Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 270 (2) (2011) 219–229
www.elsevier.com/locate/entcs

Scalar System F for Linear-Algebraic λ-Calculus: Towards a Quantum Physical Logic

Pablo Arrighi[1] Alejandro Díaz-Caro[2]

Université de Grenoble, Laboratoire d'Informatique de Grenoble
220 rue de la Chimie, 38400 Saint Martin d'Hères, France

Abstract

The Linear-Algebraic λ-Calculus [2] extends the λ-calculus with the possibility of making arbitrary linear combinations of terms $\alpha.\mathbf{t} + \beta.\mathbf{u}$. Since one can express fixed points over sums in this calculus, one has a notion of infinities arising, and hence indefinite forms. As a consequence, in order to guarantee the confluence, $\mathbf{t} - \mathbf{t}$ does not always reduce to $\mathbf{0}$ – only if $\mathbf{t}$ is closed normal. In this paper we provide a System F like type system for the Linear-Algebraic λ-Calculus, which guarantees normalisation and hence no need for such restrictions, $\mathbf{t} - \mathbf{t}$ always reduces to $\mathbf{0}$. Moreover this type system keeps track of 'the amount of a type'. As such it can be seen as probabilistic type system, guaranteeing that terms define correct probabilistic functions. It can also be seen as a step along the quest toward a *quantum physical logic* through the Curry-Howard isomorphism [12].

Keywords: Type Theory, Quantum Logic, Probabilistic Lambda Calculus, Curry-Howard

1 Introduction

In recent years a number of researchers have sought to develop quantum programming languages [6,10]. One of the purposes of such languages is to express quantum programs, but this is not a good enough reason as not many algorithms are known. A more important reason is to provide a framework for reasoning about the programs expressing those algorithms – and about quantum information in general.

Indeed, in classical computer science it is possible to express the reasoning behind a program via several formally-defined logics. They provide an important framework in which to reason and prove properties about the computational processes. Usually

[1] Email: Pablo.Arrighi@imag.fr

[2] Email: Alejandro.Diaz-Caro@imag.fr

[3] A complete version, with detailed proofs, can be found at arXiv:0903.3741, and a full journal version is ongoing

doi:10.1016/j.entcs.2011.01.033

these logics arise via the study of type systems for the language [12]. Related to our motivation there is already a quantum logic [4], which was developed before quantum computing. However it is not known to have a clear relation to quantum programs.

So there is a need for a logic that could aid us in isolating the reasoning behind some quantum algorithms; *i.e.* that would provide a tool to explore whether or not there is some typically 'quantum piece of thinking' behind some algorithms such as Grover's [9] and Shor's [11]. Rather than coming up with some 'ad hoc' logics that would only reflect our current lack of understanding of the deep nature of quantum information, we would like such a logic to arise naturally and legitimately from the study of type system or a quantum programming language. The reason why we use Linear-Algebraic λ-Calculus (*Lineal*) [2] is because it has the advantage of not being bound to a particular type system (it is untyped), and it is minimal, yet it is general enough to describe any quantum computation.

The aim of this work is to set up a System F type system *à la* Curry for the Linear-Algebraic λ-Calculus, able to handle scalars within the types, and hence in some way characterise the *amount of a type*, following the idea of superposition in the sense of how much a term belongs to a type. In this sense this *scalar* type system is a step in a research program which seeks for a form *quantum physical logic* obtained via the Curry-Howard isomorphism; it is also interesting in itself because of its relations with probabilistic systems, Linear Logic ($\mathcal{LL}$), cloning, *etc.* (see section 5).

There are also two other products of this research. First, since *Lineal* can express fixed points over sums (*e.g.* nothing forbids that a term $\mathbf{t}$ should reduce to $\mathbf{t}+\mathbf{t}$), it has infinities, and hence indefinite forms (*e.g.* $\mathbf{t}-\mathbf{t}$). As a consequence, in order to guarantee confluence, $\mathbf{t}-\mathbf{t}$ should not reduce to $\mathbf{0}$. This rule has to be restricted to closed normal terms [2]. As this paper provides a System F like type system for the Linear-Algebraic λ-Calculus and hence guarantees normalisation of all terms, this removes all need for such restrictions, making *Lineal* a more intuitive language: in the typed version $\mathbf{t}-\mathbf{t}$ always reduces to $\mathbf{0}$. Moreover by keeping track of 'the amount of a type' our type system is ready-functioning as a probabilistic type system; guaranteeing that terms define correct probabilistic functions. Although this latter result was not our primary purpose, this does define a probabilistic typed λ-calculus [5].

We now present a small overview of the Linear-Algebraic λ-Calculus (*Lineal*) [2]. Section 2 presents the *scalar* type system with its grammar, equivalences and inference rules. Section 3 shows the subject reduction property giving consistency to the system. Section 4 shows the strong normalisation property for this system, allowing us to lift the above discussed restrictions in the reduction rules. In section 5 we discuss the meaning of this type system; *i.e.* its relationship with probabilistic calculus and Linear Logic ($\mathcal{LL}$). Section 6 presents our conclusions.

1.1 Linear-Algebraic λ-Calculus

Simply, the Linear-Algebraic λ-Calculus is λ-calculus together with the possibility to make arbitrary linear combinations of terms. Consider a first-order language, called *the language of scalars*, containing at least constants 0 and 1 and binary function symbols + and ×. Then the *language of vectors* is a two-sorted language, with a sort for vectors and a sort for scalars, described by the following term grammar:

$$\mathbf{t} ::= x \mid \lambda x\ \mathbf{t} \mid (\mathbf{t}\ \mathbf{t}) \mid \mathbf{0} \mid \alpha.\mathbf{t} \mid \mathbf{t} + \mathbf{t}$$

where α has the sort of scalars and the terms contain vector variables but no scalar variables (see [2, sec. III] for details).

There are rewrite rules for the scalar rewrite system which are compatible with the axioms of rings, and there are 16 AC-rewrite rules for vectors divided in 4 groups:

- Elementary rules such as $\mathbf{u} + \mathbf{0} \to \mathbf{u}$ and $\alpha.(\mathbf{u} + \mathbf{v}) \to \alpha.\mathbf{u} + \alpha.\mathbf{v}$.
- Factorisation rules such as $\alpha.\mathbf{u} + \beta.\mathbf{u} \to (\alpha + \beta).\mathbf{u}$.
- Application rules such as $\mathbf{u}\ (\mathbf{v} + \mathbf{w}) \to (\mathbf{u}\ \mathbf{v}) + (\mathbf{u}\ \mathbf{w})$.
- Beta reduction: $(\lambda x\ \mathbf{t})\ \mathbf{b} \to \mathbf{t}[\mathbf{b}/x]$, where $\mathbf{b}$ is a *base vector* (*i.e.* an abstraction or variable).

Factorisation and application rules have restrictions of the kind '$\mathbf{v} + \mathbf{w}$ is closed normal' to avoid problems occurring when handling non-terminating terms. These kind of problems are akin to indefinite forms $\infty - \infty$.

Example 1.1 It is possible to define fixed point operators such as

$$\mathbf{Y} = \lambda y\ ((\lambda x\ (y + (x\ x)))\ (\lambda x\ (y + (x\ x))))$$

Then, let $\mathbf{b}$ be a base vector, so $(\mathbf{Y}\ \mathbf{b})$ reduces to $\mathbf{b} + (\mathbf{Y}\ \mathbf{b})$, so if there were no restriction on rules as the factorisation rules, then the term $(\mathbf{Y}\ \mathbf{b}) - (\mathbf{Y}\ \mathbf{b})$ would reduce to $(1 + (-1)).(\mathbf{Y}\ \mathbf{b})$ which reduces to $\mathbf{0}$. However, note that it would also reduce to $\mathbf{b} + (\mathbf{Y}\ \mathbf{b}) - (\mathbf{Y}\ \mathbf{b})$ breaking the confluence.

By asking for $\mathbf{u}$ closed normal in the factorisation rules this problem is avoided (see [2, sec. II] for more details).

2 The Scalar Type System

The set of types is defined by the following abstract grammar:

$$\mathcal{T} = \mathcal{U} \mid \forall X.\mathcal{T} \mid \alpha.\mathcal{T} \mid \overline{0}, \qquad \mathcal{U} = X \mid \mathcal{U} \to \mathcal{T} \mid \forall X.\mathcal{U}$$

where $\alpha \in \mathcal{S}$ and $(\mathcal{S}, +, \times)$ is a commutative ring. Note that the grammar for $\mathcal{U}$, called *unit types*, does not allow scalars except to the right of an arrow.

We also define an equivalence between types as follows:

Definition 2.1 Let $\alpha, \beta \in \mathcal{S}$ and $T \in \mathcal{T}$. We define a type equivalence $\equiv$

- $\alpha.\overline{0} \equiv \overline{0}$
- $0.T \equiv \overline{0}$
- $1.T \equiv T$
- $\alpha.(\beta.T) \equiv (\alpha \times \beta).T$
- $\forall X.\alpha.T \equiv \alpha.\forall X.T$

The typical System F rules are changed to handle scalars as showed below

$$\frac{}{\Gamma, x:U \vdash x:U}\, ax[U] \qquad \frac{\Gamma \vdash \mathbf{u}:\alpha.(U \to T) \qquad \Gamma \vdash \mathbf{v}:\beta.U}{\Gamma \vdash (\mathbf{u}\ \mathbf{v}):(\alpha \times \beta).T} \to E \qquad \frac{\Gamma, x:U \vdash \mathbf{t}:T}{\Gamma \vdash \lambda x\ \mathbf{t}:U \to T} \to I[U]$$

$$\frac{\Gamma \vdash \mathbf{u}:\forall X.T}{\Gamma \vdash \mathbf{u}:T[U/X]}\, \forall E[X := U] \qquad \frac{\Gamma \vdash \mathbf{u}:T}{\Gamma \vdash \mathbf{u}:\forall X.T}\, \forall I[X] \text{ with } X \notin FV(\Gamma)$$

$$\frac{}{\Gamma \vdash \mathbf{0}:\overline{0}}\, ax_{\overline{0}} \qquad \frac{\Gamma \vdash \mathbf{u}:\alpha.T \qquad \Gamma \vdash \mathbf{v}:\beta.T}{\Gamma \vdash \mathbf{u}+\mathbf{v}:(\alpha+\beta).T} +I \qquad \frac{\Gamma \vdash \mathbf{u}:T}{\Gamma \vdash \alpha.\mathbf{u}:\alpha.T}\, sI[\alpha]$$

Where $U \in \mathcal{U}$ and $Name[Cond]$ represents a family of rules – one for each condition.

3 Subject reduction

The following theorem ensures that typing is preserved by reduction, making our type system consistent.

Theorem 3.1 (Subject Reduction) *Let $t \to^* t'$. Then $\Gamma \vdash t:T \Rightarrow \Gamma \vdash t':T$.*

Proof. The proof proceeds by checking that every reduction rule preserves the type. Due to the large number of such reduction rules the proof is highly technical, with 27 auxiliary lemmas. Some of the interesting ones are stated below. □

Lemma 3.2 (α-Unit) *$\forall T \in \mathcal{T},\ \exists U \in \mathcal{U}, \exists \alpha \in \mathcal{S}$ such that $T \equiv \alpha.U$.*

Lemma 3.3 (Complement) *Let denote by S_n to a sequent which admits a proof of depth n. Let $S_n = \Gamma \vdash \boldsymbol{u}+\boldsymbol{v}:\alpha.T$. Then $\exists \delta, \gamma, r, s$ with $\delta+\gamma = \alpha$ and $\max(r,s) < n$ such that $S_r = \Gamma \vdash \boldsymbol{u}:\delta.T$ and $S_s = \Gamma \vdash \boldsymbol{v}:\gamma.T$.*

Lemma 3.4 (Base vectors are in unit) *Let $\boldsymbol{b}$ be a base vector,* i.e. *a variable or an abstraction. Then $\Gamma \vdash \boldsymbol{b}:T \Rightarrow \exists U \in \mathcal{U}$ s.t. $T \equiv U$.*

4 Strong normalisation

As our system is derived from System F, it can be proved that it has the strong normalisation property. In order to do so we set up another type system, a straightforward extension of System F, called $\lambda 2^{la}$, and prove strong normalisation for it. Then we show that every term which has a type in *scalar* has a type in $\lambda 2^{la}$.

In this section we use $\Gamma \Vdash \mathbf{t}:T$ to say that it is possible to derive the type $T \in \mathbb{T}(\lambda 2^{la})$ for the term $\mathbf{t}$ in the context of Γ by using the typing rules from $\lambda 2^{la}$. And we just use $\vdash$ for *scalar*. In addition, we use $R^{\triangleleft}$ to refer to a type rule in $\lambda 2^{la}$.

Definition 4.1 The type rules of $\lambda 2^{la}$ are the same as System F plus the following rules:

$$\frac{}{\Gamma \Vdash \mathbf{0}:A}\, ax_0^{\triangleleft} \qquad \frac{\Gamma \Vdash \mathbf{u}:A \qquad \Gamma \Vdash \mathbf{v}:A}{\Gamma \Vdash \mathbf{u}+\mathbf{v}:A} +I^{\triangleleft} \qquad \frac{\Gamma \Vdash \mathbf{t}:A}{\Gamma \Vdash \alpha.\mathbf{t}:A}\, \alpha I^{\triangleleft}$$

In order to prove strong normalisation we extend the proof for $\lambda 2$ which is given in [3, sec 4.3]. Most of the definitions are taken from this reference – with slight modifications to handle the extra $\lambda 2^{la}$ rules.

Definition 4.2 $SN = \{\mathbf{t} \in \Lambda \mid \mathbf{t} \text{ is strongly normalising}\}$.

Definition 4.3

(i) A subset $X \subseteq SN$ is called *saturated* if
- (a) $\forall n \geq 0, (((x\ \mathbf{t}_1)\dots)\ \mathbf{t}_n) \in X$ where $\mathbf{t}_i \in SN$;
- (b) $\mathbf{v}[\mathbf{b}/x]\ \overrightarrow{\mathbf{t}} \in X \Rightarrow (\lambda x\ \mathbf{v})\ \mathbf{b}\ \overrightarrow{\mathbf{t}} \in X$;
- (c) $\mathbf{t}, \mathbf{u} \in X \Rightarrow \mathbf{t} + \mathbf{u} \in X$;
- (d) $\forall \alpha, \mathbf{t} \in X \Rightarrow \alpha.\mathbf{t} \in X$;
- (e) $\forall i \in I, ((\mathbf{u}_i\ \mathbf{w}_1)\dots\ \mathbf{w}_n) \in X \Rightarrow \left(\left(\sum_{i\in I} \mathbf{u}_i\right)\ \mathbf{w}_1\right)\dots\ \mathbf{w}_n \in X$;
- (f) $\forall i \in I, (\mathbf{u}\ \mathbf{w}_i) \in X \Rightarrow \mathbf{u}\ \left(\sum_{i\in I} \mathbf{w}_i\right) \in X$;
- (g) $\alpha.((\mathbf{t}_1\ \mathbf{t}_2)\dots\mathbf{t}_n) \in X \Leftrightarrow ((\mathbf{t}_1\ \mathbf{t}_2)\dots\alpha.\mathbf{t}_k)\dots\mathbf{t}_n \in X\ (1 \leq k \leq n)$;
- (h) $\mathbf{0} \in X$;
- (i) $\forall \overrightarrow{\mathbf{t}} \in SN, (\mathbf{0}\ \overrightarrow{\mathbf{t}}) \in X$;
- (j) $\forall \mathbf{t}, \overrightarrow{\mathbf{u}} \in SN, (\mathbf{t}\ \mathbf{0})\ \overrightarrow{\mathbf{u}} \in X$.

(ii) $SAT = \{X \subseteq \Lambda \mid X \text{ is saturated}\}$

Lemma 4.4 (i) $SN \in SAT$, (ii) $A, B \in SAT \Rightarrow A \to B \in SAT$, (iii) *For all collection A_i of members of SAT, $\bigcap_i A_i \in SAT$.*

Definition 4.5 Let $\xi(\cdot) : TVar \to SAT$. We define the following mapping

$$[\![X]\!]_\xi = \xi(X) \qquad [\![A \to B]\!]_\xi = [\![A]\!]_\xi \to [\![B]\!]_\xi \qquad [\![\forall X.T]\!]_\xi = \bigcap_{Y\in SAT} [\![T]\!]_{\xi(X:=Y)}$$

Lemma 4.6 *Given a valuation ξ and a type T, $[\![T]\!]_\xi \in SAT$*

Definition 4.7 For $\Gamma = x_1 : A_1, \dots, x_n : A_n$, $\Gamma \vDash \mathbf{t} : T$ means that $\forall \xi$, $[x_1 \in [\![A_1]\!]_\xi, \dots x_n \in [\![A_n]\!]_\xi \Rightarrow \mathbf{t} \in [\![T]\!]_\xi]$.

Theorem 4.8 (Soundness) $\Gamma \Vdash \boldsymbol{t} : T \Rightarrow \Gamma \vDash \boldsymbol{t} : T$.

Proof. (Sketch) We prove this by induction on the derivation of $\Gamma \Vdash \mathbf{t} : T$ (In fact the definition of $\vDash$ is slightly different to strengthen the induction hypothesis). □

Theorem 4.9 (SN for $\lambda 2^{la}$) $\Gamma \Vdash \boldsymbol{t} : T \Rightarrow \boldsymbol{t}$ *is strongly normalising.*

Proof. Let $\Gamma \Vdash \mathbf{t} : T$. Then by theorem 4.8, $\Gamma \vDash \mathbf{t} : T$ and so by definition 4.7, if $\forall (x_i : A_i) \in \Gamma$, $x_i \in [\![A_i]\!]_\xi$ then $\mathbf{t} \in [\![T]\!]_\xi$. Note that by lemma 4.6, $[\![A_i]\!]_\xi$ is saturated, then $x_i \in [\![A_i]\!]_\xi$, and so $\mathbf{t}$ is strong normalising since $[\![T]\!]_\xi \subseteq SN$ also by lemma 4.6. □

Definition 4.10 Let $(\cdot)^\natural$ be a map from $\mathcal{T} \setminus \{\overline{0}\}$ to $\mathbb{T}(\lambda 2^{la})$ defined as follows.

$$(\alpha.X)^\natural = X \qquad (\alpha.\forall X.T)^\natural = \forall X.T^\natural \qquad (\alpha.A \to B)^\natural = A^\natural \to B^\natural$$
$$(A[B/X])^\natural = A^\natural[B^\natural/X] \qquad \forall T_1 \equiv T_2,\ T_1^\natural = T_2^\natural$$

Notation $\Gamma^\natural = \{(x : T^\natural) \mid (x : T) \in \Gamma\}$ and $\overline{0}^\natural = T$ for whatever type $T \in \mathbb{T}(\lambda 2^{la})$.

Lemma 4.11 (Correspondence with $\lambda 2^{la}$) $\Gamma \vdash \boldsymbol{t} : T \Rightarrow \Gamma^\natural \Vdash \boldsymbol{t} : T^\natural$.

Proof. Let n be the minimum number of steps to derive $\Gamma \vdash \mathbf{t} : T$. We proceed by induction over n.

Base cases $(n = 1)$

(i) $\dfrac{}{\Gamma, x:U \vdash x:U}\, ax[U]$ $\quad (\Gamma, x:U)^\natural = \Gamma^\natural, x:U^\natural$, so by $ax^\lhd$, $(\Gamma, x:U)^\natural \Vdash x:U^\natural$.

(ii) $\dfrac{}{\Gamma \vdash \mathbf{0}:\overline{0}}\, ax_{\overline{0}}$ $\quad$ By $ax_0{}^\lhd$, $\Gamma^\natural \Vdash \mathbf{0}:T$ for any $T \in \mathbb{T}(\lambda 2^{la})$, so take $\overline{0}^\natural = T$.

Inductive cases In all cases, if $A \equiv \overline{0}$ we can take $A^\natural = T$ for any $T \in \mathbb{T}(\lambda 2^{la})$ and it is still valid by using the type equivalences.

(i) $\dfrac{\Gamma \vdash \mathbf{u}:\alpha.(U \to B) \qquad \Gamma \vdash \mathbf{v}:\beta.U}{\Gamma \vdash \mathbf{u}\ \mathbf{v}:(\alpha \times \beta).B} \to E$ $\quad$ By the induction hypothesis $\Gamma^\natural \Vdash \mathbf{u}:U^\natural \to B^\natural$ and $\Gamma^\natural \Vdash \mathbf{v}:U^\natural$, so by $\to E^\lhd$, $\Gamma^\natural \Vdash \mathbf{u}\ \mathbf{v}:B^\natural = ((\alpha \times \beta).B)^\natural$.

(ii) $\dfrac{\Gamma, x:U \vdash \mathbf{t}:A}{\Gamma \vdash \lambda x\ \mathbf{t}:U \to A} \to I[U]$ $\quad$ By the induction hypothesis $\Gamma^\natural, x:U^\natural \Vdash \mathbf{t}:A^\natural$, so by $\to I^\lhd$, $\Gamma^\natural \Vdash \lambda x\ \mathbf{t}:U^\natural \to A^\natural = (U \to A)^\natural$.

(iii) $\dfrac{\Gamma \vdash \mathbf{t}:\forall X.B}{\Gamma \vdash \mathbf{t}:B[U/X]} \forall E[X := U]$ $\quad$ By the induction hypothesis $\Gamma^\natural \Vdash \mathbf{t}:(\forall X.B)^\natural = \forall X.B^\natural$, so by $\forall E^\lhd$, $\Gamma^\natural \Vdash \mathbf{t}:B^\natural[U^\natural/X]$.

(iv) $\dfrac{\Gamma \vdash \mathbf{t}:T}{\Gamma \vdash \mathbf{t}:\forall X.T} \forall I[X]$ $\quad$ By the induction hypothesis $\Gamma^\natural \Vdash \mathbf{t}:T^\natural$, so by $\forall I^\lhd$, $\Gamma^\natural \Vdash \mathbf{t}:\forall X.T^\natural = (\forall X.T)^\natural$.

(v) $\dfrac{\Gamma \vdash \mathbf{u}:\alpha.A \qquad \Gamma \vdash \mathbf{v}:\beta.A}{\Gamma \vdash \mathbf{u}+\mathbf{v}:(\alpha+\beta).A} +I$ $\quad$ By the induction hypothesis $\Gamma^\natural \Vdash \mathbf{u}:A^\natural$ and $\Gamma^\natural \Vdash \mathbf{v}:A^\natural$, so by $+I^\lhd$, $\Gamma^\natural \Vdash \mathbf{u}+\mathbf{v}:A^\natural = ((\alpha+\beta).A)^\natural$.

(vi) $\dfrac{\Gamma \vdash \mathbf{t}:A}{\Gamma \vdash \alpha.\mathbf{t}:\alpha.A} sI[\alpha]$ $\quad$ By the induction hypothesis $\Gamma^\natural \Vdash \mathbf{t}:A^\natural$, so by $\alpha I^\lhd$, $\Gamma^\natural \Vdash \alpha.\mathbf{x}:A^\natural = (\alpha.A)^\natural$.

□

Theorem 4.12 (Strong normalisation) $\Gamma \vdash \boldsymbol{t}:T \Rightarrow \boldsymbol{t}$ *is strongly normalising.*

Proof. By Lemma 4.11 $\Gamma^\natural \Vdash \mathbf{t}:T^\natural$, then by theorem 4.9 $\mathbf{t}$ is strong normalising. □

Theorem 4.12 ensures that all the typable terms have a normal form, hence we do not need the restriction of closed normal forms on the reduction rules of *Lineal*, as they were there only due to the impossibility of checking the normalisation property in the typeless setting. This is an important simplification of the language.

Taking up again example 1.1, terms like $\mathbf{Y}$ are simply not allowed in this typed setting, as all the terms are strong normalising. So we do not have infinities, and hence the restrictions to have closed normal forms on factorising reductions can be dropped – giving us a more powerful and intuitive tool to represent superpositions and work with them. This reinforces also the idea there is a formal correspondence between normalisation in rewriting and expressions of finite norm in algebra.

Example 4.13 Given some complicated term $\mathbf{t}$. If $\mathbf{t}$ has a type, then it must be strong normalising, and so a term like $\alpha.\mathbf{t} - \alpha.\mathbf{t}$ can be reduced by a factorisation rule into $(\alpha - \alpha).\mathbf{t}$. This reduces in one step to $\mathbf{0}$, without the need to reduce $\mathbf{t}$.

5 Discussion

5.1 Computational content

By restricting our scalars to positive reals, the *scalar* type system can be seen as a probabilistic *type system*. For example, one can type functions such as $f ::= \lambda x\ \{x\ [\frac{1}{2}.(\text{true} + \text{false})]\ [\frac{1}{4}.\text{true} + \frac{3}{4}.\text{false}]\} : \mathcal{B} \to \mathcal{B}$ with the type system serving as a guarantee that the function conserves probabilities summing to one (see [2, sec. III] for details about the {} and [] notations, which are not relevant to this discussion).

The term can be seen as a function such that, if it receives true, it returns a balanced distribution of true and false, but if it receives false, it returns *more* false *than* true. We can ask what would the result be if it receives $\frac{1}{2}.(\text{true} + \text{false})$: $f\ \left(\frac{1}{2}.(\text{true} + \text{false})\right) \longrightarrow^* \frac{3}{8}.\text{true} + \frac{5}{8}.\text{false}$, and find that everything works as expected, with probabilities summing to one.

To make this intuition more formal, let us define a type system with the rules and grammar of *scalar*, where the valid types are the classic ones (*i.e.* types exempt of any scalar) and all other types are intermediate types:

Definition 5.1 We define the *probabilistic* type system to be the *scalar* type system with the following restrictions:
• $\mathcal{S} = \mathbb{R}^+$,
• Contexts in the *probabilistic* type system are sets of tuples $(x : C)$ such that C is in the set $\mathcal{C} \subsetneq \mathcal{U}$ of classic types, that is types exempt of any scalar,
• Type variables run over classic types instead of unit types, *i.e.* the family of $\forall E[X := C]$ rules accepts only $C \in \mathcal{C}$,
• The final sequent is well-formed in the following sense: $\forall C \in \mathcal{C}$, any derivable sequent $\Gamma \vdash \mathbf{t} : C$ is well-formed, even if the derivation has scalars appearing at intermediate stages.

We define a weight function to check when a term is a probability distribution of terms:

Definition 5.2 Let $\omega : \Lambda \to \mathbb{R}^+$ be a function defined inductively by:

$$\omega(\mathbf{0}) = 0 \qquad \omega(\mathbf{t}_1 + \mathbf{t}_2) = \omega(\mathbf{t}_1) + \omega(\mathbf{t}_2) \qquad \omega(\mathbf{b}) = 1$$
$$\omega(\alpha.\mathbf{t}) = \alpha \times \omega(\mathbf{t}) \qquad \omega(\mathbf{t}_1\ \mathbf{t}_2) = \omega(\mathbf{t}_1) \times \omega(\mathbf{t}_2)$$

where $\mathbf{b}$ is a base vector.

So, we can enunciate the following theorem that shows that every term with a well-formed typing in the *probabilistic* type system reduces to a term with weight 1:

Theorem 5.3 (Terms in *probabilistic* have weight 1) *Let $\Gamma \vdash \boldsymbol{t} : C$ be well-formed, then $\omega(\boldsymbol{t}\downarrow) = 1$.*

Proof. (Sketch) We prove that $\Gamma \vdash \mathbf{t} : \alpha.C \Rightarrow \omega(\mathbf{t}\downarrow) = \alpha$ by structural induction over $\mathbf{t}\downarrow$. □

By [2, Proposition 2], closed normal terms have form $\sum_{i=1}^{n} \alpha_i.\lambda x\ \mathbf{t}_i + \sum_{j=1}^{m} \lambda x\ \mathbf{u}_j$, then the above theorem entails that $\sum_{i=1}^{n} \alpha_i + m = 1$.

Hence the *probabilistic* type system, an easy variation of the *scalar* type system, specializes *Lineal* into a probabilistic higher-order λ-calculus.

5.2 Computational content: quantum computation

In [2, sec. IV] it is shown how to encode quantum bits and gates in an original manner. Such an encoding no longer works under the *scalar* type system. This is because *conditional* functions have to have the same type on each branch, which in this case entails that they must have the same scalar in their type.

Example 5.4 The Hadamard gate is encoded in the untyped calculus in the following way: $H = \lambda y\ \{y\ [\frac{1}{\sqrt{2}}.(\text{false} + \text{true})]\ [\frac{1}{\sqrt{2}}.(\text{false} - \text{true})]\}$. In this system the types for each branch of the gate are different: $\frac{1}{\sqrt{2}}.(\text{false} + \text{true}) : \frac{1}{\sqrt{2}}.((1+1).\mathcal{B}) \equiv \sqrt{2}.\mathcal{B}$ and $\frac{1}{\sqrt{2}}.(\text{false} - \text{true}) : \frac{1}{\sqrt{2}}.((1-1).\mathcal{B}) \equiv \overline{0}$. Hence it is not possible to give a type to such an encoding of the Hadamard gate, as each branch has a different type.

This is a trivial consequence of the fact that we are characterising vectors by only magnitude and sign, not taking into account their directions. In a future version of the type system we plan to be able to distinguish qubit vectors from non-qubit vectors, allowing each branch to return qubit vectors in this example. This could be done by keeping track of the orthogonality of types. Orthogonality questions have already been raised in quantum programing languages such as QML [1].

5.3 Working out the logical content

We can use the Curry-Howard isomorphism in an original way: instead of making the relationship between an already set logic and our type system, we can remove all the terms from the inference rules of *scalar* and set up the *Scalar Logic*. This Scalar Logic ($\mathcal{SL}$) is the first non-trivial logic obtained from a Curry-Howard isomorphism of a type system for a language inspired by quantum computing. In this sense it is not an *ad hoc* logic that we have fabricated in order to convey an *a priori* meaning, but rather a logic that arises naturally and legitimately by applying a well-established method upon a quantum programming language. This discussion section is an attempt to understand the *a posteriori* meaning of this logic. It is somewhat informal, experimental, yet provides several intuitions ending with a clear-cut result (theorem 5.7).

$\mathcal{SL}$ seems to count the quantity of proofs of a proposition that we have present in the proof of a sequent. For instance, the $+I$ rule suggests that we need two proofs of A present in the proof tree in order to prove $2.A$. However, we immediately see that there is a problem with the family of rules $sI[\cdot]$, because they trivially say that if we have one proof of a proposition then we have many proofs. If we were to remove this family of rules we would lose subject reduction (specifically in the rule $\alpha.\mathbf{u} + \beta.\mathbf{u} \to (\alpha + \beta).\mathbf{u}$), so we would need to add an alternative typing rule such as

$$\frac{\Gamma \vdash \alpha.\mathbf{u}:\delta.T \qquad \Gamma \vdash \beta.\mathbf{u}:\gamma.T}{\Gamma \vdash (\alpha+\beta).\mathbf{u}:(\delta+\gamma).T}$$

From this alternative rule we would still be able to derive $sI[\cdot]$ as a theorem, but in this process we would have to repeat the proof of A α times in order to get to $\alpha.A$, and so this proof-counting interpretation would continue to hold (note that every α would now be a natural number) to some extent.

Nevertheless, consider the duplicator $\vdash \lambda x\ 2.x:\forall X.X \to 2.X$, which allows

$$\dfrac{\dfrac{\dfrac{}{\vdash \lambda x\ 2.x:\forall X.X \to 2.X}}{\vdash \lambda x\ 2.x:A \to 2.A}\forall E[X:=A] \qquad \dfrac{\vdots}{\vdash \mathbf{t}:A}}{\vdash (\lambda x\ 2.x)\ \mathbf{t}:2.A} \to E$$

without needing to prove A twice; *i.e.* there is a fixed proof method for proving $2.A$ from a single proof of A. Hence we can say only that if we do use the alternative rule instead of $sI[\cdot]$ then the proof-counting interpretation holds, but only after cut-elimination; *i.e.* the removal of all $\to E$ in the derivation tree. The idea of counting proofs, and hence considering them as resources, is very similar to *linear logic* ($\mathcal{LL}$) [7] or *bounded linear logic* ($\mathcal{BLL}$) [8]. However $\mathcal{LL}$ and $\mathcal{BLL}$ do not only *count* the amount of resources available, they make it impossible to add new resources. In $\mathcal{LL}$ the context puts a definite limit on how many resources we can use. In $\mathcal{SL}$, on the other hand, we are counting the amount of proofs we *used* modulo cut-elimination, but nothing prevented us from using many more proofs. To make this more formal let us introduce some notation:

Remark 5.5 We denote by $\Gamma \vdash A \Rightarrow \Delta \vdash B$ the fact that we have a witness derivation:

$$\dfrac{\dfrac{\Gamma \vdash A}{\vdots}}{\Delta \vdash B}$$

That is without any other leaf than $\Gamma \vdash A$ (all other leaves of the tree are $ax[\cdot]$ or $ax_{\overline{0}}$ rules) we can have $\Delta \vdash B$ as the conclusion of the derivation tree. We call this the meta-implication, because in general we may have $\Gamma \vdash A \Rightarrow \Delta \vdash B$ but not $A \to B$, due to the particular form the $\to I[\cdot]$ rules may have. For instance in the $\mathcal{SL}$ we simply cannot have any type left of an arrow, only $\mathcal{U}$ types. The distinction between $\Gamma \vdash A \Rightarrow \Delta \vdash B$ and $A \to B$ refines our discussion.

In $\mathcal{SL}$ we do not have $A \to 2.A$ for all A, only for $\mathcal{U}$, which could suggest interpreting $\mathcal{U}$ as the *banged* propositions of $\mathcal{LL}$. However, this interpretation is too fragile; consider lemma 3.2: we know that any $A \equiv \alpha.U$, and hence the above duplicator $\forall X X \to 2.X$ would still be able to produce $2.A$, without even the need to depend upon A. Therefore, we clearly have $\Gamma \vdash A \Rightarrow \Gamma \vdash 2.A$, unlike in $\mathcal{LL}$, whether A is $\mathcal{U}$ or not.

We will now see whether a stronger connection with $\mathcal{LL}$ can arise in a different manner. The 'copy' of $\mathcal{LL}$ could be interpreted in our calculus as $\lambda x\ x \otimes x$ (where $\otimes$ is the classical encoding for tuples, see [2, sec IV]). Here again we do not have $A \to A \otimes A$ for all A, only for $\mathcal{U}$ (more formally consider an extra type equivalence

$(\alpha.U) \otimes (\beta.V) \equiv \alpha.\beta.(U \otimes V))$. Again, this suggests interpreting $\mathcal{U}$ as banged propositions of $\mathcal{LL}$, but this again fails since we have that $\Gamma \vdash A \Rightarrow \Delta \vdash A \otimes A$, unlike $\mathcal{LL}$, whether A is $\mathcal{U}$ or not. However the crucial difference this time is that there is no fixed proof method for doing the cloning. Indeed, if $A \equiv \alpha.U$ then a copy machine of the form $\forall X.X \to X \otimes X$ will definitely not yield $(\alpha.U) \otimes (\alpha.U) = A \otimes A$, but $\alpha.(U \otimes U)$, which is not expressible in $\mathcal{LL}$. The proof method which derives $A \otimes A$ from A crucially depends upon A for non-unit types. We capture this intuition as follows:

Lemma 5.6 (Scalar Logic rules are deterministic) *Let R be a typing rule and let Q_i, Q'_i with $i = 1, \ldots, n$ be sequents. Then*

$$\{\frac{Q_1, \ldots, Q_n}{S} R \wedge \frac{Q'_1, \ldots, Q'_n}{S'} R \wedge \forall i,\ Q_i \equiv Q'_i\} \Rightarrow S \equiv S'$$

Hence we can define Π as a tree of typing rules and think of Π as a function from lists of sequents to proofs. This is what we called informally a *proof method* in the previous discussion.

Theorem 5.7 (No-cloning) *$\nexists \Pi$ such that $\forall A, \Pi(\Gamma \vdash A)$ is a witness of $\Gamma \vdash A \Rightarrow \Delta \vdash A \otimes A$.*

Note that our no-cloning allows the existence of a $\Pi(\Gamma \vdash A)$ that is a witness of $\Gamma \vdash A \Rightarrow \Delta \vdash A \otimes A$, but does not allow the *same* proof method Π to work for any type. Hence the non-existence of a universal proof method Π for doing cloning in $\mathcal{SL}$ is very much akin to the non-existence of a universal context Γ for doing cloning in $\mathcal{LL}$. Notice that by replacing quantification over Π by quantification over Γ, then theorem 5.7 is a well-known result of $\mathcal{LL}$. However, the $\mathcal{SL}$ way of phrasing no-cloning might be more in line with quantum theory. Quantum theory states that it is not possible to have a *universal* cloning machine, but does allow cloning machines of *specific* vectors. The interpretation of Π as a machine is possibly more natural than thinking about resources in Γ as machines.

6 Summary of conclusions

- The *scalar* type system can be seen as a probabilistic type system guaranteeing probabilistic functions to be well defined.
- As a direct consequence of theorem 4.12, some restrictions were lifted in the reduction rules, allowing the factorisation not only of closed normal terms but also of strong normalising terms, which all the typable terms are.
- Theorem 5.7 possibly indicates that this approach is more in line with quantum theoretical thinking than the $\mathcal{LL}$ approach, as in $\mathcal{SL}$ the restrictions is placed on cloning machines rather than and resources.
- This *scalar* type system is the first step towards a future *vectorial* type system. The scalar type system is able to handle the *magnitude* and *signs* for type vectors. In a future system we will deal with the *direction*, *i.e.* addition and orthogonality of types.

Acknowledgement

We would like to thank to Gilles Dowek, Benoît Valiron, Frédéric Prost, Jonathan Grattage, Simon Perdrix and Philippe Jorrand for enlightening discussions.

References

[1] Altenkirch, T., J. J. Grattage, J. K. Vizzotto and A. Sabry, *An algebra of pure quantum programming*, Electronic Notes in Theoretical Computer Science **170** (2007), pp. 23–47.

[2] Arrighi, P. and G. Dowek, *Linear-algebraic λ-calculus: higher-order, encodings and confluence*, Lecture Notes in Computer Science (RTA'08) **5117** (2008), pp. 17–31.

[3] Barendregt, H. P., "Lambda calculi with types," Handbook of Logic in Computer Science **2**, Clarendon Press, Oxford, 1992.

[4] Birkhoff, G. and J. von Neumann, *The logic of quantum mechanics*, Annals of Mathematics **37** (1936), pp. 823–843.

[5] Di Pierro, A., C. Hankin and H. Wiklicky, *Probabilistic λ-calculus and quantitative program analysis*, Journal of Logic and Computation **15** (2005), pp. 159–179.

[6] Gay, S. J., *Quantum programming languages: survey and bibliography*, Mathematical Structures in Computer Science **16** (2006), pp. 581–600.

[7] Girard, J.-Y., *Linear logic*, Theoretical Computer Science **50** (1987), pp. 1–102.

[8] Girard, J.-Y., A. Scedrov and P. J. Scott, *Bounded linear logic: a modular approach to polynomial-time computability*, Theoretical Computer Science **97** (1992), pp. 1–66.

[9] Grover, L. K., *A fast quantum mechanical algorithm for database search*, in: *STOC '96: Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (1996), pp. 212–219.

[10] Selinger, P., *A brief survey of quantum programming languages*, in: *Functional and Logic Programming*, Lecture Notes in Computer Science **2998** (2004), pp. 1–6.

[11] Shor, P. W., *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing **26** (1997), pp. 1484–1509.

[12] Sørensen, M. H. and P. Urzyczyn, "Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics)," Elsevier Science Inc., New York, NY, USA, 2006.