Semantics of a Typed Algebraic Lambda-Calculus

Benoît Valiron

Laboratoire d'Informatique de Grenoble
Université Joseph Fourier
Grenoble
France

benoit.valiron@monoidal.net

Algebraic lambda-calculi have been studied in various ways, but their semantics remain mostly untouched. In this paper we propose a semantic analysis of a general simply-typed lambda-calculus endowed with a structure of vector space. We sketch the relation with two established vectorial lambda-calculi. Then we study the problems arising from the addition of a fixed point combinator and how to modify the equational theory to solve them. We sketch an algebraic vectorial PCF and its possible denotational interpretations.

Keywords: typed lambda-calculus, module over ring and semi-ring, fixpoints, semantics, computational model.

1   Introduction

Notions of lambda-calculus with vectorial structures have at least three distinct origins. A first line of work [6, 5, 3], from which the term "algebraic lambda-calculus" comes from, focuses on general algebraic rewrite systems and studies the conditions needed for obtaining properties such as confluence or strong normalization. The second one is the calculus of Vaux [17], building up upon the work of Ehrhard and Regnier [7]. The goal here is to capture a notion of differentiation within lambda-calculus. Finally, algebraic lambda-calculus also arises in the work of Arrighi and Dowek [2] where they define a lambda-calculus oriented towards quantum computation, in the style of Van Tonder [16].

Both [2] and [17] are concerned with a lambda-calculus endowed with a structure of vector space. They both acknowledge the fact that for an untyped lambda-calculus, a naive rewrite system renders the language inconsistent, as any term can be made equal to the zero of the vectorial space of terms. However, coming from different backgrounds, they provide different solutions to the problem. In [2], the rewriting system is restrained in order to avoid unwanted equalities of terms. In [17], the rewriting system is untouched, but the scalars over which the vectorial structure is built are made into a semiring with particular properties, making the system consistent. Finally, [1] shows that a type system enforcing strong normalization is also a mean of solving the problem.

In this paper, we turn to the question of a semantics for a lambda-calculus endowed with a structure of vector space (or more generally, a structure of module). Starting with an untyped lambda-calculus and a naive rewrite system, we recall where inconsistencies occur. Then we construct a simply-typed version of the untyped language together with an equational description. In this restricted setting, the rewrite system is sound, and we describe a denotational semantics using a computational model a la Moggi [12]. We also show how one can relate this language to the one described in [2] and [17]. We then re-read the problems that occurred in the untyped world, and find a simple solution for making the system sound again in the presence of diverging terms, finding an agreement with the solution in [17]. The solution in this paper goes however a step further, proposing a denotational framework for the calculus.

1.1 An untyped calculus

Consider a ring $(\mathscr{A}, +, 0, \times, 1)$. Elements of $\mathscr{A}$ are called *scalars*. We define a call-by-value language as follows.

$$s,t \ ::= \ x \mid \lambda x.s \mid st \mid s+t \mid \alpha \cdot s \mid \mathbf{0} \mid [\,s\,] \mid \{\,s\,\},$$
$$u,v \ ::= \ x \mid \lambda x.u \mid uv \mid [\,s\,],$$

where α ranges over $\mathscr{A}$, and where x ranges over a fixed set of variables. Terms of the form s,t are called *computations* and terms of the form u,v are called *values*. We define variable substitution as usual and consider terms up to α-equivalence. The meanings of the unusual terms are explained in the next section.

1.2 A naive reduction system

A very naive reduction is to make the set of terms into a module over a ring $\mathscr{A}$, with the term $\mathbf{0}$ as unit of the addition. More precisely, a term s reduces to a term t, written $s \to t$, if there exist terms s' and t' respectively equivalent modulo congruence, associativity and commutativity of $+$ to s and t such that the relation $s' \to t'$ is derived from the rules of Table 1. Although we do not describe formally the system here (a complete development is done in Section 2.1), the reduction should be straightforward enough for the remainder of the discussion.

In particular, the addition is commutative and associative, the terms $t - t$ and $0 \cdot t$ equate the term $\mathbf{0}$. All term constructs are linear with respect to addition and scalar multiplication except $[\,\_\,]$, which "lifts" a computation into a value. One can unlift it using $\{\,\_\,\}$, and retrieve the computation. Finally, the system is call-by-value: the beta-reduction $(\lambda x.s)v$ reduces to $s[x \leftarrow v]$ only if v is a value.

<div align="center">

Group E

$$\alpha \cdot \mathbf{0} \ \to \ \mathbf{0} \qquad \mathbf{0} + s \ \to \ s \qquad \alpha \cdot (\beta \cdot s) \ \to \ (\alpha\beta) \cdot s$$
$$(*) \quad 0 \cdot s \ \to \ \mathbf{0} \qquad 1 \cdot s \ \to \ s \qquad \alpha \cdot (s+t) \ \to \ \alpha \cdot s + \alpha \cdot t$$

Group F

$$\alpha \cdot s \ + \ \beta \cdot s \ \to \ (\alpha + \beta) \cdot s$$
$$\alpha \cdot s \ + \ s \ \to \ (\alpha + 1) \cdot s$$
$$s \ + \ s \ \to \ (1+1) \cdot s$$

Group A

$$(s+t)r \to sr + tr \qquad (\alpha \cdot s)r \to \alpha \cdot (sr) \qquad \mathbf{0}r \to \mathbf{0}$$
$$r(s+t) \to rs + rt \qquad r(\alpha \cdot s) \to \alpha \cdot (rs) \qquad r\mathbf{0} \to \mathbf{0}$$
$$\lambda x.(s+t) \to \lambda x.s + \lambda x.t \qquad \lambda x.(\alpha \cdot s) \to \alpha \cdot \lambda x.s \qquad \lambda x.\mathbf{0} \to \mathbf{0}$$
$$\{\,s+t\,\} \to \{\,s\,\} + \{\,t\,\} \qquad \{\,\alpha \cdot s\,\} \to \alpha \cdot \{\,s\,\} \qquad \{\,\mathbf{0}\,\} \to \mathbf{0}$$

Group B

$$(\lambda x.s)v \to s[x \leftarrow v] \qquad \{\,[\,s\,]\,\} \to s$$

</div>

Table 1: Reduction system L.

For example, the term $(\lambda fx.(fx)x)(y+z)$ reduces to $\lambda f.(fy)y + \lambda f.(fz)z$. On the contrary, the computation $(\lambda xf.(f\{\,x\,\})\{\,x\,\})[\,y+z\,]$ reduces to the sum of terms $\lambda f.(fy)y + \lambda f.(fz)y + \lambda f.(fy)z + \lambda f.(fz)z$.

It is possible to build the same term constructs as with the regular untyped lambda-calculus [4]. For example, the product $\langle\,s,t\,\rangle$ of two terms s and t can be encoded as $\lambda f.(fs)t$, the first projection $\pi_1(s)$ of a pair s as the term $s(\lambda xy.x)$ and the second projection $\pi_2(s)$ as $s(\lambda xy.y)$. Note that, since all usual lambda-term constructs are linear with respect to addition and scalar multiplication in each variable, the new term constructs $\langle\,-,-\,\rangle$, π_1, π_2 are also linear in each variable. In particular, one can check that $\langle\,s+s',t+t'\,\rangle = \langle\,s,t\,\rangle + \langle\,s',t\,\rangle + \langle\,s,t'\,\rangle + \langle\,s',t'\,\rangle$. These term constructs are introduced in the simply-typed lambda-calculus of Section 2.

1.3 Breaking consistency

Although the set of requirements looks reasonable, as was shown in [2], the equational system is not sound. Indeed, given any term b one can construct the term $Y_b = \{\,(\lambda x.[\,\{\,xx\,\}+b\,])(\lambda x.[\,\{\,xx\,\}+b\,])\,\}$ verifying the reduction

$$Y_b \to Y_b + b. \tag{1}$$

This creates a problem of consistency, as enlightened in the following sequence of equalities:

$$\mathbf{0} = Y_b - Y_b = (Y_b + b) - Y_b = b + (Y_b - Y_b) = b. \tag{2}$$

This successfully shows that any term can be equated to $\mathbf{0}$, rendering the system inconsistent.

2 A simply-typed lambda-calculus

The problem occurring in Section 1.3 is due to the possibility of constructing diverging terms. In this section we study a simply-typed, algebraic lambda-calculus. Equipped with a naive reduction system, it verifies strong normalization. This allows us in Section 3 to analyze carefully the pitfalls occurring when adding divergence.

$$
\begin{array}{lll}
& \Delta, x:A \vdash x:A, & \\
& \Delta \vdash * : \top, & \left.\begin{array}{l}\Delta \vdash s:A \to B\\ \Delta \vdash t:A\end{array}\right\} \;\Rightarrow\; \Delta \vdash st:B,\\
& \Delta \vdash \mathbf{0}:A & \\
\Delta, x:A \vdash s:B \;\Rightarrow\; & \Delta \vdash \lambda x.s : A \to B, & \\
\Delta \vdash s:A \times B \;\Rightarrow\; & \Delta \vdash \pi_1(s):A, & \left.\begin{array}{l}\Delta \vdash s:A\\ \Delta \vdash t:B\end{array}\right\} \;\Rightarrow\; \Delta \vdash \langle\,s,t\,\rangle : A \times B,\\
\Delta \vdash s:A \times B \;\Rightarrow\; & \Delta \vdash \pi_2(s):B, & \\
\Delta \vdash s:A \;\Rightarrow\; & \Delta \vdash \alpha \cdot s:A, & \\
\Delta \vdash s:MA \;\Rightarrow\; & \Delta \vdash \{\,s\,\}:A, & \left.\begin{array}{l}\Delta \vdash s:A\\ \Delta \vdash t:A\end{array}\right\} \;\Rightarrow\; \Delta \vdash s+t:A,\\
\Delta \vdash s:A \;\Rightarrow\; & \Delta \vdash [\,s\,]:MA. &
\end{array}
$$

Table 2: Typing rules.

Definition 2.1. We suppose the existence of a ring $\mathscr{A}$, containing a multiplication and an addition. A simply-typed, call-by-value, algebraic lambda-calculus called the *computational algebraic lambda-calculus* is constructed as follows. Types are of the form

$$A,B \;::=\; \iota \mid A \to B \mid A \times B \mid \top \mid MA,$$

where ι ranges over a set of type constants. Terms again come in two flavors:

$$s, t \ ::= \ x \mid \lambda x.s \mid st \mid \langle\, s, t \,\rangle \mid \pi_1(s) \mid \pi_2(s) \mid * \mid s + t \mid \alpha \cdot s \mid \mathbf{0} \mid [\, s \,] \mid \{\, s \,\},$$
$$u, v \ ::= \ x \mid \lambda x.u \mid uv \mid \langle\, u, v \,\rangle \mid \pi_1(u) \mid \pi_2(u) \mid * \mid [\, s \,],$$

where $\alpha \in \mathscr{A}$. Terms of the form s, t are called *computations* and terms of the form u, v are called *values*. The term $[\, s \,]$ is the closure of a computation: such a term is not linear and can be duplicated "as it". The term construct $\{ - \}$ breaks such a closure and "runs" the computation.

We define the notions of typing context Δ and of typing derivation $\Delta \vdash s : A$ in the usual way [13]. Terms are considered up to α-equivalence, and valid typing derivations are built using the rules of Table 2.

2.1   Small-step semantics

The type system is valid with respect to the reduction system described in Table 1, modulo the addition of rules for the added term constructs concerning the product. In the following, we use the terminology of [2].

Definition 2.2. Given any relation R on terms, we say that it is a *call-by-value congruent relation* if for all pairs $(s, s'), (t, t') \in R$, the pairs (st, st'), $(st, s't)$, $(s + t, s + t')$, $(s + t, s' + t)$, $(\langle\, s, t \,\rangle, \langle\, s, t' \,\rangle)$, $(\langle\, s, t \,\rangle, \langle\, s', t \,\rangle)$, $(\pi_2 s, \pi_2 s')$, $(\pi_1 s, \pi_1 s')$, $(\alpha \cdot s, \alpha \cdot s')$ and $(\{\, s \,\}, \{\, s' \,\})$ are in R. We say that R is *congruent* if it is call-by-value congruent and if for all pairs $(s, s') \in R$, we also have $(\lambda x.s, \lambda x.s')$, $([\, s \,], [\, s' \,])$ in R.

Definition 2.3. We define $\simeq_{AC}$ to be the smallest congruent, equivalent relation on terms satisfying $s + t \simeq_{AC} t + s$ and $r + (s + t) \simeq_{AC} (r + s) + t$. We say that a relation R is *consistent with* $\simeq_{AC}$ if $s \simeq_{AC} s' R t' \simeq_{AC} t$ implies sRt.

Definition 2.4. A *normal* term s is such that there does not exist a term t with $s \rightarrow t$. A *rewrite sequence* is a sequence $(s_i)_i$ of terms such that for all i, either $s_i \rightarrow s_{i+1}$ or s_i is normal and i is the last index of the sequence.

Definition 2.5. We define the call-by-value reduction systems E, F, A and B of terms as the smallest call-by-value congruent relations consistent with $\simeq_{AC}$, satisfying the rules in Table 1 where B is augmented with the rules $\pi_1 \langle\, u, v \,\rangle \rightarrow u$ and $\pi_2 \langle\, u, v \,\rangle \rightarrow v$. In all the given rules, the terms u, v are assumed to be values. We write L for the relation $A \cup B \cup E \cup F$.

Convention 1. If R is a relation, we write $s \rightarrow_R t$ in place of $(s, t) \in R$. We simply write $\rightarrow$ in place of $\rightarrow_L$, and if $s \rightarrow t$, we say that s reduces to t. We denote with $\rightarrow_R^*$ the reflexive, transitive closure of $\rightarrow_R$.

Lemma 2.6 (Substitution). *Let $\Delta \vdash v : A$ and $\Delta, x : A \vdash s : B$ be two valid typing derivations, where v is a value. Then $\Delta \vdash s[x \leftarrow v] : B$ is a valid typing derivation.*

Proof. By structural induction on the typing derivation of $\Delta, x : A \vdash s : B$. $\qquad\square$

Lemma 2.7 (Subject reduction). *Let $\Delta \vdash s : A$ be a valid typing judgment such that $s \rightarrow t$. Then $\Delta \vdash t : A$ is also valid.*

Proof. Proof by structural induction on the term s and inspection of the reduction rules, using Lemma 2.6 for the first rule of group B. $\qquad\square$

Theorem 2.8 (Safety). *Suppose that* $\vdash s : A$ *is a valid typing judgment. Then either* $s \to t$ *with* $\vdash t : A$, *or s is normal.*

Proof. By case distinction on the structure of s, using Lemma 2.7. $\qquad\square$

As for the simply-typed lambda-calculus, the reduction system is normalizing. The proof uses the fact that the rewrite system consists of two parts: the rules of groups E,F,A and the rules of group B.

Lemma 2.9. *Let s be any term. There exists an index* n_s *such that any rewrite sequence* $(s_i)_i$ *in* $E \cup F \cup A$ *with* $s_0 = s$ *consists of at most* n_s *elements.*

Proof. We define two measures on terms. First, the "plus-number of s", written $np(s)$, and defined by $np(\mathbf{0}) = np(x) = np(*) = 1$, $np(\lambda x.s) = np(\pi_2(s)) = np(\pi_1(s)) = np(\{ s \}) = np(\alpha \cdot s) = 2np(s)$, $np(st) = np(\langle s,t \rangle) = 2np(s)np(t)$, and $np(s+t) = 1 + np(s) + np(t)$. Then, the "scalar-complexity of s", written $cx(s)$, and defined by $cx(\mathbf{0}) = cx(x) = cx(*) = 1$, $cx(\lambda x.s) = cx(\pi_2(s)) = cx(\pi_1(s)) = cx(\{ s \}) = 2cx(s)$, $cx(st) = cx(\langle s,t \rangle) = cx(s+t) = 2np(s)np(t)$, and $cx(\alpha \cdot s) = 1 + cx(s)$. The lemma is proved by induction on $(np(s), cx(s))$ with the lexicographic order. $\qquad\square$

Theorem 2.10 (Normalization). *Let* $\vdash s : A$ *be a valid typing judgment. There exists an index* n_s *such that any rewrite sequence* $(s_i)_i$ *with* $s_0 = s$ *is finite and of at most* n_s *elements.*

Proof. The proof uses reducibility candidates, and follows the proof provided in [9]. Lemma 2.9 is used to handle the cases where addition and scalar multiplication are involved. $\qquad\square$

Theorem 2.11 (Confluence). *Suppose that s is typable. If* $s \to^* t$ *and* $s \to^* t'$, *there exists a term r such that* $t \to^* r$ *and* $t' \to^* r$.

Proof. We first prove that for all terms s, if $s \to t$ and $s \to t'$ then there exists a term r such that $t \to^* r$ and $t' \to^* r$. We then prove the theorem using strong normalization, by induction on the length of the longest sequence of reductions. $\qquad\square$

2.1.1 Example: simulating quantum computation

As an example of the expressiveness of the language, we follow the motivation of [2] and show that we can simulate quantum computation using the computational algebraic lambda-calculus.

Quantum computation is a paradigm where data is encoded on the state of objects governed by the law of quantum physics. The mathematical description of a quantum boolean is a (normalized) vector in a 2-dimensional Hilbert space $\mathbb{H}$. In order to give sense to this vector, we choose an orthonormal basis $\{|0\rangle, |1\rangle\}$. A vector $\alpha|0\rangle + \beta|1\rangle$ is understood as the "quantum superposition" of the boolean 0 and the boolean 1.

For simulating quantum computation, we therefore choose the ring $\mathscr{A}$ to be the field of complex numbers. Given an arbitrary type X, we can represent a quantum boolean in the computational algebraic lambda-calculus as a closed value of type $qbool = MX \to (MX \to MX)$. We encode $\alpha|0\rangle + \beta|1\rangle$ as $\lambda xy.[\, \alpha \cdot \{ x \} + \beta \cdot \{ y \} \,]$. We write *tt* for $\lambda xy.[\, \{ x \} \,]$ and *ff* for $\lambda xy.[\, \{ y \} \,]$.

The operations we can perform on quantum booleans are of two sorts: Quantum gates and measurements. In the mathematical description, the former correspond to unitary maps. The Hadamard gate is such a unitary, sending $|0\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. It can be written as the term

$$H = \lambda x.\lambda ab.[\, \{x[\, \tfrac{1}{\sqrt{2}}\cdot(\{a\} + \{b\})\,][\, \tfrac{1}{\sqrt{2}}\cdot(\{a\} - \{b\})\,]\} \,]$$

of type $qbool \rightarrow qbool$. Applying the Hadamard gate to a quantum boolean b is computing the term Hb.

A measurement has a probabilistic outcome and does not have a satisfactory description as function of $\mathbb{H}$. It is customary to represent quantum booleans with *density matrices*, that is, positive matrices of norm one. The measurement operation becomes the map sending a matrix to its diagonal.

In order to model measurements, we can use the fact that the language features higher-order terms and we encode a positive matrix as a term of type $qbool \rightarrow qbool$. The quantum boolean $\alpha|0\rangle + \beta|1\rangle$ is encoded as the term v equal to

$$\lambda x.\lambda ab.[\ \{x[\ \alpha\bar{\alpha}\cdot\{a\} + \alpha\bar{\beta}\cdot\{b\}\ ][\ \bar{\alpha}\beta\cdot\{a\} + \beta\bar{\beta}\cdot\{b\}\ ]\}\ ].$$

The application of the Hadamard gate to v is $H'v$, where H' is the term $H' = \lambda x.H(xH)$ of type $(qbool \rightarrow qbool) \rightarrow (qbool \rightarrow qbool)$. The measurement is also of type $(qbool \rightarrow qbool) \rightarrow (qbool \rightarrow qbool)$ and can be encoded as the term P equal to $\lambda v.\lambda x.\lambda ab.[\ \{(vx)[\ \{a\}\ ][\ \mathbf{0}\ ] + (vx)[\ \mathbf{0}\ ][\ \{b\}\ ]\}\ ]$. We can check that Pv is indeed equal to $\lambda x.\lambda ab.[\ \{x[\ \alpha\bar{\alpha}\cdot\{a\}\ ][\ \beta\bar{\beta}\cdot\{b\}\ ]\}\ ]$.

2.2　Equational theory

$(*)$　$0\cdot s \simeq_{ax} \mathbf{0}$	$s+\mathbf{0} \simeq_{ax} s$
$1\cdot s \simeq_{ax} s$	$\alpha\cdot s + \alpha\cdot t \simeq_{ax} \alpha\cdot(s+t)$
$\alpha\cdot s + \beta\cdot s \simeq_{ax} (\alpha+\beta)\cdot s$	$(r+s)+t \simeq_{ax} r+(s+t)$
$\alpha\cdot(\beta\cdot s) \simeq_{ax} (\alpha\beta)\cdot s$	$s+t \simeq_{ax} t+s$

$\langle\, r+\alpha\cdot s, t\,\rangle \simeq_{ax} \langle\, r,t\,\rangle + \alpha\cdot\langle\, s,t\,\rangle$	$\pi_1(s+\alpha\cdot t) \simeq_{ax} \pi_1(s) + \alpha\cdot\pi_1(t)$
$\langle\, r, s+\alpha\cdot t\,\rangle \simeq_{ax} \langle\, r,s\,\rangle + \alpha\cdot\langle\, r,t\,\rangle$	$\pi_2(s+\alpha\cdot t) \simeq_{ax} \pi_2(s) + \alpha\cdot\pi_2(t)$
$\langle\, \mathbf{0}, t\,\rangle \simeq_{ax} \mathbf{0}$	$\pi_1(\mathbf{0}) \simeq_{ax} \mathbf{0}$
$\langle\, t, \mathbf{0}\,\rangle \simeq_{ax} \mathbf{0}$	$\pi_2(\mathbf{0}) \simeq_{ax} \mathbf{0}$
$(r+\alpha\cdot s)t \simeq_{ax} rt + \alpha\cdot(st)$	$\mathbf{0}t \simeq_{ax} \mathbf{0}$
$r(s+\alpha\cdot t) \simeq_{ax} rs + \alpha\cdot(rt)$	$t\mathbf{0} \simeq_{ax} \mathbf{0}$
$\lambda x.(s+\alpha\cdot t) \simeq_{ax} \lambda x.s + \alpha\cdot(\lambda x.t)$	$\lambda x.\mathbf{0} \simeq_{ax} \mathbf{0}$
$\{\, s+\alpha\cdot t\,\} \simeq_{ax} \{\, s\,\} + \alpha\cdot\{\, t\,\}$	$\{\,\mathbf{0}\,\} \simeq_{ax} \mathbf{0}$

$\pi_1\langle\, u, v\,\rangle \simeq_{ax} u$	$[\,\{\, u\,\}\,] \simeq_{ax} u$
$\pi_2\langle\, u, v\,\rangle \simeq_{ax} v$	$\{\,[\, s\,]\,\} \simeq_{ax} s$
$\langle\, \pi_1(u), \pi_2(u)\,\rangle \simeq_{ax} u$	$(\lambda x.\{\, s\,\})t \simeq_{ax} \{\,(\lambda x.s)t\,\}$
$(\lambda x.u)v \simeq_{ax} u[v/x]$	$((\lambda xy.r)s)t \simeq_{ax} ((\lambda yx.r)t)s$
$\lambda x.(ux) \simeq_{ax} u$	$(\lambda x.r)((\lambda y.s)t) \simeq_{ax} (\lambda y.(\lambda x.r)s)t$
$(\lambda x.x)s \simeq_{ax} s$	$u \simeq_{ax} *$

Table 3: Axiomatic equivalence relation.

Together with its type system, the computational algebraic lambda-calculus shares some strong similarities with Moggi's computational lambda-calculus [12] (although the notations used for the monad term constructs are closer to [8]). We follow the same path for defining a model for the algebraic lambda-calculus.

Definition 2.12. We define an equivalence relation $\simeq_{ax}$ on terms as the smallest congruent equivalence relation consistent with $\simeq_{AC}$, closed under α-equivalence and the equations of Table 3. The relation is the symmetric closure of the reduction L of Table 1, together with the rules taking into account the new term constructs.

Two valid typing judgments $\Delta \vdash s, t : A$ are said to be *axiomatically equivalent*, written $\Delta \vdash s \simeq_{ax} t : A$, if $s \simeq_{ax} t$ is provable.

Definition 2.13. We define a $\mathscr{A}$-enriched computational category to be a cartesian closed category $(\mathscr{C}, \times, \Rightarrow, \mathbf{1})$, together with a strong monad (M, η, μ, t), such that the Kleisli category is enriched over the category of $\mathscr{A}$-modules. We refer the reader to the literature for the definitions (e.g. [12, 10, 11]).

Example 2.14. The category of sets and functions together with the monad M sending a set X to the free module generated by X is a $\mathscr{A}$-enriched computational category.

Definition 2.15. We define the category $\mathscr{C}_l$ as follows: objects are types and morphisms $A \to B$ are axiomatic equivalent classes of typing judgments $x : A \vdash v : B$ (where v is a value).

Theorem 2.16. *The category $\mathscr{C}_l$ is a $\mathscr{A}$-enriched computational category. The cartesian closed structure is given by the classical subset of the language in the usual way (see e.g. [11]). The monad M sends A to MA and $x : A \vdash u : B$ to $y : MA \vdash [(\lambda x.u)\{y\}] : MB$, and the three required morphisms are $\eta_A = x : A \vdash [x] : MA$, $\mu_A = x : MMA \vdash [\{\{x\}\}] : MA$, $t_{A,B} = x : MA \times B \vdash [\langle\{\pi_1(x)\}, \pi_2(x)\rangle] : M(A \times B)$. The enrichment of $\mathscr{C}_l(A, MB)$ is given by the module structure of the term algebra. Consider the two maps $f = (x : A \vdash u : MB)$ and $g = (x : A \vdash v : MB)$. We define $0 = (x : A \vdash [\mathbf{0}] : MB)$, $f + g = (x : A \vdash [\{u\} + \{v\}] : MB)$, $\alpha \cdot f = (x : A \vdash [\alpha \cdot \{u\}] : MB)$.* $\square$

Definition 2.17. Consider a $\mathscr{A}$-enriched computational category $\mathscr{C}$. We define the interpretation of a computation $[\![\Delta \vdash t : B]\!]^c$ as a morphism in $\mathscr{C}_M$ and the interpretation of a value $[\![\Delta \vdash v : B]\!]^v$ as a morphism in $\mathscr{C}$. They are defined inductively, together with their obvious meanings.

Theorem 2.18. *If we interpret the computational algebraic lambda-calculus in $\mathscr{C}_l$ then the equations $[\![x : A \vdash v : B]\!]^v \simeq_{ax} (x : A \vdash v : B)$ and $[\![x : A \vdash t : B]\!]^c \simeq_{ax} (x : A \vdash [t] : MB)$ hold.* $\square$

2.3 Relation with other algebraic lambda-calculi

In this section, we relate the computational algebraic lambda-calculus we described in the previous section and the algebraic lambda-calculus λ_{alg} of Vaux [17] and lineal, the algebraic lambda-calculus λ_{lin} of Arrighi, Dowek and Dìaz-Caro [2, 1]. Both languages can be written using the term grammar $s, t ::= x \mid \lambda x.s \mid st \mid s + t \mid \mathbf{0} \mid \alpha \cdot s$. A possible simple type system is $A, B ::= \iota \mid A \to B$, where ι is a base type. The typing rules are the usual ones for the application and the lambda-abstraction. For the sum, the zero and the scalar multiplication, we use the typing rules found in Table 2.

The main difference between the two languages is the reduction system.

Vaux's lambda-calculus. In λ_{alg}, the lambda-abstraction is linear: $\lambda x.(s+t) \to \lambda x.s + \lambda x.t$, the application is linear on the left and non-linear on the right: $(r+s)t \to rt + st$ but $r(s+t) \not\to rs + rt$. However, $(\lambda x.s)t \to s[t/x]$ for any term t.

This language is call-by-name: a function is fed with a computation (that is, a term in superposition). One can encode λ_{alg} in the computational algebraic lambda-calculus as follows: $(\!|x|\!)_{alg} = \{\,x\,\}$, $(\!|\lambda x.s|\!)_{alg} = \lambda x.(\!|s|\!)_{alg}$, $(\!|st|\!)_{alg} = (\!|s|\!)_{alg}[\,(\!|t|\!)_{alg}\,]$. Types are encoded as follows: $(\!|\iota|\!)_{alg} = \iota$, $(\!|A \to B|\!)_{alg} = M(\!|A|\!)_{alg} \to (\!|B|\!)_{alg}$.

If $x : A \vdash s : B$ is a valid typing judgment in λ_{alg}, $x : MA \vdash (\!|s|\!)_{alg} : (\!|B|\!)_{alg}$ is valid in the computational algebraic lambda-calculus. In particular, if $\mathscr{C}$ is a $\mathscr{A}$-enriched computational model, s described a map $M[\![A]\!] \to M[\![B]\!]$ in the category $\mathscr{C}$.

Lineal. In λ_{lin}, the lambda-abstraction is non-linear: $\lambda x.(s+t) \not\to \lambda x.s + \lambda x.t$. In this calculus, the application is bilinear. In particular, $(\lambda x.s)u \to s[u/x]$ only if u is a value.

This calculus is call-by-value: the argument of a function is first reduced to a value before being substituted in the body of the function. One can encode λ_{lin} in the computational algebraic lambda-calculus as follows: $(\!|x|\!)_{lin} = x$, $(\!|\lambda x.s|\!)_{lin} = \lambda x.[\,(\!|s|\!)_{lin}\,]$, $(\!|st|\!)_{lin} = \{\,(\!|s|\!)_{lin}(\!|t|\!)_{lin}\,\}$. Types are encoded as follows: $(\!|\iota|\!)_{lin} = \iota$, $(\!|A \to B|\!)_{lin} = (\!|A|\!)_{lin} \to M(\!|B|\!)_{lin}$.

If $x : A \vdash s : B$ is a valid typing judgment in λ_{lin}, $x : A \vdash (\!|s|\!)_{lin} : (\!|B|\!)_{lin}$ is valid in the computational algebraic lambda-calculus. In particular, if $\mathscr{C}$ is a $\mathscr{A}$-enriched computational model, s describes a morphism $[\![A]\!] \to M[\![B]\!]$ of $\mathscr{C}$.

3   Adding controlled divergence

Because of Theorem 2.10, the term Y_b of Equation (1) is not constructable in the computational algebraic lambda-calculus. In this section, we add to the language a notion of fixpoint in order to understand what goes wrong in the untyped system.

3.1   A fixpoint operator

In order to stay typed and to be able to keep most of the computational interpretation of Section 2.2 but still to be able to have a term Y_b, we add to the language a unary term operator Y satisfying the reduction $Y(v) \to \{\,v[\,Y(v)\,]\,\}$, linear with respect to the module structure and satisfying the typing rule

$$\Delta \vdash s : MA \to MA \quad \Longrightarrow \quad \Delta \vdash Y(s) : A. \tag{3}$$

We can now build a term Y_b behaving as required in Equation (1):

$$Y_b \equiv Y(\lambda x.[\,b + \{\,x\,\}\,]). \tag{4}$$

Indeed, $Y(\lambda x.[\,b + \{\,x\,\}\,])$ reduces to the term $\{\,(\lambda x.[\,b + \{\,x\,\}\,])[\,Y(\lambda x.[\,b + \{\,x\,\}\,])\,]\,\}$, which reduces to $\{\,[\,b + \{\,[\,Y(\lambda x.[\,b + \{\,x\,\}\,])\,]\,\}\,]\,\}$, itself reducing to $b + Y(\lambda x.[\,b + \{\,x\,\}\,])$. Provided that $\Delta \vdash b : B$, the typing judgment $\Delta \vdash Y_b : B$ is valid. Of course, if we keep the operational semantics of Section 2, the system becomes as inconsistent as with the untyped calculus.

3.2   The zero in the algebra of terms

To understand what goes wrong, consider the typing judgment $x : MA \vdash x - x : MA$. With the equational system of Section 2.2, this typing judgment is equivalent to $x : MA \vdash \mathbf{0} : MA$. We claim that this interpretation is correct as long as the term x "does not contain any potential infinity". With the additional

construct Y, we can replace x with $[\,Y_a\,]$ (where Y_a is constructed as in Equation (4)) for some term a of type A. Consider the two terms

$$(\lambda y.*)((\lambda x.(x-x))[\,Y_a\,]), \quad (5) \qquad (\lambda y.\{\,y\,\})((\lambda x.(x-x))[\,Y_a\,]). \quad (6)$$

Term (5) reduces to $(\lambda y.*)(0 \cdot [\,Y_a\,])$ and then to $0 \cdot *$. It is reasonable to think that this is equivalent to $\mathbf{0}$, thus making $0 \cdot [\,Y_a\,]$ also equivalent to $\mathbf{0}$. Term (6), on the contrary, reduces to $Y_a - Y_a$, the flawed term of Equation (2).

The problem does not show up when writing the equation $[\,Y_a\,] - [\,Y_a\,] = 0 \cdot [\,Y_a\,]$ but when one equates it with $\mathbf{0}$. The term $0 \cdot [\,Y_a\,]$ is a "weak zero". It makes a computation "null" as long as it does not diverge (and there is always a diverging term of any inhabited type by using the construction (4)). Therefore, despite the fact that $\mathscr{A}$ is a ring, the set of terms of the form $\alpha \cdot s$ for a fixed term s is only a commutative monoid: addition does not admit an inverse, it only has an identity element $0 \cdot s$. This is consistent with previous studies [17, 15].

3.3 Recasting the equational theory

With the addition of fixpoints, the equational theory given in Section 2.2 is not valid. In the discussion of the previous section, we noted that the module of terms needs to be weakened to a commutative monoid by removing the rule $0 \cdot s \simeq_{ax} \mathbf{0}$. This is the only required modification, and one can rewrite the whole theory without this rule.

In the following, we do not consider the language extended with the fixpoint combinator; instead, we give a general theory for possible divergence in the context of a simple type system.

Definition 3.1. A *weak $\mathscr{A}$-module* is a module over $\mathscr{A}$ where $\mathscr{A}$ is seen as a semiring. In particular, a weak $\mathscr{A}$-module is only a commutative monoid, and $v - v = 0 \cdot v \neq 0$. Given a set X, the *free weak $\mathscr{A}$-module over X* is the structure consisting of all the finite sums $\sum_i \alpha_i \cdot x_i$, where $\alpha_i \in \mathscr{A}$ and $x_i \in X$.

Definition 3.2. A *weak $\mathscr{A}$-enriched computational category* consists of a cartesian closed category $(\mathscr{C}, \times, \Rightarrow, \mathbf{1})$, together with a strong monad (M, η, μ, t), such that the Kleisli category $\mathscr{C}_M$ is enriched over the category of weak $\mathscr{A}$-modules.

Remark 3.3. As we saw in Section 3.2, the two zero-functions $x : A \vdash \mathbf{0} : A$ and $x : A \vdash 0 \cdot x : A$ behave differently in general. In a weak $\mathscr{A}$-enriched computational category, the former is interpreted as the unit element of the monoid $\mathscr{C}_M(A, A)$ whereas the latter is of the form $0 \cdot id_A$, where id_A is the identity map in $\mathscr{C}_M(A, B)$.

Lemma 3.4. *Any $\mathscr{A}$-enriched computational category is also a weak $\mathscr{A}$-enriched computational category.*

Proof. Any $\mathscr{A}$-module is also a weak $\mathscr{A}$-module. $\square$

Remark 3.5. In particular, in a $\mathscr{A}$-enriched computational category, the two zero-functions $x : A \vdash \mathbf{0} : B$ and $x : A \vdash 0 \cdot x : B$ are identified.

Definition 3.6. Consider the typed language of Definition 2.1, with the axiomatic equivalence of Table 3 minus the very first rule, marked as $(*)$, stating $0 \cdot u \simeq_{ax} \mathbf{0}$. Let us call this language the *weak algebraic computational lambda-calculus* and the corresponding category of values $\mathscr{C}_l^w$.

Theorem 3.7. *1) The weak computational algebraic lambda-calculus is confluent. 2) $\mathscr{C}_l^w$ is a $\mathscr{A}$-enriched computational category. 3) The weak computational algebraic lambda-calculus is an internal language for weak $\mathscr{A}$-enriched computational categories.* $\square$

3.3.1   Extension of the language.

Here, we assume that the language is extended to a call-by-value PCF with a fixpoint combinator Y and an algebraic structure, as follows

$$A, B \; ::= \; bit \mid int \mid A \to B \mid A \times B \mid \top \mid MA,$$
$$r, s, t \; ::= \; x^A \mid \lambda x^A.s \mid st \mid \langle\, s, t\,\rangle \mid \pi_1(s) \mid \pi_2(s) \mid * \mid Y(s) \mid s + t \mid \alpha \cdot s \mid \mathbf{0} \mid$$
$$[\, s\,] \mid \{\, s\,\} \mid \mathit{tt} \mid \mathit{ff} \mid \mathit{if}\ r\ \mathit{then}\ s\ \mathit{else}\ t \mid \bar{0} \mid succ(s) \mid pred(s) \mid iszero(s),$$

where $\alpha \in \mathscr{A}$. The meaning of the terms is the usual one for PCF[14]. The terms tt and ff respectively stand for the boolean true and the boolean false; the term $\mathit{if}\ r\ \mathit{then}\ s\ \mathit{else}\ t$ is the test function on r; the term $\bar{0}$ stands for the natural number 0; the term $iszero(s)$ tests whether s is null or not; $pred$ and $succ$ are respectively the predecessor and the successor function; finally Y is the fixpoint combinator of Section 3.1. The notion of value is defined as in Definition 1.1.

The rewrite system of Section 2.1 can be reformulated for the algebraic PCF. Again, apart from the rule $(*)$ of Table 1 which is not valid, all the other ones are correct. The reduction systems E, F, A and B of terms as the smallest congruent relations consistent with $\simeq_{AC}$, satisfying the rules in Table 1 where B is augmented with the rules $Y(v) \to \{\, v[\, Y(v)\,]\,\}$, $succ(pred(u)) \to u$, $iszero(\bar{0}) \to \mathit{tt}$, $iszero(succ(u)) \to \mathit{ff}$, $\pi_1\langle\, u, v\,\rangle \to u$, $\pi_2\langle\, u, v\,\rangle \to v$, $\mathit{if}\ \mathit{tt}\ \mathit{then}\ s\ \mathit{else}\ t \to s$, $\mathit{if}\ \mathit{ff}\ \mathit{then}\ s\ \mathit{else}\ t \to t$, In all the given rules, the terms u, v are assumed to be values. We write L' for the relation $A \cup B \cup E \cup F$, and as before we write $\to$ in place of $\to_{L'}$.

Remark 3.8. Again, the rewrite system verifies subject reduction and progress. However, the system does not satisfy weak normalization. For example, the typing derivation $\vdash Y\lambda x.[\, \{\, x\,\}\,] : A$ is valid, and the term $Y\lambda x.[\, \{\, x\,\}\,]$ reduces to itself.

Example 3.9. An element of $M(int)$ can be regarded as the encoding of a polynomial as follows. The function

$$Exp = Y\lambda f.[\, \lambda nx.\mathit{if}\ iszero(n)\ \mathit{then}\ \{\, x\,\}\ \mathit{else}\ \{\, f\,\}(pred(n))x\,]$$

of type $int \to (M\top \to M\top)$ takes an integer n and returns the map sending $[\, \alpha \cdot *\,]$ to $[\, \alpha^n \cdot *\,]$. The map $Pow : M(int) \to (M\top \to M\top)$ defined as $\lambda x.Exp\, x$ takes as input $[\, \sum_i \beta_i \cdot \bar{n}_i\,]$ and return the map sending $[\, \alpha \cdot *\,]$ to $[\, (\sum_i \beta_i\, \alpha^{n_i}) \cdot *\,]$.

3.3.2   Concrete models based on Set

The category **Set** of sets and functions can be made into a weak $\mathscr{A}$-enriched computational category. It is also possible to model the PCF extension of the language: $[\![\top]\!] = \{*\}$, the one-element set, $[\![int]\!] = \mathbb{N}$, the set of natural numbers, and $[\![bit]\!] = \{0, 1\}$, the two-elements sets. The denotation of the product is the product in **Set** and the denotation of $A \to B$ is the set of **Set**-function between $[\![A]\!]$ and $[\![B]\!]$. The corresponding term constructs have their obvious meanings. Provided that the ring $\mathscr{A}$ is endowed with a suitable notion of limit (for example, taking $\mathscr{A}$ to be the reals with the usual topology), we give two monads that can be used and an intuition on their operational interpretation.

Strong convergence.   The monad M_s defined as $M_s(X) = \langle\, X\,\rangle_{\mathscr{A}} \cup \{\bot\}$, with $\langle\, X\,\rangle_{\mathscr{A}}$ is the free weak $\mathscr{A}$-module generated from X. We can define a fixpoint of $f : M_s(A) \to M_s(A)$ as $\lim_n f^n(\bot)$ if it exists, $\bot$ otherwise. We define $[\![Y(s)]\!]$ as the fixpoint of $[\![s]\!]$.

In this model, the morphism $[\![x : A \vdash \mathbf{0} : B]\!]$ is the constant function of value $0 \in \langle X \rangle_{\mathscr{A}}$ and the morphism $[\![x : A \vdash Y\lambda x.[\{x\}] : B]\!]$ is the constant function of value $\bot$. Moreover any non-converging well-typed term s have the same denotation $\bot$.

The set $M\mathbb{N}$ is $\bot$ together with all the finite linear combinations $\sum_i \alpha_i \cdot n_i$. The image of $M\mathbb{N}$ by the operator $[\![Pow]\!]$ of Example 3.9 is a set of functions $\overline{p} : \mathscr{A} \cup \{\bot\} \to \mathscr{A} \cup \{\bot\}$ sending $\bot$ to $\bot$ and $\beta \in \mathscr{A}$ to $p(\beta)$. The functions p are either constant of value $\bot$ (when f is the image of $\bot$) or polynomials (when f is the image of a linear combination).

Weak convergence.   Define the semiring $\mathscr{A} \cup \{\omega\}$ by extending the semiring $\mathscr{A}$ with a new element ω. The sum and the multiplication are extended as follows: $\alpha\omega = \omega$, $\alpha + \omega = \omega$. We set $M_w(X) = (\mathscr{A} \cup \{\omega\})^X$, the functions from X to $\mathscr{A} \cup \{\omega\}$. The fixpoint of $f : M_w(A) \to M_w(A)$ is defined as the map sending $x \in X$ to $\lim_n f^n(0)(x)$ if it exists, ω otherwise. As previously, the denotation of $Y(s)$ is the fixpoint of $[\![s]\!]$.

Here, $[\![x : A \vdash \mathbf{0} : B]\!]$ and $[\![x : A \vdash Y\lambda x.[\{x\}] : B]\!]$ are the constant functions of value $0 \in \langle X \rangle_{\mathscr{A}}$. However, all diverging terms do not have the same image. For example, the term $Y\lambda x.[\bar{0} + succ\{x\}]$ of type *int* corresponds to the element $f \in M_w(\mathbb{N})$ sending all $n \in \mathbb{N}$ to $1 \in \mathscr{A}$.

In this model, the image of $M(\mathbb{N})$ by *Pow* is the set of (generalized) entire functions $\mathscr{A} \to \mathscr{A}$, sending β to $\sum_i \alpha_i(\beta)^{n_i}$. By "generalized", we mean that the functions may send some β to ω.

4   Conclusion

In this paper, we sketched the required structures for a semantics for a typed algebraic lambda-calculus and discussed relation with previous works. We showed that the problems occurring with divergence can be solved by using a weak module. Finally, we described an algebraic PCF and its interpretation in two concrete **Set**-based models.

This raises the question of the complete description of the possible operational behaviors of the algebraic PCF and the study of their denotational semantics.

5   Acknowledgments

References

[1] Pablo Arrighi & Alejandro Díaz-Caro (2009). *A System F accounting for scalars*. Preprint: arXiv:0903.3741.

[2] Pablo Arrighi & Gilles Dowek (2008): *Linear-algebraic lambda-calculus: higher-order, encodings, and confluence.* In: *Proceedings of the 19th international conference on Rewriting Techniques and Applications (RTA'08)*, Lecture Notes in Computer Science 5117, pp. 17–31.

[3] Franco Barbanera & Maribel Fernández (1993): *Combining first and higher-order rewrite systems with type assignment systems.* In: *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, Lecture Notes in Computer Science 664, pp. 60–74.

[4] Henk P. Barendregt (1984): *The Lambda-Calculus, its Syntax and Semantics.* North Holland.

[5] Frédéric Blanqui, Jean-Pierre Jouannaud & Mitsuhiro Okada (1999): *The Calculus of algebraic Constructions*. In: *RtA '99: Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, Springer-Verlag, London, UK, pp. 301–316.

[6] Val Breazu-Tannen & Jean Gallier (1991): *Polymorphic rewriting conserves algebraic strong normalization*. Theoretical Computer Science 83(1), pp. 3–28.

[7] Thomas Ehrhard & Laurent Regnier (2003): *The differential lambda-calculus*. Theoretical Computer Science 309(1–2), pp. 1–41.

[8] Andrzej Filinski (1996): *Representing Monads*. In: *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 446–457.

[9] Jean-Yves Girard, Yves Lafont & Paul Taylor (1990): *Proofs and Types*. Cambridge University Press.

[10] Gregory M. Kelly (1982): *Basic Concepts of Enriched Category Theory*, *London Mathematical Society Lecture Notes Series* 64. Cambridge University Press. Avalaible in Reprint in Theory and Application of Categories, No 10, 1982.

[11] Joachim Lambek & Philip Scott (1989): *Introduction to Higher Order Categorical Logic*. Cambridge University Press.

[12] Eugenio Moggi (1991): *Notions of Computation and Monads*. Information and Computation 93, pp. 55–92.

[13] Benjamin C. Pierce (2002): *Types and Programming Languages*. MIT Press.

[14] Gordon D. Plotkin (1977): *LCF Considered as a Programming Language*. Theoretical Computer Science 5, pp. 223–255.

[15] Peter Selinger (2003): *Order-Incompleteness and Finite Lambda-Reduction Models*. Theoretical Computer Science 309, pp. 43–63.

[16] André van Tonder (2004): *A Lambda Calculus for Quantum Computation*. SIAM Journal of Computing 33, pp. 1109–1135.

[17] Lionel Vaux (2008): *Algebraic lambda-calculus*. Mathematical Structures in Computer Science To appear.