction, we prove it by induction on the derivation of the reduction.

- For the rule

$$[Q, (\lambda x.M)V] \longrightarrow_1 [Q, M[V/x]].$$

The typing judgment $\Phi \rhd (\lambda x.M)V : B$ is derived by the typing tree

$$\frac{!\Delta, \Gamma_1 \rhd V : A \quad \dfrac{!\Delta, \Gamma_2, x : A \rhd M : B}{!\Delta, \Gamma_2 \rhd \lambda x.M : A \multimap B}}{!\Delta, \Gamma_1, \Gamma_2 \rhd (\lambda x.M)V : B,}$$

when Φ splits into $(!\Delta, \Gamma_1, \Gamma_2)$. Using Lemma 7.1.7, since

$$!\Delta, \Gamma_1 \rhd V : A \quad \text{and} \quad !\Delta, \Gamma_2, x : A \rhd M : B,$$

the typing tree $!\Delta, \Gamma_1, \Gamma_2 \rhd M[V/x]{:}B$ is valid. the theorem is true, with $L = L'$.

- The rules for *meas* are

$$\frac{}{[\alpha|Q_0\rangle + \beta|Q_1\rangle, \mathit{meas}\ \ p_i] \longrightarrow_{\mu_0} [|Q_0\rangle, 0],}$$

$$\frac{}{[\alpha|Q_0\rangle + \beta|Q_1\rangle, \mathit{meas}\ \ p_i] \longrightarrow_{\mu_1} [|Q_1\rangle, 1].}$$

We study the first case, the second is similar. If $\Gamma, !\Delta, x : \mathit{qbit} \rhd \mathit{meas}\ \ x : B$ is valid it must come from:

$$\frac{\dfrac{!(\mathit{qbit} \multimap \mathit{bit}) <: A \multimap B}{\Gamma_1 !\Delta \rhd \mathit{meas} : A \multimap B}\ (ax) \quad \dfrac{\mathit{qbit} <: A}{\Gamma_2, !\Delta, x : \mathit{qbit} \rhd x : A}\ (ax_1)}{\Gamma, !\Delta, x : \mathit{qbit} \rhd \mathit{meas}\ \ x : B,}\ (app)$$

with $\Gamma = (\Gamma_1, \Gamma_2)$.

From the subtyping rule $(\multimap_2)$, $\mathit{bit} <: B$ and $A <: \mathit{qtype}$. Hence $A = \mathit{qbit}$ and $B = \mathit{bit}$, and $\Gamma_1, \Delta \rhd 0 : \mathit{bit}$ is a valid typing judgment. Using Lemma 7.1.2, $\Gamma_1, !\Delta, x : \mathit{qbit} \rhd 0 : \mathit{bit}$ is also valid: The theorem is true in this case.

- The rules for *new* are, if Q is in a space of dimension 2^n,

$$\frac{}{[Q, new\ \ 0] \longrightarrow_1 [Q \otimes |0\rangle, p_n],}$$

$$\frac{}{[Q, new\ \ 1] \longrightarrow_1 [Q \otimes |1\rangle, p_n].}$$

  We study the first case, the second is similar.

  If $\Gamma \rhd new\, 0{:}B$ is valid it comes from

$$\frac{\dfrac{!(bit \multimap qbit) <: A \multimap B \quad \Gamma_1, !\Delta \rhd new : A \multimap B}{\Gamma_1, \Gamma_2, !\Delta \rhd new\, 0{:}B,} \quad \dfrac{!\, bit <: A \quad \Gamma_2, !\Delta \rhd 0{:}A}{}}{}\ (app)$$

  for some splitting $\Gamma = (\Gamma_1, \Gamma_2, !\Delta)$. Thus $qbit <: B$, and then $B = qbit$. The state $[Q \otimes |0\rangle, p_n]$ is $[Q \otimes |0\rangle, L \cup \{x \mapsto n\}x]$, if $[Q, new\ \ 0] = [Q, L, new\ \ 0]$. In particular one can choose a variable x which is not in $|\Gamma|$, by α-equivalence. Then the typing judgment $\Gamma, x{:}\, qbit \rhd x{:}\, qbit$ is valid, and the theorem is true in this case.

- The idea is the same for H.

- The first induction rule is:

$$\frac{N \longrightarrow N'}{MN \longrightarrow MN'.}$$

  Since N and N' have the same type by induction hypothesis, MN and MN' have the same type by (*app*).

- The second induction rule is:

$$\frac{M \longrightarrow M'}{MV \longrightarrow M'V.}$$

  Since M and M' have the same type by induction hypothesis, MV and $M'V$ have the same type by (*app*).

- For the *if* rules, it follows directly from the typing rule.

□

7.3 Progress theorem

Definition 7.3.1 A *program* is defined as a quantum state $[Q, L, M]$, where there exists a type B such that, if $\Delta = \{x : qbit \mid x \in FV(M)\}$,

$$\Delta \rhd M : B$$

is a valid quantum typing judgment.

Theorem 7.3.2 (Progress) *Let $[Q, M]$ be a typable program, then it is consistent, as defined in Section 5.2 on page 66, i.e. it can never reduce to an error state. Hence any closed well-typed term either converges to a value, or diverges.*

Proof We prove that for all programs $[Q, M]$, either it is a value, or there exists at least one M' such that $M \longrightarrow M'$. We do it by induction on the proof of validity of the typing judgment. There are two cases. Either it is a value, in which case there is nothing to do, or it is not, and the only 2 rules that apply are (app) and (if).

(app) In this case $M = PQ$.

$$\frac{\Delta_1 \rhd P : B \multimap A \quad \Delta_2 \rhd Q : B}{\Delta \rhd PQ : A,}$$

with

$$\Delta = (\Delta_1, \Delta_2) = \{x : qbit \,|x \in FV(M)\}.$$

Since $FV(M) = FV(P) \cup FV(Q)$, and they are disjoint, the two typing judgments we have are of the form required by the theorem. So by induction hypothesis, either we can reduce Q, and we are done, or it is a value. If it is a value, let us study P: P is also either reducible, and then we are done, or it is a value. If it is a value, then either it is an abstraction and PQ is reducible, or it is a constant function, *new*, *meas* or H. Since the typing judgment is valid, we are done, we can reduce in this last case.

(if) The *if* statement is similar: $M = if(P; Q; R)$, and either we can reduce P, or it is a value, so 0 or 1 and we can reduce M in Q or R.

So by induction any closed well-typed term is consistent. □

Chapter 8

Extension of the language

8.1 Extended language

Let us extend the language with product types . Extended terms and types are defined in Tables 14 and 15. In this case we allow the U^n to be unitary operations of n qubits. For example if U^2 is a binary unitary gate, we use it as follows:

$$U^2 : !(qbit \otimes qbit \multimap qbit \otimes qbit)$$

We add to the previous definition a notion of pairs: as in simply-typed lambda-calculus, we will denote a pair by

$$\langle M_1, M_2 \rangle.$$

Tuples are defined as

$$\otimes_{i=0}^{n} A_i = A_1 \otimes (A_2 \otimes (A_3 \ldots) \ldots),$$

$$\langle M_1, \ldots M_n \rangle = \langle M_1, \langle M_2, \langle M_3 \ldots \rangle \ldots \rangle \rangle.$$

Free variable, substitution We extend the notion of free variable and substitution with the same definition as in Chapter 3, Tables 2 and 3.

Typing rules and reduction steps The typing rules to add are in Table 16. The reduction procedure for these new terms is found in Table 17

$$
\begin{array}{rcl}
RawTerm\ M,N,P & ::= & x \\
& | & MN \\
& | & \lambda x.M \\
& | & if(M;N;P) \\
& | & 0 \\
& | & 1 \\
& | & meas \\
& | & new \\
& | & U^n \\
& | & * \\
& | & \langle M,N\rangle \\
& | & let\ \langle x,y\rangle{=}M\ in\ N \\
\\
Value\ U,V & ::= & x \\
& | & \lambda x.M \\
& | & 0 \\
& | & 1 \\
& | & meas \\
& | & new \\
& | & U^n \\
& | & * \\
& | & \langle U,V\rangle
\end{array}
$$

Table 14: Extended terms

$$
\begin{array}{rcl}
qType\ A, B & ::= & (n)(\alpha) \\
& | & (n)(\top) \\
& | & (n)(X) \\
& | & (n)(A \multimap B) \\
& | & (n)(A \otimes B)
\end{array}
$$

The subtyping relation is extended to

$$\frac{(m = 0) \vee (n \geq 1)}{(n)(\top) <: (m)(\top)}\ (\top)$$

$$\frac{(m = 0) \vee (n \geq 1) \quad A_1 <: B_1 \quad A_2 <: B_2}{(n)(A_1 \otimes A_2) <: (m)(B_1 \otimes B_2)}\ (\otimes)$$

Table 15: Extended types

8.2 Cartesian product versus Tensor product

We use in our language the tensor product instead of a cartesian product. The reason is the following: If we define our product as cartesian, we need 2 projections π_1 and π_2:

$$\pi_1 : A \times B \to A$$

$$\pi_2 : A \times B \to B$$

Then there has to be a bijection

$$\langle \pi_1(M), \pi_2(M) \rangle \leftrightarrow M$$

But such a projection cannot exists: if M is not duplicable, we do not have the right to write $\langle \pi_1(M), \pi_2(M) \rangle$. This is not linear in M.

Thus, we have to take care of the fact that we can have non-duplicable terms in a tuple. Let us take an example:

$$\left[\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \langle p_0, p_1 \rangle \right]$$

is a perfectly valid quantum state: in the term $M = \langle p_0, p_1 \rangle$ we have stored two qubits. Let us say we want to apply the H gate on p_1 and then the $CNOT$ gate on

First let us define the type of the new term constants:

$$\begin{array}{rcl} * & \mapsto & !\top \\ U^n & \mapsto & !(\otimes_{i=1}^{n} qbit \multimap \otimes_{i=1}^{n} qbit) \end{array}$$

If A_1 and A_2 are not exponential,

$$\frac{!\Delta, \Gamma_1 \rhd M_1 : (n+m)(A_1) \quad !\Delta, \Gamma_2 \rhd M_2 : (n+l)(A_2)}{!\Delta, \Gamma_1, \Gamma_2 \rhd \langle M_1, M_2 \rangle : (n)((m)(A_1) \otimes (l)(A_2))} \ (\otimes.I)$$

$$\frac{!\Delta, \Gamma_1 \rhd M{:}(n)(A_1 \otimes A_2) \quad !\Delta, \Gamma_2, x_1{:}(n)(A_1), x_2{:}(n)(A_2) \rhd N{:}A}{!\Delta, \Gamma_1, \Gamma_2 \rhd \text{let } \langle x_1, x_2 \rangle = M \text{ in } N{:}A} \ (\otimes.E)$$

Table 16: Extended typing rules

both of them. The *CNOT* gate is:

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

Using projections π_1 and π_2, we would have to write this as:

$$CNOT\langle H(\pi_1 M), \pi_2 M \rangle$$

and that is not a valid program since we are duplicating M. If we want to stay linear, we have either to forget p_1 in doing π_2 or to forget p_2 in doing π_1. So we cannot use cartesian products to model all the programs we need.

With tensor product, the linearity is kept: we can retrieve information in both A and B of a product $A \otimes B$ in a linear manner using

$$let\ \langle x, y \rangle{=}M\ in\ N,$$

as we do in Chapter 3.

If V_1, V_2 are values,
$[Q, \text{let } \langle x_1, x_2\rangle = \langle V_1, V_2\rangle \text{ in } N] \longrightarrow_1 [Q, N[V_1/x_1, V_2/x_2]]$

One reduces a tuple from left to right:

$$\frac{[Q, M_1] \longrightarrow_p [Q', M_1']}{[Q, \langle M_1, M_2\rangle] \longrightarrow_p [Q', \langle M_1', M_2\rangle]}$$

$$\frac{[Q, M_2] \longrightarrow_p [Q', M_2']}{[Q, \langle V_1, M_2\rangle] \longrightarrow_p [Q', \langle V_1, M_2'\rangle]}$$

Table 17: Extended call-by-value reduction

The above problem has the following solution:

$$\text{let } \langle x, y\rangle = M \text{ in } (CNOT\, \langle (Hx), y\rangle$$

since linearity of the product's elements is preserved.

Remark. We have obtained the structure for a monoidal category. Indeed we can define linear functions:

$$\sigma : A \otimes B \multimap B \otimes A$$

$$\alpha : (A \otimes B) \otimes C \multimap A \otimes (B \otimes C)$$

$$\lambda : A \otimes \top \multimap A$$

$$\rho : A \multimap A \otimes \top$$

as follows:

$$\sigma = \lambda p.(let\ \ \langle x, y\rangle = p\ \ in\ \ \langle y, x\rangle)$$

$$\alpha = \lambda p.(let\ \ \langle x, y\rangle = p\ \ in \quad let\ \ \langle z, t\rangle = x\ \ in \quad \langle z, \langle t, y\rangle\rangle)$$

$$\lambda = \lambda p.(let\ \ \langle x, y\rangle = p\ \ in\ \ x)$$

$$\rho = \lambda x.\langle x, * \rangle$$

And moreover, given

$$f : A \multimap B$$
$$g : C \multimap D$$

one can define

$$f \otimes g : A \otimes C \multimap B \otimes D$$

like this:

$$f \otimes g = \lambda p.(let \ \ \langle x, y \rangle = p \ \ in \langle fx, gy \rangle)$$

8.3 Compatibility with the previous results

Lemma 8.3.1 *All the previous lemmas still hold in the extended language.*

Proof. the lemmas we need to prove are 7.1.1, 7.1.2, 7.1.4, 7.1.5, 7.1.6 and 7.1.7.

Lemmas 7.1.1 and 7.1.2 are completely similar to the ones in the background chapter.

Proof of Lemma 7.1.4. We want to show that if $\Delta \rhd N : A$ is valid and if $\Gamma <: \Delta$ and $A <: B$, then $\Gamma \rhd N : B$ is also valid

We do it by induction on the structure of N. We have to check for the new cases.

If $N = \langle M_1, M_2 \rangle$, then $\Delta \rhd \langle M_1, M_2 \rangle : (n)(A_1 \otimes A_2)$ comes from

$$\frac{!\Delta_1, \Delta_2 \rhd M_1 : (n)(A_1) \quad !\Delta_1, \Delta_3 \rhd M_k : (n)(A_2)}{!\Delta_1, \Delta_2, \Delta_3 \rhd \langle M_1, M_2 \rangle : (n)(A_1 \otimes A_2)} \otimes.I$$

Since $\Gamma <: \Delta$, $\Gamma = (!\Gamma_1, \Gamma_2, \Gamma_3)$ with $!\Gamma_1 <: !\Delta_1$, $\Gamma_2 <: \Delta_2$ and $\Gamma_3 <: \Delta_3$. There is a bang on $!\Gamma_1$ since $!\Gamma_1 <: !\Delta_1$. Since $A <: B$, $B = \langle B_1, B_2 \rangle$ with $A_1 <: B_1$ and $A_2 <: B_2$. Hence the induction hypothesis can be applied, and

$$!\Gamma_1, \Gamma_2 \rhd M_1 : (n)(B_1) \quad \text{and} \quad !\Gamma_1, \Gamma_3 \rhd M_2 : (n)(B_2)$$

are valid. Applying $(\otimes.I)$, we obtain the result.

If $N = let\ \langle x_1, x_2\rangle{=}M\ in\ P$, then the typing judgement comes from

$$\frac{!\Delta_1, \Delta_2 \rhd M{:}(n)(A_1 \otimes A_2) \quad !\Delta_1, \Delta_3, x_1{:}(n)(A_1), x_2{:}(n)(A_2) \rhd N{:}A}{!\Delta_1, \Delta_2, \Delta_3 \rhd \text{let } \langle x_1, x_2\rangle = M \text{ in } P{:}A}\ \otimes.E$$

Since $\Gamma <: \Delta$, $\Gamma = (!\Gamma_1, \Gamma_2, \Gamma_3)$ with $!\Gamma_1 <: !\Delta_1$, $\Gamma_2 <: \Delta_2$ and $\Gamma_3 <: \Delta_3$. There is a bang on $!\Gamma_1$ since $!\Gamma_1 <: !\Delta_1$. Applying induction hypothesis,

$$!\Gamma_1, \Gamma_2 \rhd M{:}(n)(A_1 \otimes A_2) \quad \text{and} \quad !\Gamma_1, \Gamma_3, x_1{:}(n)(A_1), x_2{:}(n)(A_2) \rhd N{:}B$$

are valid. Applying $(\otimes.E)$ gives the result

If $M = *$, the proof is done similarly to the axioms already done.

Proof of Lemma 7.1.5. We want to prove that if V is a value such that $\Delta \rhd V : !A$ is valid then for all x in $FV(V)$ there exists U in *qType* such that $\Delta_f(x) = !U$.

The proof was started by structural induction on V.

If $V = \top$, the term is closed. So by vacuity the result is true.

If $V = \langle V_1, V_2\rangle$ with V_1 and V_2 values, the typing tree starts with

$$\frac{!\Delta, \Gamma_1 \rhd V_1 : (n+1)(A_1) \quad !\Delta, \Gamma_2 \rhd V_2 : (n+1)(A_2)}{!\Delta, \Gamma_1, \Gamma_2 \rhd \langle V_1, V_2\rangle : (n+1)(A_1 \otimes A_2)}\ \otimes.I$$

By induction hypothesis,

$$FV(V_1) \cap |\Gamma_1| = FV(V_2) \cap |\Gamma_2| = \emptyset.$$

Since $FV(\langle V_1, V_2\rangle) = FV(V_2) \cup FV(V_1)$, $FV(\langle V_1, V_2\rangle) = \emptyset$. And so the result is also true in that case.

Proof of Lemma 7.1.6. We want to prove that for A and B elements of *qType* and V a value, if $!\Delta, \Gamma_1 \rhd V : A$ and $!\Delta, \Gamma_2, x : A \rhd M : B$ are valid, then $\Gamma_1, \Gamma_2, !\Delta \rhd M[V/x] : B$ is valid.

The proof was done by induction on the typing tree of $!\Delta, \Gamma_2, x : A \rhd M : B$. We have 3 cases to add:

$(\otimes.I)$ is done as in the (app) case.

$(\otimes.E)$ is like combination of an application and an abstraction rule.

$(*)$ is done as in the constant case.

Proof of Corollary 7.1.7. We want to prove that if $\Gamma_1, !\Delta, x : A \rhd M : B$ and $\Gamma_2, !\Delta \rhd V : (\tau)A$ then $\Gamma_1, \Gamma_2, !\Delta \rhd M[V/x] : B$

This is still a corollary from Lemma 7.1.6 and Lemma 7.1.4.

□

Theorem 8.3.2 *Subject reduction still holds.*

Proof.

We have to check that the new structures added have rules that are compatible with subject reduction.

- The rules for the pairing are just an extension of the application rules, so using a similar method, it is working.

□

Theorem 8.3.3 *The progress theorem still hold.*

Proof. By inspection of the new rules. □

8.4 Examples

Example: implementing the Deutsch algorithm. The formalism of higher-order functional programming language is adequate for writing the Deutsch's algorithm. Indeed it can be done in that way:

$$
\begin{array}{l}
\mathit{let}\ \mathbf{Deutsch}\ U_f\ = \\
\quad \mathit{let}\ \mathbf{tens}\ f\ g\ \langle x, y\rangle\ = \langle fx, gy\rangle \\
\quad \mathit{in}\ \mathit{let}\ \langle x, y\rangle = \\
\qquad (\mathbf{tens}\ H\ (\lambda x.x))(U_f\langle H(\mathit{new}\ 0), H(\mathit{new}\ 1)\rangle) \\
\qquad\quad \mathit{in}\ \mathit{meas}\ x,
\end{array}
$$

in *ML* notations. Note that U_f is a variable that stands for a function from a two-qubit state to a two-qubit state. And indeed the function **Deutsch** is a higher-order function:

$$\triangleright\mathbf{Deutsch}{:}!((qbit \otimes qbit \multimap qbit \otimes qbit) \multimap bit)$$

is a well-typed typing judgment. Note that **Deutsch** is duplicable, and that U_f does not need to be duplicable, since it is used only once.

Example: implementing the teleportation procedure. We can embed each quantum circuit part of the procedure in a function. There is a function **EPR** : $!(\top \multimap (qbit \otimes qbit))$ that creates an entangled state, as in the step (1):

$$\mathbf{EPR} = \lambda x.CNOT\langle H(new\,0), new\,0\rangle.$$

There is a function **BellMeasure** : $!(qbit \multimap (qbit \multimap bit \otimes bit))$ that takes two qubits, rotates and measures them, as in steps (2) and (3):

$$\mathbf{BellMeasure} = \lambda q_2.\lambda q_1.(let\ \langle x, y\rangle = CNOT\langle q_1, q_2\rangle\ in\ \langle meas(Hx), meas\,y\rangle$$

We also can define a function $\mathbf{U} : !(qbit \multimap (bit \otimes bit \multimap qbit))$ that takes a qubit q and two bits x, y and returns $U_{xy}q$, as in step (4):

$$\begin{aligned}\mathbf{U} = \lambda q.\lambda\langle x, y\rangle.if\ x\ &then\ (if\ y\ then\ U_{11}q\ \ else\ U_{10}q)\\ &else\ (if\ y\ then\ U_{01}q\ \ else\ U_{00}q),\end{aligned}$$

where U_{xy} are defined as on page 16 when the measured qubits were x and y.

The teleportation procedure can be seen as the creation of two non-duplicable functions f and g

$$\begin{aligned}f &: qbit \multimap bit \otimes bit,\\ g &: bit \otimes bit \multimap qbit,\end{aligned}$$

such that $f \circ g(x) = x$ for an arbitrary qubit x. We can construct such a pair of functions by the following code:

$$\begin{aligned}&let\ \langle x, y\rangle{=}\mathbf{EPR} *\\ &\quad in\ let\ \mathbf{f}\ = \mathbf{BellMeasure}\ x\\ &\quad in\ let\ \mathbf{g}\ = \mathbf{U}\ y.\\ &\quad in\ \langle f, g\rangle.\end{aligned}$$

Note that, since f and g depend on the state of the qubits x and y, respectively, these functions cannot be duplicated, which is reflected in the fact that the types of f and g do not contain a top-level "!".

Chapter 9

Type inference algorithm

Up to now we have defined a quantum programming language, mixing quantum and classical data types, together with a type system to certify the good behavior of programs during reduction. However, a big problem is not solved: how can we say whether or not a program is well-typed ? An algorithm that can solve such a problem is called a *type inference algorithm.*

9.1 A first example

Our goal is to find an inference algorithm. One can try to base it on the one from the simply-typed lambda-calculus from Chapter 3. Recall that one key-point in this algorithm was, given a well-typed M, the existence of a most general typing judgment $\Delta \rhd M : A$ such that each possible typing judgment would be an instance of $\Delta \rhd M : A$

However, in MAILL, such a type does not exists. Indeed, consider the following example: let

$$M = \lambda xy.xy$$

be a lambda term. Note that M is a well-typed closed term. Here are some valid typing judgments:

$$\rhd M : (U \multimap Y) \multimap (U \multimap V),$$
$$\rhd M :!(U \multimap V) \multimap !(U \multimap V),$$
$$\rhd M :!(U \multimap V) \multimap (U \multimap V).$$

The most general type W such that $\overline{\sigma}W = !(U \multimap V) \multimap !(U \multimap V)$ and $\overline{\tau}W = (U \multimap V) \multimap (U \multimap V)$ for some substitutions σ and τ is $X \multimap Y$. But

$$\triangleright M : X \multimap Y$$

is not valid: the notion of substitution is therefore not sufficient to describe the validity of a typing judgment.

On linear types, there is another natural ordering relation: the subtyping relation. For example, $(U \multimap V) \multimap !(U \multimap V)$ is the greatest element smaller than all types above, but

$$\triangleright M : (U \multimap V) \multimap !(U \multimap V)$$

is not a valid typing judgment. So there is no smallest type for this typing judgment using the subtyping relation.

However, one can consider the exponential symbols as decorations on linear types, as suggested by V. Danos, J.-B. Joinet and H. Schelling [7]. One can note that all possible types of M are of the form $(X \multimap Y) \multimap (X \multimap Y)$. If one replace $\multimap$ with $\Rightarrow$, this type gives a valid typing judgment $\blacktriangleright M : (X \Rightarrow Y) \Rightarrow (X \Rightarrow Y)$ in the simply-typed lambda-calculus of Chapter 3. The type in quantum lambda-calculus is therefore a decoration of a simple-type. We define these notions formally in the next section. This work is similar to [7].

9.2 Syntactic Skeleton

We define the class of *type skeletons* by

$$\begin{array}{llll} Skel & A, B & ::= & \alpha \\ & & | & X \\ & & | & (A \Rightarrow B) \\ & & | & (A \times B) \\ & & | & \top, \end{array}$$

where α ranges over the type constants and X over the type variables. We define the *typing-skeleton of A in qType* to be:

$$\begin{aligned} {}^{\dagger}(n)(\alpha) &= \alpha \\ {}^{\dagger}(n)(X) &= X \\ {}^{\dagger}(n)(A \multimap B) &= {}^{\dagger}A \Rightarrow {}^{\dagger}B \\ {}^{\dagger}(n)(A \otimes B) &= {}^{\dagger}A \times {}^{\dagger}B \\ {}^{\dagger}(n)(\top) &= \top. \end{aligned}$$

It corresponds to the *structure* of the type, or to erasing all "!".

Lemma 9.2.1 *If $U <: V$, then ${}^{\dagger}U = {}^{\dagger}V$.*

Proof. By induction on the derivation of $U <: V$ using set (2) of rules from page 72.

(var_2) ${}^{\dagger}(n)(X) = X = {}^{\dagger}(m)(X)$. Hence it is true in that case.

(α) The only type variables we have are *bit* and *qbit*, and they are not comparable using the subtyping relation. So if $\alpha \leq \beta$, then $\alpha = \beta$. So if $(n)(\alpha) <: (m)(\beta)$, then ${}^{\dagger}(n)(\alpha) = \alpha = \beta = {}^{\dagger}(n)(\beta)$.

($\multimap_2$) If the derivation starts with

$$\frac{A <: A' \quad B <: B' \quad (m = 0) \vee (n \geq 1)}{(n)(A' \multimap B) <: (m)(A \multimap B'),}$$

by induction hypothesis, ${}^{\dagger}A = {}^{\dagger}A'$ and ${}^{\dagger}B = {}^{\dagger}B'$. Using the definition of the skeleton, ${}^{\dagger}(n)(A' \multimap B) = {}^{\dagger}(m)(A \multimap B')$.

($\top$) By definition of skeleton, ${}^{\dagger}(n)(\top) = {}^{\dagger}(m)(\top)$.

($\otimes$) If the derivation starts with

$$\frac{(m = 0) \vee (n \geq 1) \quad A_1 <: B_1 \quad A_2 <: B_2}{(n)(A_1 \otimes A_2) <: (m)(B_1 \otimes B_2),}$$

then by induction hypothesis,

$${}^{\dagger}A_1 = {}^{\dagger}B_1 \quad \text{and} \quad {}^{\dagger}A_2 = {}^{\dagger}B_2$$

Using the definition of the skeleton,

$$^\dagger(n)(A_1 \otimes A_2) = {}^\dagger(m)(B_1 \otimes B_2)$$

□

We *extend* the notion to contexts and typing judgment as follows:

$$\begin{aligned} {}^\dagger\{x_1{:}A_1, \ldots, x_n{:}A_n\} &= \{x_1{:}{}^\dagger A_1, \ldots, x_n{:}{}^\dagger A_n\} \\ {}^\dagger(\Delta \rhd M{:}A) &= ({}^\dagger\Delta \rhd M{:}{}^\dagger A). \end{aligned}$$

If $\Delta \rhd M : A$ is a valid typing judgment in the quantum lambda-calculus, the following remark shows that ${}^\dagger(\Delta \rhd M : A) = ({}^\dagger\Delta \rhd M : {}^\dagger A)$ is a valid typing judgment in the skeleton lambda-calculus. The rules of the skeleton lambda-calculus are shown in Table 18. They are equivalent to the rules of simply-typed lambda-calculus from Table 7 in Chapter 3, modulo application of the weakening property. The reason for this slight reformulation of the rules is so that the skeleton calculus is the exact image of the quantum lambda-calculus under the skeleton operations, as shown in the following remark:

Remark 9.2.2 If a typing judgment T is valid, then its skeleton admits a proof tree constructed with the rules in Table 18. Such a proof tree is the exact image of the typing tree of T by †. □

Lemma 9.2.3

1. *The weakening property is verified: if $\Delta \rhd M : A$ is true, then $\Delta, \Gamma \rhd M : A$ is also true.*

2. *If $\Delta, x{:}B \rhd M : A$ with $x \notin FV(M)$ is true, then $\Delta \rhd M : A$ is also true.*

Proof. The proof is done by induction on the typing derivation of $\Delta \rhd M : A$, as it was for the quantum-typed case. □

Lemma 9.2.4 *Given a term M of the quantum lambda-calculus, $\Delta \rhd M{:}A$ if and only if $\Delta \blacktriangleright M{:}A$*

$$\frac{}{\Delta \rhd c : {}^{\dagger}A_c}$$

$$\frac{}{\Delta, x : A \rhd x : A}$$

$$\frac{\Delta, x : A \rhd M : B}{\Delta \rhd \lambda x.M : A \Rightarrow B}$$

$$\frac{\Delta_1, \Gamma \rhd M : A \Rightarrow B \quad \Delta_2, \Gamma \rhd N : A}{\Delta_1, \Delta_2, \Gamma \rhd MN : B}$$

$$\frac{\Gamma_1, \Delta \rhd P : bit \quad \Gamma_2, \Delta \rhd M : A \quad \Gamma_2, \Delta \rhd N : A}{\Gamma_1, \Gamma_2, \Delta \rhd \mathit{if}(P; M; N) : A}$$

$$\frac{\Delta, \Gamma_1 \rhd M_1 : A_1 \quad \Delta, \Gamma_2 \rhd M_2 : A_2}{\Delta, \Gamma_1, \Gamma_2 \rhd \langle M_1, M_2 \rangle : A_1 \times A_2}$$

$$\frac{\Delta, \Gamma_1 \rhd M : A_1 \times A_2 \quad \Delta, \Gamma_2, x_1 : A_1, x_2 : A_2 \rhd N : A}{\Delta, \Gamma_1, \Gamma_2 \rhd \text{let } \langle x_1, x_2 \rangle = M \text{ in } N : A}$$

Table 18: Induced typing rules for skeleton

Proof. First note that the set of quantum lambda-terms is the set of the simply-typed lambda-terms.

$\Leftarrow$ Each rule of the simply-typed lambda-calculus is an instance of the corresponding rule in skeleton lambda-calculus.

$\Rightarrow$ Using the weakening property of Lemma 3.2.1, one can prove by induction on the typing derivation of $\Delta \rhd M{:}A$ that $\Delta \blacktriangleright M{:}A$ is true.

□

Remark 9.2.5 Given a well-typed quantum term M, there exists a most general typing judgment for $x_1 : X_1 \ldots x_n : X_n \rhd M : Y$, $|\Delta| = \{x_1 \ldots x_n\}$.

Proof. Given the previous lemma, if $\Gamma \blacktriangleright M : A$ is a most general typing judgment for $x_1 : X_1 \ldots x_n : X_n \blacktriangleright M : Y$, a most general typing judgment for $x_1 : X_1 \ldots x_n : X_n \triangleright M : Y$ is $\Gamma \triangleright M : A$. $\square$

Definition 9.2.6 Given $A \in \mathit{Skel}$, one define a quantum type with the following inductive definition:

$$\begin{aligned} {}^{\clubsuit}X &= X \\ {}^{\clubsuit}\alpha &= \alpha \\ {}^{\clubsuit}(A \Rightarrow B) &= {}^{\clubsuit}A \multimap {}^{\clubsuit}B \\ {}^{\clubsuit}(A \times B) &= {}^{\clubsuit}A \otimes {}^{\clubsuit}B \end{aligned}$$

Lemma 9.2.7 $A = {}^{\dagger}{}^{\clubsuit}A$

Proof. by induction on the derivation of ${}^{\clubsuit}A$. $\square$

We now turn to the question of how a skeleton type can be "decorated" with exponentials to yield a quantum type. These decorations are going to be the heart of the quantum type inference algorithm.

Definition 9.2.8 Given $U \in \mathit{qType}$ and $A \in \mathit{Skel}$, we define the *decoration* $A \looparrowright U \in \mathit{qType}$ *of* A *along* U by

1) $A \looparrowright (n)(U) = (n)(A \looparrowright U)$ where U is not banged,
2) $(A \Rightarrow B) \looparrowright (U \multimap V) = (A \looparrowright U \multimap B \looparrowright V)$,
3) $(A \times B) \looparrowright (U \otimes V) = (A \looparrowright U \otimes B \looparrowright V)$,

and in all other cases,

4) $A \looparrowright U = {}^{\clubsuit}A$.

Lemma 9.2.9 *If* $A \in \mathit{Skel}$ *and* $U, V \in \mathit{qType}$, *then the following are true:*

a) $A \looparrowright (n)(U) = (n)(A \looparrowright U)$,
b) $(A \Rightarrow B) \looparrowright (U \multimap V) = A \looparrowright U \multimap B \looparrowright V$,
c) $(A \times B) \looparrowright (U \otimes V) = A \looparrowright U \otimes B \looparrowright V$,
d) *If* ${}^{\dagger}U = A$ *then* $A \looparrowright U = U$,
e) ${}^{\dagger}(A \looparrowright U) = A$,
f) *If* $U <: V$ *then* $A \looparrowright U <: A \looparrowright V$.

Proof.

a) $U = (m)(V)$ with V not banged. Then $(n)(U) = (m+n)(V)$.

$$\begin{aligned} & A \multimap (n)(U) \\ = \; & A \multimap (n)(m)(V) \\ = \; & A \multimap (n+m)(V) \\ = \; & (m+n)(A \multimap V) \\ = \; & (n)((m)(A \multimap V)) \\ = \; & (n)(A \multimap (m)(V)) \\ = \; & (n)(A \multimap U). \end{aligned}$$

b) **and** *c*) are the definition.

d) By induction on the derivation of $A \multimap U$:

1) The formula is $A \multimap (n)(U)$, U not exponential. By definition, $A \multimap (n)(U) = (n)(A \multimap U)$. By induction hypothesis, $A \multimap U = U$. Then $A \multimap (n)(U) = (n)(U)$.

2) The formula is $(A \Rightarrow B) \multimap U \multimap V$ with ${}^{\dagger}(U \multimap V) = (A \Rightarrow B)$. From the definition of the skeleton, ${}^{\dagger}U = A$ and ${}^{\dagger}V = B$. So by induction hypothesis $A \multimap U = U$ and $B \multimap V = V$. So $(A \Rightarrow B) \multimap (U \multimap V) = (A \multimap U \multimap B \multimap V) = (U \multimap V)$

3) This case is similar, replacing $\multimap$ with $\otimes$ and $\Rightarrow$ with $\times$.

4) If $A = {}^{\dagger}U$ then this case is reach only if $A = U = \alpha, \top$ or X a type variable. Then ${}^{\clubsuit}A = U$, and $A \multimap U = U$.

e) By induction on the derivation of $A \multimap U$:

1) The formula is $A \multimap (n)(U)$, U not exponential. Since $A \multimap (n)(U) = (n)(A \multimap U)$, ${}^{\dagger}(A \multimap (n)(U)) = {}^{\dagger}(A \multimap U)$. By induction hypothesis, this is equal to A.

2) The formula is $(A \Rightarrow B) \multimap (U \multimap V)$. By induction hypothesis,

$$^{\dagger}(A \multimap U) = A \quad \text{and} \quad ^{\dagger}(B \multimap V) = B.$$

So $^{\dagger}(A \multimap U \multimap B \multimap V) = {}^{\dagger}(A \multimap U) \Rightarrow {}^{\dagger}(B \multimap V) = A \Rightarrow B$.

3) This case is similar, replacing $\multimap$ with $\otimes$ and $\Rightarrow$ with $\times$.

4) $A \multimap U = \clubsuit A$. By Lemma 9.2.7, $^{\dagger}(A \multimap U) = A$.

f) By induction on the derivation of $U <: V$.

(ax) In this case, since α can only be *bit* or *qbit*, the rule is $U <: U$. Then $A \multimap U = A \multimap V$. By reflexivity, $A \multimap U <: A \multimap V$.

(var) The rule is $X <: X$. By reflexivity, $A \multimap X <: A \multimap X$.

($\top$) is similar to the previous case.

(D) The rule is

$$\frac{U <: V.}{!U <: V}$$

By induction hypothesis, $A \multimap U <: A \multimap V$. Applying (D), $!(A \multimap U) <: A \multimap V$. From (a), $!(A \multimap U) = (A \multimap !U)$. Hence $A \multimap !U <: A \multimap V$.

(!) The rule is

$$\frac{!U <: V.}{!U <: !V}$$

By induction hypothesis, $A \multimap !U <: A \multimap V$. From (a), $!(A \multimap U) <: A \multimap V$. Applying (!), $!(A \multimap U) <: !(A \multimap V)$. And from (a), $A \multimap !U <: A \multimap !V$.

($\multimap$) The rule is

$$\frac{V <: U \quad U' <: V'}{U \multimap U' <: V \multimap V'}$$

By induction hypothesis, $A \multimap V <: A \multimap U$ and $A \multimap U' <: A \multimap V'$. Applying $(\multimap)$, $A \multimap U \multimap A \multimap U' <: A \multimap V \multimap A \multimap V'$. From (b), $A \multimap (U \multimap U') <: A \multimap (V \multimap V')$

($\otimes$) The rule is

$$\frac{U <: V \quad U' <: V'}{U \otimes U' <: V \otimes V'}$$

Using the same method as for the ($\multimap$) case, and from (c), one have $A \twoheadrightarrow (U \otimes U') <: A \twoheadrightarrow (V \otimes V')$

□

The following lemma is the key to the quantum type inference algorithm:

Lemma 9.2.10 *If M is well-typed in the quantum lambda-calculus with typing judgment $\Gamma \rhd M : U$, then for any valid typing judgment $\Delta \triangleright M : A$ in simply-typed lambda-calculus with $|\Delta| = |\Gamma|$, the typing judgment $\Delta \twoheadrightarrow \Gamma \rhd M : A \twoheadrightarrow U$ is valid in the quantum lambda-calculus and admits a proof which has for skeleton the proof of $\Delta \triangleright M : A$.*

Proof. By structural induction on the typing-tree of $\Gamma \rhd M : U$.

(c) $M = c$ and the typing judgment is $\Gamma \rhd c : U$, $A_c <: U$. Any valid typing judgment in simply typed λ-calculus is of the form $\Delta \triangleright c : {}^\dagger A_c$. Since $A_c <: U$, ${}^\dagger A_c = {}^\dagger U$. Then from Lemma 9.2.9.d one can deduce that ${}^\dagger A_c \twoheadrightarrow U = U$. And so the Lemma is true in that case: $\Delta \twoheadrightarrow \Gamma \rhd c : U$

(x) $M = x$ and the typing judgment is $\Gamma, x : U \rhd x : V$, with $U <: V$. A typing judgment in simply typed lambda-calculus is of the form $\Delta, x : A \triangleright x : A$. From Lemma 9.2.9.f, $A \twoheadrightarrow U <: A \twoheadrightarrow V$. And then $\Delta \twoheadrightarrow \Gamma, x : A \twoheadrightarrow U \rhd x : A \twoheadrightarrow V$ is valid in qType. And so the Lemma is true in this case.

(λ_1) $M = \lambda x.N$ and the last rule of the typing derivation is

$$\frac{\Gamma, x : U \rhd M : V.}{\Gamma \rhd \lambda x.M : U \multimap V}$$

The typing tree in simply typed lambda-calculus starts with

$$\frac{\Delta, x : A \triangleright M : B.}{\Delta \triangleright \lambda x.M : A \Rightarrow B}$$

The induction hypothesis applies for $\Gamma, x : U \rhd M : V$ and $\Delta, x : A \rhd M : B$. We have:

$$\Delta \Rightarrow \Gamma, x : A \Rightarrow U \rhd M : B \Rightarrow V.$$

One can apply (λ_1), and from Lemma 9.2.9.b, we obtain

$$\Delta \Rightarrow \Gamma \rhd \lambda x.M : (A \Rightarrow B) \Rightarrow (U \multimap V).$$

(λ_2) Given

$$\frac{\Gamma_2, !\Gamma_1, x{:}U \rhd M{:}V}{\Gamma_2, !\Gamma_1 \rhd \lambda x.M{:}(n+1)(U \multimap V)}$$

with $FV(\lambda x.N) \in |\Gamma_1|$ and

$$\frac{\Delta, x{:}A \rhd M{:}B}{\Delta \rhd \lambda x.M{:}A \Rightarrow B,}$$

since $|\Delta| = |\Gamma_2, !\Gamma_1|$, one can split Δ in (Δ_1, Δ_2), with $|\Delta_i| = |\Gamma_i|$. By induction hypothesis

$$\Delta_2 \Rightarrow \Gamma_2, \Delta_1 \Rightarrow !\Gamma_1, x{:}A \Rightarrow U \rhd M{:}B \Rightarrow V$$

is valid. From Lemma 9.2.9.a, $\Delta \Rightarrow !\Gamma_1$ is of the form $!\Gamma'_1$. The free variable of $\lambda x.N$ are still in $|\Gamma'_1|$, and then (λ_2) apply in place of (λ_1) in the previous case: we obtain

$$\Delta \Rightarrow \Gamma \rhd \lambda x.M : (A \Rightarrow B) \Rightarrow (n+1)(U \multimap V).$$

(app) $M = NP$ and the typing tree starts with

$$\frac{!\Gamma_1, \Gamma_2 \rhd N : U \multimap V \quad !\Gamma_1, \Gamma_3 \rhd P : U}{!\Gamma_1, \Gamma_2, \Gamma_3 \rhd NP : V}$$

In simply typed lambda calculus the typing tree is:

$$\frac{\Delta_1, \Delta_2 \rhd N : A \Rightarrow B \quad \Delta_1, \Delta_3 \rhd P : A}{\Delta_1, \Delta_2, \Delta_3 \rhd NP : B}$$

We have from the hypothesis that $|\Delta_1, \Delta_2, \Delta_3| = |!\Gamma_1, \Gamma_2, \Gamma_3|$. From the weakening property of Lemma 9.2.3.1 we can find a proof tree starting with:

$$\frac{\Delta_1, \Delta_2, \Delta_3 \rhd N : A \Rightarrow B \quad \Delta_1, \Delta_2, \Delta_3 \rhd P : A}{\Delta_1, \Delta_2, \Delta_3 \rhd NP : B}$$

The variable in Γ_3 are not free in N, and the variable in Γ_2 are not free in P. Using the strength property of Lemma 9.2.3.2, one can remove these variables to obtain

$$\frac{\Delta'_1, \Delta'_2 \rhd N : A \Rightarrow B \quad \Delta'_1, \Delta'_3 \rhd P : A}{\Delta'_1, \Delta'_2, \Delta'_3 \rhd NP : B},$$

with $|\Delta'_i| = |\Gamma_i|$. The induction hypothesis allows us to write that

$$\Delta'_1 \looparrowright !\Gamma_1, \Delta'_2 \looparrowright \Gamma_2 \rhd N : (A \Rightarrow B) \looparrowright (U \multimap V)$$

and

$$\Delta'_1 \looparrowright !\Gamma_1, \Delta'_3 \looparrowright \Gamma_3 \rhd P : A \looparrowright U$$

are valid. Since $(A \Rightarrow B) \looparrowright (U \multimap V) = A \looparrowright U \multimap B \looparrowright V$ and $\Delta'_1 \looparrowright !\Gamma_1$ is of the form $!\Gamma'_1$ using Lemma 9.2.9.a, we can apply the application rule and get

$$\Delta'_1 \looparrowright !\Gamma_1, \Delta'_2 \looparrowright \Gamma_2, \Delta'_3 \looparrowright \Gamma_3 \rhd NP : B \looparrowright V$$

and see that the lemma is verified in that case.

(if) $M = if\,(P; N; Q)$ and the typing tree starts with

$$\frac{!\Gamma_1, \Gamma_2 \rhd P{:}\, bit \quad !\Gamma_1, \Gamma_3 \rhd N{:}U \quad !\Gamma_1, \Gamma_3 \rhd Q{:}U}{!\Gamma_1, \Gamma_2, \Gamma_3 \rhd if\,(P; N; Q) : U}$$

In simply typed lambda calculus the typing tree is:

$$\frac{\Delta_1, \Delta_2 \rhd P{:}\, bit \quad \Delta_1, \Delta_3 \rhd N{:}A \quad \Delta_1, \Delta_3 \rhd Q{:}A}{\Delta_1, \Delta_2, \Delta_3 \rhd if\,(P; N; Q) : A}$$

Using the same trick as in the (app), one can rearrange the contexts to obtain

$$\frac{\Delta'_1, \Delta'_2 \rhd P{:}\, bit \quad \Delta'_1, \Delta'_3 \rhd N{:}A \quad \Delta'_1, \Delta'_3 \rhd Q{:}A}{\Delta'_1, \Delta'_2, \Delta_3 \rhd if\,(P; N; Q) : A}$$

with $|\Delta'_i| = |\Gamma_i|$. The induction hypothesis allows us to write that

$$\Delta'_1 \looparrowright !\Gamma_1, \Delta'_2 \looparrowright \Gamma_2 \rhd P{:}\, bit,$$

$$\Delta'_1 \looparrowright !\Gamma_1, \Delta'_3 \looparrowright \Gamma_3 \rhd N{:}A \looparrowright U,$$

and

$$\Delta'_1 \not\hookrightarrow !\Gamma_1, \Delta'_3 \not\hookrightarrow \Gamma_3 \rhd Q : A \not\hookrightarrow U$$

are valid. Since $\Delta'_1 \not\hookrightarrow !\Gamma_1$ is of the form $!\Gamma'_1$ using Lemma 9.2.9.a, we can apply the (if) rule and get

$$\Delta'_1 \not\hookrightarrow !\Gamma_1, \Delta'_2 \not\hookrightarrow \Gamma_2, \Delta'_3 \not\hookrightarrow \Gamma_3 \rhd if(P; N; Q) : A \not\hookrightarrow U$$

and see that the lemma is verified in that case.

$(\otimes)$ $M = \langle N, P\rangle$ and the typing tree starts with

$$\frac{!\Gamma_1, \Gamma_2 \rhd N{:}(m+n)(U) \quad !\Gamma_1, \Gamma_3 \rhd P{:}(n+l)(V)}{!\Gamma_1, \Gamma_2, \Gamma_3 \rhd \langle N, P\rangle : (n)((m)U) \otimes (l)(V))}$$

In simply typed lambda calculus the typing tree is:

$$\frac{\Delta_1, \Delta_2 \rhd N{:}A \quad \Delta_1, \Delta_3 \rhd P{:}B}{\Delta_1, \Delta_2, \Delta_3 \rhd \langle N, P\rangle : A \times B}$$

Using the same trick as in the (app), one can rearrange the contexts to obtain

$$\frac{\Delta'_1, \Delta'_2 \rhd N{:}A \quad \Delta'_1, \Delta'_3 \rhd P{:}B}{\Delta'_1, \Delta'_2, \Delta_3 \rhd \langle N, P\rangle : A \times B}$$

with $|\Delta'_i| = |\Gamma_i|$. The induction hypothesis allows us to write that

$$\Delta'_1 \not\hookrightarrow !\Gamma_1, \Delta'_2 \not\hookrightarrow \Gamma_2 \rhd N{:}A \not\hookrightarrow (n+m)(U),$$

and

$$\Delta'_1 \not\hookrightarrow !\Gamma_1, \Delta'_3 \not\hookrightarrow \Gamma_3 \rhd P : B \not\hookrightarrow (n+l)(V)$$

are valid. Since $\Delta'_1 \not\hookrightarrow !\Gamma_1$ is of the form $!\Gamma'_1$ using Lemma 9.2.9.a, we can apply the $(\otimes)$ rule and get

$$\Delta'_1 \not\hookrightarrow !\Gamma_1, \Delta'_2 \not\hookrightarrow \Gamma_2, \Delta'_3 \not\hookrightarrow \Gamma_3 \rhd \langle N, P\rangle : (A \times B) \not\hookrightarrow (n)(U \otimes V)$$

and see that the lemma is verified in that case.

- The proof for *let* is based on the same model as the ones above. The proof for $(*)$ is the same as the one for the constant terms.

□

9.3 Template

We want to be able to say whether a given term is typable. Note that if M is not typable in simply typed lambda calculus the M is not quantum typable by Remark 9.2. On the other hand, if M admits an intuitionistic typing judgment $\Gamma \rhd M : A$ (with typing derivation π, say), then M is quantum typable if and only if M has a quantum derivation whose skeleton is π. Thus we can perform type inference in the quantum lambda-calculus in two steps:

1. Find an intuitionistic typing derivation π, if any,

2. and find a decoration of π which is a valid quantum typing derivation, if possible.

Step (1) is already decidable, using Remark 9.2. In step (2), note that the set of decorations of π is in general infinite, due to the presence of multiple exponentials of the form $(n)(N)$ for arbitrary n. However, as we show in the next section, it suffices to consider the cases $n = 0$ and $n = 1$ making the search space for step (2) finite.

We define formally the *template of a term M and a term variables set E* to be the set

$$\mathcal{T}(E, M) = \left\{ \begin{array}{c} \vdots\ \pi \\ \Delta \rhd M : A \end{array} \text{ valid typing tree with } |\Delta| = E \right\}$$

9.4 A subclass of $qType$

We define a *SqType* to be a quantum type without repeated exponentials. Formally:

$$\begin{array}{rrcl} SType & C, D & ::= & !A \\ & & | & A \\ AqType & A & ::= & \alpha \\ & & | & X \\ & & | & (C \multimap D) \\ & & | & (C \otimes D) \\ & & | & \top \end{array}$$

There is a canonical projection:

$$\downarrow : qType \longmapsto SqType$$

defined by the following, using the set of rules (2):

$$\begin{aligned}
\downarrow(A \multimap B) &= \downarrow(A) \multimap \downarrow(B) \\
\downarrow((n+1)(A \multimap B)) &= !(\downarrow(A) \multimap \downarrow(B)) \\
\downarrow(\alpha) &= \alpha \\
\downarrow((n+1)(\alpha)) &= !\alpha \\
\downarrow(A \otimes B) &= \downarrow(A) \otimes \downarrow(B) \\
\downarrow((n+1)(A \otimes B)) &= !(\downarrow(A) \otimes \downarrow(B))) \\
\downarrow(\top) &= \top \\
\downarrow((n+1)(\top)) &= !\top
\end{aligned}$$

We extend this function to typing judgments, proofs and type substitutions in a canonical way. We define ϵ_n to be 0 if $n = 0$, 1 else.

Lemma 9.4.1 *For all A in qType, $\downarrow A \doteqdot A$.*

Proof. By structural induction on A, where $\epsilon_n = 0$ if $n = 0$, 1 else:

$$\begin{aligned}
\downarrow((n)(X)) = (\epsilon_n)(X) \doteqdot (n)(X), \\
\downarrow((n)(\alpha)) = (\epsilon_n)(\alpha) \doteqdot (n)(\alpha), \\
\downarrow((n)(\top)) = (\epsilon_n)(\top) \doteqdot (n)(\top),
\end{aligned}$$

from the set (2) of subtyping rules, and by induction hypothesis: $\downarrow((n)(C \multimap D)) \doteqdot (n)(C \multimap D)$ and $\downarrow((n)(C \otimes D)) \doteqdot (n)(C \otimes D)$, using the definition of subtyping. □

Lemma 9.4.2 *Any given skeleton is the image by † of only a finite number of elements of SqType.*

Proof. By structural induction on a skeleton A.

A is X, α or $\top$. The only two possible *qType* of such a skeleton are A and $!A$. Then there is a finite number of them.

$A = B \Rightarrow C$ a type U in *SqType* such that ${}^{\dagger}U = A$ can only be of the form $(\epsilon)(V \multimap W)$, with V being one of the finitely many *SqType* of skeleton B and W being one of the finitely many *SqType* of skeleton C, and ϵ being 0 or 1. So there are only finitely many U satisfying this condition.

$A = B \otimes C$ is similar to the $\Rightarrow$ case, replacing $\multimap$ with $\otimes$.

□

Lemma 9.4.3 *Given any valid typing judgment $\Delta \rhd M : U$ in qType with typing tree ω, the projection $\downarrow(\omega)$ is a valid typing tree for the typing judgment $\downarrow\Delta \rhd M : \downarrow U$. So a term M is valid in qType if and only if there is a typing tree for it in the co-domain of $\downarrow$.*

Proof. Follows directly from Lemma 9.4.1 and Lemma 7.1.4. □

Theorem 9.4.4 *There is a deterministic algorithm to check if a given term M is valid:*

- *Find a typing derivation for M. For example the one given by the type inference algorithm of Chapter 3.*
- *There is only a finite number of possible decoration, and M is valid if and only if one one find a valid proof tree for one of those decorations.*

Now we have a deterministic algorithm to decide if a term M is well-typed or not. However this algorithm is exponential in the size of the typing-tree of the most general unifier.

9.5 A polynomial-time decision procedure

The naive application of the procedure from Theorem 9.4.4 yields a search space which is finite, but exponential in the size of the intuitionistic typing derivation. However, it is easy to organize the search in a more efficient way.

Let *xSqType* be an extension of *qType*:

$$\begin{array}{llll} xSqType & V, W & ::= & (\rho)(U), \\ xAqType & U & ::= & X \mid \alpha \mid V \multimap W \mid V \otimes W, \end{array}$$

when ρ ranges over a countable set of variables, called *flags*.

Let Π be a map from *Skel* to *xSqType*, defined recursively, where ρ is a fresh flag at each step:

$$\begin{array}{rcl} \Pi(X) & = & (\rho)(X) \\ \Pi(\alpha) & = & (\rho)(\alpha) \\ \Pi(A \Rightarrow B) & = & (\rho)(\Pi(A) \multimap \Pi(B)) \\ \Pi(A \times B) & = & (\rho)(\Pi(A) \otimes \Pi(B)) \end{array}$$

One can canonically extend Π to skeleton judgments and skeleton typing-proofs.

Let A be an *xSqType*, and let F be the set of flags occurring in A. Given a function τ from *xSqType* to *SqType*, we can define an *SqType* $\tau(A)$ by:

$$\begin{array}{rcl} \text{If } A & \text{is} & \text{in } xAqType \\ \tau((\rho)(A)) & = & (\tau\rho)A \\ \text{If } V, W & \text{are} & \text{in } xSqType \\ \tau(X) & = & X \\ \tau(\alpha) & = & \alpha \\ \tau(V \multimap W) & = & \tau(V) \multimap \tau(W) \\ \tau(V \otimes W) & = & \tau(V) \otimes \tau(W) \end{array}$$

One can canonically extend τ to the domain of Π.

Given a skeleton typing tree, an inference algorithm needs only to place constraints on τ in order to obtain a valid typing tree in *SqType*. Given a valid skeleton typing judgment $(\Delta \rhd M : A)$ with its typing tree, one construct a set of constraints for τ in the following manner:

(x) The typing tree of $(\Delta \rhd M : A)$ is

$$\overline{x_1 : A_1, \ldots x_n : A_n \rhd x_i : A_i}.$$

Π outputs

$$x_1 : U_1, \ldots x_n : U_n \rhd x_i : V$$

The constraint for τ is (it is fully explained in the proof of Lemma 9.5.1):

$$\tau(U_i) <: \tau(V)$$

(c) The typing tree of $(\Delta \rhd M : A)$ is

$$\frac{}{x_1 : A_1, \ldots x_n : A_n \rhd c : {}^{\dagger}A_c.}$$

Π outputs

$$x_1 : U_1, \ldots x_n : U_n \rhd c : V.$$

The constraint for τ is (it is fully explained in the proof of Lemma 9.5.1):

$$A_c <: \tau(V)$$

(λ) The typing tree of $(\Delta \rhd M{:}A)$ is

$$\frac{\begin{array}{c}\vdots\ \omega\\ \Delta, x{:}C \rhd N{:}D\end{array}}{\Delta \rhd \lambda x.N{:}C \Rightarrow D.}$$

One must make matching the output of Π with the form

$$\frac{\begin{array}{c}\vdots\ \Pi(\omega)\\ \Delta', x{:}U \rhd N{:}V\end{array}}{\Delta' \rhd \lambda x.N{:}(\rho)(U \multimap V).}$$

If $\Delta' = \{y_1{:}(\rho_1)(U_1) \ldots y_n{:}(\rho_n)(U_n)\}$, the constraints for τ are the ones of the previous call to Π together with:

$$\forall y_i \quad \tau(\rho) = 1 \Rightarrow \tau(\rho_i) = 1$$

(app) The typing tree of $(\Delta \rhd M{:}A)$ is

$$\frac{\begin{array}{cc}\vdots\ \omega_1 & \vdots\ \omega_2\\ \Delta_1, \Delta_2 \rhd N{:}A \Rightarrow B & \Delta_1, \Delta_3 \rhd P{:}A\end{array}}{\Delta_1, \Delta_2, \Delta_3 \rhd NP{:}B}.$$

One must make matching the output of Π with the form

$$\frac{\begin{array}{cc}\vdots\ \Pi(\omega_1) & \vdots\ \Pi(\omega_2)\\ \Delta'_1,\Delta'_2 \rhd N{:}(\rho)(U \multimap V) & \Delta'_1,\Delta'_3 \rhd P{:}U\end{array}}{\Delta'_1,\Delta'_2,\Delta'_3 \rhd NP{:}V}.$$

If $\Delta'_1 = \{y_1{:}(\rho_1)(U_1) \ldots y_n{:}(\rho_n)(U_n)\}$, the constraints for τ are the ones of the two previous calls to Π together with

$$\tau(\rho) = 0$$

$$\forall y_i \quad \tau(\rho_i) = 1$$

(if) The typing tree of $(\Delta \rhd M{:}A)$ is

$$\frac{\begin{array}{ccc}\vdots\ \omega_1 & \vdots\ \omega_2 & \vdots\ \omega_3\\ \Delta_3,\Delta_1 \rhd P{:}bit & \Delta_2,\Delta_1 \rhd Q{:}A & \Delta_2,\Delta_1 \rhd N{:}A\end{array}}{\Delta_1,\Delta_2,\Delta_3 \rhd \mathit{if}(P;Q;N) : A}.$$

One must make matching the output of Π with the form

$$\frac{\begin{array}{ccc}\vdots\ \Pi(\omega_1) & \vdots\ \Pi(\omega_2) & \vdots\ \Pi(\omega_3\\ \Delta'_3,\Delta'_1 \rhd P{:}(\rho)(bit) & \Delta'_2,\Delta'_1 \rhd Q{:}U & \Delta'_2,\Delta'_1 \rhd N{:}U\end{array}}{\Delta'_1,\Delta'_2,\Delta'_3 \rhd \mathit{if}(P;Q;N) : U}.$$

If $\Delta'_1 = \{y_1{:}(\rho_1)(U_1) \ldots y_n{:}(\rho_n)(U_n)\}$, the constraints for τ are the ones of the three previous calls to Π together with

$$\tau(\rho) = 0$$

$$\forall y_i \quad \tau(\rho_i) = 1$$

$(\times.I)$ The typing tree of $(\Delta \rhd M{:}A)$ is

$$\frac{\begin{array}{cc}\vdots\ \omega_1 & \vdots\ \omega_2\\ \Delta,\Gamma_1 \rhd M_1 : A_1 & \Delta,\Gamma_2 \rhd M_2 : A_2\end{array}}{\Delta,\Gamma_1,\Gamma_2 \rhd \langle M_1, M_2\rangle : A_1 \times A_2}$$

One must make matching the output of Π with the form

$$\frac{\begin{array}{cc}\vdots\ \Pi(\omega_1) & \vdots\ \Pi(\omega_2)\\ \Delta', \Gamma_1' \rhd M_1 : (\sigma_1')(A_1) & \Delta', \Gamma_2' \rhd M_2 : (\sigma_2')(A_2)\end{array}}{\Delta', \Gamma_1', \Gamma_2' \rhd \langle M_1, M_2\rangle : (\rho)((\sigma_1)(A_1) \times (\sigma_2)(A_2))}$$

If $\Delta' = \{y_1{:}(\rho_1)(U_1) \ldots y_n{:}(\rho_n)(U_n)\}$, the constraints for τ are the ones of the two previous calls to Π together with

$$\forall y_i \quad \tau(\rho_i) = 1,$$

$$\tau(\rho) = 1 \Rightarrow \tau(\sigma_1') = 1 \quad \text{and} \quad \tau(\rho) = 1 \Rightarrow \tau(\sigma_2') = 1,$$

$$\tau(\sigma_1) = 1 \Rightarrow \tau(\sigma_1') = 1 \quad \text{and} \quad \tau(\sigma_2) = 1 \Rightarrow \tau(\sigma_2') = 1.$$

$(\times.E)$ The typing tree of $(\Delta \rhd M{:}A)$ is

$$\frac{\begin{array}{cc}\vdots\ \omega_1 & \vdots\ \omega_2\\ \Delta, \Gamma_1 \rhd M : A_1 \times A_2 & \Delta, \Gamma_2, x_1 : A_1, x_2 : A_2 \rhd N : A\end{array}}{\Delta, \Gamma_1, \Gamma_2 \rhd \mathit{let}\ \langle x_1, x_2\rangle{=}M\ \mathit{in}\ N{:}A}$$

One must make matching the output of Π with the form

$$\frac{\begin{array}{cc}\vdots\ \Pi(\omega_1) & \vdots\ \Pi(\omega_2)\\ \Delta', \Gamma_1' \rhd M : (\rho')(A_1 \times A_2) & \Delta', \Gamma_2', x_1 : (\rho')(A_1), x_2 : (\rho')(A_2) \rhd N : (\rho)(A)\end{array}}{\Delta', \Gamma_1', \Gamma_2' \rhd \mathit{let}\ \langle x_1, x_2\rangle{=}M\ \mathit{in}\ N{:}(\rho)(A)}$$

If $\Delta' = \{y_1{:}(\rho_1)(U_1) \ldots y_n{:}(\rho_n)(U_n)\}$, the constraints for τ are the ones of the two previous calls to Π together with

$$\forall y_i \quad \tau(\rho_i) = 1,$$

Lemma 9.5.1 *This algorithm is well-defined and given $\Delta \rhd M : A$ a skeleton typing judgment, the set of τ that satisfy the constraints is in 1 to 1 correspondence with the set of quantum typing derivations, using SqType, whose images by $\dagger$ are $(\Delta \rhd M : A)$. Moreover, the set of constraints for τ are all of the form*

$$\tau(\rho) = 1,$$
$$\tau(\rho) = 0 \text{ or}$$
$$\tau(\rho) = 1 \Rightarrow \tau(\rho') = 1$$

Proof. By induction on the typing-tree, any typing judgment $(\Gamma \rhd M{:}U)$ using *SqType* whose image by † is $(\Delta \rhd M : A)$ will give a map τ such that $\tau\Pi(\Delta \rhd M : A) = (\Gamma \rhd M{:}U)$. On the converse, all the constraints placed for τ are sufficient to make the image valid, by inspection of the rules.

Finally, the only constraints that are not of the claimed form are the ones that are for the variables and for the constants: $\tau(U) <: \tau(V)$ and $A_c <: \tau(V)$ But such a constraint can be re-written, using this recursive procedure which is a translation of the set (2) of subtyping rules on page 72:

(α) Since α is only *bit* and *qbit*, and since there is no subtyping relation between them, the rule is $\tau((\rho)(\alpha)) <: \tau((\rho')(\alpha))$ if and only if $\tau(\rho') = 1 \Rightarrow \tau(\rho) = 1$

(X) $\tau((\rho)(X)) <: \tau((\rho')(X))$ if and only if $\tau(\rho') = 1 \Rightarrow \tau(\rho) = 1$

($\multimap$) $\tau((\rho)(U \multimap V)) <: \tau((\rho')(U' \multimap V'))$ if and only if $\tau(V) <: \tau(V')$ and $\tau(U') <: \tau(U)$ and $\tau(\rho') = 1 \Rightarrow \tau(\rho) = 1$

($\otimes$) $\tau((\rho)(U \otimes V)) <: \tau((\rho')(U' \otimes V'))$ if and only if $\tau(V) <: \tau(V')$ and $\tau(U) <: \tau(U')$ and $\tau(\rho') = 1 \Rightarrow \tau(\rho) = 1$

and this gives a set of constraints that are of the right form. The constraint $A_c <: \tau(V)$ are translated with a similar algorithm. □

Theorem 9.5.2 *The algorithm gives a decidability criterion for the quantum typability of M, and given a valid skeleton typing judgment $\Delta \rhd M{:}A$, the algorithm is polynomial in the size of skeleton typing tree of $\Delta \rhd M{:}A$.*

Proof. From Lemma 9.2.10 and Lemma 9.5.1, if M is intuitionistic-typable, then it is quantum typable if and only if the set of constraints for τ is consistent. This can be done in a polynomial manner on the number of elements in the set. Indeed, an algorithm based on tableau-system is the following:

- Set all the values to 1 and 0 according to the clauses $\tau(\rho) = \ldots$. If there is a contradiction there, then fail.

- For each formula $\tau(\rho) = 1 \Rightarrow \tau(\rho') = 1$, if the value of ρ is 1, remove $\tau(\rho) = 1 \Rightarrow \tau(\rho') = 1$ and set ρ' to 1 if it is not set to 0.
- If it was, then fails.
- Continue until nothing can be done anymore, and then output success.

Since the number of constraints is polynomial in the size of the typing derivation, and since the algorithm given in this proof is polynomial in the size of the number of constraints, the quantum typability of a given valid M in simply-typed lambda calculus can be decided in polynomial time on the size of the intuitionistic typing derivation. $\square$

Chapter 10

Conclusion and further work

In this thesis, we have defined a higher-order quantum programming language based on a linear typed lambda calculus. Compared to the quantum lambda calculus of van Tonder [26, 27], our language is characterized by the fact that it combines classical as well as quantum features; thus, we have classical data types as well as quantum ones. We also provide both unitary operations and measurements as primitive features of our language. As the language shows, linearity constraints do not just exist at base types, but also at higher types, due to the fact that higher-order function are represented as closures which may in turns contain embedded quantum data. We have shown that affine intuitionistic linear logic provides precisely the right type system to deal with this situation.

There are many open problems for further work. An interesting question is whether the syntax of this language can be extended to include recursion. In addition to the multiplicative types, one can wonder whether it is possible to extend the type system to additive types, as in linear logic. Another question is to study more carefully the relation with affine intuitionistic linear logic, and compare with a type-system for a call-by-name reduction strategy. A very important open problem is to find a satisfactory denotational semantics for a higher order quantum programming language. One approach for finding such a semantics is to extend the framework of Selinger [21] and to identify an appropriate higher-order version of the notion of a superoperator.

Bibliography

[1] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1-2):3–57, 1993.

[2] H. P. Barendregt. *The Lambda-Calculus, its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North Holland, second edition, 1984.

[3] P. Benioff. The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines. *Journal of Statistical Physics*, 22:563–591, 1980.

[4] N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes In Computer Science*, pages 75–90. Springer Verlag, 1993.

[5] S. Bettelli, T. Calarco, and L. Serafini. Toward an architecture for quantum programming. *The European Physical Journal D*, 25(2):181–200, August 2003.

[6] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[7] V. Danos, J.-B. Joinet, and H. Schellinx. On the linear decoration of intuitionistic derivations. *Archive for Mathematical Logic*, 33:387–412, 1995.

[8] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 400(1818):97–117, July 1985.

[9] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[10] J.-Y. Girard. La logique linéaire. *Pour La Science*, 150:74–85, April 1990.

[11] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1990.

[12] S. Kleene. A theory of positive integers in formal logic. *American Journal of Mathematics*, 57:153–173 and 219–244, 1935.

[13] E. Knill. Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory, 1996.

[14] P. Naur, J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, and B. Vauquois. Report on the algorithmic language ALGOL 60. *Communications of the Association of Computing Machinery*, 3(5):299–314, May 1960.

[15] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.

[16] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[17] J. Preskill. Lecture notes for Physics 229, quantum computation. Available from http://www.theory.caltech.edu/people/preskill/ph229/#lecture, 1999.

[18] A. P. Propylov. Decidability of linear affine logic. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 496–504, San Diego, California, June 1995. IEEE, IEEE Computer Society Press.

[19] J. W. Sanders and P. Zuliani. Quantum programming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction: 5th International*

Conference, volume 1837 of *Lecture Notes in Computer Science*, pages 80–99, Ponte de Lima, Portugal, July 2000. Springer-Verlag.

[20] P. Selinger. Lecture notes on the lambda calculus. Available from the web site http://quasar.mathstat.uottawa.ca/∼selinger/papers/, 2001.

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

[22] P. W. Shor. Algorithms for quantum computation: Discrete log and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. Institute of Electrical and Electronic Engineers Computer Society Press, November 1994.

[23] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997. This is an expanded version of [22].

[24] A. Turing and J.-Y. Girard. *La machine de Turing*, volume 131 of *Points Sciences*. Editions du Seuil, 1995.

[25] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2*, 42, 1936. Can be found commented by J.-Y. Girard in [24].

[26] A. van Tonder. Quantum computation, categorical semantics and linear logic. On arXiv: quant-ph/0312174, 2003.

[27] A. van Tonder. A lambda calculus for quantum computation. *SIAM Journal of Computing*, 33(5):1109–1135, 2004. Available from arXiv:quant-ph/0307150.

[28] P. Wadler. A syntax for linear logic. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *9th International Conference on the Mathematical Foundations of Programming Semantics*, volume 802 of *Lecture Notes in Computer Science*, pages 513–529, New Orleans, Louisiana, 1993. Springer Verlag.