Orthogonality and Algebraic Lambda-Calculus

Benoît Valiron

March 28, 2010

Abstract

Directly encoding lambda-terms on quantum strings while keeping a quantum interpretation is a hard task. As shown by van Tonder [2004], requiring a unitary reduction forces the lambda-terms in superposition to be mostly equivalent.

Following instead [Arrighi and Díaz-Caro, 2009], we show in this note how one can conceive a lambda-calculus with algebraic features and that admits a general notion of orthogonality amongst lambda-terms, by providing a compiler of the system into unitary maps.

1 Introduction

In the literature, there have been two canonical ways for building a lambda-calculus for quantum computation. One can either consider the program outside of the quantum store: this is known as "classical control", and it has been studied in detail in various publications [Selinger and Valiron, 2006; Valiron, 2008]. One can also adopt the "quantum control" approach [Altenkirch and Grattage, 2005; van Tonder, 2004; Arrighi and Dowek, 2008]. In this setting, the lambda-terms are written directly on the quantum array. In his famous paper, van Tonder [2004] considers a lambda-calculus featuring constants for booleans and unitary gates, where terms are encoded directly on quantum bits. He shows that to be able to have a unitary beta-reduction, all terms in superpositions need to be equal up to the booleans. This morally states that the resulting lambda-calculus is classical.

Other attempts have been proposed. First, if one does not look for a beta-reduction but merely an interpreter, a functional approach is still possible, and a language together with a compiler was proposed with QML [Grattage, 2007]. One can also forget altogether the requirement that the beta-reduction be realizable, leaving it for later. This is the approach taken by Arrighi and Dowek [2008].

In this note, we conciliate these two approaches by showing how to write a (strictly linear) lambda-calculus capturing a notion of orthogonality and (non-trivial) superposition of terms. The superposition is validated by interpreting the language in a typed version of lineal, and the orthogonality is validated by providing a compilation in quantum circuit, in a QML style.

The resulting language, although quite simple, is powerful enough for encoding regular quantum circuits consisting of a given set of gates. To the knowledge of the author, this is the first mention of a truly "purely quantum" language with higher-order features and non-trivial superposition of terms.

2 A lambda-calculus

Consider the following call-by-value, linear, simply-typed lambda-calculus. The constant c stands for a unitary gate, say the Hadamard. We call this language the *orthogonal lambda-calculus*.

$$
\begin{array}{lll}
\textit{Type} & A, B & ::= \top \mid A \multimap B \mid A \oplus B \mid A \otimes B, \\
\textit{Value} & U, V & ::= x^A \mid * \mid c \mid \lambda x^A.M \mid UV \mid U \otimes V \mid \\
 & & \quad \langle \alpha \cdot M, \beta \cdot N \rangle, \\
\textit{Term} & M, N & ::= U \mid MN \mid M \otimes N \mid \langle \alpha \cdot M, \beta \cdot N \rangle \mid \\
 & & \quad \textit{let } x^A \otimes y^B = M \textit{ in } N \mid \\
 & & \quad \textit{let } * = M \textit{ in } N \mid \\
 & & \quad \textit{match } P \textit{ in } (x^A \mapsto M \mid y^B \mapsto N).
\end{array}
$$

$$\frac{}{x : A \rhd x : A}\ (x) \qquad \frac{}{\rhd * : \top}\ (\top_I) \qquad \frac{}{\rhd c : I \oplus I \multimap I \oplus I}\ (c)$$

$$\frac{\Delta, x : A \rhd M : C}{\Delta \rhd \lambda x^A.M : A \multimap C}\ (\lambda) \qquad \frac{\Delta \rhd M : A \multimap B \quad \Gamma \rhd N : A}{\Delta, \Gamma \rhd MN : B}\ (\varepsilon)$$

$$\frac{\Delta \rhd M : A \quad \Gamma \rhd N : B}{\Delta, \Gamma \rhd M \otimes N : A \otimes B}\ (\otimes_I) \qquad \frac{\Delta \rhd M : A \otimes B \quad \Gamma, x : A, y : B \rhd N : C}{\Delta, \Gamma \rhd \ let\ x^A \otimes y^B = M\ in\ N : C}\ (\otimes_E)$$

$$\frac{\Delta \rhd M : \top \quad \Gamma \rhd N : C}{\Delta, \Gamma \rhd let * = M\ in\ N : C}\ (\top_E) \qquad \frac{\Delta \rhd M : A \quad \Delta \rhd N : B \quad |\alpha|^2 + |\beta|^2 = 1}{\Delta \rhd \langle \alpha \cdot M, \beta \cdot N \rangle : A \oplus B}\ (\Sigma)$$

$$\frac{\Delta \rhd P : A \oplus B \quad \Gamma, x : A \rhd M : C \quad \Gamma, y : B \rhd N : D}{\Delta, \Gamma \rhd match\ P\ in\ (x^A \mapsto M \mid y^B \mapsto N) : C \oplus D}\ (\oplus)$$

Table 1: Typing rules for the orthogonal lambda-calculus

Typing rules are given in Table 2 It is to be noted that the match-term can be simulated by a product in the type system ; this product would be different from the type operator $\oplus$.

Reduction steps are

$$\begin{aligned}
(\lambda x.M)U &\to M[U/x], \\
let\ x \otimes y = U \otimes V\ in\ M &\to M[U/x, V/y], \\
let * = *\ in\ M &\to M, \\
match\ \langle \alpha \cdot M', \beta \cdot N' \rangle\ in\ (x \mapsto M \mid y \mapsto N) & \\
&\to \langle \alpha \cdot (\lambda x.M)M', \beta \cdot (\lambda y.N)N' \rangle, \\
&\quad\ c\ \langle \alpha \cdot *, \beta \cdot * \rangle, \\
&\to \langle \frac{1}{\sqrt{2}}(\alpha + \beta) \cdot *, \frac{1}{\sqrt{2}}(\alpha - \beta) \cdot * \rangle,
\end{aligned}$$

plus the usual call-by-value ξ-reductions.

Theorem 2.1. *The language features the usual safety properties: if a closed term M is well-typed of type A, then either it reduces to some other term, or it is a value. Next, any term N to which M reduce to is also of type A. Finally, the language features confluence and strong normalization.* □

3 Embedding in an algebraic System F

Consider lineal with a system-F-like type system, in the style of [Arrighi and Díaz-Caro, 2009].

$$\begin{aligned}
Type \quad A, B \quad ::= \quad & X \mid A \to B \mid \forall X \cdot A, \\
Term \quad M, N \quad ::= \quad & x^A \mid \lambda x^A.M \mid MN \mid \alpha \cdot M \mid \\
& M + N \mid M[A] \mid \Pi X.M.
\end{aligned}$$

One can encode types of Section 1 as

$$\begin{aligned}
A \to B &\mapsto A \to B, \\
A \oplus B &\mapsto \forall X \cdot (A \to X) \to (B \to X) \to X, \\
A \otimes B &\mapsto \forall X \cdot (A \to B \to X) \to X, \\
\top &\mapsto \forall X \cdot X \to X.
\end{aligned}$$

A base term of type $A \oplus B$ is constructed using two terms, one of type A and one of type B. As it should be, there are only two possible base elements of type $A \oplus B$.

A term M from the orthogonal lambda-calculus is encoded to a term $\overline{M}$ from lineal inductively as follows.

$$\begin{aligned}
* &\mapsto \Pi X.\lambda x^X.x, \\
M \otimes N &\mapsto \Pi X.\lambda f.fMN, \\
let\ x \otimes y = M\ in\ N : C &\mapsto M[C](\lambda xy.N), \\
let * = M\ in\ N : C &\mapsto M[C]N, \\
\langle \alpha \cdot M, \beta \cdot N \rangle & \\
&\mapsto \Pi X.\lambda fg.(\alpha \cdot fM + \beta \cdot gN) \\
match\ P\ in\ (x \mapsto M \mid y \mapsto N) & \\
&\mapsto \Pi X.\lambda fg.P[X](\lambda x.fM)(\lambda y.gN).
\end{aligned}$$

The constant term c is represented by

$$\begin{aligned}
\lambda x.\Pi X.\lambda fg.x[X]\,(\lambda y.(\tfrac{1}{\sqrt{2}} \cdot fy + \tfrac{1}{\sqrt{2}} \cdot gy)) & \\
(\lambda y.(\tfrac{1}{\sqrt{2}} \cdot fy - \tfrac{1}{\sqrt{2}} \cdot gy)). &
\end{aligned}$$

This encoding is sound with respect to the calculus of Section 1.

Theorem 3.1. *Given M a term in the orthogonal lambda-calculus, if $M \to N$ then $\overline{M} \to^* \overline{N}$ in the algebraic system F.*

Theorem 3.2. *If M is well-typed in the orthogonal calculus, so is $\overline{M}$ in the algebraic system F.*

4 Embedding into quantum circuits

Theorem 3.1 and 3.2 capture the fact that the orthogonal lambda-calculus is in a sense an algebraic lambda-calculus. We now want to show that this language is also capturing a "correct" notion of orthogonality. For this purpose we show that the language can be compiled into a quantum circuit.

4.1 An interpreter

The main difficulty resides in encoding the lambda-abstraction. For this purpose, we use a trick already sketched in [Preskill, 1999] and show that one can build an interpreter of a quantum circuit as a quantum circuit. This interpreter takes two sets of wires, one for a code describing a quantum circuit using a given set of gates, and the other one for the desired input of the encoded circuit.

Consider an alphabet $\mathcal{A}$. This alphabet contains: symbols for each gate in $\mathcal{U}$; the natural numbers; tuples of symbols of the form x_n, where n are positive integers; the booleans tt and ff. One consider a Hilbert space $\mathcal{H}$ freely generated by the space $\mathcal{A}$. The assembly language consists of instructions of the form $\mathtt{U}\ (x_1, \ldots, x_n)$, where $\mathtt{U}$ is a gate in $\mathcal{U}$. The integer n is the number of entries to the gate $\mathtt{U}$ and $x_1, \ldots x_n$ are indexes referencing quantum data on a quantum array.

A program of k instructions is in a base vector of $\mathcal{H}^{\otimes(2\cdot k)}$ of the form $|I_1 \ldots I_k\rangle$.

An emulator step ε_l is a unitary gate of the space $\mathcal{H}^{\otimes(2+l)}$. The 2 first elements correspond to an instruction and the last l correspond to the processed data. For a given base vector input, the emulator step behaves as follows.

- If the input is of the form

  $$|\phi\rangle = |\mathtt{U}\ (x_1, \ldots, x_k)\rangle \otimes |s_1 \ldots s_l\rangle,$$

  where U is a gate accepting k input and where for all j, $x_j \leqslant l$, then $\varepsilon_l|\phi\rangle$ is

  $$|\mathtt{U}\ k\ (x_1, \ldots, x_k)\rangle \otimes V|s_1 \ldots s_l\rangle$$

  where V is the action $\mathtt{U}$ on the selected entries.

- Otherwise, it acts as the identity on the input.

An emulator $\varepsilon_{k,l}$ taking k instructions and performing the whole program is simply the concatenation of k emulator steps.

Provided that the set of gates $\mathcal{U}$ is closed under controlled operation (that is, if U is in $\mathcal{U}$, so is $C-U$), it is possible to encode the gate ε_l in the assembly language: this is the key for what is following. Note however that one will need emulator steps with larger memory to be able to run it. For simplification we overlook this subtlety: since the language is strictly linear, it is possible to know in advance the size of the various emulators that will be needed.

4.2 Compilation of a typing derivation

In this section we show how to encode the orthogonal lambda-calculus on a quantum circuit. For simplifying, we only consider sum types of the form $A \oplus A$.

We consider quantum circuit with sorted wires: each wire will either be of sort boolean, of sort program, or of sort unit (i.e. the wire corresponding to the zero-dimensional vector space). To each type we assign a list of wires, possibly identified, attached to the structure of the type, as follows: $\top^1$ (1 is of sort unit), $A^{\mathbf{1}} \otimes B^{\mathbf{2}}$, $A^{\mathbf{1}} \oplus^2 A^{\mathbf{3}}$ (2 is of sort boolean), $A^{\mathbf{1}} \multimap^2 B^{\mathbf{3}}$ (2 is of sort program). The bold numbers corresponds to the list of numbers inside the corresponding type. The identification of wires is done as in Table 4.2. We only show a subset of the rules. When bold numbers are identified, this means that all numbers inside the type are identified one-to-one. We call *indexed typing derivation* a typing derivation with such numbers.

$$\frac{}{x : A^{\mathbf{1}} \rhd x : A^{\mathbf{1}}}\ (x) \qquad \frac{}{\rhd * : \top^1}\ (\top_I) \qquad \frac{}{\rhd c : I^1 \oplus^2 I^1 \multimap^3 I^1 \oplus^2 I^1}\ (c)$$

$$\frac{\Delta^{\mathbf{1}}, x : A^{\mathbf{2}} \rhd M : C^{\mathbf{3}}}{\Delta^{\mathbf{1}} \rhd \lambda x.M : A^{\mathbf{2}} \multimap^4 C^{\mathbf{3}}}\ (\lambda) \qquad \frac{\Delta^{\mathbf{1}} \rhd M : A^{\mathbf{4}} \multimap^5 B^{\mathbf{3}} \quad \Gamma^{\mathbf{2}} \rhd N : A^{\mathbf{4}}}{\Delta^{\mathbf{1}}, \Gamma^{\mathbf{2}} \rhd MN : B^{\mathbf{3}}}\ (\varepsilon)$$

$$\frac{\Delta \rhd M : A^{\mathbf{2}} \quad \Delta \rhd N : A^{\mathbf{2}} \quad |\alpha|^2 + |\beta|^2 = 1}{\Delta^{\mathbf{1}} \rhd \langle \alpha \cdot M, \beta \cdot N \rangle : A^{\mathbf{2}} \oplus^3 A^{\mathbf{2}}}\ (\Sigma)$$

$$\frac{\Delta^{\mathbf{1}} \rhd P : A^{\mathbf{3}} \oplus^2 A^{\mathbf{3}} \quad \Gamma^{\mathbf{2}}, x : A^{\mathbf{3}} \rhd M : C^{\mathbf{4}} \quad \Gamma^{\mathbf{2}}, y : A^{\mathbf{3}} \rhd N : C^{\mathbf{4}}}{\Delta^{\mathbf{1}}, \Gamma^{\mathbf{2}} \rhd \mathit{match}\ P\ \mathit{in}\ (x \mapsto M \mid y \mapsto N) : C^{\mathbf{4}} \oplus^2 C^{\mathbf{4}}}\ (\oplus)$$

Table 2: Indexed typing rules

Lemma 4.1. *If $\Delta^{\mathbf{1}} \rhd M : A^{\mathbf{2}}$ has typing derivation π and if $M \to N$ then $\Delta^{\mathbf{1}} \rhd N : A^{\mathbf{2}}$ and it comes with a typing derivation π' with the same set of indices as the one in π.* □

The quantum circuit corresponding to a given indexed typing derivation has wires of two kinds: the ones assigned with values (called *closed wires*), and the others (called *open wires*). The circuit consists of three successive groups of gates: The first one, called *initializer*, initializes the closed wires of sort program (setting them in some state); the last one, called *closing*, that does the opposite process, and the middle one, called *computational circuit*, which is the actual computation. The whole circuit is built inductively as follows.

(x):A bunch of open wires, with no gates. $(\top_I)$: A closed wire 1 of type unit with no gate. (c): Two open wires 1 and 2 and one closed wire 3. 2 is of sort boolean, and 3 is of sort program, initialized with value $|H(2)\rangle$, the program applying the Hadamard gates on the wire 2.

(λ). Takes the circuit coming from $\Delta^{\mathbf{1}}, x : A^{\mathbf{2}} \rhd M : C^{\mathbf{3}}$. Return the wires 1, 2 and 3 with no gates on it, and initialize a closed wire 4 to the program corresponding to M.

(ε): Place the circuit corresponding to N, then connect an emulator evaluating the wire 5.

(Σ): Create a closed wire 3 with value $\alpha|0\rangle + \beta|1\rangle$. Consider the computational circuit coming from N. It is written in terms of elementary quantum gates (Hadamard gates and controlled Hadamard, controlled-controlled Hadamard,...). Transform this circuit by controlling over the wire 3 each elementary gate so that N is applied on the subspace directed by $|0\rangle$. Do the same thing for M but for the subspace directed by $|1\rangle$. Build the resulting computational circuit by plugging the circuits next to each other. The last step consists in entangling the closed wires of sort program appearing both in the typing derivation of N and in the typing derivation of M with the boolean wire 3. The entangling process is done in the initializer and the unentangling process in the closing circuit.

$(\oplus)$: First write the circuit for P. It generates a wire 2. Use the same trick as above to control M and N over it.

Thanks to the strict linearity of the calculus, we have the following result.

Theorem 4.2. *In the context of Lemma 4.1, the quantum circuit corresponding to M is the same as the one corresponding to N.* □

4.3 Encoding quantum circuits in the orthogonal lambda-calculus

The encoding of two quantum bits in the orthogonal lambda-calculus is as follows :

$$\begin{aligned}&\alpha_0|0\rangle \otimes (\alpha_{00}|0\rangle + \alpha_{01}|1\rangle) + \alpha_1|1\rangle \otimes (\alpha_{10}|0\rangle + \alpha_{11}|1\rangle)\\ &\quad = \langle \alpha_0 \cdot \langle \alpha_{00} \cdot *, \alpha_{01} \cdot * \rangle, \alpha_1 \cdot \langle \alpha_{10} \cdot *, \alpha_{11} \cdot * \rangle \rangle\end{aligned}$$

of type $(\top \oplus \top) \oplus (\top \oplus \top)$ We write $[\![qbit^{\otimes 2}]\!]$ for this type. The type $[\![qbit^{\otimes n}]\!]$ is generated on the same principle.

Remark 4.3. One could have been tempted to write it as $(\top \oplus \top) \otimes (\top \oplus \top)$. This is not the expected result: such a type encode non-entangled quantum bits.

To be able to encode quantum circuits in the orthogonal lambda-calculus, we are missing something: in a two quantum bit system $(\top^1 \oplus^2 \top^1) \oplus^3 (\top^1 \oplus^2 \top^1)$, one cannot apply an Hadamard gate on the qubit labeled 3, said to be *at the top of the stack*. One can only apply an Hadamard gate to the qubit 2, *at the bottom of the stack*.

To solve the problem, one term constructor sw is needed, to exchange the role of two adjacent qubits, i.e. moving them along the *stack* of qubits. This term constructor is typed as follows:

$$x : (\top^1 \oplus^2 \top^1) \oplus^3 (\top^1 \oplus^2 \top^1) \rhd sw(x) : (\top^1 \oplus^3 \top^1) \oplus^2 (\top^1 \oplus^3 \top^1)$$

and comes with the reduction rule

$$sw\langle \alpha_0 \cdot \langle \alpha_{00} \cdot *, \alpha_{01} \cdot * \rangle, \alpha_1 \cdot \langle \alpha_{10} \cdot *, \alpha_{11} \cdot * \rangle \rangle \rightarrow \langle \beta_0 \cdot \langle \frac{\alpha_0 \alpha_{00}}{\beta_0} \cdot *, \frac{\alpha_1 \alpha_{10}}{\beta_0} \cdot * \rangle, \beta_1 \cdot \langle \frac{\alpha_0 \alpha_{01}}{\beta_1} \cdot *, \frac{\alpha_1 \alpha_{11}}{\beta_1} \cdot * \rangle \rangle$$

where β_0 and β_1 are renormalization coefficients. The swap operation does not add anything in the compilation. It is only allowing two exchange the role of two "qubit" wires. We are now ready to state the theorem:

Theorem 4.4. *Consider the set of gates closed under controlled operation and generated by the only gate Hadamard. Consider a quantum circuit on n qubits built upon this set of gates and denote by A the unitary map described by this circuit. There exists a typing judgment* $x : [\![qbit^{\otimes n}]\!] \rhd M : [\![qbit^{\otimes n}]\!]$ *whose compiled circuit precisely denote the map A.*

Proof. To apply an Hadamard gate on some quantum bit, send the quantum bit at the bottom of the stack and uses a sequence of *match* operator and the term operator c at the very end. Controlled operations are done similarly, using the possibilities offered by the *match* operator. □

5 Conclusion

In this note we sketched a possible merge between the QML approach of compilation into quantum circuits and a algebraic lambda-calculus.

This raises several questions. First, is it possible to extend the compilation to a calculus allowing duplication ? And recursion ? Then, about the encoding into lineal, the superposition of terms is still quite restrictive. It would be interesting to know if it is possible to loosen it.

References

Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *Proceedings of LICS'05*, pages 249–258, 2005.

Pablo Arrighi and Alejandro Díaz-Caro. A system F accounting for scalars. Preprint: arXiv:0903.3741, July 2009.

Pablo Arrighi and Gilles Dowek. Linear-algebraic lambda-calculus: higher-order, encodings, and confluence. In *Proceedings of RTA'08*, vol 5117 of *LNCS*, pages 17–31, 2008.

Jonathan Grattage. *QML: A functional quantum programming language*. PhD thesis, University of Nottingham, 2007.

John Preskill. Plug-in quantum software. *Nature*, 402: 357–358, 1999.

Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. *MSCS*, 16:527–552, 2006.

Benoît Valiron. *Semantics for a Higher Order Functional Programming Language for Quantum Computation*. PhD thesis, University of Ottawa, 2008.

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