Algebras and Coalgebras in the Light Affine Lambda Calculus

Marco Gaboardi
University of Dundee, United Kingdom

Romain Péchoux
Université de Lorraine, France

Abstract

Algebra and coalgebra are widely used to model data types in functional programming languages and proof assistants. Their use permits to better structure the computations and also to enhance the expressivity of a language or of a proof system.

Interestingly, parametric polymorphism *à la* System F provides a way to encode algebras and coalgebras in strongly normalizing languages without losing the good logical properties of the calculus. Even if these encodings are sometimes unsatisfying because they provide only limited forms of algebras and coalgebras, they give insights on the expressivity of System F in terms of functions that we can program in it.

With the goal of contributing to a better understanding of the expressivity of Implicit Computational Complexity systems, we study the problem of defining algebras and coalgebras in the Light Affine Lambda Calculus, a system characterizing the complexity class FPTIME. This system limits the computational complexity of programs but it also limits the ways we can use parametric polymorphism, and in general the way we can write our programs.

We show here that while the restrictions imposed by the Light Affine Lambda Calculus pose some issues to the standard System F encodings, they still permit to encode some form of algebra and coalgebra. Using the algebra encoding one can define in the Light Affine Lambda Calculus the traditional inductive types. Unfortunately, the corresponding coalgebra encoding permits only a very limited form of coinductive data types. To extend this class we study an extension of the Light Affine Lambda Calculus by distributive laws for the modality §. This extension has been discussed but not studied before.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda Calculus and Related Systems.

Keywords implicit computational complexity; algebra and coalgebra; light logics

1. Introduction

Algebras and coalgebras Data types shape the style we can use to write our programs, contributing in this way to determining the *expressivity* of a programming language. Algebras and coalgebras of a functor (see [26] for an extended introduction) are important tools coming from category theory that are useful to specify data types in a uniform way. This uniformity has been exploited in the design of functional programming languages, via the use of abstract data types. In this particular setting, algebraic types correspond usually to finite data types and coalgebraic ones correspond to infinite data types.

Despite the fact that coalgebras correspond to infinite data types, interestingly algebras and coalgebras can be also added to languages that are strongly normalizing by preserving the strong normalization property, as shown by Hagino [23]. Moreover, algebras and coalgebras can also be encoded by using parametric polymorphism in strongly normalizing languages as System F as shown by Reynolds and Plotkin [37], Wraith [42]. Preserving strong normalization corresponds to preserving the consistency property of the language. It is this last feature that allows the integration of algebras and coalgebras in proof assistants such as Coq and Agda, where they can be used to define inductive and coinductive data types, respectively.

Different notions of algebras and coalgebras can provide different forms of recursion and corecursion that can be used to program algorithms in different ways. Moreover, algebras and coalgebras also provide some form of induction and coinduction that we can use to prove program properties. So, algebras and coalgebras are abstractions that are useful to compare in the abstract the expressivity of distinct languages.

Implicit Computational Complexity (ICC) ICC aims at characterizing complexity classes by means that are independent from the underlying machine model. A characterization of a complexity class C is traditionally determined by a system S obtained by restricting the class of proofs of a given logical system or the class of programs of a given programming language L. In order to characterize C, the system S needs to satisfy two properties: 1) the evaluation process of proofs or programs of S must lie within the given complexity class C—this ensures that S is *sound* with respect to C; 2) any function (or decision problem) in C must be implementable by a proof or program in S—this ensures that S is *complete* with respect to C.

This approach for characterizing complexity classes where all the functions or problems of the given class C can be encoded by some proof or program in S is traditionally referred to as *extensionally complete*. On the other hand, an *intensionally complete* characterization requires that all the proofs or programs that can be evaluated within the complexity class C must lie in the restriction S. From a programmer's perspective intensionally complete characterizations are certainly preferable to extensionally complete ones since they capture all the algorithms of the language L that lie in the class C. However, providing intensional characterizations of well-known and interesting complexity classes is in general problematic: for polynomial time the problem of providing an intensional characterization is Σ_2^0-complete in the arithmetical hierarchy—and so undecidable—as proved by Hájek [24].

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...$15.00
http://dx.doi.org/10.1145/2784731.2784759

This contrast between extensional and intensional completeness has motivated researchers in ICC in the search of restrictions to logical systems and programming languages that are more and more expressive in terms of proofs or programs in L that they can fit. This is usually achieved in two ways: by weakening the restrictions, and by enriching the language with new programming constructs. See the survey by Hofmann [25] for more information.

Light Logics The Light Logics [21, 27] approach to ICC is based on the idea of providing characterizations of complexity classes by means of subsystems of Girard's (second order) Linear Logic [20]. Proofs of second order linear logic can be seen through the proof-as-programs correspondence as terms of System F typed under a refined typing discipline using the contraction and weakening rules in a more principled way via the exponential modality !.

Following the light logic approach one can design type systems for the lambda calculus and its extensions where only programs that are in a particular computational complexity class can be assigned a type. This approach has been used to provide characterizations of several complexity classes like FPTIME [2, 5, 14, 40], PSPACE [16], LOGSPACE [39], NP [15, 32], P/Poly [33], etc.

A bird's eye view on Light Affine Logic One of the most successful examples of light logic is certainly Light Affine Logic (LAL), the affine version of Light Linear Logic (LLL). Similarly to LLL, LAL provides a characterization of the class FPTIME by limiting the way the modality ! is used in proofs and by introducing a new modality § to compensate for some of these limitations. Roughly, the modality ! is used as a marker for objects that can be iterated, the modality § is used as a marker of objects that are the result of an iteration and cannot be iterated anymore. The combined use of these two modalities provides a way to limit the iterations that one can write in proofs, and so the complexity of the system.

More precisely, LAL enforces a design principle named *stratification* by adopting the following rules for modalities[1]:

$$\frac{\Gamma \vdash \tau \quad \Gamma \subseteq \{\sigma\}}{!\Gamma \vdash !\tau}\ (!) \qquad \frac{\Gamma, !\tau, !\tau \vdash \sigma}{\Gamma, !\tau \vdash \sigma}\ (C) \qquad \frac{\Gamma, \Delta \vdash \tau}{!\Gamma, \S\Delta \vdash \S\tau}\ (\S)$$

The stratification is obtained by limiting the introduction of the two modalities to the two rules (!) and (§), respectively. These rules can be seen as boxes that stratify the proofs. In other words, stratification corresponds roughly to ruling out the logical principles $!A \multimap A$ and $!A \multimap !!A$ but allowing the principles $!A \multimap \S A$. Enforcing stratification is not sufficient to characterize polynomial time—it provides a characterization of Elementary time [21]. For this reason, LAL further restrict the power of the modality ! by requiring that the environment Γ in the rule (!) has at most one assumption, this is the meaning of the premise $\Gamma \subseteq \{\sigma\}$. This requirement corresponds roughly to ruling out the logical principles $!A \otimes !B \multimap !(A \otimes B)$ and only allowing instead the restricted principle $!A \otimes !B \multimap \S(A \otimes B)$.

Despite their rather technical definitions, LLL and LAL provide natural and quite expressive characterizations of the class FPTIME. For this reason, their principles have been used to design a lambda calculus [40], a type system [5] and an extended language [6] for polynomial time computations. In these languages one can program several natural polynomial time algorithms over different data structures.

Our contribution In this work we study the definability of algebras and coalgebras in the Linear Affine Lambda Calculus (LALC), a term language for LAL, with the aim of better understanding the expressivity of LALC with respect to the definability of inductive and coinductive data structures, in particular with a focus on infinite data structures like streams.

Since LALC can be seen as a subsystem of System F, we study how to adapt the encoding of algebras and coalgebras in System F to the case of LALC. Not surprisingly, the standard System F encoding from Wraith [42] cannot be straightforwardly adapted to LALC because of the stratification principle. Indeed variable duplication in the terms enforces the modalities ! and § to appear. The presence of these modalities enforces types encoding initial algebras and final coalgebras that differ from the ones of the standard encoding. The initial algebra for the functor F can be encoded in LALC by terms of type

$$\forall X.!(F(X) \multimap X) \multimap \S X$$

The final coalgebra for the same functor F can instead be encoded by terms of type

$$\exists X.!(X \multimap F(X)) \otimes \S X$$

Initial algebras and final coalgebras definable in System F are only *weak* [37, 42]. In the case of LALC the two types above provide an even more restricted class of initial algebras and final coalgebras: intuitively, the ones that *behave well under* § as a marker for iteration. These definitions will be made precise in Section 4 and Section 5, respectively. A further restriction comes from the fact that to obtain these classes of algebras and coalgebras we need to consider only functors that behave well with respect to the modality §. More precisely, for initial algebras we need functors that *left-distribute* over §, i.e. functors F such that $F(\S X) \multimap \S F(X)$. Conversely, for final algebras we need functors that *right-distribute* over §, i.e. functors F such that $\S F(X) \multimap F(\S X)$.

Functors that left-distribute over § are quite common in LALC and so we can define several standard inductive data types. Unfortunately, only few functors right-distribute over §. In particular, we cannot encode standard coinductive data structures. The main reason is that the modality § does not *distribute* with respect to the connectives tensor and plus. More precisely, in LALC we cannot derive the distributive[2] laws $\S(A \otimes B) \multimap \S A \otimes \S B$ and $\S(A \oplus B) \multimap \S A \oplus \S B$ for generic A and B. We overcome this situation by adding terms for these distributive laws to LALC. Thanks to this extension we are able to write programs working on infinite streams of booleans (or of any finite data type) and other infinite data types.

Quite interestingly, Girard [19, §16.5.3] remarked that adding the principle $\S(A \oplus B) \multimap \S A \oplus \S B$ ("supposedly doing what one thinks") to LAL would bring to the absurd situation where we can decide in linear time all the polynomial time problems. The informal argument is that this principle would allow us to extract the output bit of a decision problem without the need of computing it. A discussion on this argument has also been used by Baillot and Mazza [4] to explain one of the differences between LAL and their Linear Logic by Level. Here we show that adding the distributivity principle $\S(A \oplus B) \multimap \S A \oplus \S B$ with a computational counterpart in the term language does not bring to the absurd situation prospected by Girard. On the contrary, we show that the full evaluation of programs containing this distributivity principle requires a more complex reduction strategy than the dept-by-depth one traditionally used for LAL [21]. This is also reflected in the polynomial time soundness proof for LAL extended with distributions that we provide in Section 6. Let us stress that our argument is not necessarily in contradiction with Girard's argument because the latter relies on the informal condition "supposedly doing what one thinks" and one can think to introduce the distributivity principles as an identity

[1] We present here the rules of the logic in sequent calculus. The corresponding typing rules will then by presented in Section 3.

[2] We use here the term "distributive" because we think of both modalities and type constructors as operations. In the literature, other people have preferred the term "commutative".

$\S(A \oplus B) = \S A \oplus \S B$ without computational content. In this case, the absurd situation would indeed arise.

2. Algebras and Coalgebras in System F

The starting point of our work is the encoding of weak initial F-algebras and weak final F-coalgebras in System F as described by Wraith [42] and Freyd [11] (see also Wadler [41]). Let us start by reviewing the definition of F-algebras and F-coalgebras.

Definition 1 (F-Algebra and F-Coalgebra)**.** *Given a category $\mathcal{C}$ and an endofunctor $F : \mathcal{C} \to \mathcal{C}$:*

- *a* F-algebra *is a pair (A, a) of an object $A \in \mathcal{C}$ together with a $\mathcal{C}$-morphism $a : F(A) \to A$,*
- *a* F-coalgebra *is a pair (A, a) of an object $A \in \mathcal{C}$ together with a $\mathcal{C}$-morphism $a : A \to F(A)$.*

Algebras and coalgebras provide the basic syntactic structure that is needed in order to define data types.

We can define two categories Alg-F and Coalg-F whose objects are F-algebras and F-coalgebras, respectively, and whose morphisms are defined as follows.

Definition 2. *A F-algebra homomorphism from the F-algebra (A, a) to the F-algebra (B, b) is a morphism $f : A \to B$ making the following diagram commute:*

$$\begin{tikzcd}
F(A) \arrow[r, "F(f)"] \arrow[d, "a"'] & F(B) \arrow[d, "b"] \\
A \arrow[r, "f"'] & B
\end{tikzcd}$$

A F-coalgebra homomorphism from the F-coalgebra (A, a) to the F-coalgebra (B, b) is a morphism $f : A \to B$ making the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "a"'] & B \arrow[d, "b"] \\
F(A) \arrow[r, "F(f)"'] & F(B)
\end{tikzcd}$$

To define the traditional inductive and coinductive data types we also need the notions of *initial algebras* and *final coalgebras*.

Definition 3 (Initial algebra and final coalgebra)**.** *A F-algebra (A, a) is* initial *if for each F-algebra (B, b), there exists a unique F-algebra homomorphism $f : A \to B$. A F-coalgebra (A, a) is* final *if for each F-coalgebra (B, b), there exists a unique F-coalgebra homomorphism $f : B \to A$.*

If the uniqueness condition is not met then the F-algebra (resp. F-coalgebra) is only weakly initial *(resp.* weakly final*).*

An initial F-algebra is an initial object in the category Alg-F. Conversely, a final F-coalgebra is a terminal object in the category Coalg-F. In the definition above, the existence of a homomorphism provides a way to build objects by (co)iteration; this corresponds to have the ability to define by iteration elements in type fixpoints. Conversely, the uniqueness of such homomorphism provides a way to prove properties of these elements by (co)induction; this is something that type fixpoint does not necessarily provide.

Example 4. *Consider the functor F defined by $F(X) = 1 + X$. The pair $(\mathbb{N}, [0, suc])$ consisting in the set of natural numbers $\mathbb{N}$ together with the morphism $[0, suc] : 1 + \mathbb{N} \to \mathbb{N}$, defined as the coproduct of $0 : 1 \to \mathbb{N}$ and $suc : \mathbb{N} \to \mathbb{N}$, is an F-algebra.*

Consider the endofunctor F over the category Set defined by $F(X) = A \times X$, for some set A. The pair $(A^\omega, \langle head, tail \rangle)$ where A^ω is the set of infinite lists over A and the morphism $\langle head, tail \rangle : A^\omega \to A \times A^\omega$ is defined by $head : A^\omega \to A$ and $tail : A^\omega \to A^\omega$, is a final F-coalgebra.

Encoding Weak Initial Algebras and Weak Final Coalgebras We here assume some familiarity with System F and existential types (see [22] and [35]). A functor $F(X)$ is definable in System F if $F(X)$ is a type scheme mapping every type A to the type $F(A)$, and if there exists a term F mapping every term of type $A \to B$ to a term of type $F(A) \to F(B)$ and such that it preserves identity and composition. We say that a functor $F(X)$ is covariant if the variable X only appears in covariant positions.

It is a well known result that for any covariant functor $F(X)$ that is definable in System F we can define an algebra that is weakly initial and a coalgebra that is weakly final [26]. This corresponds to defining the least and the greatest fixpoint of $F(X)$ as a type scheme.

Proposition 5 (Weak Initial Algebra)**.** *Let $F(X)$ be a covariant functor definable in System F and $T = \forall X.(F(X) \to X) \to X$. Consider the morphisms defined by:*

$$\begin{aligned}
&\mathtt{in}_T : F(T) \to T,\\
&\mathtt{in}_T = \lambda \mathtt{s} : F(T).\Lambda X.\lambda \mathtt{k} : F(X) \to X.\mathtt{k}(F\ (\mathtt{fold}_T\ X\ \mathtt{k})\ \mathtt{s}),\\
&\mathtt{fold}_T : \forall X.(F(X) \to X) \to T \to X,\\
&\mathtt{fold}_T = \Lambda X.\lambda \mathtt{k} : F(X) \to X.\lambda \mathtt{t} : T.\mathtt{t}\ X\ \mathtt{k}.
\end{aligned}$$

Then, $(T, \mathtt{in}_T)$ is a weak initial F-algebra: for every F-algebra $(A, g : F(A) \to A)$ there is an F-homomorphism $h : T \to A$ defined as $h = \mathtt{fold}_T\, A\, g$.

We will sometimes write T as $\mu X.F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the least fixpoint of F.

Proposition 6 (Weak Final Coalgebra)**.** *Let F be a covariant functor definable in System F and $T = \exists X.(X \to F(X)) \times X$. Consider the morphisms defined by:*

$$\begin{aligned}
&\mathtt{out}_T : T \to F(T),\\
&\mathtt{out}_T = \lambda \mathtt{t} : T.\mathtt{unpack}\ \mathtt{t}\ \mathtt{as}\ (X, \mathtt{z})\ \mathtt{in}\\
&\qquad \mathtt{let}\ (\mathtt{k}, \mathtt{x}) = \mathtt{z}\ \mathtt{in}\ \mathtt{F}(\mathtt{unfold_T}\ \mathtt{X}\ \mathtt{k})(\mathtt{k}\ \mathtt{x}),\\
&\mathtt{unfold}_T : \forall X.(X \to F(X)) \to X \to T,\\
&\mathtt{unfold}_T = \Lambda X.\lambda \mathtt{k} : X \to F(X).\lambda \mathtt{x} : X.\mathtt{pack}\ ((\mathtt{k}, \mathtt{x}), X)\ \mathtt{as}\ T.
\end{aligned}$$

Then, $(T, \mathtt{out}_T)$ is a weak final F-coalgebra: for every F-coalgebra $(A, g : A \to F(A))$ there is a F-homomorphism $h : A \to T$ defined as $h = \mathtt{unfold}_T\, A\, g$.

Similarly to the case of F-algebras, we will write T as $\nu X.F(X)$ when we want to stress the underlying functor F and the fact that T corresponds to the greatest fixpoint of F.

Example 7. *Let us consider a functor defined on types as $F(X) = \mathbf{1} + X$ and on terms as:*

$$\begin{aligned}
&\lambda \mathtt{f} : X \to Y.\lambda \mathtt{x} : \mathbf{1} + X.\mathtt{case}\ \mathtt{x}\ \mathtt{of}\\
&\{\mathtt{inj}_0^{1+X}(\mathtt{z}) \to \mathtt{inj}_0^{1+Y}(()), \mathtt{inj}_1^{1+X}(\mathtt{z}) \to \mathtt{inj}_1^{1+Y}(\mathtt{f}\ \mathtt{z})\,\}.
\end{aligned}$$

Let $\mathbb{N} = \mu X.F(X)$. Proposition 5 ensures that $(\mathbb{N}, \mathtt{in}_{\mathbb{N}})$ is a weak initial algebra: the weak initial algebra of natural numbers. In particular, we can define $\underline{0} = \mathtt{in}_{\mathbb{N}}(\mathtt{inj}_0^{1+\mathbb{N}}(()))$, $\underline{n+1} =$

$\mathtt{in}_{\mathbb{N}}(\mathtt{inj}_1^{\mathbf{1}+\mathbb{N}}(\underline{n}))$*, and more in general the successor function as* $\mathtt{succ} = \lambda\mathtt{x}.\mathtt{in}_{\mathbb{N}}(\mathtt{inj}_1^{\mathbf{1}+\mathbb{N}}(\mathtt{x}))$*. We can use the fact that* $\mathbb{N}$ *is a weak initial algebra to define an addition function. We just need to consider a term like the following (we omit some type for conciseness):*

$$g = \lambda\mathtt{x} : \mathbf{1} + (\mathbb{N} \to \mathbb{N}).\mathtt{case}\ \mathtt{x}\ \mathtt{of}$$
$$\{\mathtt{inj}_0(\mathtt{z}) \to \lambda\mathtt{y} : \mathbb{N}.\mathtt{y}, \mathtt{inj}_1(\mathtt{z}) \to \lambda\mathtt{y} : \mathbb{N}.\mathtt{succ}(\mathtt{z}\,\mathtt{y})\ \}.$$

Then, Proposition 5 ensures that we can define $\mathtt{add}$ *as* $\mathtt{fold}_{\mathbb{N}}\ (\mathbb{N} \to \mathbb{N})\ g$*.*

Example 8. *Let us consider a functor defined on types as* $F(X) = \mathbb{N} \times X$ *and on terms as:*

$$\lambda\mathtt{f} : X \to Y.\lambda\mathtt{x} : \mathbb{N} \times X.\mathtt{let}\ \langle\mathtt{x}_1, \mathtt{x}_2\rangle = \mathtt{x}\ \mathtt{in}\ \langle\mathtt{x}_1, \mathtt{f}\,\mathtt{x}_2\rangle.$$

Let $\mathbb{N}^\omega = \nu X.F(X)$*. Proposition 5 ensures that* $(\mathbb{N}^\omega, \mathtt{out}_{\mathbb{N}^\omega})$ *is a weak final coalgebra: the weak final coalgebra of streams over natural numbers. We can define the usual operations on streams as* $\mathtt{head} = \lambda\mathtt{x} : \mathbb{N}^\omega.\mathtt{let}\ \langle\mathtt{x}_1, \mathtt{x}_2\rangle = (\mathtt{out}_{\mathbb{N}^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{x}_1$*, and* $\mathtt{tail} = \lambda\mathtt{x} : \mathbb{N}^\omega.\mathtt{let}\ \langle\mathtt{x}_1, \mathtt{x}_2\rangle = (\mathtt{out}_{\mathbb{N}^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{x}_2$*. We can use the fact that* $\mathbb{N}^\omega$ *is a weak final coalgebra to define streams. As an example we can define a constant stream of ks by using a function:*

$$g = \lambda\mathtt{x} : \mathbf{1}.\mathtt{let}\ () = \mathtt{x}\ \mathtt{in}\ \langle k, ()\rangle.$$

Proposition 6 ensures that we can define $\mathtt{const} = \mathtt{unfold}_{\mathbb{N}^\omega}\ \mathbf{1}\ g()$*. Similarly, we can define a function that extracts from a stream the elements in even position. This time we need a function:*

$$g = \lambda\mathtt{x} : \mathbb{N}^\omega.\langle\mathtt{hd}\,\mathtt{x}, \mathtt{tl}\,(\mathtt{tl}\,\mathtt{x})\rangle.$$

Proposition 6 ensures that we can define $\mathtt{even} = \mathtt{unfold}_{\mathbb{N}^\omega}\ \mathbb{N}^\omega\ g$*.*

3. The Light Affine Lambda Calculus

The Light Affine Lambda Calculus is the affine version of the Light Linear Lambda Calculus [40] and provide a concrete syntax for Intuitionistic Light Affine Logic [1].

3.1 The Language

The syntax of the Light Affine Lambda Calculus (LALC) is inspired by the restrictions provided by Light Affine Logic. The focus of our work is on the expressivity of the calculus rather than on other properties, so to make our examples more clear we adopt an explicitly typed version of LALC. The types and the terms of LALC are presented in Figure 1. As basic type we consider only the multiplicative unit $\mathbf{1}$, while as type constructors we consider the linear implication $\multimap$, the tensor product $\otimes$, and the additive disjunction $\oplus$. Moreover, we have type variables X and type universal and existential quantifications: $\forall X.\tau$, and $\exists X.\tau$. We also have two modalities, $!$ and $\S$. The connectives $\otimes$, $\oplus$ and the existential quantification $\exists X.\tau$ can be defined by using only the linear implication $\multimap$, the modalities $!$, $\S$, and the universal quantification $\forall$ but we prefer here to consider them as primitive. The reason behind this choice is that in the second part of the paper we will introduce explicit rules for distributing the $\S$ modality over $\otimes$ and $\oplus$, so it is natural to consider them as primitive.

Every type constructor comes equipped with a term constructor and a term destructor. Since we consider explicitly typed terms we avoid confusions by denoting $\hat{!}$ and $\hat{\S}$ the term level constructors for the modalities $!$ and $\S$, respectively. The semantics of LALC is defined in terms of the reduction relation $\to$ described in Figure 2 where we use the notation $[\mathtt{M}/\mathtt{x}]$ for the usual capture avoiding term substitution, the notation $[\tau/\mathtt{X}]$ for the usual capture avoiding type substitution, and $\dagger, \ddagger$ to denote the modalities $!$ or $\S$.

We have three kinds of reduction rules: the *exponential* rules describe the interaction of a constructor and a destructor for modalities, the *beta* rules describe instead the interaction of a constructor and a destructor for all the other types, the *commuting conversion* rules describe the interaction of different destructors. In Figure 2 we have omitted several commuting rules. The number of these rules is quite high and their behavior is standard. We consider only two such rules (com-1) and (com-2) as representative of this class.

$$\begin{array}{rl}
\tau, \sigma ::= & X \mid \mathbf{1} \mid !\tau \mid \S\tau \mid \tau \oplus \sigma \mid \tau \otimes \sigma \mid \tau \multimap \sigma \\
& \mid \forall X.\tau \mid \exists X.\tau \\
\mathtt{M}, \mathtt{N}, \mathtt{L} ::= & \mathtt{x} \mid () \mid \lambda\mathtt{x} : \tau.\mathtt{M} \mid \mathtt{M}\,\mathtt{N} \mid \Lambda\mathtt{X}.\mathtt{M} \mid \mathtt{M}\,\tau \mid \hat{!}\mathtt{M} \mid \hat{\S}\mathtt{M} \\
& \mid \mathtt{let}\ \hat{\S}\mathtt{x} : \tau = \mathtt{M}\ \mathtt{in}\ \mathtt{N} \mid \mathtt{let}\ \hat{!}\mathtt{x} : \tau = \mathtt{M}\ \mathtt{in}\ \mathtt{N} \\
& \mid \langle\mathtt{M}, \mathtt{N}\rangle \mid \mathtt{let}\ \langle\mathtt{x} : \tau_1, \mathtt{y} : \tau_2\rangle = \mathtt{M}\ \mathtt{in}\ \mathtt{N} \\
& \mid \mathtt{pack}\ (\mathtt{M}, \sigma)\ \mathtt{as}\ \tau \mid \mathtt{unpack}\ \mathtt{M}\ \mathtt{as}\ (\mathtt{X}, \mathtt{x})\ \mathtt{in}\ \mathtt{N} \\
& \mid \mathtt{let}\ () = \mathtt{M}\ \mathtt{in}\ \mathtt{N} \mid \mathtt{inj}_{\mathtt{i}}^{\tau}(\mathtt{M}) \mid \\
& \mid \mathtt{case}\ \mathtt{M}\ \mathtt{of}\ \{\mathtt{inj}_0^{\tau}(\mathtt{x}) \to \mathtt{N} \mid \mathtt{inj}_1^{\tau}(\mathtt{x}) \to \mathtt{L}\}
\end{array}$$

Figure 1. LALC: grammar for types and terms.

3.2 Type System

A typing judgment is of the shape $\Gamma \vdash \mathtt{M} : \tau$, for some typing environment Γ (an environment assigning types to term variables), some term $\mathtt{M}$ and some type τ. The standard typing rules, inherited from Light Affine λ-calculus, are given in Figure 3(a) and additional rules for the extra constructs are given in Figure 3(b). As usual, this system uses the notion of discharged formulas, which are expressions of the form $[\tau]_\dagger$. Given a typing environment $\Gamma = \mathtt{x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n$, $[\Gamma]_\dagger$ is a notation for the environment $\mathtt{x}_1 : [\tau_1]_\dagger, \ldots \mathtt{x}_n : [\tau_n]_\dagger$. Discharged formulas are not types, so they cannot be abstracted and we do not want them to appear in final judgments. They are just syntactic artifacts introduced by the rule $(!I)$ and $(\S I)$, used by the rule (C), and eliminated by the rules $(!E)$ and $(\S E)$, respectively. These five rules implement in a natural deduction style the three sequent calculus LAL rules we discussed in the introduction and the *cut rule* on modalities. All the other rules are the linear versions of the standard System F rules. We assume a *multiplicative* management of environments: when we write Γ, Δ we assume that the set of free variables in Γ and Δ are disjoint. The only rule that uses in part an *additive* management of environments is the rule $(\oplus E)$ where we have a sharing of variables in the two branches of the $\mathtt{case}$ construction. We adopt here the same convention as in Light Linear Logic (LLL) [21] and we consider a *lazy* reduction that reduces redexes with variable bound in the two branches only when the argument is closed.

The polynomial soundness of LALC can be expressed in terms of the *depth* $d(\mathtt{M})$ of a term $\mathtt{M}$: the maximal number of nested $\hat{!}$ or $\hat{\S}$ that can be found in any path of the term syntax tree. Moreover, we will call depth of a subterm $\mathtt{N}$ of $\mathtt{M}$ the number of $\hat{!}$ and $\hat{\S}$ that one has to cross to reach the root of $\mathtt{N}$ starting from the root of $\mathtt{M}$.

3.3 Properties of LALC

LALC provides a characterization of the FPTIME complexity class. However, it also enjoys standard properties of typed lambda calculi. In particular, it enjoys subject reduction.

Theorem 9 (Subject Reduction)**.** *Let* $\Gamma \vdash \mathtt{M} : \tau$ *and* $\mathtt{M} \to \mathtt{N}$*, then* $\Gamma \vdash \mathtt{N} : \tau$*.*

In fact, LALC enjoys also a stronger version of subject reduction that ensures not only that types are preserved, but also that a reduction $\mathtt{M} \to \mathtt{N}$ corresponds to a rewriting of the type derivation of $\mathtt{M}$ in the type derivation of $\mathtt{N}$.

Polynomial time soundness for LALC can be stated as follow:

Theorem 10 (Polynomial Time Soundness)**.** *Consider a term* $\Gamma \vdash \mathtt{M} : \tau$*. Then,* $\mathtt{M}$ *can be reduced to normal form by a Turing Machine*

$$(\lambda \mathtt{x} : \tau.\mathtt{M})\, \mathtt{N} \to \mathtt{M}[\mathtt{N}/\mathtt{x}] \qquad \text{(beta-}\lambda\text{)}$$
$$\mathtt{let}\ () = ()\ \mathtt{in}\ \mathtt{M} \to \mathtt{M} \qquad \text{(beta-}\mathbf{1}\text{)}$$
$$(\Lambda X.\mathtt{M})\, \tau \to \mathtt{M}[\tau/X] \qquad \text{(beta-}\forall\text{)}$$
$$\mathtt{case}\ \mathtt{inj}^{\tau}_{\mathtt{i}}(\mathtt{M})\ \mathtt{of}\ \{\mathtt{inj}^{\tau}_{0}(\mathtt{x}) \to \mathtt{N}_0 | \mathtt{inj}^{\tau}_{1}(\mathtt{x}) \to \mathtt{N}_1\} \to \mathtt{N}_{\mathtt{i}}[\mathtt{M}/\mathtt{x}] \qquad \text{(beta-}\oplus\text{)}$$
$$\mathtt{let}\ \langle \mathtt{x} : \tau, \mathtt{y} : \sigma \rangle = \langle \mathtt{N}_0, \mathtt{N}_1 \rangle\ \mathtt{in}\ \mathtt{M} \to \mathtt{M}[\mathtt{N}_0/\mathtt{x}, \mathtt{N}_1/\mathtt{y}] \qquad \text{(beta-}\otimes\text{)}$$
$$\mathtt{unpack}\ (\mathtt{pack}\ (\mathtt{M}, \sigma)\ \mathtt{as}\ \exists \mathtt{X}.\tau)\ \mathtt{as}\ (\mathtt{X}, \mathtt{x})\ \mathtt{in}\ \mathtt{N} \to \mathtt{N}[\mathtt{M}/\mathtt{x}, \sigma/X] \qquad \text{(beta-}\exists\text{)}$$

$$\mathtt{let}\ \hat{\S}\mathtt{x} : \S\tau = \hat{\S}\mathtt{N}\ \mathtt{in}\ \mathtt{M} \to \mathtt{M}[\mathtt{N}/\mathtt{x}] \qquad \text{(exp-}\hat{\S}\text{)}$$
$$\mathtt{let}\ \hat{!}\mathtt{x} : !\tau = \hat{!}\mathtt{N}\ \mathtt{in}\ \mathtt{M} \to \mathtt{M}[\mathtt{N}/\mathtt{x}] \qquad \text{(exp-}\hat{!}\text{)}$$

$$(\mathtt{let}\ \hat{\dagger}\mathtt{x} : \dagger\tau = \mathtt{N}\ \mathtt{in}\ \mathtt{M})\mathtt{L} \to \mathtt{let}\ \hat{\dagger}\mathtt{x} : \dagger\tau = \mathtt{N}\ \mathtt{in}\ (\mathtt{ML}) \qquad \text{(com-1)}$$
$$\mathtt{let}\ \hat{\dagger}\mathtt{y} : \dagger\tau = (\mathtt{let}\ \hat{\ddagger}\mathtt{x} : \ddagger\sigma = \mathtt{N}\ \mathtt{in}\ \mathtt{L})\ \mathtt{in}\ \mathtt{M} \to \mathtt{let}\ \hat{\ddagger}\mathtt{x} : \ddagger\sigma = \mathtt{N}\ \mathtt{in}\ (\mathtt{let}\ \hat{\dagger}\mathtt{y} : \dagger\tau = \mathtt{L}\ \mathtt{in}\ \mathtt{M}) \qquad \text{(com-2)}$$

Figure 2. Light Affine Lambda Calculus reduction rules.

working in time polynomial in $|\mathtt{M}|$ *with exponent proportional to* $d(\mathtt{M})$.

The original proof of this theorem by Girard [21] as well as other subsequent proofs [2, 40] are based on three main observations about reductions in LALC:

1. reductions cannot increase the depth of a term,
2. a beta reduction at depth i decreases the size of the term at depth i and cannot increase the size of the term at other depths,
3. any sequence of exponential reduction at depth i can only square the size of the term at every depth j greater than i.

These properties suggest a depth-by-depth reduction strategy whose length is polynomial in the size of the term and exponential in the depth. So, we have a polynomial bound when the depth is fixed. A key argument for the use of depth-by-depth reduction is that this reduction is *enough to reach normal forms*. This will not be the case for the extension to LALC that we will present in Section 6.

For expressing the FPTIME completeness statement for LALC we need a data type $\mathbb{B}^*$ for strings of Booleans that can be easily defined by a standard Church encoding.

Theorem 11 (FPTIME completeness)**.** *For every polynomial time function* $f : \{0,1\}^* \to \{0,1\}^*$ *there exists a natural number* n *and a term* $\vdash \mathtt{f} : \mathbb{B}^* \multimap \hat{\S}^n \mathbb{B}^*$ *such that* $\mathtt{f}$ *represents* f.

The proof of this statement requires to show that one can program in LALC all the polynomial expressions that one can define data types for Turing Machine's configurations, and that transitions between configurations are definable.

4. Algebras in LALC

We want now to adapt the encoding of algebras in System F to the case of LALC. The first thing that we need is to find a type that permits to express a weak initial F-algebra. One can consider the straightforward linear type $T = \forall X.(F(X) \multimap X) \multimap X$ but this is not enough for typing an analog of the term $\mathtt{in}_{\mathtt{T}}$ that contains a duplicated variable $\mathtt{k}$. Consequently, modalities are required in the corresponding type as the duplication in the term $\mathtt{in}_{\mathtt{T}}$ is needed for iteration. So, a more natural choice is instead to use the type:

$$T = \forall X.!(F(X) \multimap X) \multimap \S X. \tag{1}$$

Indeed, this type can be seen as the analog of the one used for the standard Church natural numbers in LALC: $\forall X.!(X \multimap X) \multimap \S(X \multimap X)$, see [21]. In this type, the modality $!$ is a marker for the duplication of the successor function and the modality $\S$ witness its iteration.

Using this type for assigning types in LALC to terms analog to the one of Proposition 5 presents two problems. First, the modality $\S$ in Equation 1—that witnesses iteration—propagates when one wants to build the F-homomorphism to another F-algebra. This implies that only a restricted form of weak initial algebras can be obtained. So, we are able to build the needed F-algebra homomorphisms only with F-algebra of the form $(\S B, g : F(\S B) \to \S B)$. There is a second problem: we need the functor F to *left-distribute* over $\S$[3]. This corresponds to requiring the existence of a morphism:

$$L_F : F(\S X) \multimap \S F(X).$$

This technical requirement comes from the fact that the modality $\S$ propagates due to the iteration, but also from the uniformity imposed by the polymorphic encoding. This uniformity corresponds to requiring that the algebra $(\S B, g)$ target of the F-algebra homomorphism comes from an underlying F-algebra (B, f) via the functoriality of $\S$ and the left-distributivity L_F of F.

By considering the two requirements, we obtain the following definition.

Definition 12. *Given a functor* F*, we say that an* F*-algebra* (A, a) *is* weakly-initial under $\S$ *if for every* F*-algebra of the form* (B, f) *there exists an* F*-algebra* $(\S B, g)$ *and an* F*-algebra homomorphism* $h : A \to \S B$ *making the following diagram commute:*

$$\begin{tikzcd}
F(A) \arrow[r, "F(h)"] \arrow[dd, "a"'] & F(\S B) \arrow[dd, "g"] \arrow[dr, "L_F"] & \\
 & & \S F(B) \arrow[dl, "\S f"] \\
A \arrow[r, "h"'] & \S B &
\end{tikzcd}$$

That is, we require the existence of an F-algebra homomorphism only for F-algebra of the form $(\S B, g)$ that comes from an underlying F-algebra (B, f) via the functoriality of $\S$ and the left-distributivity L_F of F. With these two restrictions in mind we can now formulate an analog of Proposition 5.

Theorem 13. *Let* F *be a functor definable in LALC that left-distributes over* $\S$*, and let* $T = \forall X.!(F(X) \multimap X) \multimap \S X$.

[3] We call this property "distribute" instead of "commute" in order to highlight the distinction with the standard commuting rules (com-n).

$$\frac{}{\mathtt{x}:\tau \vdash \mathtt{x}:\tau}\ (Ax) \qquad \frac{\Gamma \vdash \mathtt{M}:\tau}{\Gamma,\Delta \vdash \mathtt{M}:\tau}\ (W) \qquad \frac{\Gamma, \mathtt{x}:[\tau]_!, \mathtt{y}:[\tau]_! \vdash \mathtt{M}:\sigma}{\Gamma, \mathtt{z}:[\tau]_! \vdash \mathtt{M}[\mathtt{z}/\mathtt{x},\mathtt{z}/\mathtt{y}]:\sigma}\ (C) \qquad \frac{\Gamma,\mathtt{x}:\tau \vdash \mathtt{M}:\sigma}{\Gamma \vdash \lambda\mathtt{x}:\tau.\mathtt{M}:\tau \multimap \sigma}\ (\multimap I)$$

$$\frac{\Gamma \vdash \mathtt{M}:\tau \multimap \sigma \quad \Delta \vdash \mathtt{N}:\tau}{\Gamma,\Delta \vdash \mathtt{MN}:\sigma}\ (\multimap E) \qquad \frac{\Gamma \vdash \mathtt{N}:!\tau \quad \Delta,\mathtt{x}:[\tau]_! \vdash \mathtt{M}:\sigma}{\Gamma,\Delta \vdash \mathtt{let}\ \hat{!}\mathtt{x}\ :!\tau = \mathtt{N}\ \mathtt{in}\ \mathtt{M}:\sigma}\ (!E) \qquad \frac{\Gamma \vdash \mathtt{N}:\S\tau \quad \Delta,\mathtt{x}:[\tau]_\S \vdash \mathtt{M}:\sigma}{\Gamma,\Delta \vdash \mathtt{let}\ \hat{\S}\mathtt{x}:\S\tau = \mathtt{N}\ \mathtt{in}\ \mathtt{M}:\sigma}\ (\S E)$$

$$\frac{\Gamma \vdash \mathtt{M}:\tau \quad \Gamma \subseteq \{\mathtt{x}:\sigma\}}{[\Gamma]_! \vdash \hat{!}\mathtt{M}\ :!\tau}\ (!I) \qquad \frac{\Gamma,\Delta \vdash \mathtt{M}:\tau}{[\Gamma]_!,[\Delta]_\S \vdash \hat{\S}\mathtt{M}:\S\tau}\ (\S I) \qquad \frac{\Gamma \vdash \mathtt{M}:\tau \quad X \notin \mathrm{FTV}(\Gamma)}{\Gamma \vdash \Lambda X.\mathtt{M}:\forall X.\tau}\ (\forall I) \qquad \frac{\Gamma \vdash \mathtt{M}:\forall X.\tau}{\Gamma \vdash \mathtt{M}\sigma:\tau[\sigma/X]}\ (\forall E)$$

(a) standard rules

$$\frac{\Gamma \vdash \mathtt{M}:\tau \quad \Delta \vdash \mathtt{N}:\sigma}{\Gamma,\Delta \vdash \langle \mathtt{M},\mathtt{N}\rangle:\tau\otimes\sigma}\ (\otimes I) \qquad \frac{\Gamma \vdash \mathtt{M}:\tau\otimes\sigma \quad \Delta,\mathtt{x}:\tau,\mathtt{y}:\sigma \vdash \mathtt{N}:\tau'}{\Gamma,\Delta \vdash \mathtt{let}\ \langle \mathtt{x}:\tau,\mathtt{y}:\sigma\rangle = \mathtt{M}\ \mathtt{in}\ \mathtt{N}:\tau'}\ (\otimes E) \qquad \frac{}{\Gamma \vdash ():\mathbf{1}}\ (\mathbf{1}I)$$

$$\frac{\Gamma \vdash \mathtt{M}:\mathbf{1} \quad \Delta \vdash \mathtt{N}:\tau}{\Gamma,\Delta \vdash \mathtt{let}\ () = \mathtt{M}\ \mathtt{in}\ \mathtt{N}:\tau}\ (\mathbf{1}E) \qquad \frac{\Gamma \vdash \mathtt{M}:\tau_i}{\Gamma \vdash \mathtt{inj}_i^{\tau_0\oplus\tau_1}(\mathtt{M}):\tau_0\oplus\tau_1}\ (\oplus I) \qquad \frac{\Gamma \vdash \mathtt{M}:\tau[\sigma/X]}{\Gamma \vdash \mathtt{pack}\ (\mathtt{M},\sigma)\ \mathtt{as}\ \exists \mathtt{X}.\tau : \exists X.\tau}\ (\exists I)$$

$$\frac{\Gamma \vdash \mathtt{M}:\tau_0\oplus\tau_1 \quad \Delta,\mathtt{x}:\tau_0 \vdash \mathtt{N}_0:\tau \quad \Delta,\mathtt{x}:\tau_1 \vdash \mathtt{N}_1:\tau}{\Gamma,\Delta \vdash \mathtt{case}\ \mathtt{M}\ \mathtt{of}\ \{\mathtt{inj}_0^{\tau_0\oplus\tau_1}(\mathtt{x}) \to \mathtt{N}_0 | \mathtt{inj}_1^{\tau_0\oplus\tau_1}(\mathtt{x}) \to \mathtt{N}_1\}:\tau}\ (\oplus E) \qquad \frac{\Gamma \vdash \mathtt{M}:\exists X.\tau \quad \Delta,\mathtt{x}:\tau \vdash \mathtt{N}:\sigma}{\Gamma,\Delta \vdash \mathtt{unpack}\ \mathtt{M}\ \mathtt{as}\ (\mathtt{X},\mathtt{x})\ \mathtt{in}\ \mathtt{N}:\sigma}\ (\exists E)$$

(b) rules for additional constructions

Figure 3. Typing rules for the Light Affine Lambda Calculus.

Consider the morphisms defined by:

$$\begin{aligned}
&\mathtt{in}_T : F(T) \multimap T,\\
&\mathtt{in}_T = \lambda\mathtt{s} : F(T).\Lambda X.\lambda\mathtt{k}\ :!(F(X) \multimap X).\\
&\qquad \mathtt{let}\ \hat{!}\mathtt{y}\ :!(F(X) \multimap X) = \mathtt{k}\ \mathtt{in}\\
&\qquad \mathtt{let}\ \hat{\S}\mathtt{z} : \S F(X) = L_F(F(\mathtt{fold}_T\ X\ \hat{!}\mathtt{y})\mathtt{s})\ \mathtt{in}\ \hat{\S}(\mathtt{y}\ \mathtt{z}),
\end{aligned}$$

$$\begin{aligned}
&\mathtt{fold}_T : \forall X.!(F(X) \multimap X) \multimap T \multimap \S X,\\
&\mathtt{fold}_T = \Lambda X.\lambda\mathtt{k}\ :!(F(X) \multimap X).\lambda\mathtt{t} : T.\mathtt{t}\ X\ \mathtt{k}.
\end{aligned}$$

Then, $(T, \mathtt{in}_T)$ is a weakly-initial F-algebra under $\S$: for every F-algebra $(B, f : F(B) \multimap B)$ we have an F-algebra $(\S B, g : F(\S B) \to \S B)$ and an F-algebra homomorphism $h : T \to \S B$ defined as $h = \mathtt{fold}_T\ B\ \hat{!}f$.

The situation described by Theorem 13 corresponds to saying that for every F-algebra (B, f) the following diagram commutes:

$$\begin{tikzcd}
F(T) \arrow[r, "F(\mathtt{fold}_T\ B\ \hat{!}f)"] \arrow[dd, "\mathtt{in}_T"'] & F(\S B) \arrow[dd, "g"'] \arrow[dr, "L_F"] & \\
 & & \S F(B) \arrow[dl, "\S f"] \\
T \arrow[r, "\mathtt{fold}_T\ B\ \hat{!}f"'] & \S B &
\end{tikzcd}$$

This diagram provides a way to encode the least fixpoint of types, similarly to what we have for System F, and so to define standard data types.

Notice that with respect to initial algebras we have now relaxed both the uniqueness and the existence property.

4.1 Algebra Examples

Before providing examples of algebras, we want to characterize a large class of functors that left-distribute.

Lemma 14. *All the functors built using the following signature left-distribute over $\S$:*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \oplus F(X) \mid F(X) \otimes F(X),$$

provided that A is a closed type for which there exists a closed term of type $A \multimap !A$ or type $A \multimap \S A$.

Proof. By induction on $F(X)$. □

Thanks to the above lemma we can give a notion of weakly-initial F-algebra under $\S$ to several standard examples.

Example 15. *Consider the functor $F(X) = \mathbf{1} \oplus X$. This is the linear analog of the functor considered in Example 7, definable by the same term (in the types annotation, implication is replaced by a linear arrow and $+$ is replaced by $\oplus$). By Lemma 14, we have that F left-distributes over $\S$, and so by Theorem 13 we have that $(\mathbb{N}, \mathtt{in}_\mathbb{N})$ is a weakly-initial F-algebra under $\S$, where by abuse of notation we again use $\mathbb{N}$ to denote $\mu X.F(X)$. Similarly to what we did in Example 7, we can define natural numbers as inhabitants of this type. Noticing that to the term g defined there we can also give the type $F(\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N})$, we have that $\mathtt{add} = \mathtt{fold}_\mathbb{N}\ (\mathbb{N} \multimap \mathbb{N})\ \hat{!}g$ has type $\mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})$.*

Example 16. *Consider the functor $F_n(X) = \mathbf{1} \oplus (\mathbb{B}_n \otimes X)$ where $\mathbb{B}_n$ is a finite type with n states. The functor $F_n(X)$ is definable by the term:*

$$\begin{aligned}
&\lambda\mathtt{f} : X \multimap Y.\lambda\mathtt{x} : F_n(X).\ \mathtt{case}\ \mathtt{x}\ \mathtt{of}\\
&\qquad\qquad \{\mathtt{inj}_0(\mathtt{z}) \to \mathtt{inj}_0(()),\\
&\mathtt{inj}_1(\mathtt{z}) \to \mathtt{let}\langle \mathtt{z}_1 : \mathbb{B}_n, \mathtt{z}_2 : X\rangle = \mathtt{z}\ \mathtt{in}\ \mathtt{inj}_1(\langle \mathtt{z}_1, \mathtt{f}\ \mathtt{z}_2\rangle)\},
\end{aligned}$$

where we omit the superscripts of the $\mathtt{inj}_i$ constructs for readability. It is easy to verify that by Lemma 14, we have that F_n left-distributes over $\S$. So, if we define $\mathbb{B}_n^* = \mu X.F_n(X)$, by Theorem 13 we have that $(\mathbb{B}_n^*, \mathtt{in}_{\mathbb{B}_n^*})$ is a weakly-initial F-algebra under $\S$. In the particular case where $n = 2$, let $\mathbb{B}_2 = \mathbf{1} \oplus \mathbf{1}$ be the type for booleans. The type $\mathbb{B}_2^*$ is inhabited by finite boolean strings: $\mathtt{nil} = \mathtt{in}_{\mathbb{B}_2^*}(\mathtt{inj}_0(()))$, $\mathtt{cons} = \lambda\mathtt{h} : \mathbb{B}_2.\lambda\mathtt{t} : \mathbb{B}_2^*.\mathtt{in}_{\mathbb{B}_2^*}(\mathtt{inj}_1(\langle \mathtt{h}, \mathtt{t}\rangle))$.

We can define a map *function on boolean strings using the function:*

$$g = \lambda \mathtt{f} : \mathbb{B}_2 \multimap \mathbb{B}_2 . \lambda \mathtt{x} : F_2(\mathbb{B}_2^*).\ \mathtt{case\ x\ of}$$
$$\{\mathtt{inj}_0(\mathtt{z}) \to \mathtt{in}_{\mathbb{B}_2^*}(\mathtt{inj}_0(())),$$
$$\mathtt{inj}_1(\mathtt{z}) \to \mathtt{let}\langle \mathtt{z}_1 : \mathbb{B}_2, \mathtt{z}_2 : \mathbb{B}_2^* \rangle = \mathtt{z\ in\ in}_{\mathbb{B}_2^*}(\mathtt{inj}_1(\langle \mathtt{f\, z}_1, \mathtt{z}_2 \rangle))).$$

Noticing that for a variable $\mathtt{f} : \mathbb{B}_2 \multimap \mathbb{B}_2$*, to the term* $g\,\mathtt{f}$ *we can give the type* $F(\mathbb{B}_2^*) \multimap \mathbb{B}_2^*$*, we have that* $\mathtt{map\ f} = \mathtt{fold}_{\mathbb{B}_2^*}\, \mathbb{B}_2^*\, \hat{!}(g\,\mathtt{f})$ *has type* $\mathbb{B}_2^* \multimap \S\mathbb{B}_2^*$.

The next section will show an extensive example in programming with these data structures.

4.2 Polynomial Time Completeness of LALC using Algebras

As a sanity check, we want to use the new encoding of algebras to prove the FPTIME completeness of LALC. This follows the same line as the standard proof from Asperti and Roversi [2] or the one from Baillot et al. [6]. The idea is to show that we can use algebras to encode polynomial expressions—that can be used as clocks for iterations—and Turing Machines and their transitions.

We have seen how to define natural numbers as inhabitants of the type $\mathbb{N} = \mu X.\mathbf{1} \oplus X$. It is important to stress that all the inhabitants of $\mathbb{N}$ can be typed with a fixed number of $(!I)$ and $(\S I)$ rules—this corresponds to having terms with constant *depth* as defined in Definition 23—this is quite standard but still important to stress in order to ensure a sound characterization of PTIME, see [28]. We want now to show that we can encode polynomial expressions and that we can use them as clocks of iterators. Let us start with the latter. Thanks to Theorem 13 and the fact that given two terms of type $\mathbf{1} \multimap A$ and $A \multimap A$ we can build a term of type $\mathbf{1} \oplus A \multimap A$ combining them, we can define an iteration scheme parametrized by the type A over natural numbers as:

$$\mathtt{iter} : \mathbb{N} \multimap (\mathbf{1} \multimap A) \multimap (A \multimap A) \multimap \S A.$$

This term has the property that for every $\mathtt{n}$, given a *base* $\mathtt{b}$ and a *step* $\mathtt{s}$ it produces the n-th iteration of $\mathtt{s}$ over $\mathtt{b}$. More precisely:

$$\mathtt{iter\,n\,b\,s} \to^* \hat{\S}(\mathtt{s}^{\mathtt{n}}\,\mathtt{b}).$$

As an example, we can make explicit the base and the iteration step for addition $\mathtt{add} : \mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})$:

$$\mathtt{b}_{\mathtt{add}} = \lambda \mathtt{z} : \mathbf{1}.\mathtt{let}\ () = \mathtt{z\ in}\ \lambda \mathtt{y.y} : \mathbf{1} \multimap (\mathbb{N} \multimap \mathbb{N}),$$
$$\mathtt{s}_{\mathtt{add}} = \lambda \mathtt{z}.\lambda \mathtt{y}.\mathtt{succ}(\mathtt{z\,y}) : (\mathbb{N} \multimap \mathbb{N}) \multimap (\mathbb{N} \multimap \mathbb{N}).$$

The type of $\mathtt{add}$ is not entirely satisfying, however we can define a coercion function:

$$\mathtt{coer} : (\mathbb{N} \multimap \S(\mathbb{N} \multimap \mathbb{N})) \multimap \mathbb{N} \multimap \S\mathbb{N} \multimap \S\mathbb{N},$$
$$\mathtt{coer} = \lambda \mathtt{f}.\lambda \mathtt{n}.\lambda \mathtt{m}.\mathtt{let}\ \hat{\S}\mathtt{u} = \mathtt{m\ in\ (let}\ \hat{\S}\mathtt{z} = \mathtt{f\,n\ in}\ \hat{\S}(\mathtt{z\,u})).$$

Moreover, we have a coercion function:

$$\mathtt{coer}' : \mathbb{N} \multimap \S\mathbb{N},$$
$$\mathtt{coer}' = \lambda \mathtt{n}.\mathtt{iter\,n}\,\underline{0}\,\mathtt{succ}.$$

Thanks to these coercion functions we can change the type of addition:

$$\mathtt{add}_c : \mathbb{N} \multimap \mathbb{N} \multimap \S\mathbb{N},$$
$$\mathtt{add}_c = \lambda \mathtt{n} : \mathbb{N}.\lambda \mathtt{m} : \mathbb{N}.\mathtt{coer\ add\ n\ (coer}'\,\mathtt{m}).$$

This is the same type that we can assign to addition in LLL. While this type is good for adding together several elements, in order to define multiplication it is also convenient to give to addition the following type:

$$\mathtt{add}_\S : \S\mathbb{N} \multimap \S^2\mathbb{N} \multimap \S^2\mathbb{N},$$
$$\mathtt{add}_\S = \lambda \mathtt{n} : \S\mathbb{N}.\lambda \mathtt{m} : \S^2\mathbb{N}.\mathtt{let}\ \hat{\S}\mathtt{u} = \mathtt{n}$$
$$\mathtt{in\ let}\ \hat{\S}\mathtt{w} = \mathtt{m\ in}\ \hat{\S}(\mathtt{coer\ add\ u\ w}).$$

We can now define the base step and the iteration step for multiplication $\mathtt{mul} : \mathbb{N} \multimap \S(!\mathbb{N} \multimap \S^2\mathbb{N})$:

$$\mathtt{b}_{\mathtt{mul}} = \lambda \mathtt{z} : \mathbf{1}.\mathtt{let}\ () = \mathtt{z\ in}$$
$$\lambda \mathtt{y} :!\mathbb{N}.\mathtt{let}\ \hat{!}\mathtt{v} = \mathtt{y\ in}\ \hat{\S}(\mathtt{coer}'\,\mathtt{v}) : \mathbf{1} \multimap (!\mathbb{N} \multimap \S^2\mathbb{N}),$$

$$\mathtt{s}_{\mathtt{mul}} = \lambda \mathtt{g} :!\mathbb{N} \multimap \S^2\mathbb{N}.\lambda \mathtt{y} :!\mathbb{N}.\mathtt{let}\ !\mathtt{z} = \mathtt{y\ in}$$
$$(\mathtt{add}_\S\, \hat{\S}\mathtt{z}\,(\mathtt{g}\,!\mathtt{z})) : (!\mathbb{N} \multimap \S^2\mathbb{N}) \multimap (!\mathbb{N} \multimap \S^2\mathbb{N}).$$

By using another coercions similar to $\mathtt{coer}$ and $\mathtt{coer}'$ we can assign to multiplication a type as: $\mathtt{mul}_c : \mathbb{N} \multimap !\mathbb{N} \multimap \S^3\mathbb{N}$. By using addition and multiplication we can prove the following.

Lemma 17. *For any polynomial* $p[x]$ *in the variable* x *there exists an integer* n *and a term* $\lambda \mathtt{x}.\underline{\mathtt{p}}$ *of type* $\mathbb{N} \multimap \hat{\S}^n\mathbb{N}$ *representing* $p[x]$.

Likewise Asperti and Roversi [2] the above lemma can be also improved by expressing the number n of $\S$ in term of the degree of the polynomial. However, in our case we would have some extra $\S$ given by the extra $\S$ that we have in the type of $\mathtt{mul}_c$.

Now we need to encode Turing Machines $\mathcal{M}$. We can encode a configuration of $\mathcal{M}$ with n states by a type $\mathbb{M}_n = \mathbb{B}_3^* \otimes \mathbb{B}_3^* \otimes \mathbb{B}_n$ where $\mathbb{B}_3^*$ is the type of strings over a three symbols alphabet, and $\mathbb{B}_n$ is a finite type of length n. The first $\mathbb{B}_3^*$ represents the left part of the tape while the second one represents the right part of the tape, starting from the scanned symbol. The type $\mathbb{B}_n$ represents the state. The transition function δ between configurations can be defined by case analysis: we can represent δ by a term $\mathtt{delta} : \mathbb{M}_n \multimap \mathbb{M}_n$. So, we can use the iteration scheme to iterate transitions starting from an initial configuration. We then have the following:

Theorem 18 (FPTIME completeness)**.** *For every polynomial time function* $f : \{0,1\}^* \to \{0,1\}^*$ *there exists a natural number* n *and a term* $\mathtt{f} : \mathbb{B}_3^* \multimap \hat{\S}^n\mathbb{B}_3^*$ *such that and* $\mathtt{f}$ *represents* f.

The proof uses the type of configurations as described above and is similar to the one presented by Asperti and Roversi [2] or the one presented by Baillot et al. [6].

5. Coalgebras in LALC

Trying to adapt the encoding of final coalgebras we hit unsurprisingly the same problem we had for the encoding of initial algebra. Knowing now the recipe we can consider the type:

$$T = \exists X.!(X \multimap F(X)) \otimes \S X. \qquad (2)$$

By duality, we are able to build F-coalgebra homomorphisms only with coalgebras of the shape $(\S B, g : \S B \multimap F(\S B))$ and we will require the functor F to *right-distribute* over $\S$. The latter corresponds to requiring the existence of a morphism:

$$R_F : \S F(X) \multimap F(\S X).$$

These requirements come once again from the fact that the modality $\S$ propagates and from the polymorphic encoding. We have the following dual of Definition 12.

Definition 19. *Given a functor* F*, we say that an* F*-coalgebra* (A, a) *is* weakly-final under $\S$ *if for every* F*-coalgebra of the form* (B, f) *there exists an* F*-coalgebra* $(\S B, g)$ *and an* F*-coalgebra homomorphism* $h : \S B \to A$ *making the following diagram* *commute:*

$$\begin{tikzcd}
A \arrow[dd, "a"'] & \S B \arrow[l, "h"'] \arrow[dd, "g"] \arrow[dr, "\S f"] & \\
 & & \S F(B) \arrow[dl, "R_F"] \\
F(A) & F(\S B) \arrow[l, "F(h)"] &
\end{tikzcd}$$

That is, we require the existence of an F-coalgebra homomorphism only for F-coalgebra of the form $(\S B, g)$ that comes from an underlying F-coalgebra (B, f) via the functoriality of $\S$ and the right-distributivity R_F of F.

We can now formulate an analog of Proposition 6.

Theorem 20. *Let F be a functor definable in LALC that right-distributes over $\S$, and let $T = \exists X.!(X \multimap F(X)) \otimes \S X$. Consider the morphisms defined by:*

$\mathtt{out}_T : T \multimap F(T)$,

$\mathtt{out}_T = \lambda \mathtt{t} : T.\mathtt{unpack\ t\ as}\ (X, \mathtt{z})\ \mathtt{in}$

$\quad \mathtt{let}\ \langle \mathtt{k} :!(\mathtt{X} \multimap \mathtt{F(X)}), \mathtt{x} : \S\mathtt{X}\rangle = \mathtt{z\ in}$

$\quad \mathtt{let}\ \hat{!}\mathtt{u} = \mathtt{k\ in}$

$\quad \mathtt{let}\ \hat{\S}\mathtt{v} = \mathtt{x\ in}\ F(\mathtt{unfold}_T\ X\ \hat{!}\mathtt{u}) R_F(\hat{\S}(\mathtt{u\ v}))$,

$\mathtt{unfold}_T : \forall X.!(X \multimap F(X)) \multimap \S X \multimap T$,

$\mathtt{unfold}_T = \Lambda X.\lambda \mathtt{k} :!(X \multimap F(X)).\lambda \mathtt{x} : \S X.\mathtt{pack}\ (\langle \mathtt{k}, \mathtt{x}\rangle, X)\ \mathtt{as}\ T.$

Then, $(T, \mathtt{out}_T)$ is a weakly-final F-coalgebra under $\S$: for every F-coalgebra $(B, f : B \multimap F(B))$ we have an F-coalgebra $(\S B, g : \S B \to F(\S B))$ and an F-coalgebra homomorphism $h : \S B \to T$ defined as $h = \mathtt{unfold}_T\ B\ \hat{!}f$.

The situation described by Theorem 13 corresponds to saying that for every F-coalgebra (B, f) the following diagram commutes:

$$\begin{tikzcd}
T \arrow[dd, "\mathtt{out}_T"'] & \S B \arrow[l, "\mathtt{unfold}_T\ B\ \hat{!}f"'] \arrow[dd, "g"] \arrow[dr, "\S f"] & \\
 & & \S F(B) \arrow[dl, "R_F"] \\
F(T) & F(\S B) \arrow[l, "F(\mathtt{unfold}_T\ B\ \hat{!}f)"] &
\end{tikzcd}$$

This diagram provides a way to encode the greatest fixpoint of a type schema, similarly to what we have for System F, and so to define standard data types.

Once again, with respect to final coalgebras we have now relaxed both the uniqueness and the existence property.

Lack of Examples We want now to give an analog of Lemma 14 describing the functors that right-distribute. Unfortunately, we are able to prove only the following weak lemma.

Lemma 21. *All the functors built using the following signature right-distribute over $\S$:*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X),$$

provided that A is a closed type for which there exists a closed term of type $\S A \multimap A$.

Clearly, this lemma is too weak for defining interesting coinductive examples. Indeed, even a simple functor such as $F(X) = \mathbf{1} \oplus X$ (the type for the natural numbers extended with an infinite object) do not right-distribute. By proof search it is easy to verify that we cannot derive a closed judgment for $\S(\mathbf{1} \oplus X) \multimap \mathbf{1} \oplus \S X$. Similarly, we cannot derive a closed judgment for the type $\S(A \otimes X) \multimap A \otimes \S X$, or the type $\S(A \otimes A \otimes X) \multimap A \otimes A \otimes \S X$. So, we cannot encode infinite lists and infinite trees using Theorem 20.

The root of this problem lies in the fact that in LALC, as in Light Linear Logic, the modality $\S$ does not distribute on the right with $\otimes$ and $\oplus$. That is, we cannot prove in general $\S(A \otimes B) \multimap \S A \otimes \S B$ or $\S(A \oplus B) \multimap \S A \oplus \S B$. Without these distributive rules it seems hard to program any interesting coinductive data in LALC. For this reason, the next step is to support these principles in our language.

6. LALC with Distributions

We want now to add to LALC the distributive principles we discussed in the previous section. The calculus we obtain by adding these principles inherits the polynomial time completeness of LAL. However, we need to do some work in order to show that it preserves the *polynomial time soundness*. Indeed the usual proof technique based on the depth by depth reduction (see [40]) cannot be applied to this calculus as distributions at depth i may create new redexes at depth $i - 1$.

6.1 Extending LALC with Distributions

We extend the grammar of Figure 1 with distributions, constructs that allow the reduction to distribute a $\hat{\S}$ constructor over a $\mathtt{inj}_i^\tau(-)$ or $\langle -, - \rangle$ constructor:

$$\begin{aligned}
\mathtt{M}, \mathtt{N}, \mathtt{L} ::= &\ \ldots \\
&\mid \mathtt{dist}\ \hat{\S}\langle \mathtt{x} : \tau_1, \mathtt{y} : \tau_2\rangle = \mathtt{M\ as\ z} = \langle \hat{\S}\mathtt{x}, \hat{\S}\mathtt{y}\rangle\ \mathtt{in\ N} \\
&\mid \mathtt{dist}\ \hat{\S}\mathtt{inj}_\mathtt{i}^{\tau \oplus \tau'}(\mathtt{x}) = \mathtt{M\ as\ z} = \mathtt{inj}_\mathtt{i}^{\S\tau \oplus \S\tau'}(\hat{\S}\mathtt{x})\ \mathtt{in\ N}.
\end{aligned}$$

We extend the reduction relation $\to$ from Figure 2 with two new distributive rules given in Figure 4 and as usual we denote by $\to^*$ its reflexive, transitive and contextual closure. Similarly, we add the two typing rules for distributions given in Figure 5. In the following, we will sometimes need to consider a term $\mathtt{M}$ with a specific type derivation Π for it[4], we will then use the notation $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$ for some environment Γ and type τ.

The most interesting logical properties of LAL, such as subjection reduction, are preserved by this extension.

Theorem 22 (Subject reduction)**.** *Let Π be a type derivation of the shape $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$. If $\mathtt{M} \to \mathtt{N}$, then there exists a type derivation Π' such that $\Pi' \rhd \Gamma \vdash \mathtt{N} : \tau$.*

Proof. As usual, this result can be proved in three steps by showing three intermediate properties: a substitution lemma, a normalization lemma and a abstraction lemma. The proof has to be slightly adapted in order to deal with the new distribution constructs. ☐

We have already discussed informally the depth of LAL terms. Here we introduce this concept more formally.

Definition 23 (depth)**.** *Given a term* $\mathtt{M}$*, the* depth of $\mathtt{M}$*, noted* $d(\mathtt{M})$*, is the maximal number of nested $\hat{\dagger}$ constructs in a path of the syntax tree of* $\mathtt{M}$*. Given a term* $\mathtt{M}$ *and one of its subterms* $\mathtt{N}$*,* $\mathtt{N}$ *is at depth* i in $\mathtt{M}$ *if it is in the scope of i nested $\hat{\dagger}$ in* $\mathtt{M}$*. Given a derivation* Π*, the* depth of Π*, noted* $d(\Pi)$*, is the maximal number of introduction rules for $\dagger$ ($\dagger I$ rules) in a branch of the derivation Π. A rule in a given type derivation is at depth i if it is preceded by i $\dagger I$ rules in a branch of* Π.

[4] For a given term $\mathtt{M}$ we can have several type derivations differing for their use of the structural rules (W) and (C).

$$\mathtt{dist}\ \hat{\S}\langle \mathtt{x}:\tau, \mathtt{y}:\sigma\rangle = \hat{\S}\langle \mathtt{M}_1, \mathtt{M}_2\rangle\ \mathtt{as}\ \mathtt{z} = \langle \hat{\S}\mathtt{x}, \hat{\S}\mathtt{y}\rangle\ \mathtt{in}\ \mathtt{N} \to \mathtt{N}[\langle \hat{\S}\mathtt{M}_1, \hat{\S}\mathtt{M}_2\rangle/\mathtt{z}] \qquad \text{(dis-1)}$$
$$\mathtt{dist}\ \hat{\S}\mathtt{inj}_{\mathtt{i}}^{\tau\oplus\tau'}(\mathtt{x}) = \hat{\S}\mathtt{inj}_{\mathtt{i}}^{\tau\oplus\tau'}(\mathtt{M})\ \mathtt{as}\ \mathtt{z} = \mathtt{inj}_{\mathtt{i}}^{\S\tau\oplus\S\tau'}(\hat{\S}\mathtt{x})\ \mathtt{in}\ \mathtt{N} \to \mathtt{N}[\mathtt{inj}_{\mathtt{i}}^{\S\tau\oplus\S\tau'}(\hat{\S}\mathtt{M})/\mathtt{z}] \qquad \text{(dis-2)}$$

Figure 4. Distributive reduction rules.

$$\frac{\Gamma \vdash \mathtt{M} : \S(\tau\otimes\sigma) \quad \Delta, \mathtt{z} : \S\tau\otimes\S\sigma \vdash \mathtt{N} : \tau'}{\Gamma, \Delta \vdash \mathtt{dist}\ \hat{\S}\langle \mathtt{x}:\tau, \mathtt{y}:\sigma\rangle = \mathtt{M}\ \mathtt{as}\ \mathtt{z} = \langle \hat{\S}\mathtt{x}, \hat{\S}\mathtt{y}\rangle\ \mathtt{in}\ \mathtt{N} : \tau'}\ (\mathtt{d}\otimes)$$

$$\frac{\Gamma \vdash \mathtt{M} : \S(\tau\oplus\sigma) \quad \Delta, \mathtt{z} : \S\tau\oplus\S\sigma \vdash \mathtt{N} : \tau'}{\Gamma, \Delta \vdash \mathtt{dist}\ \hat{\S}\mathtt{inj}_{\mathtt{i}}^{\tau\oplus\sigma}(\mathtt{x}) = \mathtt{M}\ \mathtt{as}\ \mathtt{z} = \mathtt{inj}_{\mathtt{i}}^{\S\tau\oplus\S\sigma}(\hat{\S}\mathtt{x})\ \mathtt{in}\ \mathtt{N} : \tau'}\ (\mathtt{d}\oplus)$$

Figure 5. Distributive typing rules for LALC

Notice that the depth of a term coincides with the depth of its typing derivations. In other words, if $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$ then $d(\mathtt{M}) = d(\Pi)$. In the following we will be interested in reductions that occur at a particular depth i, in such cases we will use the notation $\to_i$. Similarly to LAL, in the extended calculus the depth of a term cannot increase in the reduction.

Lemma 24 (Depth preservation)**.** *Given two terms* M *and* N*, if* $\mathtt{M} \to^* \mathtt{N}$ *then* $d(\mathtt{N}) \leq d(\mathtt{M})$.

Proof. By a case analysis of the reduction rules. □

6.2 Depth-by-Depth Reduction is not Enough

We want now to show that the depth-by-depth reduction is not enough to evaluate terms to normal form. Let us first introduce the notion of potential redex.

Definition 25 (Potential redexes at depth i)**.** *A* potential redex *in* M *is a subterm whose outermost construct is either a destructor or a distribution. Given a type derivation* $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$*, we denote by* $\mathcal{E}_i(\Pi)$ *the number of* elimination *rules,* $(\mathtt{d}\oplus)$ *and* $(\mathtt{d}\otimes)$ *rules that are at depth* i *in* Π*. An* actual redex *in* M *is a potential redex that can be reduced at the outermost level by applying either a beta rule, an exponential rule or a distribution rule. A* stuck redex *in* M *is a potential redex that is not an actual redex.*

Notice that $\mathcal{E}_i(\Pi)$ is exactly equal to the number of potential redexes at depth i in M. Indeed, each elimination or distribution rule corresponds to the introduction of exactly one destructor or distribution construct in the typed term and conversely.

Fact 26 (From potential redex of depth i to actual redex of depth $i-1$)**.** *A reduction* $\mathtt{M} \to_i \mathtt{N}$ *at depth* i *can turn a stuck redex at depth* $i-1$ *in an actual redex at depth* $i-1$.

We show that this fact holds by providing an illustrating example. Consider a term M of depth i having the following subterm $\mathtt{M}_1$ occurring at depth $i-1$:

$$\mathtt{dist}\ \hat{\S}\langle \mathtt{x}:\tau, \mathtt{y}:\sigma\rangle = \hat{\S}(\lambda \mathtt{u}:\tau'.\langle \mathtt{u},\mathtt{v}\rangle)\mathtt{w}\ \mathtt{as}\ \mathtt{z} = \langle \hat{\S}\mathtt{x}, \hat{\S}\mathtt{y}\rangle\ \mathtt{in}\ \mathtt{M}_2.$$

This subterm is a stuck redex as it is a potential redex (indeed a distribution) that is not an actual redex as the term $(\lambda\mathtt{u}:\tau'.\langle \mathtt{u},\mathtt{v}\rangle)\mathtt{w}$ is not a constructor. Indeed, this term needs to be evaluated first in order to turn $\mathtt{M}_1$ in an actual redex. Notice that this term $(\lambda\mathtt{u}:\tau'.\langle \mathtt{u},\mathtt{v}\rangle)\mathtt{w}$ is of depth i in M as it is located under an extra $\hat{\S}$ construct in $\mathtt{M}_1$. Consequently, in order to reduce the distribution in $\mathtt{M}_1$ we must perform a reduction $\mathtt{M} \to_i \mathtt{N}$ where N is the term obtained from M by substituting $\mathtt{M}_1$ with the term $\mathtt{N}_1$ below:

$$\mathtt{dist}\ \hat{\S}\langle \mathtt{x}:\tau, \mathtt{y}:\sigma\rangle = \hat{\S}\langle \mathtt{w},\mathtt{v}\rangle\ \mathtt{as}\ \mathtt{z} = \langle \hat{\S}\mathtt{x}, \hat{\S}\mathtt{y}\rangle\ \mathtt{in}\ \mathtt{M}_2.$$

$\mathtt{N}_1$ is a potential redex of depth $i-1$ in N (no enclosing †-box has been changed wrt to the initial term) and is not a stuck redex since it reduces to $\mathtt{M}_2[\langle \hat{\S}\mathtt{w}, \hat{\S}\mathtt{v}\rangle/\mathtt{z}]$.

This example shows that the depth-by-depth reduction strategy is not enough to reach a normal form. So we cannot use this strategy to prove the polynomial time soundness of LALC extended with distributions. Indeed, we need a reduction strategy that can round trip on the depth and reach in this way normal forms. In the following section we will prove the polynomial time soundness by using a global argument that provides an upper bound on the number and size of subterms generated in any reduction. This will help us in showing that each reduction has a polynomially bounded length.

It is also worth noticing that in the reduction of $\mathtt{N}_1$ above the constructor $\langle\cdot,\cdot\rangle$ for the product passes from depth i (from $\hat{\S}\langle \mathtt{w},\mathtt{v}\rangle$) to depth $i-1$ (to $\mathtt{M}_2[\langle \hat{\S}\mathtt{w}, \hat{\S}\mathtt{v}\rangle/\mathtt{z}]$ where z is at depth 0 in $\mathtt{M}_2$). Nevertheless, this is achieved by erasing the distribute in $\mathtt{N}_1$ that is also at depth $i-1$. So the number of syntax tree nodes at depth $i-1$ does not changes. This property will also be useful in proving the soundness in the next section.

6.3 Soundness of the Extension

In this section, we show that the well-typed terms of our language can be reduced in polynomial time by a Turing Machine.

As we discussed in the previous section, the usual proof for LALC based on bounding the length of the depth-by-depth reduction is not enough to reach normal forms in presence of distributions. Indeed, we can have a *stuck redex* at a given depth i that can be turned in an *actual redex* by reductions at depth $i+1$. However, it is important to stress that a reduction at depth $i+1$ cannot create new *potential redexes* but only turning a stuck redex in an actual one. This suggests that the complexity of the extended system is the same as the one of LALC but that we also need a more global argument to prove it.

As discussed in Section 3, the proof for LALC relies on the following three properties:

1. reductions cannot increase the depth of a term,
2. a beta reduction at depth i decreases the size of the term at depth i and cannot increase the size of the term at other depths,
3. any sequence of exponential reduction at depth i can only square the size of the term at every depth j greater than i.

These properties still holds for LALC extended with distributions, so we can use them to provide a global argument giving a polynomial bound on the number of potential redexes that can be generated in any possible reduction (Corollary 32). This combined with a lexicographic argument on the decrease on the number of potential redexes (Lemma 36) will give the polynomial time soundness.

Preliminary notations We write $\mathtt{M} \rightarrow_{\mathtt{c}} \mathtt{N}$ (resp. $\mathtt{M} \rightarrow_{\mathtt{nc}} \mathtt{N}$) to stress that $\mathtt{M} \rightarrow \mathtt{N}$ and that the reduction uses (resp. does not use) a commutation rule (com-n). Similarly, we write $\mathtt{M} \rightarrow_{\mathtt{k},i} \mathtt{N}$, $\mathtt{k} \in \{\mathtt{c}, \mathtt{nc}\}$, to stress that $\mathtt{M} \rightarrow_{\mathtt{k}} \mathtt{N}$ with a reduction at depth i.

While subject reduction only claims the existence of a type derivation Π', the proof gives us a concrete way to build Π' starting from Π. Thanks to this we can lift our reasoning from terms to type derivations. Indeed every reduction in terms corresponds to some transformation on the type derivation. In general we will write $\Sigma : \Pi \rhd \mathtt{M}\, \mathcal{R}^*\, \Pi' \rhd \mathtt{N}$, with $\mathcal{R} \in \{\rightarrow, \rightarrow_i, \rightarrow_{\mathtt{c}}, \rightarrow_{\mathtt{nc}}, \rightarrow_{\mathtt{c},i}, \rightarrow_{\mathtt{nc},i}\}$, when we want to explicitly give a name Σ to the reduction and the corresponding type derivation Π' obtained by reducing $\mathtt{M}$ to $\mathtt{N}$ wrt the reflexive and transitive closure of $\mathcal{R}$ starting with the type derivation Π. This will be useful when discussing about the structural rules—contraction and weakening—that do not have a corresponding syntactic construct in the language. So, they can only be seen in the typing derivation.

Length and size We also need to clarify the notions of length of a derivation and size of a term (or of a typing derivation). The reduction name Σ is useful when we want to deal with reduction length: $|\Sigma|$ will denote the length of the reduction (i.e. number of applications of $\mathcal{R}$), while $|\Sigma|^{\mathtt{c}}$ (respectively $|\Sigma|^{\mathtt{nc}}$) will denote the number of commutation rules (resp. rules that are not commutation rules) in Σ. Trivially, $|\Sigma| = |\Sigma|^{\mathtt{c}} + |\Sigma|^{\mathtt{nc}}$ as each reduction rule is either a commutation or not. The size of a term $\mathtt{M}$ is the number of symbols in the syntax tree that are at any depth. The size of a typing derivation Π is the total number of rules in it. Straightforwardly, the size of a term is bounded by the size of its typing derivations:

Lemma 27 (Size)**.** *If $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$ then $|\mathtt{M}| \leq |\Pi|$.*

Proof. Any constructor (destructor resp.) of the language corresponds to exactly one introduction (elimination resp.) rule in the type system. □

Polynomial size reducts The key property that we will use to prove the soundness with a global argument is that the size of each intermediate type derivation obtained in a reduction is bounded polynomially by the size of the initial type derivation. To express this fact we need to refer to specific parts of a given term or derivation that are at a particular depth.

Definition 28 (size at depth i)**.** *Given a term $\mathtt{M}$ we denote by $|\mathtt{M}|_i$ the number of symbols of $\mathtt{M}$ that are at depth i in $\mathtt{M}$. Trivially, the following equality holds $|\mathtt{M}| = \sum_{i=0}^{d(\mathtt{M})} |\mathtt{M}|_i$. In the same way, we denote by $|\Pi|_i$ the number of rules that are at depth i and the equality $|\Pi| = \sum_{i=0}^{d(\Pi)} |\Pi|_i$ also holds.*

We also need to count the contraction rules at each depth.

Definition 29 (contractions at depth i)**.** *Given a type derivation $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$, we denote by $\mathcal{C}_i(\Pi)$ the number of* contraction rules *that are at depth i in Π.*

The notion of *potential* for a type derivation Π is introduced in order to bound the size of typing derivations in a reduction.

Definition 30. *Given a typing derivation $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$ we define its* weight at depth i*, denoted $\mathcal{W}_i(\Pi)$, by:*

$$\mathcal{W}_0(\Pi) = \mathcal{C}_0(\Pi)$$
$$\mathcal{W}_{i+1}(\Pi) = \prod_{j=0}^{i} (\mathcal{C}_j(\Pi) + 1)^{2^{i-j}} \cdot \mathcal{C}_{i+1}(\Pi)$$

The potential *of Π, denoted $\mathcal{P}(\Pi)$ is defined as:*

$$\mathcal{P}(\Pi) = \sum_{i=0}^{d(\Pi)} (\mathcal{W}_{i-1}(\Pi) + 1) \cdot |\Pi|_i$$

with the convention that $\mathcal{W}_{-1}(\Pi) = 0$.

We can now formulate the key property of the potential.

Lemma 31 (Polynomial size)**.** *Consider a reduction $\Sigma : \Pi \rhd \mathtt{M} \rightarrow^* \Pi' \rhd \mathtt{N}$. Then, there is a polynomial $\mathcal{P}(\Pi)$ in $|\Pi|$ such that the following hold:*

- $|\Pi'| \leq P(\Pi)$,
- $P(\Pi) = \mathcal{O}(|\Pi|^{2^{d(\Pi)}+2})$.

Proof. By induction on the depth, using an upper bound on the maximal number of contraction rules that might exist at each depth in a reduct. □

As a corollary, we obtain an upper bound on the number of potential redexes at each depth:

Corollary 32 (Polynomial number of potential redexes at any depth)**.** *Consider a reduction $\Sigma : \Pi \rhd \mathtt{M} \rightarrow^* \Pi' \rhd \mathtt{N}$. Then, for any $i \geq 0$, we have:*

$$\mathcal{E}_i(\Pi') \leq \mathcal{P}(\Pi).$$

Proof. By Lemma 31 as $\mathcal{E}_i(\Pi') \leq |\Pi'| \leq \mathcal{P}(\Pi)$. □

Polynomial length reductions Now we are ready to show that the reduction length of a typed term is polynomially bounded by the size of its typing derivation. We proceed in two steps. First we show that the number of non commuting reduction steps is bounded polynomially in the size (and exponentially in the depth - that is fixed) of a term using a lexicographic decrease on the number of potential redexes (and the fact that they are bounded by the potential by Corollary 32). In a second step, we show that the number of commuting reduction steps is bounded polynomially in the size using a rewriting argument based on the structure of such rules.

Lemma 33. *Consider a reduction $\Sigma : \Pi \rhd \mathtt{M} \rightarrow_{\mathtt{k},i} \Pi' \rhd \mathtt{N}$, with $\mathtt{k} \in \{\mathtt{c}, \mathtt{nc}\}$.*

(i) For each $j < i$, we have $\mathcal{E}_j(\Pi') = \mathcal{E}_j(\Pi)$.
(ii) Moreover,
1. *if $\mathtt{k} = \mathtt{c}$ then we have: for each $j \geq i$, $\mathcal{E}_j(\Pi') = \mathcal{E}_j(\Pi)$;*
2. *if $\mathtt{k} = \mathtt{nc}$ then we have: $\mathcal{E}_i(\Pi') < \mathcal{E}_i(\Pi)$.*

Proof. By a case analysis on the reduction rules. □

The above lemma tells us that a commutative reduction does not change potential redexes at all while non commutative reductions at depth i preserve potential redexes at depth strictly smaller than i and decrease by one the number of potential redexes at depth i. Of course, in this latter case, the number of potential redexes at depth strictly higher than i may increase.

Definition 34 (Strength)**.** *Given a type derivation $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$, the* strength *of Π, noted $s(\Pi)$, is a $d(\Pi) + 1$ tuple defined by:*

$$s(\Pi) = \langle \mathcal{E}_0(\Pi), \mathcal{E}_1(\Pi), \ldots, \mathcal{E}_{d(\Pi)}(\Pi) \rangle.$$

Corollary 35. *Consider a reduction $\Sigma : \Pi \rhd \mathtt{M} \rightarrow_{\mathtt{k},i} \Pi' \rhd \mathtt{N}$, with $\mathtt{k} \in \{\mathtt{c}, \mathtt{nc}\}$.*

1. *if $\mathtt{k} = \mathtt{c}$ then we have $s(\Pi) =_{\mathtt{lex}} s(\Pi')$;*
2. *if $\mathtt{k} = \mathtt{nc}$ then we have: $s(\Pi) >_{\mathtt{lex}} s(\Pi')$.*

where $>_{\mathtt{lex}}$ is the lexicographic strict order induced by $>$ on tuples.

Proof. (1) is a consequence of Lemma 33(i) and (ii.1) together with the fact that depth does not increase in a reduction, by Lemma 24, while (2) is a consequence of Lemma 33(i), (ii.2) and Lemma 24. □

Now we take benefits of the above Corollary together with the previous upper bound on size of reduced proof in order to infer an upper bound on the length of non-commutative reductions.

Lemma 36. *Consider a reduction* $\Sigma : \Pi \rhd \mathtt{M} \to^* \Pi' \rhd \mathtt{N}$*. Then, we have:*

$$|\Sigma|^{\mathtt{nc}} \leq (\mathcal{P}(\Pi))^{d(\Pi)+1}.$$

Proof. Let $t(\Pi) = \langle \mathcal{P}(\Pi), \ldots, \mathcal{P}(\Pi) \rangle$ be a tuple with $d(\Pi) + 1$ times elements. By several applications of Corollary 35 and Corollary 32, for any reduction of the shape $\Pi_1 \rhd \mathtt{M}_1 \to^*_{\mathtt{c}} \Pi_2 \rhd \mathtt{M}_2 \to_{\mathtt{nc}} \Pi_3 \rhd \mathtt{M}_3 \to^*_{\mathtt{c}} \Pi_4 \rhd \mathtt{M}_4$, the following holds:

$$\begin{array}{ccccccc} t(\Pi) & & t(\Pi) & & t(\Pi) & & t(\Pi) \\ \geq & & \geq & & \geq & & \geq \\ s(\Pi_1) & =_{\mathtt{lex}} & s(\Pi_2) & >_{\mathtt{lex}} & s(\Pi_3) & =_{\mathtt{lex}} & s(\Pi_4), \end{array}$$

where $\geq$ is the pointwise partial order induced by $\geq$ on tuples.

Consequently, there can be no more than $(\mathcal{P}(\Pi))^{d(\Pi)+1}$ strict lexicographic decreases in a reduction. This provides us a bound on the number of reduction rules that are not commutation rules. □

It is worth noticing that the above lemma diverges from the classical soundness proof for LAL. In particular, we combine a global argument given by the potential $\mathcal{P}(\Pi)$ of the type derivation Π with the lexicographic order that provides a local argument. In this way we have an argument that is independent from the reduction strategy. While this approach has the consequence that the bound we provide is looser than the usual one, the difference is just in a small exponential constant that leaves the bound polynomial once the depth is fixed. A tighter bound—similar to the usual one—can be obtained by considering instead a specific reduction strategy where lower depth redexes are reduced with higher priority.

As usual, the length of reductions only involving commutation is bounded quadratically in the size of the initial type derivation using a term rewriting argument.

Lemma 37. *For each reduction* $\Sigma : \Pi \rhd \mathtt{M} \to^*_{\mathtt{c}} \Pi' \rhd \mathtt{N}$*, we have* $|\Sigma| \leq |\Pi|^2$.

Proof. Commuting rules form a rewriting system that terminates in quadratic time. That is, each commuting rule can move a subterm around but every time it is applied the destructor providing the actual redex decreases in outermost-rightmost order in the syntactic tree of the term. By Lemma 33, we have that the number of potential redexes is preserved by commutation rules, so each of them can just be moved along that order. For example, consider the rule (com-1) $(\mathtt{let}\ \hat{\dagger}\mathtt{x} : \dagger\tau = \mathtt{N}\ \mathtt{in}\ \mathtt{M})\mathtt{L} \to \mathtt{let}\ \hat{\dagger}\mathtt{x} : \dagger\tau = \mathtt{N}\ \mathtt{in}\ (\mathtt{ML})$ which can be represented by the following rule on syntactic trees:

$$\begin{tikzcd}
 & & @ \arrow[dl] \arrow[dr] & & & & \mathtt{let}\hat{\dagger} \arrow[dl] \arrow[dr] & & \\
 & \mathtt{let}\hat{\dagger} \arrow[dl] \arrow[dr] & & \mathtt{L} & \to & \mathtt{N} & & @ \arrow[dl] \arrow[dr] & \\
\mathtt{N} & & \mathtt{M} & & & & \mathtt{M} & & \mathtt{L}
\end{tikzcd}$$

As expected, one can see that the let-$\hat{\dagger}$ destructor moves in outermost-rightmost order in the syntactic tree to which this rule is applied. So are the other commutation rules. By Lemma 27, the number of potential redexes is bounded by $|\Pi|$ and so we have the bound. □

Now we are ready to show that any reduction has a polynomially bounded length:

Lemma 38 (Polynomially bounded reduction length)**.** *Consider a reduction* $\sigma : \Pi \rhd \mathtt{M} \to^* \Pi' \rhd \mathtt{N}$*. Then, we have:*

$$|\Sigma| \leq \mathcal{P}(\Pi)^{d(\Pi)+1} \cdot (\mathcal{P}(\Pi)^2 + 1).$$

Proof. The inequality follows by combining Lemma 37, Lemma 31 and Lemma 36. □

FPtime soundness We can now show the soundness properties of our type system:

Theorem 39 (Polynomial Time Soundness)**.** *Consider a type derivation* $\Pi \rhd \Gamma \vdash \mathtt{M} : \tau$*. Then,* $\mathtt{M}$ *can be reduced to normal form by a Turing Machine working in time polynomial in* $|\mathtt{M}|$ *with exponent proportional to* $d(\mathtt{M})$.

Proof. By definition of depth, the equality $d(\Pi) = d(\mathtt{M})$ holds. Moreover, for any typable term $\mathtt{M}$, there is at least one *normal* type derivation Π (with no superfluous contraction rule and weakening rule). For such a type derivation, $|\Pi| = \mathcal{O}(|\mathtt{M}|)$ holds. By Lemma 31, the potential of a derivation is bounded by a polynomial in the size of Π with exponent proportional to $d(\Pi)$. By Lemma 38, each typable term $\mathtt{M}$ has at most a polynomially bounded number of reductions in $|\mathtt{M}|$. By Corollary 31 the size of every intermediate term in the reduction is bounded polynomially in $|\mathtt{M}|$. As the reduction of LAL can be easily implemented by a Turing Machine in quadratic time (see [40]), the conclusion follows. □

7. Coalgebra Examples

We can now improve Lemma 21 and obtain the following analog of Lemma 14.

Lemma 40. *All the functors built using the following signature right-distribute over* $\S$*:*

$$F(X) ::= \mathbf{1} \mid X \mid A \mid \S F(X) \mid F(X) \otimes F(X) \mid F(X) \oplus F(X),$$

provided that A *is a closed type for which there exists a closed term of type* $\S A \multimap A$.

Proof. By induction on $F(X)$. □

Similarly to Example 8 we would like to consider the case of streams of natural numbers. Unfortunately, Lemma 40 is not enough to show that the functor $F(X) = \mathbb{N} \otimes X$ distributes to the right. The problem is that we do not have a coercion $\S\mathbb{N} \multimap \mathbb{N}$, but only the converse one. Nevertheless, we can consider streams of booleans (or more in general of every finite type $\mathbf{1} \oplus \cdots \oplus \mathbf{1}$). Let us consider the functor defined on types as $F(X) = \mathbb{B}_2 \otimes X$ and on terms as:

$$\lambda\mathtt{f} : X \multimap Y.\lambda\mathtt{x} : \mathbb{B}_2 \otimes X.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = \mathtt{x}\ \mathtt{in}\ \langle \mathtt{x}_1, \mathtt{f}\ \mathtt{x}_2 \rangle.$$

This functor right-distributes by Lemma 40. Let $\mathbb{B}_2^\omega = \nu X.F(X)$. Theorem 20 ensures that $(\mathbb{B}_2^\omega, \mathtt{out}_{\mathbb{B}_2^\omega})$ is a weak final coalgebra. Let us use this property to define a constant stream of 1 (as a boolean). Similarly to what we did in Example 8 this can be defined by considering the function:

$$g = \lambda\mathtt{x} : \mathbf{1}.\mathtt{let}\ () = \mathtt{x}\ \mathtt{in}\ \langle 1, () \rangle.$$

By Theorem 20 we can then define $\mathtt{ones} = \mathtt{unfold}\,\mathbf{1}\,\hat{!}g\,\hat{\S}()$. Similarly to what happens in System F we can define the usual operations on streams as:

$$\begin{aligned} \mathtt{head} &= \lambda\mathtt{x} : \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{x}_1, \\ \mathtt{tail} &= \lambda\mathtt{x} : \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2 \rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x})\ \mathtt{in}\ \mathtt{x}_2. \end{aligned}$$

Unfortunately, using these operations is often inconvenient in presence of linearity, and it is more convenient to use directly the coalgebra structure provided by $\mathtt{out}_{\mathbb{B}_2^\omega}$. Consider for example the operation that extracts from a stream of booleans the elements in even position – we have seen a similar operation encoded in System F in Example 8. We can define this operation by using the function:

$$g = \lambda \mathtt{x} : \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2\rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x})\ \mathtt{in}\\ \mathtt{let}\ \langle \mathtt{x}_{21}, \mathtt{x}_{22}\rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x}_2)\ \mathtt{in}\ \langle \mathtt{x}_1, \mathtt{x}_{22}\rangle.$$

This function has type $g : \mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes \mathbb{B}_2^\omega$. So, by Theorem 20 $\mathtt{even} = \mathtt{unfold}_{\mathbb{B}_2^\omega}\ \mathbb{B}_2^\omega\ \hat{!}g$. Another interesting example is a function that merges two streams. This can be defined by the term:

$$g = \lambda \mathtt{x} : \mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2\rangle = \mathtt{x}\ \mathtt{in}\\ \mathtt{let}\ \langle \mathtt{x}_{11}, \mathtt{x}_{12}\rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{x}_1)\ \mathtt{in}\ \langle \mathtt{x}_{11}, \langle \mathtt{x}_2, \mathtt{x}_{12}\rangle\rangle.$$

This function has type $g : \mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega \multimap \mathbb{B}_2 \otimes (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega)$. So, by Theorem 20 again, $\mathtt{merge} = \mathtt{unfold}_{\mathbb{B}_2^\omega}\ (\mathbb{B}_2^\omega \otimes \mathbb{B}_2^\omega)\ \hat{!}g$.

We can combine algebra examples and coalgebra ones. For instance, we can write a standard inductive function take that for a given n returns the first n elements of a stream as a string. This can be obtained by the function:

$$g = \lambda \mathtt{x} : \mathbf{1} \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*).\mathtt{case}\ \mathtt{x}\ \mathtt{of}\\ \{\mathtt{inj}_0(\mathtt{z}) \to \lambda \mathtt{y} : \mathbb{B}_2^\omega.\mathtt{nil} \mid \mathtt{inj}_1(\mathtt{z}) \to\\ \lambda \mathtt{y} : \mathbb{B}_2^\omega.\mathtt{let}\ \langle \mathtt{y}_1, \mathtt{y}_2\rangle = (\mathtt{out}_{\mathbb{B}_2^\omega}\ \mathtt{y})\ \mathtt{in}\ \mathtt{cons}(\mathtt{y}_1, \mathtt{z}\,\mathtt{y}_2)\}.$$

This function has type $g : \mathbf{1} \oplus (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*) \multimap (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*)$. So, by Theorem 13, $\mathtt{take} = \mathtt{fold}_{\mathbb{B}_2^\omega}\ (\mathbb{B}_2^\omega \multimap \mathbb{B}_2^*)\ \hat{!}g$.

Even if we cannot define a stream of the standard inductive natural numbers, we can have a stream of extended natural numbers. Let us define the latter first. Consider the functor $F(X) = \mathbf{1} \oplus X$. By Lemma 40, we have that F right-distributes over §, and so by Theorem 20 we have that $(\overline{\mathbb{N}}, \mathtt{out}_{\overline{\mathbb{N}}})$ is a weakly-final F-coalgebra under §, where $\overline{\mathbb{N}}$ denotes $\nu X.F(X)$. The inhabitants of the type $\overline{\mathbb{N}}$ correspond to the natural numbers extended with a limit element ∞. We can think about $\mathtt{out}_{\overline{\mathbb{N}}}$ as a predecessor function mapping 0 to (), n to $n-1$ and ∞ to ∞. We can define the addition of two extended natural numbers by considering the term:

$$g = \lambda \mathtt{x} : \overline{\mathbb{N}} \otimes \overline{\mathbb{N}}.\mathtt{let}\ \langle \mathtt{x}_1, \mathtt{x}_2\rangle = \mathtt{x}\ \mathtt{in}\ \Big(\mathtt{case}\ (\mathtt{out}_{\overline{\mathbb{N}}}\ \mathtt{x}_1)\ \mathtt{of}\\ \Big\{\mathtt{inj}_0(\mathtt{z}) \to \mathtt{case}\ (\mathtt{out}_{\overline{\mathbb{N}}}\ \mathtt{x}_2)\ \mathtt{of}\ \{\mathtt{inj}_0(\mathtt{z}') \to \mathtt{inj}_0(())\\ \mid \mathtt{inj}_1(\mathtt{z}') \to \mathtt{inj}_1(\langle \mathtt{z}, \mathtt{z}'\rangle)\}\\ \mid \mathtt{inj}_1(\mathtt{z}) \to \mathtt{inj}_1(\langle \mathtt{z}, \mathtt{x}_2\rangle)\Big\}\Big).$$

This function has type $g : \overline{\mathbb{N}} \otimes \overline{\mathbb{N}} \multimap \mathbf{1} \otimes (\overline{\mathbb{N}} \otimes \overline{\mathbb{N}})$. So, by Theorem 20 $\mathtt{add} = \mathtt{unfold}_{\overline{\mathbb{N}}}\ (\overline{\mathbb{N}} \otimes \overline{\mathbb{N}})\ \hat{!}g$. For the extended natural numbers, we have a term $\mathtt{coer}_{\overline{\mathbb{N}}} : \S\overline{\mathbb{N}} \multimap \overline{\mathbb{N}}$, this is given by Theorem 20 as $\mathtt{coer}_{\overline{\mathbb{N}}} = \mathtt{unfold}_{\overline{\mathbb{N}}}\ \overline{\mathbb{N}}\ \hat{!}\mathtt{out}_{\overline{\mathbb{N}}}$. Thanks to this coercion we can define streams of extended natural numbers.

One would like also to consider infinite trees labelled with elements in A. This could be defined using the functor $F(X) = A \otimes (X \otimes X)$. Unfortunately, the term defining this functor :

$$\lambda \mathtt{f}.\lambda \mathtt{x}.\mathtt{let}\ \langle \mathtt{y}, \mathtt{z}\rangle \ = \ \mathtt{x}\ \mathtt{inlet}\ \langle \mathtt{u}, \mathtt{v}\rangle \ = \ \mathtt{z}\ \mathtt{in}\ \langle \mathtt{y}, \langle \mathtt{f}\,\mathtt{u}, \mathtt{f}\,\mathtt{v}\rangle\rangle$$

cannot be assigned a linear type because of the duplication of the variable f.

8. Related works

Infinite data structures in ICC Several works have studied properties related to ICC in the context of infinite data structures.

Burrell et al. [8] proposed Pola as a programming language characterizing FPTIME. The design idea of Pola comes from safe recursion on notation [7] and interestingly, Pola permits the programmer to write polynomial time functional programs working both on inductive and coinductive data types. This work is close to ours but there are two main differences. First, the use of safe recursion on notation and the use of linear types are quite different and produce two different programming methodologies. Second, we studied how to define algebras and coalgebras in the language while Pola takes inductive and coinductive types as primitive.

Leivant and Ramyaa [29] have studied a framework based on equational programs that is useful to reason about programs over inductive and coinductive types. They used such a framework to obtain an ICC characterization of primitive corecurrence (a weak form of productivity). Ramyaa and Leivant [36] also shows that a ramified version of corecurrence gives an ICC characterization of the class of functions over streams working in logarithmic space. Leivant and Ramyaa [30] further studied the correspondence between ramified recurrence and ramified corecurrence. In our work, in contrast we focus on the restrictions directly provided by Light Affine Logic. It can be an interesting future direction to study whether one can express some form of ramified corecurrence in LAL, along the lines of what has been done for ramified recursion [34].

Using an approach based on quasi-interpretation, in [12, 13] we have studied space upper bounds properties and input-output properties of programs working on streams. Using a similar approach Férée et al. [10] showed that interpretations can be used on stream programs also to characterize type 2 polynomial time functions by providing a characterization of the class of the Basic Feasible Functionals of Cook and Urquhart [9].

Dal Lago et al.[3] have developed a technique inspired by quasi-interpretations to study the complexity of higher-order programs. In their framework infinite data are first class citizens in the form of higher order functions. However, they do not consider programs working on declarative infinite data structures as streams.

Expressivity of Light Logics Several works have studied ways to better understand and improve the expressivity of light logics, and more in general of ICC systems. Unfortunately, we do not yet have a general method for comparing different systems and improvements.

Hofmann [25] provides a survey of the different approaches to ICC. In this survey he also discusses the expressivity and the limitations of the different light logics in the encoding of traditional algorithms. Murawski and Ong [34] and, more recently, Roversi and Vercelli [38] have studied the expressivity of light logics by comparing them with the one provided by ramified recursion. In particular, they provide different embeddings of (fragments of) ramified recursion in LLL and its extensions. Dal Lago et al.[28] have studied and compared the expressivity of different light logics obtained by adding or removing several type constructions, like tensor product, polymorphism and type fixpoints. Our work follows in spirit the same approach focusing on the encoding of (co)algebras.

Gaboardi et al. [17] have studied the expressivity of the different light logics by designing embeddings from the light logics to Linear Logic by Level [4], another logic providing a characterization of polynomial time but based on more general principles. Interestingly, in Linear Logic by Level the § modality commutes with all the other type constructions. It would be interesting to study what is the expressivity of this logic with respect to the encoding of algebras and coalgebras. Baillot et al. [6] have approached the problem of improving the expressivity of LAL by designing a programming language with recursion and pattern matching around it. We drawn inspiration from their work but instead of adding extra constructions we focus on the constructions that can be defined in LAL itself.

9. Conclusion and Future Works

We have studied the definability of algebras and coalgebras in the LALC along the lines of the encoding of algebras and coalgebras in the polymorphic lambda calculus. By extending the calculus with distributive rules for the modality § we are able to program several natural examples over infinite data structures.

It is well-known that the encoding of algebras and coalgebras in System F is rather limited and it also does not behave well from the type theory point of view [18]. For this reason, several works have studied how to directly extend the polymorphic lambda calculus with different notions of algebras and coalgebras that behave better, e.g. [31]. We expect that a similar approach can be also followed for LALC: it would be interesting to understand how the different extensions studied in the literature can fit the LALC setting. We expect that also other extensions would require the terms for distribution we introduced in this work.

We have approached the study of algebras and coalgebras in a term language for Light Affine Logic (LALC). There are also other light logics for which we could ask the same question. Obviously, an encoding similar to the one we studied here can be used for Elementary Linear Logic. It would instead be more interesting to understand whether there is an encoding for Soft Linear Logic that allows a large class of coinductive data structures to be defined.

Acknowledgements

We thank Michele Pagani and Shin-ya Katsumata for some helpful comments on a preliminary version of this work. We also thank the anonymous reviewers for their close reading and helpful comments. Part of this work was done while Marco Gaboardi was visiting Harvards Center for Research on Computation and Society (CRCS). Romain Péchoux work was partially supported by ANR-14-CE25-0005 Elica: Expanding Logical Ideas for Complexity Analysis.

References

[1] A. Asperti. Light affine logic. In LICS ’98, pages 300–308. IEEE, 1998.
[2] A. Asperti and L. Roversi. Intuitionistic light affine logic. *ACM TOCL*, 3(1):137–175, 2002.
[3] P. Baillot and U. Dal Lago. Higher-Order Interpretations and Program Complexity. In *CSL’12*, volume 16, pages 62–76. LIPIcs, 2012.
[4] P. Baillot and D. Mazza. Linear logic by levels and bounded time complexity. *TCS*, 411(2):470–503, 2010.
[5] P. Baillot and K. Terui. Light types for polynomial time computation in lambda-calculus. In LICS ’04, pages 266–275. IEEE, 2004.
[6] P. Baillot, M. Gaboardi, and V. Mogbil. A polytime functional language from light linear logic. In *ESOP’10*, volume 6012 of *LNCS*. Springer, 2010.
[7] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the poly-time functions. *Comp. Complexity*, 2:97–110, 1992.
[8] M. J. Burrell, R. Cockett, and B. F. Redmond. Pola: a language for PTIME programming. In *LCC’09*, 2009.
[9] S. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. STOC ’89, pages 107–112. ACM, 1989.
[10] H. Férée, E. Hainry, M. Hoyrup, and R. Péchoux. Interpretation of stream programs: characterizing type 2 polynomial time complexity. In *ISAAC’10*, pages 291–303. Springer, 2010.
[11] P. Freyd. Structural polymorphism. *TCS*, 115(1):107–129, 1993.
[12] M. Gaboardi and R. Péchoux. Upper bounds on stream I/O using semantic interpretations. In *CSL ’09*, volume 5771 of *LNCS*, pages 271 – 286. Springer, 2009.
[13] M. Gaboardi and R. Péchoux. Global and local space properties of stream programs. In *FOPARA’09*, volume 6324 of *LNCS*, pages 51 – 66. Springer, 2010.
[14] M. Gaboardi and S. Ronchi Della Rocca. A soft type assignment system for λ-calculus. In *CSL ’07*, volume 4646 of *LNCS*, pages 253–267. Springer, 2007.
[15] M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. Soft linear logic and polynomial complexity classes. In *LSFA 2007*, volume 205 of *ENTCS*, pages 67–87. Elsevier.
[16] M. Gaboardi, J.-Y. Marion, and S. Ronchi Della Rocca. A logical account of PSPACE. In *POPL’08*. ACM, 2008.
[17] M. Gaboardi, L. Roversi, and L. Vercelli. A by-level analysis of Multiplicative Exponential Linear Logic. In *MFCS’09*, volume 5734 of *LNCS*, pages 344 – 355. Springer, 2009.
[18] H. Geuvers. Inductive and coinductive types with iteration and recursion. In *Types for Proofs and Programs*, pages 193–217. 1992.
[19] J. Girard. *The Blind Spot: Lectures on Logic*. European Mathematical Society, 2011. ISBN 9783037190883.
[20] J.-Y. Girard. Linear logic. *TCS*, 50:1–102, 1987.
[21] J.-Y. Girard. Light linear logic. *Information and Computation*, 143(2): 175–204, 1998.
[22] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge university press, 1989.
[23] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory and Computer Science*, volume 283, pages 140–57. LNCS, 1987.
[24] P. Hájek. Arithmetical hierarchy and complexity of computation. *TCS*, 8:227–237, 1979.
[25] M. Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(1):31–42, 2000.
[26] B. Jacobs and J. Rutten. A tutorial on (co) algebras and (co) induction. *EATCS*, 62:222–259, 1997.
[27] Y. Lafont. Soft linear logic and polynomial time. *TCS*, 318(1-2):163–180, 2004.
[28] U. D. Lago and P. Baillot. On light logics, uniform encodings and polynomial time. *MSCS’06*, 16(4):713–733, 2006.
[29] D. Leivant and R. Ramyaa. Implicit complexity for coinductive data: a characterization of corecurrence. In *DICE’12*, volume 75 of *EPTCS*, pages 1–14, 2012.
[30] D. Leivant and R. Ramyaa. The computational contents of ramified corecurrence. In *FOSSACS*, 2015. To appear.
[31] R. Matthes. Monotone (co)inductive types and positive fixed-point types. *ITA*, 33(4/5):309–328, 1999.
[32] F. Maurel. Nondeterministic light logics and NP-time. In *TLCA ’03*, volume 2701 of *LNCS*, pages 241–255. Springer, 2003.
[33] D. Mazza. Non-uniform polytime computation in the infinitary affine lambda-calculus. In *ICALP’14*, pages 305–317, 2014.
[34] A. S. Murawski and C. L. Ong. On an interpretation of safe recursion in light affine logic. *TCS*, 318(1-2):197–223, 2004.
[35] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
[36] R. Ramyaa and D. Leivant. Ramified corecurrence and logspace. *ENTCS*, 276(0):247 – 261, 2011. MFPS’11.
[37] J. C. Reynolds and G. Plotkin. On functors expressible in the polymorphic typed lambda calculus. *Information and Computation*, 105 (1):1–29, 1993.
[38] L. Roversi and L. Vercelli. Safe recursion on notation into a light logic by levels. In *DICE’10*, pages 63–77, 2010.
[39] U. Schöpp. Stratified bounded affine logic for logarithmic space. In LICS ’07, pages 411–420, Washington, DC, USA, 2007. IEEE.
[40] K. Terui. Light affine lambda calculus and polytime strong normalization. In LICS ’01, pages 209–220. IEEE, 2001.
[41] P. Wadler. Recursive types for free! Technical report, University of Glasgow, 1990.
[42] G. C. Wraith. A note on categorical datatypes. In *CTCS*, volume 389 of *LNCS*, pages 118–127. Springer, 1993.