Equivalence of Algebraic λ-calculi – extended abstract*–

Alejandro Díaz-Caro
LIG, Université de Grenoble, France
Alejandro.Diaz-Caro@imag.fr

Simon Perdrix
CNRS, LIG, Université de Grenoble, France
Simon.Perdrix@imag.fr

Christine Tasson
CEA-LIST, MeASI, France
Christine.Tasson@cea.fr

Benoît Valiron
LIG, Université de Grenoble, France
Benoit.Valiron@imag.fr

We examine the relationship between the *algebraic λ-calculus* (λ_{alg}) [9] a fragment of the differential λ-calculus [4]; and the *linear-algebraic λ-calculus* (λ_{lin}) [1], a candidate λ-calculus for quantum computation. Both calculi are algebraic: each one is equipped with an additive and a scalar-multiplicative structure, and the set of terms is closed under linear combinations. We answer the conjectured question of the simulation of λ_{alg} by λ_{lin} [2] and the reverse simulation of λ_{lin} by λ_{alg}. Our proof relies on the observation that λ_{lin} is essentially call-by-value, while λ_{alg} is call-by-name. The former simulation uses the standard notion of thunks, while the latter is based on an algebraic extension of the continuation passing style. This result is a step towards an extension of call-by-value / call-by-name duality to algebraic λ-calculi.

1 Introduction

Context. Two algebraic versions of the λ-calculus arise independently in distinct contexts: the algebraic λ-calculus (λ_{alg}) and the linear algebraic λ-calculus (λ_{lin}). The former has been introduced in the context of linear logic as a fragment of the differential λ-calculus. The latter has been introduced as a candidate λ-calculus for quantum computation: in λ_{lin}, a linear combination of terms reflects the phenomenon of superposition, i.e. the capability for a quantum system to be in two or more states at the same time.

Linearity of functions and arguments. In both languages, functions which are linear combinations of terms are interpreted pointwise: $(\alpha.f+\beta.g)\ x = \alpha.(f)\ x+\beta.(g)\ x$, where "." is the external product. The two languages differ on the treatment of the arguments. In λ_{lin}, any function is considered as a linear map: $(f)\ (\alpha.x+\beta.y) \to^*_\ell \alpha.(f)\ x+\beta.(f)\ y$, reflecting the fact that any quantum evolution is a linear map; while λ_{alg} has a call-by-name evolution: $(\lambda x M)\ N \to_a M[x := N]$, without restriction on N. As a consequence, the evolutions are different as illustrated by the following example. In λ_{lin}, $(\lambda x(x)\ x)\ (\alpha.y+\beta.z) \to^*_\ell \alpha.(y)\ y+\beta.(z)\ z$ while in λ_{alg}, $(\lambda x(x)\ x)\ (\alpha.y+\beta.z) \to_a (\alpha.y+\beta.z)\ (\alpha.y+\beta.z) =_a \alpha^2.(y)\ y+(\alpha\beta).(y)\ z+(\beta\alpha).(z)\ y+\beta^2.(z)\ z$.

Simulations. These two languages behave in different manner. An essential question is whether they would nonetheless be equivalent (and in which manner). Indeed, a positive answer would link two distinct research areas and unify works done in linear logic and works on quantum computation. It has been conjectured [2] that λ_{lin} simulates λ_{alg}. Our contribution is to prove it formally (Section 3.1) and to provide also the other way around proof of λ_{alg} simulating λ_{lin} (Section 3.2). The first simulation

*A full version of this paper with all the proofs is available in the arXiv

HOR 2010
EPTCS ??, 20??, pp. 1–6, doi:10.4204/EPTCS.??.??

uses the encoding, known as "thunk" in the folklore [6], which is based on "freezing" the evaluation of arguments by systematically encapsulating them into abstractions (that is, making them into values). It has been extensively studied in the case of the regular, untyped lambda-calculus [5]. The other way around is based on an algebraic extension of continuation passing style encoding [8].

Modifications to the original calculi. In this paper we slightly modify the two languages. The unique modification to λ_{alg} consists in avoiding reduction under λ, so that for any M, $\lambda x M$ is a value. As a consequence, λ is not linear: $\lambda x(\alpha.M+\beta.N) \neq \alpha.\lambda xN+\beta.\lambda xN$. In λ_{lin}, we restrict the application of several rewriting rules in order to make the rules more coherent with a call-by-value leftmost-redex evaluation. For instance, the rule $(M+N)\,L \to_\ell (M)\,L+(N)\,L$ is restricted to the case where both $M+N$ and L are values.

Finally, several distinct techniques can be used to make an algebraic calculus confluent. In λ_{lin}, restrictions on reduction rules are introduced, e.g. $\alpha.M+\beta.M \to_\ell (\alpha+\beta).M$ if M is closed normal. In λ_{alg} a restriction to positive scalars is proposed. Finally, one can use a typing system to guarantee confluence. In this paper, we assume that one of these techniques – without specifying explicitly which one – is used to make the calculi confluent.

2 Algebraic λ-calculi

The languages λ_{lin} and λ_{alg} share the same syntax, defined as follows:

$$\begin{array}{rcll} M,N,L & ::= & V \mid (M)\,N \mid M+N \mid \alpha.M & \text{(terms)},\\ U,V,W & ::= & 0 \mid B \mid \alpha.B \mid V+W & \text{(values)},\\ B & ::= & x \mid \lambda xM & \text{(basis terms)}. \end{array}$$

where α represents scalars which may themselves be defined by a term grammar, and endowed with a term rewrite system compatible with their basic ring operations $(+,\times)$. Formally it is captured in the definition [1, sec. III – def. 1] of a scalar rewrite system, but for our purpose it is sufficient to think of them as a ring.

The main differences between the two languages are the β-reduction and the algebraic linearity of function arguments. If U,V and W are values, and B is a basis term, the rules are defined by:

$$\begin{array}{rcll} (\lambda xM)\,N & \to_a & M[x:=N] & \beta_{\lambda_{alg}},\\ (\lambda xM)\,B & \to_\ell & M[x:=B] & \beta_{\lambda_{lin}},\\ (U)\,(V+W) & \to_\ell & (U)\,V+(U)\,W & \gamma_{\lambda_{lin}},\\ (V)\,(\alpha.W) & \to_\ell & \alpha.(V)\,W & \gamma_{\lambda_{lin}},\\ (V)\,0 & \to_\ell & 0 & \gamma_{\lambda_{lin}}. \end{array}$$

In both languages, $+$ is associative and commutative, i.e. $(M+N)+L = M+(N+L)$ and $M+N = N+M$. Notwithstanding their different axiomatizations – one based on equations and the other one on rewriting rules – linear combination of terms is treated in the same way: the set of terms behaves as a module over the ring of scalar in both languages.

In λ_{alg} the following algebraic equality is defined[1]:

[1] The reader should not be surprised by noticing that two terms that are equal under $=_a$ may reduce to terms that are not equal any more. Indeed, it is already the case with the syntactical equality of the λ-calculus.

$$\begin{array}{rcll}(M+N)\,L & =_a & (M)\,L+(N)\,L & (\lambda_{alg})\\ (\alpha.M)\,N & =_a & \alpha.(M)\,N & (\lambda_{alg})\\ 0+M & =_a & M & (\lambda_{alg})\\ \alpha.(M+N) & =_a & \alpha.M+\alpha.N & (\lambda_{alg})\\ \alpha.M+\beta.M & =_a & (\alpha+\beta).M & (\lambda_{alg})\end{array}\qquad\begin{array}{rcll}\alpha.(\beta.M) & =_a & (\alpha\times\beta).M & (\lambda_{alg})\\ (0)\,M & =_a & 0 & (\lambda_{alg})\\ 1.M & =_a & M & (\lambda_{alg})\\ 0.M & =_a & 0 & (\lambda_{alg})\\ \alpha.0 & =_a & 0 & (\lambda_{alg})\end{array}$$

In the opposite, the ring structure and the linearity of functions in λ_{lin} are provided by reduction rules. Let U,V and W stand for values[2], the rules are defined as follows.

$$\begin{array}{rcll}(U+V)\,W & \to_\ell & (U)\,V+(V)\,W & (\lambda_{lin})\\ (\alpha.V)\,W & \to_\ell & \alpha.(V)\,W & (\lambda_{lin})\\ \alpha.(M+N) & \to_\ell & \alpha.M+\alpha.N & (\lambda_{lin})\\ \alpha.M+\beta.M & \to_\ell & (\alpha+\beta).M & (\lambda_{lin})\\ \alpha.M+M & \to_\ell & (\alpha+1).M & (\lambda_{lin})\\ M+M & \to_\ell & (1+1).M & (\lambda_{lin})\end{array}\qquad\begin{array}{rcll}0+M & \to_\ell & M & (\lambda_{lin})\\ \alpha.(\beta.M) & \to_\ell & (\alpha\times\beta).M & (\lambda_{lin})\\ (0)\,V & \to_\ell & 0 & (\lambda_{lin})\\ 1.M & \to_\ell & M & (\lambda_{lin})\\ 0.M & \to_\ell & 0 & (\lambda_{lin})\\ \alpha.0 & \to_\ell & 0 & (\lambda_{lin})\end{array}$$

The context rules for both languages are

$$\frac{M\to M'}{(M)\,N\to(M')\,N}\qquad\frac{M\to M'}{M+N\to M'+N}\qquad\frac{N\to N'}{M+N\to M+N'}\qquad\frac{M\to M'}{\alpha.M\to\alpha.M'}$$

together with the additional context rule only for λ_{lin}

$$\frac{M\to_l M'}{(V)\,M\to_l(V)\,M'}.$$

The β-reduction of λ_{alg} corresponds to a call-by-name evaluation, while the β-reduction of λ_{lin} occurs only if the argument is a basis term, i.e. a variable or an abstraction. The γ-rules, only available in λ_{lin}, allows linearity in the arguments.

3 Simulations

3.1 λ_{lin} simulates λ_{alg}

We consider the following encoding $(\!|\cdot|\!) : \lambda_{alg}\to\lambda_{lin}$. The variables f and z are chosen fresh.

$$\begin{array}{rcl}(\!|0|\!) & = & 0,\\ (\!|x|\!) & = & (x)\,f,\\ (\!|\lambda x M|\!) & = & \lambda x\,(\!|M|\!),\end{array}\qquad\begin{array}{rcl}(\!|(M)\,N|\!) & = & ((\!|M|\!))\,\lambda z\,(\!|N|\!),\\ (\!|M+N|\!) & = & (\!|M|\!)+(\!|N|\!),\\ (\!|\alpha.M|\!) & = & \alpha.(\!|M|\!).\end{array}$$

One could be tempted to prove a result in the line of $M\to_a N$ implies $(\!|M|\!)\to_\ell^*(\!|N|\!)$. Unfortunately this does not work. Indeed, the encoding brings "administrative" redexes, as in the following example (where $I=\lambda xx$). Although $(\lambda x\lambda y(y)\,x)\,I\to_a\lambda y(y)\,I$,

$$\begin{array}{l}(\!|(\lambda x\lambda y(y)\,x)\,I|\!)=(\lambda x\lambda y((y)\,f)\,(\lambda z(x)\,f))\,(\lambda z\lambda x(x)\,f)\to_\ell^*\lambda y((y)\,f)\,(\lambda z(\lambda z\lambda x(x)\,f)\,f),\\ (\!|\lambda y(y)\,I|\!)=\lambda y((y)\,f)\,(\lambda z\lambda x(x)\,f)\end{array}$$

are not equal: there is an "administrative" redex hidden in the first expression. This redex does not bring any information, it is only brought by the encoding.

In order to clear these redexes, we define the map *Admin* as follows.

[2]Notice that in λ_{lin} a value is not necessarily in normal form. For instance the value $\lambda xx+\lambda xx$ reduces to $2.\lambda xx$. The reductions of values result solely from the ring structure, and all values are normalizing terms.

$$\begin{array}{rcl@{\qquad}rcl}
Admin\,0 & = & 0, & Admin\,\lambda x M & = & \lambda x Admin\,M,\\
Admin\,x & = & x, & Admin\,M+N & = & Admin\,M + Admin\,N,\\
Admin\,(\lambda f M)\ f & = & Admin\,M, & Admin\,\alpha.M & = & \alpha.Admin\,M.\\
Admin\,(M)\ N & = & (Admin\,M)\ Admin\,N, & & &
\end{array}$$

Theorem 3.1 *For any program (i.e. closed term) M, if $M \to_a N$ and $(\!|N|\!) \to_\ell^* V$ for a value V, then there exists M' such that $(\!|M|\!) \to_\ell^* M'$ and $Admin\,M' = Admin\,V$.*

Proof Proof by induction on the derivation of $M \to_a N$. □

Lemma 3.2 *If W is a value and M a term such that $Admin\,W = Admin\,M$, then there exists a value V such that $M \to_\ell^* V$ and $Admin\,W = Admin\,V$.*

Lemma 3.3 *If V is a closed value, then $(\!|V|\!)$ is a value.*

Theorem 3.4 (Simulation) *For any program (i.e. closed term) M, if $M \to_a^* V$ a value, then there exists a value W such that $(\!|M|\!) \to_\ell^* W$ and $Admin\,W = Admin\,(\!|V|\!)$.*

Proof The proof is done by induction on the size of the sequence of reductions $M \to_a^* V$. If $M = V$, this is trivially true by choosing $W = (\!|V|\!)$, which is a value since V is closed, by Lemma 3.3. Now, suppose the result true for the reduction $N \to_a^* V$ and suppose that $M \to_a N$. By induction hypothesis, $(\!|N|\!) \to_\ell^* W$, for some value W such that $Admin\,W = Admin\,(\!|V|\!)$. From Theorem 3.1, there exists M' such that $(\!|M|\!) \to_\ell^* M'$ and $Admin\,M' = Admin\,W$. From Lemma 3.2, without loss of generality we can choose this M' to be a value W'. This closes the proof of the theorem: we have indeed the equality $Admin\,W' = Admin\,(\!|V|\!)$. □

3.2 λ_{alg} simulates λ_{lin}

To prove the simulation of λ_{lin} with λ_{alg} we use the following encoding. This is an algebraic extension of the continuation passing style used to prove that call-by-name simulates call-by-value in the regular λ-calculus [8].

Let $[\![\cdot]\!] : \lambda_{lin} \to \lambda_{alg}$ be the following encoding. The variables f, g and h are chosen fresh.

$$\begin{array}{rcl@{\qquad}rcl}
[\![x]\!] & = & \lambda f\,(f)\ x, & [\![(M)\ N]\!] & = & \lambda f\,([\![M]\!])\ \lambda g\,([\![N]\!])\ \lambda h\,((g)\ h)\ f,\\
[\![0]\!] & = & 0, & [\![\alpha.M]\!] & = & \lambda f\,(\alpha.[\![M]\!])\ f,\\
[\![\lambda x M]\!] & = & \lambda f\,(f)\ \lambda x [\![M]\!], & [\![M+N]\!] & = & \lambda f\,([\![M]\!] + [\![N]\!])\ f.
\end{array}$$

Let Ψ be the encoding for values defined by:

$$\begin{array}{rcl@{\qquad}rcl}
\Psi(x) & = & x, & \Psi(\alpha.V) & = & \alpha.\Psi(V),\\
\Psi(0) & = & 0, & \Psi(V+W) & = & \Psi(V)+\Psi(W).\\
\Psi(\lambda x M) & = & \lambda x [\![M]\!], & & &
\end{array}$$

Using this encoding, it is possible to prove that λ_{alg} simulates λ_{lin} for any program reducing to a value:

Theorem 3.5 (Simulation) *For any program M, if $M \to_\ell^* V$ where V is a value, then*

$$[\![M]\!]\ (\lambda x x) \to_a^* \Psi(V).$$

Thanks to the subtle modifications done to the original algebraic calculi (presented in the introduction), the proof in [8] can easily be extended to the algebraic case. We first define a convenient infix operation (:) that captures the behaviour of the translated terms. For example, if B is a base term, *i.e.* a variable or an abstraction, then its translation into λ_{alg} is $[\![B]\!] \mapsto \lambda f\,(f)\ \Psi(B)$. If we apply this translated term to a certain K, we obtain $\lambda f\,(f)\ \Psi(B)\ K \to_a K\,\Psi(B)$. We capture this by defining $B : K = K\,\Psi(B)$. In general, $M : K$ is the reduction of the λ_{alg} term $[\![M]\!]\ K$, as Lemma 3.7 states.

Definition 3.6 *Let* $(:) : \Lambda_{\lambda_{lin}} \times \Lambda_{\lambda_{alg}} \to \Lambda_{\lambda_{alg}}$ *be the infix binary operation defined as follows:*

$$
\begin{array}{ll}
B : K = (K)\ \Psi(B) & \text{(with } B \text{ a base term),} \\
(M)\ N : K = M : \lambda g\,(\llbracket N \rrbracket)\ \lambda h\,((g)\ h)\ K & \text{(with } M \text{ not a value),} \\
(M)\ N : K = N : \lambda f\,((\Psi(M))\ f)\ K & \text{(with } M, \text{ but not } N, \text{ being a value),} \\
(M)\ N : K = ((\Psi(M))\ \Psi(N))\ K & \text{(with } M \text{ a value, and } N \text{ a base term),} \\
(M)\ (N_1 + N_2) : K = ((M)\ N_1 + (M)\ N_2) : K & \text{(with } M \text{ and } N_1 + N_1 \text{ values),} \\
(M)\ (\alpha.N) : K = \alpha.(M)\ N : K & \text{(with } M \text{ and } \alpha.N \text{ values),} \\
(M)\ 0 : K = 0 & \text{(with } M \text{ a value),} \\
(M + N) : K = M : K + N : K, & \\
\alpha.M : K = \alpha.(M : K), & \\
0 : K = 0. &
\end{array}
$$

Lemma 3.7 *If* K *is a value, then* $\forall M,\ \llbracket M \rrbracket\, K \to_a^* M : K$.

Lemma 3.8 *If* $M \to_\ell N$ *then* $\forall K$ *value,* $M : K \to_a^* N : K$

The proof of the Theorem 3.5 is now stated as follows.

Proof of Theorem 3.5. From Lemma 3.7, $\llbracket M \rrbracket\,(\lambda xx)$ reduces to $M : (\lambda xx)$. From Lemma 3.8, it reduces to $V : (\lambda xx)$. We now proceed by structural induction on V.

- Let V be a base term. Then $V : (\lambda xx) = (\lambda xx)\,\Psi(V) \to \Psi(V)$.
- Let $V = V_1 + V_2$. Then $V : (\lambda xx) = V_1 : (\lambda xx) + V_2 : (\lambda xx)$, which by the induction hypothesis, reduces to $\Psi(V_1) + \Psi(V_2) = \Psi(V)$.
- Let $V = \alpha.V'$. Then $V : (\lambda xx) = \alpha.(V' : (\lambda xx))$, which by the induction hypothesis, reduces to $\alpha.\Psi(V') = \Psi(V)$.

□

4 Conclusion and perspectives

In this paper we proved the conjectured [2] simulation of λ_{alg} by λ_{lin} and its inverse, on valid programs (that is, programs reducing to values), answering an open question about the equivalence of the algebraic λ-calculus (λ_{alg}) [9] and the linear-algebraic λ-calculus (λ_{lin}) [1].

As already shown by Plotkin [8], if the simulation of call-by-value by call-by-name is sound, it fails to be complete for general (possibly non-terminating) programs. To make it complete, a known solution is to consider the problem from the point of view of Moggi's computational calculus [7]. A direction for study is to consider an algebraic computational λ-calculus instead of a general algebraic λ-calculus. This raises the question of finding a correct notion of monad for capturing both algebraicity and non-termination in the context of higher-order structures. Another direction of study is the relation between the simulation of call-by-name by call-by-value using thunks and the CPS encoding. A first direction of study is [5]

Concerning semantics, the algebraic λ-calculus admits finiteness spaces as a model [3]. What is the structure of the model of the linear algebraic λ-calculus induced by the continuation-passing style translation in finiteness spaces? The algebraic lambda-calculus can be equipped with a differential operator. What is the corresponding operator in λ_{lin} through the translation?

References

[1] Pablo Arrighi & Gilles Dowek (2008): *Linear-algebraic lambda-calculus: higher-order, encodings, and confluence*. In: Andrei Voronkov, editor: *RTA 2008, Lecture Notes in Computer Science* 5117, Springer, Hagenberg, Austria, pp. 17–31.

[2] Pablo Arrighi & Lionel Vaux (2009): *Embeding AlgLam into Lineal*. Private communication.

[3] Thomas Ehrhard (2005): *Finiteness spaces*. *Mathematical Structures in Computer Science* 15(4), pp. 615–646.

[4] Thomas Ehrhard & Laurent Regnier (2003): *The differential lambda-calculus*. *Theoretical Computer Science* 309(1), pp. 1–41.

[5] John Hatcliff & Olivier Danvy (1997): *Thunks and the lambda-calculus*. *Journal of Functional Programming* 7(03), pp. 303–319.

[6] Peter Zilahy Ingerman (1961): *Thunks: a way of compiling procedure statements with some comments on procedure declarations*. *Communication of the ACM* 4(1), pp. 55–58.

[7] Eugenio Moggi (1989): *Computational Lambda-Calculus and Monads*. In: *LICS*, IEEE Computer Society, pp. 14–23.

[8] Gordon D. Plotkin (1975): *Call-by-name, call-by-value and the lambda-calculus*. *Theoretical Computer Science* 1(2), pp. 125–159.

[9] Lionel Vaux (2009): *The algebraic lambda calculus*. *Mathematical Structures in Computer Science* 19(5), pp. 1029–1059.