1

Quantum Lambda Calculus

Peter Selinger
Dalhousie University, Canada

Benoît Valiron
INRIA and École Polytechnique, LIX, Palaiseau, France.

Abstract

We discuss the design of a typed lambda calculus for quantum computation. After a brief discussion of the role of higher-order functions in quantum information theory, we define the quantum lambda calculus and its operational semantics. Safety invariants, such as the no-cloning property, are enforced by a static type system that is based on intuitionistic linear logic. We also describe a type inference algorithm, and a categorical semantics.

1.1 Introduction

The lambda calculus, developed in the 1930's by Church and Curry, is a formalism for expressing higher-order functions. In a nutshell, a higher-order function is a function that inputs or outputs a "black box", which is itself a (possibly higher-order) function. Higher-order functions are a computationally powerful tool. Indeed, the pure untyped lambda calculus has the same computational power as Turing machines (Turing, 1937). At the same time, higher-order functions are a useful abstraction for programmers. They form the basis of functional programming languages such as LISP (McCarthy, 1960), Scheme (Sussman and Steele Jr, 1975), ML (Milner, 1978) and Haskell (Hudak et al., 2007).

In this chapter, we discuss how to combine higher-order functions with quantum computation. We believe that this is an interesting question for a number of reasons. First, the combination of higher-order functions with quantum phenomena raises the prospect of *entangled functions*. Certain well-known quantum phenomena can be naturally described in

terms of entangled functions, and we will give some examples of this in Section 1.2.

Another interesting aspect of higher-order quantum computation is the interplay between classical objects and quantum objects in a higher-order context. A priori, quantum computation operates on two distinct kinds of data: classical data, which can be read, written, duplicated, and discarded as usual, and quantum data, which has state preparation, unitary maps, and measurements as primitive operations. The higher-order computational paradigm introduces a third kind of data, namely functions, and one may ask whether functions behave like classical data, quantum data, or something intermediate.

The answer is that there will actually be two kinds of functions: those that behave like "quantum" objects, and those that behave like "classical" objects. Functions of the first kind carry the potential to be entangled, and can only be used once. They are effectively a "quantum state with an interface". Functions of the second kind carry no possibility of being entangled, and can be duplicated and reused at will. Interestingly, this classification of functions in not directly related to the types of their inputs and outputs: as we will see, there exist quantum-like functions that act on classical data, as well as classical-like functions that act on quantum data. In Section 1.3.2, we will discuss a type system by which one can mechanically determine, for every expression of the quantum lambda calculus, which of these categories it falls into.

When designing a quantum lambda calculus, one naturally has to make a number of choices. We do not attempt to present all possible design choices that could have been made; instead, we present one set of coherent choices that we think is particularly useful, with occasional comments on possible alternatives. Various aspects of our quantum lambda calculus and its semantics were developed in (Valiron, 2004a,b; Selinger and Valiron, 2005, 2006, 2008b,a; Valiron, 2008). We briefly discuss some of the main design choices, as well as related work, in Section 1.3.8.

The remainder of this paper is organized as follows. In Section 1.2, we give some examples of the use of higher-order functions in quantum computation. In Section 1.3, we introduce the quantum lambda calculus, its syntax, type system, and operational semantics. A type inference algorithm is described in Section 1.4. We discuss an extension of the language with infinite data types in Section 1.5. Finally, in Section 1.6 we discuss the category-theoretic semantics of the quantum lambda calculus.

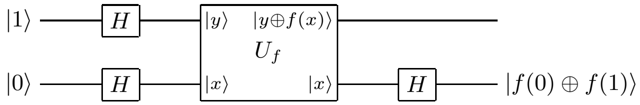

Fig. 1.1. The Deutsch-Jozsa algorithm.

Errata. This version of the paper differs from the published version. A typo in Definition 1.6.21 has been corrected.

1.2 Higher-order quantum computation examples

At the heart of the higher-order computation paradigm is the idea that "functions are data". As such, functions can appear in any context in which data is normally used. In particular, they can appear as inputs to other functions, appear as outputs of other functions, or be stored in data structures (such as a pair of functions, or a list of functions). An important restriction of higher-order computation is that all functions must be accessed via their interface. In other words, a program may interact with a function-as-data by applying it to an argument, but not, for instance, by examining its code.

We give some examples illustrating how some common phenomena in quantum computation can be interpreted in terms of higher-order functions. In these examples, we will informally use some concepts (such as types) that will be more rigorously defined in later sections.

1.2.1 The Deutsch-Jozsa algorithm

Maybe the easiest example to interpret in the context of higher-order is the Deutsch-Jozsa algorithm (Deutsch and Jozsa, 1992). In its simplest form, this algorithm decides whether a boolean function $f : \mathit{bit} \to \mathit{bit}$ is balanced or constant. It does so by interacting with an encoding of f, and uses this encoding only once (contrary to the classical case, where two calls to f are needed).

The function f must be provided in the form of a unitary 2-qubit circuit U_f, where $U_f(|x\rangle \otimes |y\rangle) = |x\rangle \otimes |y \oplus f(x)\rangle$. The algorithm constructs and runs the circuit shown in Figure 1.1. It measures the indicated qubit and returns $f(0)+f(1)$. As the Deutsch-Jozsa algorithm inputs a binary gate, which is a function of type $\mathit{qbit} \otimes \mathit{qbit} \to \mathit{qbit} \otimes \mathit{qbit}$, and outputs

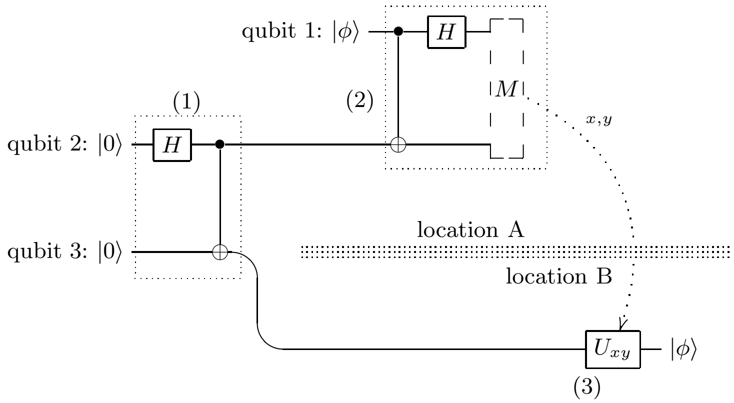

Fig. 1.2. Quantum teleportation protocol.

a classical bit, its type is:

$$(\mathit{qbit} \otimes \mathit{qbit} \to \mathit{qbit} \otimes \mathit{qbit}) \to \mathit{bit}.$$

We note that the preparation of the encoding U_f from the original boolean function f is itself a higher-order function of type

$$(\mathit{bit} \to \mathit{bit}) \to (\mathit{qbit} \otimes \mathit{qbit} \to \mathit{qbit} \otimes \mathit{qbit}).$$

However, this latter function must still call f twice (or else, it must examine the implementation of f, which is not allowed in the higher-order paradigm).

1.2.2 The teleportation procedure

The teleportation algorithm (Bennett et al., 1993) is a procedure for sending a quantum bit in an unknown state $|\phi\rangle$ from a location A to a location B using two classical bits. The procedure can be reversed to send two classical bits using a quantum bit. In this case it is called the *dense coding algorithm* (Bennett and Wiesner, 1992). The teleportation procedure is summarized in Figure 1.2. The parts (1), (2) and (3) can be described functionally as follows:

(1) takes no input and outputs an EPR pair of entangled quantum bits. Its type is therefore $\top \to \mathit{qbit} \otimes \mathit{qbit}$.
(2) performs a Bell measurement on two quantum bits and outputs two classical bits x, y. Its type is thus $\mathit{qbit} \otimes \mathit{qbit} \to \mathit{bit} \otimes \mathit{bit}$.

(3) performs a correction. It takes one quantum bit, two classical bits, and outputs a quantum bit. It has a type of the form $qbit \otimes bit \otimes bit \to qbit$.

If one curries† parts (2) and (3), one gets two respective functions

$$qbit \to (qbit \to bit \otimes bit), \qquad qbit \to (bit \otimes bit \to qbit).$$

Tensoring them and composing with (1) yields a map

$$\top \to qbit \otimes qbit \to (qbit \to bit \otimes bit) \otimes (bit \otimes bit \to qbit).$$

That is, the teleportation algorithm produces a pair of entangled functions

$$f : qbit \to bit \otimes bit, \qquad g : bit \otimes bit \to qbit,$$

such that $g(f(|\phi\rangle)) = |\phi\rangle$ for all qubits $|\phi\rangle$, and $f(g(x,y)) = (x,y)$ for all booleans x and y. These two functions are each other's inverse, but because they contain an embedded qubit each, they can only be used once. They can be said to form a "single-use isomorphism" between the (otherwise non-isomorphic) types $qbit$ and $bit \otimes bit$.

Note that the two functions f and g are entangled, since they still share the created EPR pair. One can also identify the "location" of each function: f is located at B while g is located at A. The tensor "$\otimes$" acts as "space-like" separation.

1.2.3 Bell's experiment

Recall Bell's experiment (Bell, 1964). Two quantum bits A and B are created in the maximally entangled state $|\phi_{AB}\rangle = \frac{1}{\sqrt{2}}(|0\rangle \otimes |1\rangle - |1\rangle \otimes |0\rangle)$. These qubits are sent to Alice and Bob, respectively. Suppose that Alice and Bob can independently choose one of the following three bases $\{a, b, c\}$ for measuring their qubit:

$$|0_a\rangle = |0\rangle,\ |0_b\rangle = \frac{1}{2}|0\rangle + \frac{\sqrt{3}}{2}|1\rangle,\ |0_c\rangle = \frac{1}{2}|0\rangle - \frac{\sqrt{3}}{2}|1\rangle,$$
$$|1_a\rangle = |1\rangle \quad |1_b\rangle = \frac{\sqrt{3}}{2}|0\rangle - \frac{1}{2}|1\rangle,\ |1_c\rangle = \frac{\sqrt{3}}{2}|0\rangle + \frac{1}{2}|1\rangle.$$

The question is to compute the probability of obtaining the same output when measuring A and B with respect to each of the nine possible choices of a pair of bases.

† Consider a function of two arguments $h : A \times B \to C$. Currying h means building a function $h' : A \to (B \to C)$ as follows: h' is the function that outputs the function $x \mapsto h(a,x)$ when given the input a.

One can interpret this experiment in the context of higher-order quantum computation. First, the state preparation can be viewed as a map $\mathbf{EPR} : \top \to qbit \otimes qbit$. Then each of Alice and Bob takes one qubit, chooses a basis, and measures: the measurement they perform is then a function $f : qbit \otimes trit \to bit$, where $trit = \{0,1,2\}$ is a classical three-element type. One can curry this function to $f' : qbit \to (trit \to bit)$.

The Bell experiment can be viewed as the composition

$$\top \xrightarrow{\mathbf{EPR}} qbit \otimes qbit \xrightarrow{f' \otimes f'} (trit \to bit) \otimes (trit \to bit),$$

which produces a term of type $(trit \to bit) \otimes (trit \to bit)$, i.e., a pair $\langle f, g\rangle$ of entangled functions. Although this final type is purely classical, the Bell inequalities prove that there is no classical probabilistic pair of functions exhibiting the same behavior, namely, such that $f(x) = g(y)$ with probability 1 if $x = y$, but with probability 1/4 if $x \neq y$.

1.3 Design of a typed quantum lambda calculus

In this section, we describe the design of a lambda calculus for quantum computation in which one can, in particular, interpret the algorithms described in Section 1.2.

1.3.1 Terms

We begin by defining a generic lambda calculus to support the following programming constructs: higher-order functions, tuples, disjoint unions, and recursive function definitions.

Definition 1.3.1 We define a set of *lambda terms* by the following abstract syntax:

$$\begin{aligned} M, N, P \ ::=\ & c \mid x \mid \lambda x.M \mid MN \mid \\ & \langle M, N\rangle \mid * \mid \mathit{let}\ \langle x, y\rangle = M\ \mathit{in}\ N \mid \\ & \mathit{inj}_l(M) \mid \mathit{inj}_r(M) \mid \mathit{match}\ P\ \mathit{with}\ (x \mapsto M \mid y \mapsto N) \mid \\ & \mathit{let\ rec}\ f\ x = M\ \mathit{in}\ N. \end{aligned}$$

Here, c ranges over a given set of term constants (to be instantiated below), the symbol x ranges over a fixed given infinite set of *term variables.*

The intended meaning of the different terms is as follows. The term $\lambda x.M$ denotes the function $x \mapsto M$. For example, $\lambda x.x$ is the identity function. The term MN stands for the application of the function M to the argument N. The term $\langle M, N\rangle$ represents a pair, and $*$ is the unique 0-tuple. The term $let\ \langle x, y\rangle = M\ in\ N$ is the program which first evaluates M to compute a pair $\langle V, W\rangle$, then assigns V to x and W to y before executing N. The terms $inj_l(M)$ and $inj_r(M)$ denote the left and right inclusion function of data into a disjoint union, whereas the term $match\ P\ with\ (x \mapsto M \mid y \mapsto N)$ evaluates P to an element of a disjoint union, and then proceeds by case distinction depending on whether P is of the form $inj_l(x)$ or $inj_r(y)$. Finally, the term $let\ rec\ f\ x = M\ in\ N$ defines a recursive function $f(x) = M$ before evaluating N.

The notions of free and bound variables, of substitution and of α-equivalence are defined in a standard way (see e.g. Barendregt, 1984). We identify terms up to renaming of bound variables, so for instance, $\lambda x.x$ and $\lambda y.y$ are regarded as equal.

Convention 1.3.2 The set of terms defined in Definition 1.3.1 is somewhat spartan. Following common practice, we will use the following shorthand notations to make terms more readable:

$$
\begin{aligned}
\lambda x_1 \ldots x_n.M &= \lambda x_1.\lambda x_2.\ldots.\lambda x_n.M \\
M_1 M_2 \ldots M_n &= (\ldots((M_1 M_2)M_3)\ldots M_n) \\
\langle M_1, \ldots, M_n\rangle &= \langle M_1, \langle M_2, \ldots\rangle\rangle \\
\lambda\langle x, y\rangle.M &= \lambda z.(let\ \langle x, y\rangle = z\ in\ M) \\
\lambda *.M &= \lambda z.M \quad \text{(where z is fresh)}, \\
let\ x = M\ in\ N &= (\lambda x.N)M \\
let\ * = M\ in\ N &= (\lambda z.N)M \quad \text{(where z is fresh)} \\
let\ f\ y_1 \ldots y_n = M\ in\ N &= let\ f = (\lambda y_1 \ldots y_n.M)\ in\ N \\
let\ rec\ f\ x\ y_1 \ldots y_n = M\ in\ N &= let\ rec\ f\ x = (\lambda y_1 \ldots y_n.M)\ in\ N \\
0 &= inj_r(*) \\
1 &= inj_l(*) \\
if\, P\ then\ M\ else\ N &= match\ P\ with\ (x \mapsto M \mid y \mapsto N) \\
& \qquad \text{(where x, y are fresh)}
\end{aligned}
$$

We now specify a set of term constants specific to quantum computation. The constants are:

- *new* is a function for state preparation. It inputs a classical bit (i.e., 0 or 1 as defined in Convention 1.3.2), and outputs a quantum bit prepared in state $|0\rangle$ or $|1\rangle$, respectively.

- *meas* is a function for measurement. It inputs a quantum bit, measures it in the standard basis $\{|0\rangle, |1\rangle\}$, and outputs the result as a classical bit.
- a set of constants U, ranging over some fixed universal set of unitary gates of varying arities.

The set of constants can be expanded when it is convenient to do so, for example to include new language features. We will make use of this prerogative in Section 1.5.1.

Example 1.3.3 In this language, one can write lambda terms that compute quantum algorithms. For example, a fair coin can be implemented as $\mathbf{c} = \lambda *.\, meas(H(new\, 0))$, where H is the Hadamard gate. More examples will be given in Section 1.3.4.

1.3.2 Types

Clearly, not every term defined in Section 1.3.1 makes sense in every possible context. For example, the term MN only makes sense if M is a function; the term $let\ \langle x, y\rangle = M\ in\ N$ only makes sense when M is a pair; the term $match\ P\ with\ (x \mapsto M \mid y \mapsto N)$ only makes sense when P is a member of a disjoint union; and the term $meas\ M$ only makes sense if M is a quantum bit.

Moreover, a term such as $\lambda x.\langle x, x\rangle$ only makes sense when x is a duplicable value, such as a classical bit, and not, for example, a quantum bit (which cannot be duplicated by the no-cloning theorem).

As is common in programming languages, we define a type system to rule out terms that "don't make sense". An important feature of a static type system is that well-typedness can be checked when a term is written, and not when it is executed. Another useful feature of some type systems is that types can be inferred automatically, rather than having to be explicitly specified by the programmer. As we will see, the type system we are about to introduce has all of the above properties.

Informally, and as a first approximation, it is useful to think of a type as a set of values. For example, *qbit* is the set of all one-qubit states, whereas *bit* is the set of classical booleans. $A \otimes B$ is the set of (possibly entangled) pairs of an element of type A and and element of type B. $\top$ is a singleton type, and $A \oplus B$ is the disjoint union of A and B. Since we are interested in higher-order computation, the functions from A to B also form a type, which we write as $A \multimap B$. Finally, we write $!A$ for the subset of values of type A that have the additional property of being re-usable (i.e., duplicable, cloneable).

The type systems is based on intuitionistic linear logic (Girard, 1987), and is formally defined as follows.

Definition 1.3.4 The set of *types* for the quantum lambda calculus is given by the following abstract syntax.

$$\mathit{Type}\ A, B \ ::=\ \mathit{qbit} \mid\, !A \mid (A \multimap B) \mid \top \mid (A \otimes B) \mid (A \oplus B).$$

We call the operator "!" the *exponential*.

Convention 1.3.5 We write $!^n A$ for $!!!\ldots!!A$, with n repetitions of !. We also use the notation $A^{\otimes n}$ for the n-fold tensor product

$$A \otimes \ldots \otimes A = (\ldots(A \otimes A)\ldots \otimes A).$$

Similarly, we write $A^{\oplus n}$ for the n-fold sum:

$$A \oplus \ldots \oplus A = (\ldots(A \oplus A)\ldots \oplus A).$$

Finally, we use the shorthand notation $\mathit{bit} = \top \oplus \top$.

When it is convenient, the set of types can be extended to support new language features. For example, in Section 1.5.1, we will define a type $\mathit{list}(A)$ of list of elements of type A.

1.3.3 Typing rules

We have stated informally that $!A$ is the set of values of type A that have the additional property of being re-usable. Consequently, any term that has type $!A$ should automatically also have type A. We say that $!A$ is a *subtype* of A. The concept of a subtype also extends to composite types, for instance, any function of type $A \multimap B$ that can accept an argument of type A can also accept a re-usable argument of type A. Therefore, $A \multimap B$ is a subtype of $!A \multimap B$. This motivates the following definition:

Definition 1.3.6 We define the *subtyping relation* $<:$ to be the smallest relation on types satisfying the rules in Table 1.1, using the overall condition on n and m that $(m = 0) \vee (n \geqslant 1)$.

Note that the rules in Table 1.1 should be read as follows: if the premises (above the horizontal line) are all true, then the conclusion (below the horizontal line) is true.

$$\frac{}{!^n\,\mathit{qbit} <: !^m\,\mathit{qbit}}\ (\mathit{qbit}), \qquad \frac{}{!^n \top <: !^m \top}\ (\top),$$

$$\frac{A_1 <: B_1 \quad A_2 <: B_2}{!^n(A_1 \otimes A_2) <: !^m(B_1 \otimes B_2)}\ (\otimes), \qquad \frac{A <: A' \quad B <: B'}{!^n(A' \multimap B) <: !^m(A \multimap B')}\ (\multimap),$$

$$\frac{A_1 <: B_1 \quad A_2 <: B_2}{!^n(A_1 \oplus A_2) <: !^m(B_1 \oplus B_2)}\ (\oplus).$$

Table 1.1. *Subtyping relation*

Lemma 1.3.7 *The subtyping relation $<:$ is reflexive and transitive.*

Lemma 1.3.8 *If $A <: !B$, then $A = !A'$ for some type A'. Dually, if A is not of the form $!A'$ and if $A <: B$, then B is not of the form $!B'$.*

The purpose of the type system is to assign a (not necessarily unique) type to each well-formed term, with the goal that a well-typed term can never perform an illegal operation. We write $M : A$ to indicate that the term M is well-typed of type A. In general, it is not sufficient to reason about judgements of the form $M : A$, because the well-typedness of a term M also depends on the types of all of the free variables that appear in M. We therefore introduce the notion of a typing context.

Definition 1.3.9 A *typing context* is a finite set $\{x_1 : A_1, \ldots, x_n : A_n\}$ of pairs of a variable and a type, such that no variable occurs more than once. We often write Δ for a typing context, and we write $|\Delta| = \{x_1, \ldots, x_n\}$ and $\Delta(x_i) = A_i$. We also write $!\Delta$ for a context of the form $\{x_1 : !A_1, \ldots, x_n : !A_n\}$. If $|\Delta|$ and $|\Delta'|$ are disjoint, we write Δ, Δ' for the union of two typing contexts. Moreover, we extend the subtyping relation to contexts as follows. We write $\Delta <: \Delta'$ if and only if $|\Delta| = |\Delta'|$ and for all $x \in |\Delta|$, $\Delta(x) <: \Delta'(x)$.

Definition 1.3.10 A *typing judgement* is an expression of the form $\Delta \rhd M : B$, where Δ is a typing context, M is a term, and B is a type. A typing derivation is called *valid* if it can be inferred from the rules in Table 1.2.

In the table, the symbol c ranges over the set of term constants $\{\mathit{meas}, \mathit{new}, U\}$. To each constant c we associate a type A_c, as follows:

$$A_{\mathit{new}} = \mathit{bit} \multimap \mathit{qbit}, \quad A_U = \mathit{qbit}^{\otimes n} \multimap \mathit{qbit}^{\otimes n}, \quad A_{\mathit{meas}} = \mathit{qbit} \multimap !\mathit{bit}.$$

$$\frac{A <: B}{\Delta, x : A \rhd x : B}\ (ax_1) \qquad \frac{!A_c <: B}{\Delta \rhd c : B}\ (ax_2)$$

$$\frac{\Delta \rhd M : !^n A}{\Delta \rhd inj_l(M) : !^n(A \oplus B)}\ (\oplus.I_1) \qquad \frac{\Delta \rhd N : !^n B}{\Delta \rhd inj_r(N) : !^n(A \oplus B)}\ (\oplus.I_2)$$

$$\frac{!\Delta, \Gamma_1 \rhd P : !^n(A \oplus B) \quad \begin{array}{c} !\Delta, \Gamma_2, x : !^n A \rhd M : C \\ !\Delta, \Gamma_2, y : !^n B \rhd N : C \end{array}}{\Gamma_1, \Gamma_2, !\Delta \rhd match\ P\ with\ (x \mapsto M \mid y \mapsto N) : C}\ (\oplus.E)$$

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

$$\frac{x : A, \Delta \rhd M : B}{\Delta \rhd \lambda x.M : A \multimap B}\ (\lambda_1) \qquad \frac{\begin{array}{c}\text{If } FV(M) \cap |\Gamma| = \emptyset: \\ \Gamma, !\Delta, x : A \rhd M : B\end{array}}{\Gamma, !\Delta \rhd \lambda x.M : !^{n+1}(A \multimap B)}\ (\lambda_2)$$

$$\frac{!\Delta, \Gamma_1 \rhd M_1 : !^n A_1 \quad !\Delta, \Gamma_2 \rhd M_2 : !^n A_2}{!\Delta, \Gamma_1, \Gamma_2 \rhd \langle M_1, M_2 \rangle : !^n(A_1 \otimes A_2)}\ (\otimes.I) \qquad \frac{}{\Delta \rhd * : !^n \top}\ (\top)$$

$$\frac{!\Delta, \Gamma_1 \rhd M : !^n(A_1 \otimes A_2) \quad !\Delta, \Gamma_2,\ x_1{:}!^n A_1,\ x_2{:}!^n A_2 \rhd N : A}{!\Delta, \Gamma_1, \Gamma_2 \rhd \text{let } \langle x_1, x_2 \rangle = M \text{ in } N : A}\ (\otimes.E)$$

$$\frac{!\Delta, f : !(A \multimap B), x : A \rhd M : B \quad !\Delta, \Gamma, f : !(A \multimap B) \rhd N : C}{!\Delta, \Gamma \rhd let\ rec\ f\ x = M\ in\ N : C}\ (rec)$$

Table 1.2. *Typing rules*

Lemma 1.3.11 *The following are derived rules:*

$$\frac{}{\rhd 0 : !^n bit,}\ (bit.I_1) \qquad \frac{}{\rhd 1 : !^n bit,}\ (bit.I_2)$$

$$\frac{!\Delta, \Gamma_1 \rhd P : bit \quad !\Delta, \Gamma_2 \rhd M, N : A}{!\Delta, \Gamma_1, \Gamma_2 \rhd if\,P\ then\ M\ else\ N : A.}\ (bit.E)$$

Proof The proof is straightforward using the typing rules and Conventions 1.3.2 and 1.3.5. □

Remark 1.3.12 The type $!A_c$ is understood as being the "most generic" one for c, as enforced by the typing rule (ax_2). For example, we defined A_{new} to be $bit \multimap qbit$. Since by the rule (ax_2), *new* can take any type B such that $!A_c <: B$, the term *new* can be typed with any type in the

following poset:

$$\begin{tikzcd}[column sep=small, row sep=tiny] & !(!bit \multimap qbit) \arrow[dr, dash, "<:"] & \\ !(bit \multimap qbit) \arrow[ur, dash, "<:"] \arrow[dr, dash, "<:"'] & & !bit \multimap qbit. \\ & bit \multimap qbit \arrow[ur, dash, "<:"'] & \end{tikzcd}$$

This implies, as expected, that no created quantum bit can have the type $!qbit$.

Remark 1.3.13 Note that the type system enforces the requirement that variables holding quantum data cannot be freely duplicated; thus $\lambda x.\langle x, x\rangle$ is not a valid term of type $qbit \multimap qbit \otimes qbit$. On the other hand, we allow variables to be discarded freely.

Note also that due to rule (λ_2) the term $\lambda x.M$ is duplicable only if all the free variables of M (other that x) are duplicable. This answers the question, raised in the introduction, what it means for a higher-order function to be "classical". A function is classical (and therefore re-usable) if and only if it does not contain any embedded non-duplicable data. This definition is of course recursive, in a way that is made precise by the typing rules.

1.3.4 Examples

Example 1.3.14 The term $\mathbf{c} = \lambda *.\ meas(H(new\,0))$ of Example 1.3.3 can be typed as $\rhd \mathbf{c} : \top \multimap bit$.

Example 1.3.15 Consider the term

$$M \quad = \quad let\ rec\ f\,x = (if\,(\mathbf{c}\,*)\ then\ H\,(f\,x)\ else\ x)\ in\ f\,p,$$

where $\mathbf{c}$ is the fair coin of Example 1.3.14. Intuitively, this term applies the Hadamard gate a random number of times to a qubit p. We claim that $p : qbit \rhd M : qbit$ is a valid typing judgement. Indeed, assuming that $\rhd\ \mathbf{c} : \top \multimap bit$ is a valid typing judgement, one can write the following typing derivation:

1	(ax_1)	$f : !(qbit \multimap qbit), x : qbit \rhd x : qbit$
2	$(\multimap)$	$!(qbit \multimap qbit) <: qbit \multimap qbit$
3	$(2, ax_1)$	$f : !(qbit \multimap qbit) \rhd f : qbit \multimap qbit$
4	$(3, 1, app)$	$f : !(qbit \multimap qbit), x : qbit \rhd (f\,x) : qbit$
5	$(2, ax_2)$	$\rhd H : qbit \multimap qbit$

6	$(5, 4, app)$	$f : !(qbit \multimap qbit), x : qbit \rhd H\,(f\,x) : qbit$
7	(Ex.1.3.14)	$\rhd\, \mathbf{c} : \top \multimap bit$
8	$(\top.I)$	$\rhd\, * : \top$
9	$(7, 8, app)$	$\rhd\, \mathbf{c}\,* : bit$
10	$(9, 6, 1, if)$	$f : !(qbit \multimap qbit), x : qbit \rhd$ $if\,(\mathbf{c}\,*)\ then\ H\,(f\,x)\ else\ x : qbit$
11	(ax_1)	$p : qbit \rhd p : qbit$
12	$(3, 11, app)$	$p : qbit, f : !(qbit \multimap qbit) \rhd f\ p : qbit$
13	$(10, 12, rec)$	$p : qbit \rhd let\ rec\ f\,x = if\,(\mathbf{c}\,*)\ then\ H\,(f\,x)\ else\ x)\ in\ f\ p : qbit$

Here, we have used the notation (x,y,z,R) for "The rule R is used with hypothesis lines x, y and z". The rule ($\multimap$) refers to Table 1.1.

Example 1.3.16 Consider the Deutsch-Jozsa algorithm introduced in Section 1.2.1. We claimed that one can interpret it as a term of type

$$(qbit \otimes qbit \multimap qbit \otimes qbit) \to bit.$$

Here is an implementation in the quantum lambda calculus, using the notations of Convention 1.3.2:

$$\begin{aligned}
&let\ \mathbf{Deutsch}\ U_f\ = \\
&\quad let\ \mathbf{tens}\ f\ g\ = \lambda\langle x, y\rangle.\langle fx, gy\rangle\ in \\
&\quad let\ \langle x, y\rangle = (\mathbf{tens}\ H\ (\lambda x.x))(U_f\langle H(new\ 0), H(new\ 1)\rangle)\ in \\
&\qquad meas\ x,
\end{aligned}$$

One can check that

$$\rhd\ \mathbf{Deutsch} : !((qbit \otimes qbit \multimap qbit \otimes qbit) \multimap bit)$$

is a well-typed typing judgement. Note that **Deutsch** is duplicable, and that U_f does not need to be duplicable, since it is used only once.

Example 1.3.17 The teleportation algorithm can similarly be defined. Part (1) is the function $\mathbf{EPR} : !(\top \multimap (qbit \otimes qbit))$, defined as

$$\mathbf{EPR} = \lambda x.CNOT\langle H(new\,0), new\,0\rangle.$$

Part (2) is the function $\mathbf{BellMeasure} : !(qbit \multimap (qbit \multimap bit \otimes bit))$, defined as

$$\begin{aligned}
&\mathbf{BellMeasure} = \\
&\qquad \lambda q_2.\lambda q_1.(let\ \langle x, y\rangle = CNOT\langle q_1, q_2\rangle\ in\ \langle meas(Hx), meas\ y\rangle.
\end{aligned}$$

Part (3) is the function $\mathbf{U} : !(qbit \multimap (bit \otimes bit \multimap qbit))$ defined as

$$\begin{aligned}
\mathbf{U} = \lambda q.\lambda\langle x, y\rangle.if\ x\ then\ &(if\ y\ then\ U_{11}q\ else\ U_{10}q) \\
else\ &(if\ y\ then\ U_{01}q\ else\ U_{00}q).
\end{aligned}$$

The teleportation procedure as described in Section 1.2.2 can be written with the following code:

$$\begin{aligned}
\mathbf{telep} = \quad &let\ \langle x, y\rangle = \mathbf{EPR}\ *\ in \\
&let\ f = \mathbf{BellMeasure}\ x\ in \\
&let\ g = \mathbf{U}\ y \\
&in\ \langle f, g\rangle.
\end{aligned}$$

It can then be shown that

$$\rhd\ \mathbf{telep} : (qbit \multimap bit \otimes bit) \otimes (bit \otimes bit \multimap qbit)$$

is a valid typing judgement (for details, see Selinger and Valiron, 2006).

1.3.5 The evaluation of terms: operational semantics

So far, we have concentrated on the static properties of lambda terms, and have only informally described their intended behavior. In this section, we will present the formal computation rules by which terms are evaluated.

Our informal description of the behavior of lambda terms has left many details undefined. For example, when evaluating a pair of terms $\langle M, N\rangle$, we have not specified whether M or N will be evaluated first. Similarly, in terms such as $let\ x = M\ in\ N$, we have not specified whether M should be evaluated immediately, or whether it should be re-evaluated each time the term N needs to access the value of the variable x. Such seemingly arbitrary choices actually can affect the outcome of the computation, as the following example shows.

Example 1.3.18 Consider the boolean addition function, defined as

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

Now consider the term

$$let\ x = \mathbf{c}*\ in\ \mathbf{plus}\,x\,x,$$

where $\mathbf{c}$ is the fair coin from Example 1.3.3. If we evaluate $\mathbf{c}*$ each time the variable x is needed, we obtain $\mathbf{plus}(\mathbf{c}*)(\mathbf{c}*)$, which will be 0 or 1 with equal probability. On the other hand, if we evaluate $\mathbf{c}*$ ahead of time, we obtain either $\mathbf{plus}\,0\,0$ or $\mathbf{plus}\,1\,1$, and in either case the final answer is 0. The former evaluation method is called the *call-by-name* method, and the latter is called the *call-by-value* method.

One of the reasons we give a formal definition of the behavior of lambda terms is to resolve such imprecisions. Thus, we will have to make a specific choice in each instance where an ambiguity might arise. For example, we will be choosing the call-by-value reduction strategy. Another reason for making a formal definition is that this will allow us to prove meta-theorems about the behavior of quantum lambda terms. For example, we will prove in Section 1.3.7 that a well-typed lambda term never executes an illegal operation such as attempting to clone a quantum bit.

Definition 1.3.19 A *quantum closure* (or *state*) is a tuple $[Q, L, M]$, where

- Q is a normalized vector of $\otimes_{i=1}^{n}\mathbb{C}^2$, for some integer $n \geqslant 0$. The vector Q is called the *quantum array*;
- L is a list of n distinct term variables, written as $|x_1, \ldots, x_n\rangle$.
- M is a lambda term whose free variables all appear in L.

We write $|L| = \{x_1, \ldots, x_n\}$, and we also write $L(x_i) = i$ for the position of a variable x_i in the list.

The purpose of a quantum closure is to provide a mechanism to talk about terms with embedded quantum data. The idea is that the variable x_i is bound in the term M to qubit number $L(x_i)$ of the state Q. So for example, the quantum closure

$$[\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \quad |pq\rangle, \quad \lambda x.xpq]$$

denotes a term $\lambda x.xpq$ with two embedded qubits p, q in the entangled state $|pq\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

The notion of α-equivalence extends naturally to quantum closures, for instance, the states $[Q, |x\rangle, \lambda y.x]$ and $[Q, |z\rangle, \lambda y.z]$ are equivalent. We identify quantum closures up to renaming of bound variables.

$$\begin{array}{c}
[\,Q, L, let\ x = V\ in\ M\,] \to_1 [\,Q, L, M[V/x]\,] \\
[\,Q, L, let\ \langle x, y\rangle = \langle V, W\rangle\ in\ N\,] \to_1 [\,Q, L, N[V/x, W/y]\,] \\
[\,Q, L, match\ inj_l(V)\ with\ (x \mapsto M \mid y \mapsto N)\,] \to_1 [\,Q, L, M[V/x]\,] \\
[\,Q, L, match\ inj_r(W)\ with\ (x \mapsto M \mid y \mapsto N)\,] \to_1 [\,Q, L, N[W/y]\,] \\
[\,Q, L, let\ rec\ f\ x = M\ in\ N\,] \\
\to_1 [\,Q, L, N[\lambda x.(let\ rec\ f\ x = M\ in\ M)\ /\ f]\,]
\end{array}$$

Table 1.3. *Reductions rules: classical control*

$$\begin{array}{c}
[\,Q, |x_1 \ldots x_n\rangle, U\langle x_{j_1}, \ldots, x_{j_n}\rangle\,] \to_1 [\,Q', |x_1 \ldots x_n\rangle, \langle x_{j_1}, \ldots, x_{j_n}\rangle\,] \\
[\,\alpha|Q_0\rangle + \beta|Q_1\rangle, |x_1 \ldots x_n\rangle, meas\ \ x_i\,] \to_{|\alpha|^2} [\,|Q_0\rangle, |x_1 \ldots x_n\rangle, 0\,] \\
[\,\alpha|Q_0\rangle + \beta|Q_1\rangle, |x_1 \ldots x_n\rangle, meas\ \ x_i\,] \to_{|\beta|^2} [\,|Q_1\rangle, |x_1 \ldots x_n\rangle, 1\,] \\
[\,Q, |x_1 \ldots x_n\rangle, new\ \ 0\,] \to_1 [\,Q \otimes |0\rangle, |x_1 \ldots x_{n+1}\rangle, x_{n+1}\,] \\
[\,Q, |x_1 \ldots x_n\rangle, new\ \ 1\,] \to_1 [\,Q \otimes |1\rangle, |x_1 \ldots x_{n+1}\rangle, x_{n+1}\,]
\end{array}$$

Table 1.4. *Reductions rules: quantum data*

The evaluation of quantum lambda terms will be defined as a probabilistic rewrite procedure on quantum closures, using a call-by-value reduction strategy.

Definition 1.3.20 A *value* is a term defined with the following grammar.

$$Value\ V, W \ ::= \ c \mid x \mid \lambda x.M \mid \ inj_r\,V \mid \ inj_l\,V \mid * \mid \langle V, W\rangle$$

The quantum closure $[Q, L, V]$ is called a *value state* if V is a value.

Definition 1.3.21 The reduction rules are shown in Tables 1.3–1.5. We write $[Q, L, M] \to_p [Q', L', M']$ for a single-step reduction of quantum closures that takes place with probability p. In the rules for *let*, *let rec*, and *match*, $M[V/x]$ denotes the term M where the free variable x has been replaced by V (renaming bound variables as necessary). In the rule for reducing the term $U\langle x_{j_1}, \ldots, x_{j_n}\rangle$, U is an n-ary built-in unitary gate, $j_1, \ldots, j_n$ are pairwise distinct, and Q' is the quantum state obtained from Q by applying this gate to qubits $j_1, \ldots, j_n$. In the

$$\frac{[\,Q, L, N\,] \to_p [\,Q', L', N'\,]}{[\,Q, L, MN\,] \to_p [\,Q', L, MN'\,]}$$

$$\frac{[\,Q, L, M\,] \to_p [\,Q', L', M'\,]}{[\,Q, L, MV\,] \to_p [\,Q', L', M'V\,]}$$

$$\frac{[\,Q, L, M_2\,] \to_p [\,Q', L', M_2'\,]}{[\,Q, L, \langle M_1, M_2\rangle\,] \to_p [\,Q', L', \langle M_1, M_2'\rangle\,]}$$

$$\frac{[\,Q, L, M_1\,] \to_p [\,Q', L', M_1'\,]}{[\,Q, L, \langle M_1, V_2\rangle\,] \to_p [\,Q', L', \langle M_1', V_2\rangle\,]}$$

$$\frac{[\,Q, L, M\,] \to_p [\,Q', L', M'\,]}{[\,Q, L, inj_l(M)\,] \to_p [\,Q', L', inj_l(M')\,]}$$

$$\frac{[\,Q, L, M\,] \to_p [\,Q', L', M'\,]}{[\,Q, L, inj_r(M)\,] \to_p [\,Q', L', inj_r(M')\,]}$$

$$\frac{[\,Q, L, P\,] \to_p [\,Q', L', P'\,]}{[\,Q, L, match\ P\ with \ldots\,] \to_p [\,Q', L', match\ P'\ with \ldots\,]}$$

$$\frac{[\,Q, L, M\,] \to_p [\,Q', L', M'\,]}{[\,Q, L, let\ \langle x, y\rangle = M\ in\ N\,] \to_p [\,Q', L', let\ \langle x, y\rangle = M'\ in\ N\,]}$$

Table 1.5. *Reductions rules: congruence rules*

rule for measurement, $|Q_0\rangle$ and $|Q_1\rangle$ are normalized states of the form

$$|Q_0\rangle = \sum_j \alpha_j |\phi_j^0\rangle \otimes |0\rangle \otimes |\psi_j^0\rangle, \qquad |Q_1\rangle = \sum_j \beta_j |\phi_j^1\rangle \otimes |1\rangle \otimes |\psi_j^1\rangle,$$

where ϕ_j^0 and ϕ_j^1 are i-qubit states (so that the measured qubit is the one pointed to by x_i). In the rule for *new*, Q is an n-qubit state, so that $Q \otimes |i\rangle$ is an $(n+1)$-qubit state.

We write $\to$ for the reduction $\to_1$, and $\to^*$ for the reflexive, transitive closure of $\to$.

Note that the only probabilistic reduction step is the one corresponding to measurement.

Example 1.3.22 Consider the term

$$M \;=\; let\ rec\ f\,x = (if\,(\mathbf{c}\,*)\ then\ H\,(f\,x)\ else\ x)\ in\ f\,p$$

from Example 1.3.15. Let us write

$$\begin{aligned} P &= if\,(\mathbf{c}\,*)\ then\ H\,(f\,x)\ else\ x \\ R &= let\ rec\ f\,x = P\ in\ P. \end{aligned}$$

We have

$$\begin{aligned}
M &\to [\,|0\rangle, |p\rangle, (f\ p)[\lambda x.R\,/\,f]\,] \\
&\to [\,|0\rangle, |p\rangle, (\lambda x.R)p\,] \\
&\to [\,|0\rangle, |p\rangle, let\ rec\ f\,x = P\ in\ (if\,(\mathbf{c}\,*)\ then\ H\,(f\,p)\ else\ p)\,] \\
&\to [\,|0\rangle, |p\rangle, (if\,(\mathbf{c}\,*)\ then\ H\,(f\,p)\ else\ p)[\lambda x.R\,/\,f]\,] \\
&\to [\,|0\rangle, |p\rangle, if\,(\mathbf{c}\,*)\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \\
&= [\,|0\rangle, |p\rangle, if\,(meas(H(new\,0)))\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \\
&\to [\,|00\rangle, |pq\rangle, if\,(meas(H\ q))\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \\
&\to [\,\tfrac{1}{\sqrt{2}}(|00\rangle + |01\rangle), |pq\rangle, if\,(meas\ \ q)\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \\
&\to \begin{cases} [\,|00\rangle, |pq\rangle, if\,0\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \\ [\,|01\rangle, |pq\rangle, if\,1\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \end{cases} \\
&\to \begin{cases} [\,|00\rangle, |pq\rangle, p\,] \\ [\,|01\rangle, |pq\rangle, H\,((\lambda x.R)\,p)\,] \end{cases} \\
&\to \begin{cases} [\,|00\rangle, |pq\rangle, p\,] \\ [\,|01\rangle, |pq\rangle, H\,(letrec\ f\,x = P\ in \\ \qquad\qquad if\,(\mathbf{c}\,*)\ then\ H\,((\lambda x.R)\,p)\ else\ p)\,] \end{cases} \\
&\to^* \begin{cases} [\,|00\rangle, |pq\rangle, p\,] \\ [\,\tfrac{1}{\sqrt{2}}(|010\rangle + |011\rangle), |pqr\rangle, \\ \qquad H\ if\,(meas\,r)\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \end{cases} \\
&\to \begin{cases} [\,|00\rangle, |pq\rangle, p\,] \\ \begin{cases} [\,|010\rangle, |pqr\rangle, H\ if\,0\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \\ [\,|011\rangle, |pqr\rangle, H\ if\,1\ then\ H\,((\lambda x.R)\,p)\ else\ p\,] \end{cases} \end{cases} \\
&\to \begin{cases} [\,|00\rangle, |pq\rangle, p\,] \\ \begin{cases} [\,|010\rangle, |pqr\rangle, H\,p\,] \\ [\,|011\rangle, |pqr\rangle, H\,(H\,((\lambda x.R)\,p))\,] \end{cases} \end{cases} \\
&\to^* \begin{cases} [\,|00\rangle, |pq\rangle, p\,] \\ \begin{cases} [\,\tfrac{1}{\sqrt{2}}(|010\rangle + |110\rangle), |pqr\rangle, p\,] \\ \begin{cases} [\,|0110\rangle, |pqrs\rangle, H\,(H\,p)\,] \\ [\,|0111\rangle, |pqrs\rangle, H\,(H\,(H\,((\lambda x.R)\,p)))\,] \end{cases} \end{cases} \end{cases}
\end{aligned}$$

$$\to^* \left\{ \begin{array}{l} [\,|00\rangle, |pq\rangle, p\,] \\ \left\{ \begin{array}{l} [\,\frac{1}{\sqrt{2}}(|010\rangle + |110\rangle), |pqr\rangle, p\,] \\ \left\{ \begin{array}{l} [\,|0110\rangle, |pqrs\rangle, p\,] \\ \left\{ \begin{array}{l} [\,\frac{1}{\sqrt{2}}(|01110\rangle + |11110\rangle), |pqrst\rangle, p\,] \\ \cdots \end{array} \right. \end{array} \right. \end{array} \right. \end{array} \right.$$

where each split occurs with probability $\frac{1}{2}$, and where the result is read as the term part of the value state. On average, the term M reduces to $|0\rangle$ with probability

$$\frac{1}{2}\sum_{n=0}^{\infty} \frac{1}{2^{2n}} = \frac{1}{2}\left(\frac{1}{1-\frac{1}{4}}\right) = \frac{2}{3}$$

and to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ with probability $\frac{1}{3}$. Note that since there is no garbage collection, the quantum array is filled up with unused quantum bits. It is possible to define an operational semantics that removes unused quantum bits (see e.g. Selinger and Valiron, 2008b).

Remark 1.3.23 (Error states) The purpose of the reduction rules is to evaluate a quantum closure until a value state is reached. However, this does not always succeed: there are states (so-called *error states*) that are not values, but from which no reduction is possible. Such states correspond to run-time errors of the programming language (and in actual implementations, these would either lead to run-time error messages, or to undefined behavior). Examples of error states are the quantum closures $[Q, L, \mathit{let}\ \langle x, y\rangle = \mathit{inj}_l(M)\ \mathit{in}\ N]$, $[Q, L, \mathit{meas}(\lambda x.x)]$, and $[Q, |xyz\rangle, U\langle x, x\rangle]$.

In Sections 1.3.2 and 1.3.3, we introduced a type system precisely for the purpose of ruling out such error states. And indeed, we will show in Section 1.3.7 that well-typed closure can never lead to an error state.

1.3.6 Properties of the type system

Before we can prove type soundness of the reduction rules, we need to record some basic properties of the type system.

Lemma 1.3.24

(1) If $x \notin FV(M)$ and $\Delta, x : A \rhd M : B$, then $\Delta \rhd M : B$.
(2) If $\Delta \rhd M : A$, then $\Gamma, \Delta \rhd M : A$.
(3) If $\Gamma <: \Delta$ and $\Delta \rhd M : A$ and $A <: B$, then $\Gamma \rhd M : B$.

Proof By structural induction on the type derivation of M. □

The next lemma is crucial in the proof of the substitution lemma. Note that it is only true for a value V, and in general fails for an arbitrary term M.

Lemma 1.3.25 *If V is a value and $\Delta \rhd V : !A$, then for all $x \in FV(V)$, there exists some type B such that $\Delta(x) = !B$. Conversely, if $FV(V) = |\Delta|$ and if $!\Delta, \Gamma \rhd V : A$ is valid, so is $!\Delta, \Gamma \rhd V : !A$.*

Proof By structural induction on V. □

Lemma 1.3.26 (Substitution) *If V is a value such that $\Gamma_1, !\Delta, x : A \rhd M : B$ and $\Gamma_2, !\Delta \rhd V : A$, then $\Gamma_1, \Gamma_2, !\Delta \rhd M[V/x] : B$.*

Proof By structural induction on the derivation of the typing judgement $\Gamma_1, !\Delta, x : A \rhd M : B$, using Lemma 1.3.24. In the case (λ_2), one uses Lemma 1.3.25. □

Remark 1.3.27 Those readers familiar with intuitionistic linear logic (Girard, 1987; Troelstra, 1992) may note that all the rules of affine intuitionistic logic are derived rules of our type system, *except* for the general promotion rule

$$\frac{!\Delta \rhd M : A}{!\Delta \rhd M : !A}.$$

The absence of this rule is necessary: indeed, although the typing judgement $\rhd\ \mathit{new}\,0 : \mathit{qbit}$ is valid, the judgement $\rhd\ \mathit{new}\,0 : !\mathit{qbit}$ should not be allowed. However, as stated in Lemma 1.3.25, the promotion rule is true if M is a value. This point will be further developed in Section 1.6 when we study the categorical semantics.

1.3.7 Safety properties

In this section, we wish to show that a well-typed program cannot reach an error state. We first define what it means for a quantum closure to be well-typed:

Definition 1.3.28 Consider the quantum closure $[\,Q, L, M\,]$ where $L = |x_1 \dots x_k\rangle$. We say that $[\,Q, L, M\,]$ is *well-typed* (or *valid*) of type C, written $[Q, L, M] : C$, if $x_1 : qbit, \ldots, x_k : qbit \rhd M : C$ is a valid typing judgement. A well-typed quantum closure is also called a *program*.

The first step in proving type safety is to show that well-typedness is preserved by the reduction rules. This property is known as the *subject reduction property*. In order to get the strongest possible result, we will prove that type safety holds even in the presence of decoherence and imprecision of the physical operations. To that end, we define a notion of reachability of quantum closures that includes reduction steps of probability 0, as well as arbitrary perturbations of the quantum state.

Definition 1.3.29 The *reachability relation* $\rightsquigarrow$ is the smallest transitive reflexive relation on quantum closures such that $[Q, L, M] \rightsquigarrow [Q', L', M']$ holds whenever $[Q, L, M] \to_p [Q', L', M']$ with some probability p (even for $p = 0$), and such that $[Q, L, M] \rightsquigarrow [Q', L, M]$ whenever Q' is a normalized vector of dimension equal to that of Q.

Theorem 1.3.30 (Subject reduction) *Suppose* $[Q, L, M] : A$ *and* $[Q, L, M] \rightsquigarrow [Q', L', M']$. *Then* $[Q', L', M'] : A$.

Proof Since well-typedness is defined without reference to Q, it suffices to show that if $[Q, L, M] : A$ and if $[Q, L, M] \to_p [Q', L', M']$, then $[Q', L', M'] : A$. This is proved by induction on the derivation of the reduction, using Lemma 1.3.26 in the "*let*", "*let rec*" and "*match*" cases.

We detail one of the less immediate cases, the case *let rec*. Suppose that

$$\dfrac{\begin{matrix}\vdots\ \pi_1 \\ !\Delta, f : !(A \multimap B), x : A \rhd M : B\end{matrix} \quad \begin{matrix}\vdots\ \pi_2 \\ !\Delta, \Gamma, f : !(A \multimap B) \rhd N : C\end{matrix}}{!\Delta, \Gamma \rhd \mathit{let\ rec}\ f\ x = M\ \mathit{in}\ N : C}$$

is a valid derivation. Then so is

$$\dfrac{\dfrac{\begin{matrix}\vdots\ \pi_1 \\ !\Delta, f : !(A \multimap B), x : A \rhd M : B\end{matrix} \quad \begin{matrix}\vdots\ \pi_1 \\ !\Delta, f : !(A \multimap B), x : A \rhd M : B\end{matrix}}{!\Delta, x : A \rhd \mathit{let\ rec}\ f\ x = M\ \mathit{in}\ M : B}}{!\Delta \rhd \lambda x.(\mathit{let\ rec}\ f\ x = M\ \mathit{in}\ M) : !(A \multimap B).}$$

By Lemma 1.3.26, one concludes

$$!\Delta, \Gamma \rhd N[\lambda x.(\mathit{let\ rec}\ f\ x = M\ \mathit{in}\ M)\ /\ f] : C.$$

□

Lemma 1.3.31 *A well-typed value is either a constant, a variable, or one of the following case occurs: if it is of type* $!^n(A \oplus B)$, *it is of the form* $inj_l(V)$ *or* $inj_r(V)$ *with* V *a value; if it is of type* $!^n(A \otimes B)$, *it is of the form* $\langle V, W \rangle$ *with* V *and* W *values; if it is of type* $!^n\top$, *it is precisely the term* $*$; *if it is of type* $!^n(A \multimap B)$, *it is a lambda-abstraction.*

Proof By structural induction on the derivation of the typing judgement. □

Theorem 1.3.32 (Progress) *Let* $[Q, L, M]$ *be a program of type* B. *Then either* $[Q, L, M]$ *is a value state, or there is a program* $[Q', L', M']$ *such that* $[Q, L, M] \to_p [Q', L', M']$. *Moreover, the total probability of all possible single-step reductions from* $[Q, L, M]$ *is* 1.

Proof By structural induction on the derivation of the reduction of $[Q, L, M]$, using Lemma 1.3.31. □

Corollary 1.3.33 (Type safety) *A well-typed program does not reach an error state. In particular, any probabilistic computation path of a well-typed program is either infinite, or reaches a value state in a finite number of steps.*

Remark 1.3.34 The evaluation rules of the quantum lambda calculus were defined on untyped quantum closures. This is in line with our remark, in Section 1.3.2, that well-typedness is a property to be checked when a term is written, and not when it is executed. In other words, once it is established that a term is well-typed, it can be safely executed, without further well-typedness checks at runtime, and the type safety property guarantees that there will be no errors. In particular, the type system introduces no computational overhead in the evaluation of terms.

1.3.8 Design choices and some related work

We briefly discuss some of the design decisions underlying our quantum lambda calculus.

Classical control. Our quantum lambda calculus has *classical control*, by which we mean that it essentially fits into the QRAM paradigm of quantum computing (Knill, 1996). This means that, while programs operate on quantum data, the programs themselves are classical. For example, we do not allow superpositions of two arbitrary lambda terms. Van Tonder (2004) gives convincing evidence that this choice is essentially necessary.

Call-by-value reduction. We have equipped the quantum lambda calculus with a *call-by-value* reduction strategy, which means that the arguments to any function call are completely evaluated before the function is called. This is in contrast to *call-by-name* or *lazy* languages, which evaluate the arguments only if they are actually needed. Example 1.3.18 shows that it is necessary to choose a reduction strategy; the actual choice is largely a matter of taste.

Uniformity of data. Our language treats classical and quantum data uniformly. This means that we do not have, for example, distinct sets of classical variables and quantum variables. For example, we use the same syntax for a function, regardless of whether the function accepts a classical argument, a quantum argument, or indeed, a more complex object (such as a pair of a classical bit and a quantum bit). We use a type system to ensure well-formedness of programs.

Implicit linearity tracking. Perhaps the most important choice that we have made, which distinguishes the quantum lambda calculus from other linear lambda calculi in the literature, is that there is no explicit syntax for linearity in the untyped language. This was motivated by the desire to design a language for practical use, which would be as natural as possible to the programmer.

The literature contains a number of lambda calculi for intuitionistic linear logic. For example, the linear lambda calculi of (Abramsky, 1993) and (Benton et al., 1993b; Benton, 1995; Benton and Wadler, 1996) have typing rules such as

$$\frac{\Delta \rhd M : !A}{\Delta \rhd \mathrm{derelict}(M) : A} \qquad \frac{\Gamma \rhd M : !A \qquad \Delta, x : !A, y : !A \rhd N : B}{\Gamma, \Delta \rhd \mathrm{copy}\ M\ \mathrm{as}\ x, y\ \mathrm{in}\ N : B}$$

$$\frac{\Gamma \rhd M : !A \qquad \Delta \rhd N : B}{\Gamma, \Delta \rhd \mathrm{discard}\ M\ \mathrm{in}\ N : B}$$

This means, if one has a variable x of duplicable type $!A$, and one wants to use this variable twice, then one must write

$$\mathrm{copy}\ x\ \mathrm{as}\ x_1, x_2\ \mathrm{in}\ f(\mathrm{derelict}(x_1), \mathrm{derelict}(x_2)),$$

whereas in the quantum lambda calculus, we would simply write

$$f(x, x).$$

Our syntax is less faithful to the inference rules of intuitionistic linear logic. For example, the term $f(g(x, x))$ in the quantum lambda calculus could ambiguously refer to either of the following two terms in the language of Benton et al.:

$$\begin{array}{l} \mathrm{copy}\ x\ \mathrm{as}\ x_1, x_2\ \mathrm{in}\ f(g(\mathrm{derelict}(x_1), \mathrm{derelict}(x_2))),\ \mathrm{or} \\ f(\mathrm{copy}\ x\ \mathrm{as}\ x_1, x_2\ \mathrm{in}\ g(\mathrm{derelict}(x_1), \mathrm{derelict}(x_2))). \end{array}$$

Our point of view is that this apparent ambiguity does not matter from a practical point of view, because the two terms will compute precisely the same thing. Therefore, the programmer should not have to worry about such details as the precise time of the duplication of a particular value.

While this makes programming in the quantum lambda calculus simpler, it leads to a more complicated meta-theory. Some of the technical complications are: we need to introduce a subtyping relation, which complicates the proofs of type soundness. Also, because of the above ambiguities, typing derivations are not unique in the quantum lambda calculus, which means that we have to work harder to prove that the meaning of terms is independent of the typing derivation used (see Theorem 1.6.30 below).

Ultimately, we feel that this is a worthwhile trade-off, as the more complicated meta-theory can be settled once and for all. Moreover, it is reassuring to know that type checking, including checking of linearity constraints, can be completely automated by a type inference algorithm (see Section 1.4), so that the slightly more complicated type system does not result in a burden to the programmer.

Type isomorphisms. In our formulation of the quantum lambda calculus, we have a type isomorphism $!!A \cong !A$, which is not valid e.g. in Benton's system. This is a necessary consequence of our choice of implicit linearity tracking, as a value of type $!A$ can be used with type $!!A$ and vice versa. For similar reasons, we have a type isomorphism $!(A \otimes B) \cong !A \otimes !B$, which is not valid in general linear logic. This is because in the quantum lambda calculus, a pair $\langle x, y\rangle$ of type $A \otimes B$ is duplicable if and only if x and y are duplicable.

Copying vs. cloning. Altenkirch and Grattage (2005) have described a quantum programming language where implicit copying of quantum data is allowed, but is interpreted as creating an entangled pair, rather than as cloning. This, however, requires an explicit operation for discarding unused data, similar to Benton et al.'s "discard" operation.

1.4 Type inference algorithm

When writing programs in the quantum lambda calculus, it is an important task, as we have seen in the previous section, to check that the program is well-typed. One way to ensure this would be to require the programmer to supply all type information. This means, the programmer would have to explicitly state the type of every variable and subexpression used in the program. In practice, this is quite cumbersome, as the types can be long and error-prone to write, and make the program hard to read.

Fortunately, the process can be automated. A *type inference algorithm* is an algorithm that inputs an untyped quantum closure, and either outputs a possible type for it, or else announces that no such type exists. When a type inference algorithm exists, the programmer is free to write terms in an untyped way and can rely on the type inference algorithm to point out any potential safety problems.

In "classical" lambda calculus, there exists a well-known type inference algorithm (for a reference, see e.g. Pierce, 2002), which is based on the fact that every term admits a *principal type*. A type is *principal* among the possible types of a term if any other possible type can be obtained from it by *instantiation* of type variables. For example, in simply typed lambda calculus, the term $M = \lambda fx.fx$ admits a principal type of the form $(X \Rightarrow Y) \Rightarrow (X \Rightarrow Y)$. Here, X and Y are type variables, and any valid type of M, such as $(A \Rightarrow (B \Rightarrow A)) \Rightarrow (A \Rightarrow (B \Rightarrow A))$, can be obtained by substituting particular types for X and Y.

The quantum lambda calculus of this chapter does not satisfy the principal type property. The main complication is the subtyping relation $<:$, which allows a term to possess multiple types that are not instances of each other. Indeed, consider again the term $M = \lambda fx.fx$. Naively, a principal type would be (1) the most general with respect to instantiation of type variables, and (2) the smallest with respect to the subtyping relation. Here are some possible types for M, ordered by the subtyping relation (from left to right) as shown in Table 1.6. Note that the types $!((X \multimap Y) \multimap !(X \multimap Y))$ and $(X \multimap Y) \multimap !(X \multimap Y)$ are not valid for the

$$\begin{tikzcd}
!((X\multimap Y)\multimap(X\multimap Y)) \arrow[rr] \arrow[dr] & & (X\multimap Y)\multimap(X\multimap Y) \arrow[dr] & \\
& !(!(X\multimap Y)\multimap(X\multimap Y)) \arrow[rr] & & !(X\multimap Y)\multimap(X\multimap Y). \\
!(!(X\multimap Y)\multimap !(X\multimap Y)) \arrow[rr] \arrow[ur] & & !(X\multimap Y)\multimap !(X\multimap Y) \arrow[ur] &
\end{tikzcd}$$

Table 1.6. *Partial ordering of some possible types of* $\lambda fx.fx$

term M. Therefore, there is no smallest element in the set of possible types for M.

Despite the absence of principal types, it is nevertheless possible to find a type inference algorithm for the quantum lambda calculus. The idea is to proceed in two steps: first, find the principal type derivation for M in a simply-typed version of the language (i.e., without the operator !, and without any restriction on duplication). Second, try to "annotate" this type derivation by putting "!" in the smallest possible number of locations to satisfy the constraints of the linear type system. It was shown in (Valiron, 2004a) that this algorithm succeeds if and only if the term M is typable in the quantum lambda calculus. We refer the reader to (Valiron, 2004a; Selinger and Valiron, 2006) for the full discussion.

1.5 Extending the language with an infinite data type

For a computational formalism to be fully expressive, it must be able to operate on data of unlimited size. For example, in the quantum circuit model, an algorithm is not given by a single quantum circuit, but rather, by a uniform family of circuits, one for each possible size of input.

In the form given in Section 1.3, the quantum lambda calculus is only able to operate on finite dimensional data, and therefore, it is not universal for quantum, or even classical, computation. This situation can be easily remedied by adding an unbounded data type, such as a type of lists. We describe such an extension in Section 1.5.1, and then illustrate its use in Section 1.5.2 with the Quantum Fourier Transform on an unbounded number of qubits.

1.5.1 The type $list(A)$

The notion of a list can be easily defined recursively: a list of objects of type A is either the empty list, or else it is a pair consisting of a *head* of type A and a *tail* that is another list of objects of type A. Therefore, if $list(A)$ is the type of lists of elements of type A, then the following recursive equation is satisfied:

$$list(A) = \top \oplus (A \otimes list(A)).$$

To add such list types to the quantum lambda calculus, we extend:

- the type formation rules: whenever A is a type, then $list(A)$ is a type;
- the term formation rules: we add three term constructors $cons(M, N)$, nil and $unfold(M)$;
- the typing rules: we add three new rules

$$\frac{\begin{array}{c} !\Delta, \Gamma_1 \rhd M : !^n A \\ !\Delta, \Gamma_2 \rhd N : !^n list(A) \end{array}}{!\Delta, \Gamma_1, \Gamma_2 \rhd cons(M, N) : !^n list(A),} \ (list.I_1)$$

$$\frac{}{!\Delta, \Gamma_1, \Gamma_2 \rhd nil : !^n list(A),} \ (list.I_2)$$

$$\frac{\Delta \rhd M : !^n list(A)}{\Delta \rhd unfold_A(M) : !^n((!\top) \oplus (A \otimes list(A)));} \ (list.E)$$

- the reduction rules:

$$[\,Q, L, unfold\ nil\,] \to_1 [\,Q, L, inj_l(*)\,],$$
$$[\,Q, L, unfold(cons(V, W))\,] \to_1 [\,Q, L, inj_r\,\langle V, W\rangle\,],$$

and the congruence rules:

$$\frac{[\,Q, L, N\,] \to_1 [\,Q, L, N'\,]}{[\,Q, L, cons(M, N)\,] \to_1 [\,Q, L, cons(M, N')\,]}$$

$$\frac{[\,Q, L, M\,] \to_1 [\,Q, L, M'\,]}{[\,Q, L, cons(M, V)\,] \to_1 [\,Q, L, cons(M', V)\,]}$$

$$\frac{[\,Q, L, M\,] \to_1 [\,Q, L, M'\,]}{[\,Q, L, unfold(M)\,] \to_1 [\,Q, L, unfold(M')\,]}$$

Note that a list is duplicable only if all of its elements are duplicable. In particular, a list of quantum bits is duplicable only if it is empty.

Remark 1.5.1 The extended language still satisfies the safety properties described in Theorems 1.3.30 and 1.3.32. Moreover, since the principal type property still applies for intuitionistic types in this context, the type inference algorithm can be extended to this situation.

1.5.2 Implementing the Quantum Fourier Transform

From the list type, we can define a data type of natural numbers $Nat = list(\top)$. Then we define the closed terms **succ** : $Nat \multimap Nat$ and **two** : Nat as follows:

$$let\ \ \mathbf{succ}\ n = cons(*, n), \qquad let\ \ \mathbf{two} = \mathbf{succ}\ (\mathbf{succ}\ nil).$$

Assume that **rn** : $Nat \multimap (qbit \otimes qbit \multimap qbit \otimes qbit)$ is a term such that **rn** n computes the gate

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{2\pi i/2^n} \end{pmatrix}.$$

Then one can define the term **rotate** : $qbit \multimap qlist \multimap Nat \multimap (qbit \otimes qlist)$:

$$\begin{array}{l} let\ rec\ \mathbf{rotate}\ h\ t\ n = match\ \ unfold(t)\ with \\ \quad x \mapsto nil \\ \quad b \mapsto let\ \langle x, y\rangle = b\ in \\ \qquad let\ \langle x, h\rangle = (\mathbf{rn}\ n)\ \langle x, h\rangle\ in \\ \qquad let\ \langle h, y\rangle = \mathbf{rotate}\ h\ y\ (\mathbf{succ}\ n)\ in \\ \qquad\quad \langle h, cons(x, y)\rangle, \end{array}$$

and the term **QFT** : $qlist \multimap qlist$:

$$\begin{array}{l} let\ rec\ \mathbf{QFT}\ l = match\ \ unfold(l)\ with \\ \quad x \mapsto nil \\ \quad b \mapsto let\ \langle h, t\rangle = b\ in \\ \qquad let\ \langle h, t\rangle = \mathbf{rotate}\ h\ t\ \mathbf{two}\ in \\ \qquad let\ t = \mathbf{QFT}\ t\ in \\ \qquad\quad cons(h, t). \end{array}$$

The **QFT** term implements the quantum Fourier algorithm but does not reverse the order of the qubits in the output list. Reversing the order of the elements in a list can be efficiently implemented as a map **rev** : $list(A) \multimap list(A)$, defined using an auxiliary map **aux** : $list(A) \otimes list(A) \multimap list(A)$:

$$\begin{array}{l} let\ rec\ \mathbf{aux}\ l_1\ l_2 = match\ \ unfold(l_1)\ with \\ \quad x \mapsto l_2 \\ \quad b \mapsto let\ \langle x, y\rangle = b\ \ in\ \ \mathbf{aux}\ y\ \ cons(x, l_2), \\ let\ \mathbf{rev}\ l = \mathbf{aux}\ l\ nil. \end{array}$$

Note that these functions are linear: the function **rev** can therefore be used to reverse the order of a list of quantum bits to implement the end of the QFT algorithm.

Remark 1.5.2 Unlike the examples of Section 1.2, the implementation of the quantum Fourier transform does not use higher-order structures.

1.6 Categorical semantics

In this section, we present some more advanced material on categorical models for the quantum lambda calculus. The mathematical area of category theory (Mac Lane, 1971) is well-suited for the study of programming languages, as it reveals the structure of programming language features, such as higher-order functions, computational effects, or linearity, at a more abstract level.

For example, there is a well-known connection between higher order functions and the theory of cartesian-closed categories (Lambek and Scott, 1986). Moggi (1991) has given a general theory of computational effects (such as probabilistic programs) in terms of the categorical concept of a strong monad. Finally, Seely (1989), Bierman (1993), and Benton et al. (1993b,a) have given categorical models of linear logic. In this section, we show how to combine these models to obtain a categorical description of the higher-order probabilistic linear features of the quantum lambda calculus.

Categorical semantics is fundamentally a theory of equations between programs. In investigating the equational theory of the quantum lambda calculus, we find that the equations can be divided into two classes: *structural equations*, such as $(\lambda x.M)V = M[V/x]$, which derive from the higher-order type and term constructors, and *ground equations*, such as $meas(H(H(new\,0))) = 0$, which encode special properties of the chosen set of elementary operators.

The ground operations for quantum computation are already well understood: they are given by state preparation, unitary operations, and measurements, and they satisfy precisely the equations that hold in the usual formalism of finite-dimensional Hilbert spaces, density matrices, and superoperators (see e.g. Selinger, 2004).

In constructing a concrete model for a programming language, one usually first focuses on the structural equations, i.e., without fixing a particular set of ground operations. This allows one to identify a whole class of categories that possess the required higher-order structure. In a second step, one may then look for a particular member of this class of categories that also supports the required ground operations.

We will focus exclusively on the first step, i.e., we will work relative to an unspecified, but fixed set of ground types (denoted α), ground operations (denoted c), and ground equations. The construction of a concrete model of higher-order quantum computation, i.e., a specific model that supports both the ground operations and the higher-order structure, is an open problem.

To keep the presentation reasonably simple, we restrict ourselves to the subset of the quantum lambda calculus without union types or recursive function definitions. We call this the "core fragment" of the quantum lambda calculus. The results presented here can be easily extended to the full language.

1.6.1 Equational description of the core fragment

Definition 1.6.1 The core fragment of the linear lambda calculus is defined as follows:

$$\begin{array}{lll} \textit{Type } A, B & ::= & \alpha \mid\ !A \mid A \multimap B \mid \top \mid A \otimes B \\ \textit{Value } V & ::= & x \mid c \mid * \mid \lambda x.M \mid \langle V, V' \rangle, \\ \textit{Term } M, N & ::= & V \mid \langle M, N \rangle \mid (MN) \mid \textit{let } \langle x, y \rangle = M \textit{ in } N. \end{array}$$

Definition 1.6.2 We define an equivalence relation on typing judgements as being the smallest equivalence relation $\approx_{ax}$ satisfying the axioms in Table 1.7, and satisfying the usual congruence relations, stating that if M is a term of the form $C[N]$, i.e. with a subterm N, such that $N \approx_{ax} N'$, then M is equivalent to $C[N']$. We use the notations $\boxdot$, $\boxdot$ and $\boxdot$ as place holders for x and $\langle x, y \rangle$ (not respectively). We also assume that V is a value.

Note that the relation $\approx_{ax}$ is on typing judgements, and not on untyped terms. We write $\Delta \rhd M \approx_{ax} M' : A$ if we want to make the typing context explicit, but we also often write $M \approx_{ax} M'$ when the typing information is implicit.

$$
\begin{array}{lll}
(\beta_\lambda) & let\ x = V\ in\ M & \approx_{ax} M[V/x], \\
(\beta_\otimes) & let\ \langle x,y\rangle = \langle V,W\rangle\ in\ M & \approx_{ax} M[V/x, W/y], \\
(\beta_*) & let\ * = *\ in\ M & \approx_{ax} M, \\
(\eta_\lambda) & \lambda x.Vx & \approx_{ax} V, \\
(\beta^2_\lambda) & let\ x = N\ in\ x & \approx_{ax} N, \\
(\eta_\otimes) & let\ \langle x,y\rangle = N\ in\ \langle x,y\rangle & \approx_{ax} N, \\
(\eta_*) & let\ * = N\ in\ * & \approx_{ax} N, \\
(let^{app}) & let\ x = M\ in\ let\ y = N\ in\ xy & \approx_{ax} MN, \\
(let^{\lambda}) & let\ x = V\ in\ \lambda y.M & \approx_{ax} \lambda y.let\ x = V\ in\ M, \\
(let^{\otimes}) & let\ x = M\ in\ let\ y = N\ in\ \langle x,y\rangle & \approx_{ax} \langle M,N\rangle, \\
(let_1) & let\ \boxdot = (let\ \boxdot = M\ in\ N)\ in\ P & \\
 & \qquad \approx_{ax} let\ \boxdot = M\ in\ let\ \boxdot = N\ in\ P, & \\
(let_2) & let\ \boxdot = V\ in\ let\ \boxdot = W\ in\ M & \\
 & \qquad \approx_{ax} let\ \boxdot = W\ in\ let\ \boxdot = V\ in\ M, &
\end{array}
$$

Table 1.7. *Axiomatic equivalence*

1.6.2 The term model

Having an equivalence relation on typing judgements, it is possible to define categories from axiomatic equivalence classes of typing judgements, as in (Lambek and Scott, 1986).

Definition 1.6.3 We define the notion of *extended value* as follows:

$$ExtValue\ E, E' ::= V \mid \langle E, E'\rangle \mid let\ x = E\ in\ E' \mid let\ \langle x,y\rangle = E\ in\ E',$$

where V is a value. To distinguish between the two sorts of values we also call the values of Definition 1.6.1 *core values*.

The relationship between extended values and core values is explained by the following lemma.

Lemma 1.6.4 *Suppose that no constant term c has a type of the form $!^n(A \otimes B)$. Then for every extended value E such that $\rhd E : A$, there exists a core value V with $\rhd E \approx_{ax} V : A$.*

Proof The proof is done by induction on the size of E. □

Remark 1.6.5 The restriction on constants in Lemma 1.6.4 is not a serious one: if a constant c of type $!^n(A \otimes B)$ were ever needed, one could instead use two constants $c_1 : !^n(A)$ and $c_2 : !^n(B)$.

Definition 1.6.6 We define three categories:

The category $\mathcal{D}$ of computations. Objects are types and arrows of the form $A \to B$ are valid typing judgements $x : A \rhd M : B$. The composition of $x : A \rhd M : B$ and $y : B \rhd N : C$ is $x : A \rhd$ *let* $y = M$ *in* $N : C$ and the identity on A is $x : A \rhd x : A$.

The category $\mathcal{C}$ of values. Objects are types and arrows $A \to B$ are valid typing judgements of the form $x : A \rhd E : B$, where E is an extended value. Composition and identity are defined in the same way as for $\mathcal{D}$.

The category $\mathcal{B}$ of classical values. Objects are types and arrows $A \to B$ are valid typing judgements of the form $x : !A \rhd E : !B$, where E is an extended value. Composition and identity are defined in the same way as for $\mathcal{D}$.

The category $\mathcal{D}$ represents the class of quantum computations, i.e., terms that must still be evaluated and can result in a probability distribution of values. The category $\mathcal{C}$ stands for the category of quantum values. The category $\mathcal{B}$ is more specific: it is composed of the duplicable values with only duplicable free variables. We therefore call it the category of classical values.

Lemma 1.6.7 *The structures defined in Definition 1.6.6 are categories.* □

The remainder of this section is devoted to a study of the abstract properties of the categories $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$.

1.6.3 Structure of the tensor operator "$\otimes$"

The first structural element coming from the type system is the tensor $\otimes$. As suggested by the notation, the tensor possesses the structure of a symmetric monoidal category.

Definition 1.6.8 A *monoidal category* $\mathcal{E}$ is a tuple $(\mathcal{E}, \otimes, \top, \alpha, \lambda, \rho)$ where $\mathcal{E}$ is a category, $\otimes : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ is a functor, $\top$ is an object of $\mathcal{E}$ and α, λ, ρ are three natural isomorphisms

$$\alpha_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C,$$
$$\lambda_A : \top \otimes A \to A, \qquad \rho_A : A \otimes \top \to A$$

such that the diagrams

(1.1)
$$\begin{tikzcd}
& & (A\otimes B)\otimes(C\otimes D) \arrow[drr, "\alpha"] & & \\
A\otimes(B\otimes(C\otimes D)) \arrow[urr, "\alpha"] \arrow[dr, "id\otimes\alpha"'] & & & & ((A\otimes B)\otimes C)\otimes D \\
& A\otimes((B\otimes C)\otimes D) \arrow[rr, "\alpha"'] & & (A\otimes(B\otimes C))\otimes D, \arrow[ur, "\alpha\otimes id"'] &
\end{tikzcd}$$

(1.2)
$$\begin{tikzcd}
A\otimes(\top\otimes C) \arrow[rr, "\alpha"] \arrow[dr, "id\otimes\lambda"'] & & (A\otimes\top)\otimes C \arrow[dl, "\rho\otimes id"] \\
& A\otimes C, &
\end{tikzcd}$$

(1.3)
$$\begin{tikzcd}
\top\otimes\top \arrow[d, shift right, "\rho"'] \arrow[d, shift left, "\lambda"] \\
\top
\end{tikzcd}$$

commute. A monoidal category is said to be *symmetric* when it is equipped with a natural isomorphism $\sigma_{A,B} : A \otimes B \to B \otimes A$ such that the diagrams

(1.4)
$$\begin{tikzcd}
A\otimes B \arrow[r, shift left, "\sigma_{A,B}"] & B\otimes A, \arrow[l, shift left, "\sigma_{B,A}"]
\end{tikzcd}$$

(1.5)
$$\begin{tikzcd}
B\otimes\top \arrow[r, "\sigma_{B,\top}"] \arrow[rr, bend right, "\rho_B"'] & \top\otimes B \arrow[r, "\lambda_B"] & B,
\end{tikzcd}$$

(1.6)
$$\begin{tikzcd}
A\otimes(B\otimes C) \arrow[r, "\alpha"] \arrow[d, "id\otimes\sigma"'] & (A\otimes B)\otimes C \arrow[r, "\sigma"] & C\otimes(A\otimes B) \arrow[d, "\alpha"] \\
A\otimes(C\otimes B) \arrow[r, "\alpha"] & (A\otimes C)\otimes B \arrow[r, "\sigma\otimes id"] & (C\otimes A)\otimes B
\end{tikzcd}$$

commute.

Lemma 1.6.9 *Equipped with the morphisms*

$$\begin{aligned}
\alpha_{A,B,C} &= x : A\otimes(B\otimes C) \rhd \textit{let } \langle y,z\rangle = x \textit{ in} \\
&\qquad \textit{let } \langle t,u\rangle = z \textit{ in } \langle\langle y,t\rangle,u\rangle : (A\otimes B)\otimes C, \\
\lambda_A &= x : \top\otimes A \rhd \textit{let } \langle y,z\rangle = x \textit{ in let } * = y \textit{ in } z : A, \\
\rho_A &= x : A\otimes\top \rhd \textit{let } \langle y,z\rangle = x \textit{ in let } * = z \textit{ in } y : A, \\
\sigma_{A,B} &= x : A\otimes B \rhd \textit{let } \langle y,z\rangle = x \textit{ in } \langle z,y\rangle : B\otimes A,
\end{aligned}$$

and with the map of arrows $(x : A \rhd V : B) \otimes (y : C \rhd W : D) = z : A\otimes B \rhd \textit{let } \langle x,y\rangle = z \textit{ in } \langle V,W\rangle : C\otimes D$, *the category* $\mathcal{C}$ *is symmetric monoidal.* □

Like the category $\mathcal{C}$, the category $\mathcal{B}$ is also a category of values. One expects it to be symmetric monoidal too. This turns out to be the case, but in fact, the category $\mathcal{B}$ has more structure: it is cartesian.

Definition 1.6.10 Given a category $\mathcal{E}$ and two objects A and B, the *cartesian product* of A and B is, if it exists, the data consisting of an object $A\times B$ and two maps $\pi^1_{A,B} : A\times B \to A$ and $\pi^2_{A,B} : A\times B \to B$, such that for any pair of maps $f : C \to A$ and $g : C \to B$, there exists a unique map $\langle f,g\rangle : C \to A\times B$ satisfying $\langle f,g\rangle;\pi^1_{A,B} = f$ and $\langle f,g\rangle;\pi^2_{A,B} = g$. Note that uniqueness is equivalent to the equation $\langle h;\pi^1_{A,B}, h;\pi^2_{A,B}\rangle = h$ for all $h : C \to A\times B$.

We say that the category $\mathcal{E}$ *has binary products* if there is a cartesian product for all A and B. We say that it is *cartesian* if it also has a terminal object T.

Lemma 1.6.11 *The category* $(\mathcal{B},\otimes,\top)$ *is cartesian.*

Proof Consider A and B any types. One constructs $\pi^1_{A,B}$ and $\pi^2_{A,B}$ respectively as follows:

$$x : !(A\otimes B) \rhd \textit{let } \langle y,z\rangle = x \textit{ in } y : !A,$$
$$x : !(A\otimes B) \rhd \textit{let } \langle y,z\rangle = x \textit{ in } z : !B.$$

In this case, given two maps f and g respectively of the form $x : !C \rhd U : !A$ and $x : !C \rhd V : !B$, one constructs the map $\langle f,g\rangle : C \to A\otimes B$ simply as $x : !C \rhd \langle U,V\rangle : !A\otimes B$. □

Remark 1.6.12 Note that, although the maps $\pi^1_{A,B}$ and $\pi^2_{A,B}$ can be similarly defined in $\mathcal{C}$ and although $\top$ is terminal in $\mathcal{C}$, the map $\langle f,g\rangle$ cannot in general be defined due to linearity: for instance, there is no diagonal map for the type *qbit*.

1.6.4 Structure of the function operator "$\multimap$"

Before explaining the nature of the function operator "$\multimap$", we must briefly comment on the distinction between *computations* and *values*, which is reflected in the definitions of the categories $\mathcal{C}$ and $\mathcal{D}$. This distinction holds in any call-by-value programming language, including the quantum lambda calculus.

A computation is an arbitrary term M, i.e., something that must be evaluated to obtain a result. On the other hand, a value V represents the result of a computation. Because of the probabilistic nature of the quantum lambda calculus, a computation M in general does not yield a value, but a probability distribution of values. In the presence of recursive function definitions, a computation M may also diverge with non-zero probability. It is therefore natural to think of a value $V : A$ as an "element" of type A, and to think of a computation $M : A$ as a sub-probability distribution of such elements. We informally write $T(A)$ for the set of sub-probability distributions on a set of values A.

In the context of higher-order computation, computations of type A can be identified with values of type $\top \multimap A$. Indeed, given any computation $M : A$, we can form the value $\lambda *.M : \top \multimap A$ (called a "thunk"); and conversely, given any value $V : \top \multimap A$, we can form a computation $V* : A$. The two constructions are mutually inverse modulo the axioms in Table 1.7. We can therefore identify $T(A)$ with $\top \multimap A$.

Moggi (1991) has established a general framework for dealing with values and computations in the context of side-effects (such as probability and non-termination), in terms of the concept of strong monad.

Definition 1.6.13 A *monad* over a category $\mathcal{E}$ is a tuple (T, η, μ), where $T : \mathcal{E} \to \mathcal{E}$ is a functor, $\eta : id \stackrel{\cdot}{\to} T$ and $\mu : T^2 \stackrel{\cdot}{\to} T$ are natural transformations and the diagrams (1.7) and (1.8) in Table 1.8 commute. The natural transformation μ is called the *multiplication* of the monad and η the *unit* of the monad.

A monad over a monoidal category $(\mathcal{E}, \otimes, \top)$ is *strong* if moreover there exists a natural transformation $t_{A,B} : A \otimes TB \to T(A \otimes B)$, called the *tensorial strength*, such that the diagrams (1.9), (1.10), (1.11) commute.

The category $\mathcal{C}$ described in Definition 1.6.6 does admit a strong monad:

Lemma 1.6.14 *In $\mathcal{C}$, define the following maps:*

$$\begin{aligned}
\eta_A &= \quad x : A \rhd \lambda *.x : \top \multimap A, \\
\mu_A &= \quad x : \top \multimap (\top \multimap A) \rhd \lambda *.(x*)* : \top \multimap A, \\
t_{A,B} &= \quad z : A \otimes (\top \multimap B) \rhd let\ \langle x, y \rangle = z\ in\ \lambda *.\langle x, y* \rangle : \top \multimap (A \otimes B),
\end{aligned}$$

and define a functor T as follows: for every object A, $TA = \top \multimap A$ and for every morphism $f : A \to B$ of the form $x : A \rhd V : B$, the image $Tf : TA \to TB$ is

$$y : \top \multimap A \rhd \lambda *.let\ x = (y*)\ in\ V : \top \multimap B.$$

Then (T, η, μ, t) is a strong monad on $\mathcal{C}$.

Definition 1.6.15 Given a monad (T, η, μ) over $\mathcal{E}$, the *Kleisli category* $\mathcal{E}_T$ is defined as follows:

$$(1.7) \quad \begin{tikzcd} T^3A \arrow[r, "T\mu_A"] \arrow[d, "\mu_{TA}"'] & T^2A \arrow[d, "\mu_A"] \\ T^2A \arrow[r, "\mu_A"] & TA, \end{tikzcd} \qquad \begin{tikzcd} TA \arrow[r, "\eta_{TA}"] \arrow[dr, "id_{TA}"'] & T^2A \arrow[d, "\mu_A"] & TA. \arrow[l, "T\eta_A"'] \arrow[dl, "id_{TA}"] \\ & TA & \end{tikzcd} \quad (1.8)$$

$$(1.9) \quad \begin{tikzcd} \top \otimes TA \arrow[r, "\lambda"] \arrow[dr, "t"] & TA \\ & T(\top \otimes A), \arrow[u, "T\lambda"'] \end{tikzcd} \qquad (1.10) \quad \begin{tikzcd} (A \otimes B) \otimes TC \arrow[d, "t"] & A \otimes (B \otimes TC) \arrow[l, "\alpha"'] \arrow[d, "id \otimes t"] \\ T((A \otimes B) \otimes C) & A \otimes T(B \otimes C) \arrow[d, "t"] \\ & T(A \otimes (B \otimes C)), \arrow[ul, "T(\alpha)"] \end{tikzcd}$$

$$\begin{tikzcd} A \otimes B \arrow[d, "id \otimes \eta"'] \arrow[dr, "\eta"] & & \\ A \otimes TB \arrow[r, "t"] & T(A \otimes B) & \\ A \otimes T^2B \arrow[u, "id \otimes \mu"] \arrow[r, "t"] & T(A \otimes TB) \arrow[r, "Tt"] & T^2(A \otimes B) \arrow[ul, "\mu"'] \end{tikzcd} \quad (1.11)$$

Table 1.8. *Equations for a strong monad*

- the objects of $\mathcal{E}_T$ are the objects of $\mathcal{E}$,
- the set $\mathcal{E}_T(A, B)$ of morphisms from A to B in $\mathcal{E}_T$ is $\mathcal{E}(A, TB)$,
- the identity on A is $\eta_A : A \to TA$,
- The composition of $f \in \mathcal{E}_T(A, B)$ and $g \in \mathcal{E}_T(B, C)$ in $\mathcal{E}_T$ is $f; Tg; \mu_C : A \to TC$.

Remark 1.6.16 The category $\mathcal{D}$ of computations is the Kleisli category $\mathcal{C}_T$ over the category of values $\mathcal{C}$, where (T, μ, η) is the monad described in Lemma 1.6.14.

The operator $\multimap$ has more structure than "just" providing a monad.

Definition 1.6.17 A symmetric monoidal category $(\mathcal{E}, \otimes, \top)$ together with a strong monad (T, η, μ) is said to have *T-exponentials* (Moggi, 1991), or *Kleisli exponentials*, if it is equipped with a bifunctor $\multimap : \mathcal{E}^{op} \times \mathcal{E} \to \mathcal{E}$, and a natural isomorphism

$$\Phi : \mathcal{E}(A, B \multimap C) \xrightarrow{\cong} \mathcal{E}(A \otimes B, TC).$$

Lemma 1.6.18 *Consider the functor $\multimap : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{C}$, defined on* *arrows by*

$$\begin{array}{rcl} A \multimap (x : B \rhd V : C) & = & y : A \multimap B \rhd \lambda z.\mathit{let}\ x = yz\ \mathit{in}\ V : A \multimap C, \\ (x : A \rhd V : B) \multimap C & = & y : B \multimap C \rhd \lambda x.yV : A \multimap C, \end{array}$$

and the map $\Phi_{A,B,C} : \mathcal{C}(A, B \multimap C) \to \mathcal{C}(A \otimes B, \top \multimap C)$*, sending the morphism* $x : A \rhd V : B \multimap C$ *to*

$$t : A \otimes B \rhd \lambda *.\mathit{let}\ \langle x, y\rangle = t\ \mathit{in}\ V y : \top \multimap C.$$

Together with these structures, $\mathcal{C}$ *has* T*-exponentials, where* (T, μ, η) *is the monad described in Lemma 1.6.14.* □

1.6.5 Structure of the exponential operator "!"

The type operator "!" is first dealt with using a subtyping relation. Using this relation, one can construct a comonad for the category $\mathcal{C}$.

Definition 1.6.19 A *comonad* in a category $\mathcal{E}$ is a tuple (L, ϵ, δ), where $L : \mathcal{E} \to \mathcal{E}$ is a functor and $\epsilon : L \dot{\to} I$ and $\delta : L \dot{\to} L^2$ are natural transformations such that the following diagrams commute:

$$(1.12) \quad \begin{tikzcd} LA \arrow[r, "\delta_A"] \arrow[d, "\delta_A"'] & L^2A \arrow[d, "L\delta_A"] \\ L^2A \arrow[r, "\delta_{LA}"'] & L^3A, \end{tikzcd} \qquad \begin{tikzcd} & LA \arrow[dl, "id_{LA}"'] \arrow[d, "\delta_A"] \arrow[dr, "id_{LA}"] & \\ LA & L^2A \arrow[l, "\epsilon_{LA}"] \arrow[r, "L\epsilon_A"'] & LA. \end{tikzcd} \quad (1.13)$$

A comonad is said to be *idempotent* if the comultiplication δ is an isomorphism.

Lemma 1.6.20 *Consider the map* ! *sending* A *to* $!A$ *and sending* $x : A \rhd V : B$ *to* $x : !A \rhd V : !B$ *(which is well-typed by Lemmas 1.3.24 and 1.3.25). Also consider the arrows*

$$\epsilon_A = x : !A \rhd x : A, \qquad \delta_A = x : !A \rhd x : !^2A.$$

Then $(!, \epsilon, \delta)$ *is an idempotent comonad in* $\mathcal{C}$. □

The goal of the type operator "!" is to indicate when and how an object is duplicable. Besides being an idempotent comonad, it has additional properties, which we will detail below. As we have already remarked, the operator "!" is taken from intuitionistic linear logic (Girard, 1987). It is therefore not surprising that its categorical semantics closely follows

the model of intuitionistic linear logic proposed by Bierman (1993), and later simplified by Benton et al. (1993b,a). Good additional references are Melliès (2003) and Maneggia (2004). We will follow the terminology of Schalk (2004) and use the concept of *linear exponential comonad*.

Definition 1.6.21 (Benton et al., 1993a; Bierman, 1993) Let $(\mathcal{E}, \otimes, \top)$ be a (symmetric) monoidal category. A *(symmetric) monoidal comonad* on $\mathcal{E}$ is a comonad (L, δ, ϵ):

- equipped with two natural transformations $d_{A,B} : LA \otimes LB \to L(A \otimes B)$ and $d_\top : \top \to L\top$ making (L, d) a lax (symmetric) monoidal functor,
- such that δ and ϵ are monoidal natural transformations, i.e. such that the following diagrams commute:

$$(1.14) \quad \begin{tikzcd} LA \otimes LB \arrow[rr, "d_{A,B}"] \arrow[dr, "\epsilon_A \otimes \epsilon_B"'] & & L(A \otimes B), \arrow[dl, "\epsilon_{A \otimes B}"] \\ & A \otimes B & \end{tikzcd} \qquad \begin{tikzcd} \top \arrow[rr, "d_\top"] \arrow[dr, "id_\top"'] & & L\top, \arrow[dl, "\epsilon_\top"] \\ & \top & \end{tikzcd} \quad (1.15)$$

$$\begin{tikzcd} LA \otimes LB \arrow[rr, "d_{A,B}"] \arrow[d, "\delta_A \otimes \delta_A"] \arrow[drr, phantom, "(1.16)"] & & L(A \otimes B), \arrow[d, "\delta_{A \otimes B}"'] \\ L^2A \otimes L^2B \arrow[r, "d_{LA,LB}"'] & L(LA \otimes LB) \arrow[r, "Ld_{A,B}"'] & L^2(A \otimes B) \end{tikzcd} \qquad \begin{tikzcd} \top \arrow[r, "d_\top"] \arrow[d, "d_\top"'] \arrow[dr, phantom, "(1.17)"] & L\top. \arrow[d, "\delta_\top"] \\ L\top \arrow[r, "Ld_\top"'] & L^2\top \end{tikzcd}$$

Definition 1.6.22 Let $(\mathcal{E}, \otimes, \top, \alpha, \lambda, \rho, \sigma)$ be a symmetric monoidal category. Let $(L, \delta, \epsilon, d^L, d^L)$ be a monoidal comonad. We say that L is a *linear exponential comonad* provided that

(i) each object in $\mathcal{E}$ of the form LA is equipped with a commutative comonoid $(LA, \triangle_A, \diamondsuit_A)$, where $\triangle_A : LA \to LA \otimes LA$ and $\diamondsuit_A : LA \to \top$;

(ii) $\triangle_A$ and $\diamondsuit_A$ are monoidal natural transformations, i.e. the following diagrams

$$\begin{tikzcd}
LA \otimes LB \arrow[r, "\triangle_A \otimes \triangle_B"] \arrow[dd, "d^L_{A,B}"'] & (LA \otimes LA) \otimes (LB \otimes LB) \arrow[d, "sw"] \\
 & (LA \otimes LB) \otimes (LA \otimes LB) \arrow[d, "d^L_{A,B} \otimes d^L_{A,B}"] \\
L(A \otimes B) \arrow[r, "\triangle_{(A\otimes B)}"'] & L(A \otimes B) \otimes L(A \otimes B)
\end{tikzcd} \tag{1.18}$$

$$(1.19) \quad \begin{tikzcd}
\top \arrow[r, "\lambda_\top^{-1}"] \arrow[d, "d^L_\top"'] & \top \otimes \top \arrow[d, "d^L_\top \otimes d^L_\top"] \\
L\top \arrow[r, "\triangle_\top"'] & L\top \otimes L\top
\end{tikzcd} \qquad \begin{tikzcd}
LA \otimes LB \arrow[r, "\Diamond_A \otimes \Diamond_B"] \arrow[d, "d^L_{A,B}"'] & \top \otimes \top \arrow[d, "\lambda_\top"] \\
L(A \otimes B) \arrow[r, "\Diamond_{A\otimes B}"'] & \top
\end{tikzcd} \quad (1.20)$$

$$\begin{tikzcd}
\top \arrow[rr, "id"] \arrow[dr, "d^L_\top"'] & & \top; \\
 & L\top \arrow[ur, "\Diamond_\top"'] &
\end{tikzcd} \quad (1.21)$$

commute.

(iii) The maps

$$\triangle_A : (LA, \delta_A) \to (LA \otimes LA, (\delta_A \otimes \delta_A); d_A),$$
$$\Diamond_A : (LA, \delta_A) \to (\top, d^L_\top)$$

are L-coalgebra morphisms, i.e.

$$(1.22) \quad \begin{tikzcd}
LA \arrow[r, "\triangle_A"] \arrow[dd, "\delta_A"'] & LA \otimes LA \arrow[d, "\delta_A \otimes \delta_A"] \\
 & L^2A \otimes L^2A \arrow[d, "d^L_{LA,LA}"] \\
L^2A \arrow[r, "L\triangle_A"'] & L(LA \otimes LA),
\end{tikzcd} \qquad \begin{tikzcd}
LA \arrow[r, "\Diamond_A"] \arrow[d, "\delta_A"'] & \top \arrow[d, "d^L_\top"] \\
L^2A \arrow[r, "L\Diamond_A"'] & L\top;
\end{tikzcd} \quad (1.23)$$

(iv) Every map δ_A is a comonoid morphism

$$(LA, \Diamond_A, \triangle_A) \to (L^2A, \Diamond_{LA}, \triangle_{LA}),$$

i.e. satisfying the diagrams

$$(1.24) \quad \begin{tikzcd}
LA \arrow[r, "\delta_A"] \arrow[d, "\triangle_A"'] & L^2A, \arrow[d, "\triangle_{LA}"] \\
LA \otimes LA \arrow[r, "\delta_A \otimes \delta_A"'] & L^2A \otimes L^2A
\end{tikzcd} \qquad \begin{tikzcd}
LA \arrow[rr, "\delta_A"] \arrow[dr, "\Diamond_A"'] & & L^2A. \arrow[dl, "\Diamond_{LA}"] \\
 & \top &
\end{tikzcd} \quad (1.25)$$

Lemma 1.6.23 *If we define the maps*

$$\begin{aligned}
d^!_{A,B} &= \ z : !A \otimes !B \rhd \mathit{let}\ \langle x, y\rangle = z\ \mathit{in}\ \langle x, y\rangle : !(A \otimes B), \\
d^!_\top &= \ z : \top \rhd * : !\top, \\
\triangle_A &= \ x : !A \rhd \langle x, x\rangle : !A \otimes !A, \\
\Diamond_A &= \ x : !A \rhd * : \top,
\end{aligned}$$

the comonad $(!, \delta, \epsilon)$ *is an idempotent linear exponential comonad on the category* $\mathcal{C}$. □

Lemma 1.6.24 *The category* $\mathcal{B}$ *is the co-Kleisli category of the comonad described in Lemma 1.6.23.* □

Remark 1.6.25 Note that the map $d^!_{A,B}$ and $d^!_\top$ in Lemma 1.6.23 are isomorphisms of $\mathcal{C}$. In other words, we require the monoidal comonad to be *strongly monoidal*. This is because, unlike the work of Bierman (1993) and Benton et al. (1993b), the types $!(A \otimes B)$ and $!A \otimes !B$ are isomorphic in the quantum lambda calculus (as explained in Section 1.3.8).

1.6.6 A linear category for duplication

Putting together the features developed on the last three sections, we can abstract from the properties of the term categories $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$, and define a sound model for the core fragment of the linear lambda calculus of Section 1.6.1.

Definition 1.6.26 A *linear category for duplication* is a category $\mathcal{E}$ with the following structure:

- a symmetric monoidal structure $(\otimes, \top, \alpha, \lambda, \rho, \sigma)$;
- an idempotent, strongly monoidal, linear exponential comonad $(L, \delta, \epsilon, d^L, d^L, \Diamond, \triangle)$;
- a strong monad (T, μ, η, t);
- a Kleisli exponential $\multimap$, that is, a natural map of arrows

  $$\Phi : \mathcal{E}(A \otimes B, TC) \to \mathcal{E}(A, B \multimap C)$$

  and a bifunctor $\multimap : \mathcal{E}^{op} \times \mathcal{E} \to \mathcal{E}$.

A linear category for duplication $\mathcal{E}$ is said to be *weak* if $\top$ is a terminal object.

Theorem 1.6.27 *The category $\mathcal{C}$ (from Definition 1.6.6) is a weak linear category for duplication.*

Proof The proof uses Lemmas 1.6.9, 1.6.14, 1.6.18 and 1.6.23. □

Remark 1.6.28 A linear category for duplication gives rise to a double adjunction

$$\begin{tikzcd}
\mathcal{E}_L \arrow[r, bend left, "U^L"] \arrow[r, phantom, "\perp"] & \mathcal{E} \arrow[l, bend left, "F^L"] \arrow[r, bend left, "U^T"] \arrow[r, phantom, "\perp"] & \mathcal{E}_T. \arrow[l, bend left, "F^T"]
\end{tikzcd}$$

Here the left adjunction arises from the co-Kleisli category $\mathcal{E}_L$ of the comonad L. It is a linear-non-linear model in the sense of Benton (1995). The right adjunction arises from the Kleisli category $\mathcal{E}_T$ of the computational monad T.

1.6.7 Interpretation of the core fragment

The core fragment of linear lambda calculus, as given in Section 1.6.1, can be interpreted in any weak linear category for duplication $\mathcal{E}$.

Suppose that for each type constant α, we are given an object $\Theta(\alpha)$ of $\mathcal{E}$. Then we can recursively assign to each type A an object $[\![A]\!]$ via $[\![\alpha]\!] = \Theta(\alpha)$, $[\![!A]\!] = L[\![A]\!]$, $[\![A \multimap B]\!] = [\![A]\!] \multimap [\![B]\!]$, $[\![\top]\!] = \top$, and $[\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!]$. Moreover, for any valid subtyping $A <: B$, there exists a canonical arrow $I_{A,B} : [\![A]\!] \to [\![B]\!]$ in $\mathcal{E}$, definable from the structure of the comonad L. If $\Delta = \{x_1 : A_1, \ldots, x_n : A_n\}$ is a typing context, we write $[\![\Delta]\!] = [\![A_1]\!] \otimes \ldots \otimes [\![A_n]\!]$.

Further, suppose that for each constant $c : A_c$, we are given an interpretation as an arrow $\Theta(c) : \top \to [\![A_c]\!]$ in $\mathcal{E}$. Then to each valid typing derivation, we can recursively assign an arrow of $\mathcal{E}$ as follows:

- If π is a typing derivation of a value judgement $x_1 : A_1, \ldots x_n : A_n \rhd V : B$, then $[\![\pi]\!]^v$ is an arrow $[\![A_1]\!] \otimes \ldots \otimes [\![A_n]\!] \to [\![B]\!]$;
- if π is a typing derivation of a general judgement $x_1 : A_1, \ldots x_n : A_n \rhd M : A$, then $[\![\pi]\!]^c$ is an arrow $[\![A_1]\!] \otimes \ldots \otimes [\![A_n]\!] \to T([\![B]\!])$.

The actual definition is by recursion on typing derivations, and consists of a large number of straightforward cases. We only give one of the cases as an example, and refer to (Selinger and Valiron, 2008a; Valiron, 2008) for complete details. Consider a typing derivation ending in the rule

$$\frac{\Gamma_1, !\Delta \rhd M : A \multimap B \quad \Gamma_2, !\Delta \rhd N : A}{\Gamma_1, \Gamma_2, !\Delta \rhd MN : B}.$$

If the two premises are interpreted, respectively, as morphisms

$$f : [\![!\Delta]\!] \otimes [\![\Gamma_1]\!] \to T([\![A]\!] \multimap [\![B]\!])$$
$$g : [\![!\Delta]\!] \otimes [\![\Gamma_2]\!] \to T([\![A]\!]),$$

then the interpretation the conclusion is defined as

$$\begin{aligned}
[\![!\Delta]\!] \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] &\xrightarrow{\tilde{\Delta}} [\![!\Delta]\!] \otimes [\![!\Delta]\!] \otimes [\![\Gamma_1]\!] \otimes [\![\Gamma_2]\!] \\
&\xrightarrow{id\otimes\sigma\otimes id} [\![!\Delta]\!] \otimes [\![\Gamma_1]\!] \otimes [\![!\Delta]\!] \otimes [\![\Gamma_2]\!] \\
&\xrightarrow{f\otimes g} T([\![A]\!]\multimap[\![B]\!]) \otimes T([\![A]\!]) \\
&\xrightarrow{\Psi_1} T(([\![A]\!]\multimap[\![B]\!]) \otimes [\![A]\!]) \\
&\xrightarrow{T(\varepsilon)} T(T([\![B]\!])) \\
&\xrightarrow{\mu} T([\![B]\!]),
\end{aligned}$$

where $\tilde{\Delta} : [\![\Delta]\!] \to [\![\Delta]\!] \otimes [\![\Delta]\!]$ is the canonical map definable from $\Delta_A : LA \to LA \otimes LA$ of the linear exponential comonad, and $\varepsilon : (A \multimap B) \otimes A \to TB$ is the canonical map arising from the Kleisli exponential.

Remark 1.6.29 Note that the interpretation is defined on typing derivations, and not on typing judgements. And indeed, a valid typing judgement can in general possess many different typing derivations. For example, consider the following two typing judgements, where V and W are values:

$$x : A \rhd V : !B, \qquad\qquad y : B \rhd W : C.$$

Then the typing judgement $x : A \rhd \mathit{let}\ y = V\ \mathit{in}\ W : C$ has two distinct valid typing derivations:

$$\frac{x : A \rhd V : !B \quad y : !B \rhd W : C}{x : A \rhd \mathit{let}\ y = V\ \mathit{in}\ W : C,} \qquad \frac{x : A \rhd V : B \quad y : B \rhd W : C}{x : A \rhd \mathit{let}\ y = V\ \mathit{in}\ W : C.}$$

Therefore, a major complication in the proof of soundness of the interpretation is that we must first show that the interpretation is well-defined on typing judgements, and independent of the particular typing derivation used to derive it. This is the subject of the following theorem.

Theorem 1.6.30 *Given a valid typing judgement with two typing derivations π and π', for any interpretation Θ we have the equality $[\![\pi]\!]^c_\Theta = [\![\pi']\!]^c_\Theta$ (and $[\![\pi]\!]^v_\Theta = [\![\pi']\!]^v_\Theta$ if the typing judgement is a value).*

The proof of this theorem is technically complicated, and occupies almost three chapters of Valiron's Ph.D. thesis (see Valiron, 2008, Corollary 11.2.6).

Theorem 1.6.31 (Soundness) *If $\Delta \rhd M \approx_{ax} M' : A$ then we have $[\![\Delta \rhd M : A]\!]^c_\Theta = [\![\Delta \rhd M' : A]\!]^c_\Theta$ (and $[\![\Delta \rhd M : A]\!]^v_\Theta = [\![\Delta \rhd M' : A]\!]^v_\Theta$ if M and M' are values) for every interpretation Θ in a weak linear category for duplication.*

Proof By induction on the definition of $\approx_{ax}$, using a substitution lemma. □

Theorem 1.6.32 (Completeness) *If for every interpretation Θ in every weak linear category for duplication, the equality $[\![\Delta \rhd M : A]\!]^c_\Theta = [\![\Delta \rhd M' : A]\!]^c_\Theta$ holds, then $\Delta \rhd M \approx_{ax} M' : A$.*

Proof It suffices to consider the term model $\mathcal{C}$ of Section 1.6.2, with Θ the identity map. □

1.6.8 Toward a concrete model

In light of the results of the last few subsections, we can now say what it means to be a "concrete model of the quantum lambda calculus". Such a model should have the structure of a weak linear category for duplication, with the further property that the ground type operations (*new*, *meas*, and unitary operations) can be interpreted with their usual meanings. More precisely, a *concrete model of the quantum lambda calculus* is a weak linear category for duplication $\mathcal{E}$ with distributive finite coproducts, preserved by L, together with an object $qbit \in |\mathcal{E}|$, such that the full monoidal subcategory of $\mathcal{E}_T$ of objects of the form $qbit^{n_1} \oplus \ldots \oplus qbit^{n_k}$ is equivalent to the category $\mathbf{Q}$ of tuples of finite dimensional Hilbert spaces and norm non-increasing completely positive maps (Selinger, 2004).

As mentioned in the introduction of Section 1.6, at the time of this writing, it is still an open problem to find such a concrete model (other than the term model).

The only partial result in this direction is a model for the strictly linear fragment of the quantum lambda calculus (i.e., without the $!A$ operator, and without the possibility of duplication or erasure of either classical or quantum information). A fully abstract model for this fragment was given in (Selinger and Valiron, 2008b).

Bibliography

Abramsky, S. (1993) Computational interpretations of linear logic. *Theoretical Computer Science* **111**:3–57.

Altenkirch, T. and Grattage, J. (2005) A functional quantum programming language. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science, LICS'05*, pages 249–258. IEEE Computer Society Press.

Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundation of Mathematics*. North Holland, 2nd edition.

Bell, J. S. (1964) On the Einstein Podolsky Rosen paradox. *Physics* **1**:195–200.

Bennett, C. H., Brassard, G., Crépeau, C., Jozsa, R., Peres, A. and Wootters, W. K. (1993) Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Physical Review Letters* **70**(13):1895–1899.

Bennett, C. H. and Wiesner, S. J. (1992) Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states. *Physical Review Letters* **69**(20):2881–2884.

Benton, N. (1995) A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In *Proceedings of the Eighth Workshop on Computer Science Logic, CSL'94, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer.

Benton, N., Bierman, G., de Paiva, V. and Hyland, M. (1993a) Linear λ-calculus and categorical models revisited. In *Proceedings of the Sixth Workshop on Computer Science Logic, CSL'92, Selected Papers*, volume 702 of *Lecture Notes in Computer Science*, pages 61–84. Springer.

Benton, N., Bierman, G., de Paiva, V. and Hyland, M. (1993b) A term calculus for intuitionistic linear logic. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, volume 664 of *Lecture Notes in Computer Science*, pages 75–90. Springer.

Benton, N. and Wadler, P. (1996) Linear logic, monads and the lambda calculus. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science, LICS'96*, pages 420–431. IEEE Computer Society Press.

Bierman, G. (1993) *On Intuitionistic Linear Logic*. Ph.D. thesis, University of Cambridge Computer Laboratory, Cambridge, England. Available as Technical Report 346, August 1994.

Deutsch, D. and Jozsa, R. (1992) Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London A* **439**:553–558.

Girard, J.-Y. (1987) Linear logic. *Theoretical Computer Science* **50**(1):1–101.

Hudak, P., Hughes, J., Peyton Jones, S. and Wadler, P. (2007) A history of Haskell: Being lazy with class. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, pages 12.1–55. Association for Computing Machinery.

Knill, E. H. (1996) Conventions for quantum pseudocode. Technical Report LAUR-96-2724, Los Alamos National Laboratory.

Lambek, J. and Scott, P. (1986) *Introduction to Higher Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press.

Mac Lane, S. (1971) *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer.

Maneggia, P. (2004) *Models of Linear Polymorphism*. Ph.D. thesis, School of Computer Science, University of Birmingham.

McCarthy, J. (1960) Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM* **3**(4):184–195.

Melliès, P.-A. (2003) Categorical models of linear logic revisited. Technical Report 22, Laboratoire Preuves, Programmes et Systèmes, CNRS and Paris 7. To appear in *Theoretical Computer Science*.

Milner, R. (1978) A theory of type polymorphism in programming. *Journal of Computer and System Sciences* **17**:348–375.

Moggi, E. (1991) Notions of computation and monads. *Information and Computation* **93**(1):55–92.

Pierce, B. C. (2002) *Types and Programming Languages*. MIT Press.

Schalk, A. (2004) What is a categorical model of linear logic? Manuscript, available from http://www.cs.man.ac.uk/∼schalk/work.html.

Seely, R. A. (1989) Linear logic, *-autonomous categories and cofree coalgebras. In *Proceedings of Categories in Computer Science and Logic, Boulder, 1987*, volume 92 of *Contemporary Mathematics*, pages 371–382. American Mathematical Society.

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

Selinger, P. and Valiron, B. (2005) A lambda calculus for quantum computation with classical control. In *Proceedings of the Seventh International Conference on Typed Lambda Calculi and Applications, TLCA'05*, volume 3461 of *Lecture Notes in Computer Science*, pages 354–368. Springer.

Selinger, P. and Valiron, B. (2006) A lambda calculus for quantum computation with classical control. *Mathematical Structures in Computer Science* **16**(3):527–552.

Selinger, P. and Valiron, B. (2008a) A linear-non-linear model for a computational call-by-value lambda calculus. In *Proceedings of the Eleventh International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2008*, volume 4962 of *Lecture Notes in Computer Science*, pages 81–96. Budapest, Hungary.

Selinger, P. and Valiron, B. (2008b) On a fully abstract model for a quantum linear functional language. In *Proceedings of the Fourth International Workshop on Quantum Programming Languages, QPL 2006*, volume 210 of *Electronic Notes in Theoretical Computer Science*, pages 123–137.

Sussman, G. J. and Steele Jr, G. L. (1975) Scheme: An interpreter for extended lambda calculus. Technical Report AIM-349, MIT.

Troelstra, A. S. (1992) *Lectures on Linear Logic*, volume 29 of *CSLI Lecture Notes*. University of Chicago Press, 2nd edition.

Turing, A. M. (1937) On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2* **42**:230–265.

Valiron, B. (2004a) *A functional programming language for quantum computation with classical control*. Master's thesis, Department of Mathematics and Statistics, University of Ottawa.

Valiron, B. (2004b) Quantum typing. In *Proceedings of the Second International Workshop on Quantum Programming Languages, QPL 2004*, TUCS General Publication 33, pages 163–178. Turku Centre for Computer Science, Turku, Finland.

Valiron, B. (2008) *Semantics for a Higher-Order Functional Programming Language for Quantum Computation*. Ph.D. thesis, Department of Mathematics and Statistics, University of Ottawa.

van Tonder, A. (2004) A lambda calculus for quantum computation. *SIAM Journal of Computing* **33**(5):1109–1135.