A Soft Type Assignment System for λ-Calculus

Marco Gaboardi and Simona Ronchi Della Rocca

Dipartimento di Informatica
Università degli Studi di Torino
Corso Svizzera 185, 10149 Torino, Italy
gaboardi,ronchi@di.unito.it

Abstract. Soft Linear Logic (SLL) is a subsystem of second-order linear logic with restricted rules for exponentials, which is correct and complete for PTIME. We design a type assignment system for the λ-calculus (STA), which assigns to λ-terms as types (a proper subset of) SLL formulas, in such a way that typable terms inherit the good complexity properties of the logical system. Namely STA enjoys subject reduction and normalization, and it is correct and complete for PTIME and FPTIME.

1 Introduction

The light logics, Light Linear Logic (LLL) [1] and Soft Linear Logic (SLL) [2], were both introduced as logical counterpart of the polynomial complexity. Namely proofs of both logics normalize in a polynomial number of cut-elimination steps, if their depth is fixed, and moreover they are complete for FPTIME and PTIME respectively. So they can be used for the design of programming languages with an intrinsically polynomial computational bound. This can be done in a straightforward way by a complete decoration of the logical proofs, but for both these logics, since the presence of modalities, the resulting languages have a very complex syntactical structure, and they cannot be reasonably proposed for programming (an example of complete decoration of SLL is in [3]). A different approach is to fix as starting points:

1. The use of λ-calculus as an abstract paradigm of programming languages.
2. The use of types to characterize program properties.

In this line, the aim becomes the design of a type assignment system for λ-calculus, where types are formulae of a light logic, in such a way that the logical properties are inherited by the well typed terms. Then types can be used for checking, beside the usual notion of correctness, also the property of having polynomial complexity. A type assignment for λ-calculus correct and complete with respect to polynomial time computations has been designed by Baillot and Terui [4], using as types formulae of Light Affine Logic (LAL), a simplified version of LLL defined in [5,6]. The aim of this paper is to design a type assignment system for λ-calculus, enjoying the same properties, but using as types formulae

J. Duparc and T.A. Henzinger (Eds.): CSL 2007, LNCS 4646, pp. 253–267, 2007.

of SLL. The motivation for doing it is twofold. The design of λ-calculi with implicit complexity properties is a very new research line, and so exploring different approaches is necessary in order to compare them. In particular, SLL formulas look simpler than LAL ones, since they have just one modality while LAL has two modalities, and this could in principle give rise to a system easier to deal with. Moreover, SLL has been proved to be complete just for PTIME, while we want a type assignment system complete for FPTIME: so we want to explore at the same time if it is possible, when switching from formulae to types, to prove this further property.

We start from the original version of second order SLL with the only connectives $\multimap$ and ! and the quantifier $\forall$, given in sequent calculus style. The principal problem in designing the desired type assignment system is that, in general, in a modal logic setting, the good properties of proofs are not easily inherited by λ-terms. In particular, there is a mismatch between β-reduction in the λ-calculus and cut-elimination in logical systems, which makes it difficult both getting the subject reduction property and inheriting the complexity properties from the logic, as discussed in [4]. To solve this problem we exploit the fact that, in the decorated sequent calculus, there is a redundancy of proofs, in the sense that the same λ-term can arise from different proofs. So we propose a restricted system, called Soft Type Assignment (STA), where the set of derivations corresponds to a proper subset of proofs in the affine version of SLL. Types are a subset of SLL formulae, where, in the same spirit of [4], the modality ! is not allowed in the right hand of an arrow, and the application of some rules (in particular the *cut* rule) is restricted to linear types. STA enjoys subject reduction, and the set of typable terms characterizes PTIME. Moreover we prove that the language of typable terms is complete both for PTIME and FPTIME. The completeness we are speaking about is a functional completeness, in the sense that we prove (through a simulation of polynomial time Turing Machines) that all polynomial functions can be computed by terms typable in STA. The algorithmic expressivity of the language has been not studied here.

This paper is a first step: we are actually working on a natural deduction version of STA, and on the type inference problem. The type inference problem for STA seems undecidable, but we think it is possible to build decidable restrictions that do not lose the complexity completeness. Moreover a comparison with respect to DLAL, the type assignment system in [4], is in order. The two systems have incomparable typability power, in the sense that there exist terms typable in STA but not in DLAL and vice versa. STA has an easier treatment of contexts, and no notion of discharged assumptions. This, together with the fact that types in STA have just one modality, can help in designing a simpler natural deduction and a more efficient type inference algorithm.

The paper is organized as follows. In Section 2 SLL is introduced and a discussion is made about the problem of subject reduction. In Section 3 STA is defined, and the proof of subject reduction is given. Section 4 contains the results about the complexity bound: polynomial bound is proved in Subsection 4.1, and the completeness for both PTIME and FPTIME is proved in Subsection 4.2.

2 Soft Linear Logic and λ-Calculus

In [7] a decoration of SLL by terms of λ-calculus has been presented, as technical tool for studying the expressive power of the system. The decoration is presented in Table 1, where $\Gamma\#\Delta$ denotes the fact that the domain of contexts Γ and Δ are disjoint sets of variables. In such system, which we call SLL_λ, we have the failure of subject reduction. Let us give a detailed analysis of the problem. In a decorated sequent calculus system, β-reduction is the counterpart, in terms, of the *cut* rule of the logic. The *cut* rule in Table 1 can be split into three different rules, according to the shape of U, of the contexts and of the derivation:

$$\frac{\Gamma \vdash_l M : U \quad \Delta, x : U \vdash_l N : V \quad U \text{ not modal}}{\Gamma, \Delta \vdash_l N[M/x] : V} \ (L\ cut)$$

$$\frac{\Pi \triangleright !\Gamma \vdash_l M :!U \quad \Delta, x :!U \vdash_l N : V \quad \Pi \text{ duplicable}}{!\Gamma, \Delta \vdash_l N[M/x] : V} \ (D\ cut)$$

$$\frac{\Pi \triangleright \Gamma \vdash_l M :!U \quad \Delta, x :!U \vdash_l N : V \quad \Pi \text{ not duplicable}}{\Gamma, \Delta \vdash_l N[M/x] : V} \ (S\ cut)$$

where $\Pi \triangleright !\Gamma \vdash_l M :!U$ is duplicable if it corresponds to a !-box in the respective proof-net of SLL [2], and L, D, S are short for *Linear*, *Duplication* and *Sharing*. The problem is that, while (L *cut*) and (D *cut*) both correspond to β-reduction, (S *cut*) is not necessarily reflected into a β-reduction.
Let us show it by an example. Consider the term $M \equiv y((\lambda z.sz)w)((\lambda z.sz)w)$ and let $Z = U \multimap U \multimap V, S = V \multimap !U$. It has the typing $y : Z, s : S, w : V \vdash_l y((\lambda z.sz)w)((\lambda z.sz)w) : V$ as proved by the following (incomplete) derivation:

$$\dfrac{\dfrac{s : S \vdash_l \lambda z.sz : S \quad t : S, w : V \vdash_l tw :!U}{s : S, w : V \vdash_l (\lambda z.sz)w :!U}\ (cut) \quad \dfrac{y : Z, r : U, l : U \vdash_l yrl : V}{y : Z, x :!U \vdash_l yxx : V}\ (m)}{y : Z, s : S, w : V \vdash_l y((\lambda z.sz)w)((\lambda z.sz)w) : V}\ (cut)$$

Table 1. SLL_λ

$$\frac{}{x : U \vdash_l x : U}\ (Id) \qquad \frac{\Gamma \vdash_l M : U \quad x : V, \Delta \vdash_l N : Z \quad \Gamma\#\Delta \quad y \text{ fresh}}{\Gamma, y : U \multimap V, \Delta \vdash_l N[yM/x] : Z}\ (\multimap L)$$

$$\frac{\Gamma \vdash_l M : U \quad \Delta, x : U \vdash_l N : V \quad \Gamma\#\Delta}{\Gamma, \Delta \vdash_l N[M/x] : V}\ (cut) \qquad \frac{\Gamma, x : U \vdash_l M : V}{\Gamma \vdash_l \lambda x.M : U \multimap V}\ (\multimap R)$$

$$\frac{\Gamma \vdash_l M : U}{!\Gamma \vdash_l M :!U}\ (sp) \qquad \frac{\Gamma, x_0 : U, ..., x_n : U \vdash_l M : V}{\Gamma, x :!U \vdash_l M[x/x_0, ..., x/x_n] : V}\ (m)$$

$$\frac{\Gamma \vdash_l M : U}{\Gamma \vdash_l M : \forall\alpha.U}\ (\forall R) \qquad \frac{\Gamma, x : U[V/\alpha] \vdash_l M : Z}{\Gamma, x : \forall\alpha.U \vdash_l M : Z}\ (\forall L)$$

where the *cut* is clearly a $(S\ cut)$. It is easy to check that $y((\lambda z.sz)w)((\lambda z.sz)w) \rightarrow_\beta y(sw)((\lambda z.sz)w)$ but unfortunately there is no derivation with conclusion: $y : Z, s : S, w : V \vdash_l y((sw)((\lambda z.sz)w) : V$.

The technical reason is that, in the previous derivation, there is a mismatch between the term and the derivation: in M there are two copies of $(\lambda z.sz)w$, while in the derivation there is just one subderivation with subject $(\lambda z.sz)w$, and this subderivation is not duplicable, since in particular S is not modal. Then the two copies of this subterm cannot be treated in a non uniform way.

The problem is not new, and all the type assignment systems for λ-calculus derived from Linear Logic need to deal with it. Until now the proposed solutions follow three different paths, all based on a natural deduction definition of the type assignment. The first one, proposed in [8], explicitly checks the duplicability condition before to perform a normalization step. So in the resulting language (which is a fully typed λ-calculus) the set of redexes is a proper subset of the set of classical β-redexes. In the light logics setting, in [9], a type assignment for the λ-calculus, based on EAL, is designed. There the authors use the call-by-value λ-calculus, where the restricted definition of reduction corresponds exactly to linear substitution and duplication. In the type assignment for λ-calculus based on LAL, made in [4], a syntax of types is used, where the modality ! is no more present, and there are two arrows, a linear and an intuitionistic one, whose elimination reflects the linear and the duplication cut respectively. The set of types corresponds to a restriction of the set of logical formulas. All the approaches need a careful control of the context, which technically has been realized by splitting it in different parts, collecting respectively the linear and modal assumptions (in case of [9] a further context is needed). Here we want to explore a different approach. A type derivation for λ-calculus based on sequent calculus is in some sense redundant, since the same typing can be proved by a plethora of different derivations, corresponding to different ways of building the term. A (cut) rule and a $(\multimap L)$ rule both correspond to a modification of the subject through a substitution. Our key observation is that a term can always be built using linear substitutions. As example, take the term before $y((\lambda z.sz)w)((\lambda z.sz)w)$. It can be seen as $yxx[(\lambda z.sz)w/x]$ but also as $yx_1x_2[(\lambda z.s_1z)w_1/x_1, (\lambda z.s_2z)w_2/x_2][s/s_1, s/s_2, w/w_1,\ w/w_2]$, where all substitutions are linear. Then we want to restrict the set of proofs, in such a way that both sharing and duplication are forbidden, and terms are built by means of linear substitutions only. Such restriction preserves subject reduction since duplication is a derived rule. In order to do this, we start from an affine version of SLL and restrict the formulae, using as types just a subset of the SLL formulae which are, in some sense, recursively linear. The gain is that the splitting of the context is no more necessary.

3 The Soft Type Assignment System

In this section we will present a type assignment system, which assigns to λ-terms a proper subset of SLL formulae, and we will prove that it enjoys subject

Table 2. Soft Type Assignment system

$$\frac{}{x : A \vdash x : A}\ (Id) \qquad \frac{\Gamma \vdash M : \tau \quad x : A, \Delta \vdash N : \sigma \quad \Gamma \# \Delta \quad y \text{ fresh}}{\Gamma, y : \tau \multimap A, \Delta \vdash N[yM/x] : \sigma}\ (\multimap L)$$

$$\frac{\Gamma, x : \sigma \vdash M : A}{\Gamma \vdash \lambda x.M : \sigma \multimap A}\ (\multimap R) \qquad \frac{\Gamma \vdash M : A \quad \Delta, x : A \vdash N : \sigma \quad \Gamma \# \Delta}{\Gamma, \Delta \vdash N[M/x] : \sigma}\ (cut)$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma, x : A \vdash M : \sigma}\ (w) \qquad \frac{\Gamma \vdash M : \sigma}{!\Gamma \vdash M : !\sigma}\ (sp) \qquad \frac{\Gamma, x : A[B/\alpha] \vdash M : \sigma}{\Gamma, x : \forall \alpha.A \vdash M : \sigma}\ (\forall L)$$

$$\frac{\Gamma, x_1 : \tau, ..., x_n : \tau \vdash M : \sigma}{\Gamma, x : !\tau \vdash M[x/x_1, ..., x/x_n] : \sigma}\ (m) \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall \alpha.A}\ (\forall R)$$

reduction. Types correspond in some sense to "recursively linear" formulae and quantification can be applied only to linear types. The type assignment system is such that all rules dealing with substitution ($(\multimap L)$ and (cut)) are applicable only if the type of the replaced variable is linear. So all terms are built through linear substitutions.

Definition 1. *i) The syntax of λ-terms is the usual one. Let α range over a countable set of type variables. The set* **T** *of* soft types *is defined as follows:*

$$A ::= \alpha \mid \sigma \multimap A \mid \forall \alpha.A \text{ (Linear Types)}$$
$$\sigma ::= A \mid !\sigma$$

Type variables are ranged over by α, β, linear types by A, B, C, and types by σ, τ, ζ. $\equiv$ denotes the syntactical equality both for types and terms (modulo renaming of bound variables).

ii) A context is a set of assumptions of the shape $x : \sigma$, where all variables are different. Contexts are ranged over by Γ, Δ. $dom(\Gamma) = \{x \mid \exists x : \sigma \in \Gamma\}$ and $\Gamma \# \Delta$ means $dom(\Gamma) \cap dom(\Delta) = \emptyset$.

iii) STA proves sequents of the shape $\Gamma \vdash M : \sigma$ where Γ is a context, M is a λ-term, and σ is a soft type. The rules are given in Table 2, where as usual the rule $(\forall R)$ has the side condition that α must not be free in Γ.

iv) Derivations are denoted by $\Pi, \Sigma, \Phi, \Psi, \Theta$. $\Pi \triangleright \Gamma \vdash M : \sigma$ denotes a derivation Π with conclusion $\Gamma \vdash M : \sigma$. $\vdash M : \sigma$ is short for $\emptyset \vdash M : \sigma$.

We will use the following:

Notation. *$FV(M)$ denotes the set of free variables of M, $n_o(x, M)$ the number of free occurrences of the variable x in M. $M\{z\}$ ($M\{zQ\}$) denotes that z (zQ) occurs once in M. $!^n\sigma$ is an abbreviation for $!...!\sigma$ n-times. $!^0\sigma \equiv \sigma$. As usual $\multimap$ associates to the right and has precedence on $\forall$, while $!$ has precedence on everything else. $\sigma[A/\alpha]$ denotes the capture free substitution in σ of all occurrences of the type variable α by the linear type A: note that this kind of substitution*

preserves the correct syntax of types. $\boldsymbol{\sigma}$ *denotes a sequence of types, and* $\sigma[\boldsymbol{A}/\boldsymbol{\alpha}]$ *denotes the replacement of the* i*-th type variable in* $\boldsymbol{\alpha}$ *by the* i*-th type in* $\boldsymbol{A}$*.* $|\boldsymbol{\sigma}|$ *denotes the length of the sequence* $\boldsymbol{\sigma}$*.* $\Gamma[\boldsymbol{A}/\boldsymbol{\alpha}]$ *is the context* Γ*, where all type* τ *has been replaced by* $\tau[\boldsymbol{A}/\boldsymbol{\alpha}]$*.* $\forall\boldsymbol{\alpha}.A$ *is an abbreviation for* $\forall\alpha_1....\forall\alpha_n.A$*, for some* $n \geq 0$*. Note that each type is of the shape* $!^n\forall\boldsymbol{\alpha}.A$

Some comments on STA are needed. The (*cut*) is a linear cut, according to the classification given in the previous section, and rule $(\multimap L)$ requires that the type of the replaced variable is linear, so in particular all substitutions in the subject act on a variable occurring at most once (as we will see in the following lemma). Moreover both the (Id) rule and (w) rule can introduce only linear types.

Note for example that the term M considered in Section 2. is typable with typing: $y : A \multimap A \multimap B, s :!(A \multimap A), w :!A \vdash y((\lambda z.sz)w)((\lambda z.sz)w) : B$

Lemma 2. *i)* $\Gamma, x : A \vdash M : \nu$ *implies* $n_o(x, M) \leq 1$*.*
ii) $\Gamma, x : A \vdash M :!\sigma$ *implies* $x \notin FV(M)$*.*

The system enjoys the classical substitution properties.

Lemma 3. *i)* $\Gamma \vdash M : \sigma$ *implies* $\Gamma[\boldsymbol{\zeta}/\boldsymbol{\alpha}] \vdash M : \sigma[\boldsymbol{\zeta}/\boldsymbol{\alpha}]$*.*
ii) $\Gamma \vdash M : \forall\boldsymbol{\alpha}.A$ *implies* $\Gamma \vdash M : A$*.*

Let $\Pi \triangleright \Gamma, x : \tau \vdash M : \sigma$. The notion of chain of x in Π will be used to remember the successive renaming of the assumptions which give rise to the assumption $x : \tau$. The notion of s-chain is necessary for dealing with the particular case of the replacements of a variable by another one in a cut rule.

Definition 4. *Let* $\Pi \triangleright \Gamma, x : \tau \vdash M : \sigma$*.*

- *A* chain *of* x *in* Π *is a sequence of variables inductively defined as follows:*
  - *Let the last applied rule of* Π *be:*

$$\frac{}{x : A \vdash x : A} \quad , \quad \frac{\Gamma' \vdash P : \sigma}{\Gamma', x : A \vdash P : \sigma} \quad or \quad \frac{\Delta \vdash P : \tau \quad z : A, \Gamma \vdash N : \sigma}{x : \tau \multimap A, \Gamma, \Delta \vdash N[xP/z] : \sigma}$$

    Then the only chain of x *is* x *itself.*
  - *Let the last applied rule of* Π *be:*

$$\frac{\Pi' \triangleright \Gamma', x_1 : \tau, ..., x_k : \tau \vdash N : \sigma}{\Gamma', x :!\tau \vdash N[x/x_1, ..., x/x_k] : \sigma}\ (m)$$

    Then a chain of x *in* Π *is every sequence* $x\mathbf{c}$*, where* $\mathbf{c}$ *is a chain of* x_i *in* Π' *for* $1 \leq i \leq k$*.*
  - *In every other case there is an assumption with subject* x *both in the conclusion of the rule and in one of its premises* Σ*. Then a chain of* x *in* Π *is every sequence* $x\mathbf{c}$*, where* $\mathbf{c}$ *is a chain of* x *in* Σ*.*
- *Let* $\mathbf{c}$ *be a chain of* x *in* Π *and let* x_k *be the last variable of* $\mathbf{c}$*. Then* x_k *is an* ancestor *of* x *in* Π*.* $n_a(x, \Pi)$ *denotes the number of ancestors of* x *in* Π*.*

- *y is an* effective ancestor *of x if and only if it is an ancestor of x, and moreover every variable z in the chain of x ending with y occurs in sequent where z belongs to the free variable set of the subject of the sequent. $n_e(x, \Pi)$ denotes the number of effective ancestors of x in Π.*
- *A* s-chain *of x in Π is inductively defined analogously to the chain of x in Π but the (cut) case:*

$$\frac{\Gamma \vdash M : A \quad \Sigma \triangleright \Delta, z : A \vdash N : \sigma}{\Gamma, \Delta \vdash N[M/z] : \sigma}\ (cut)$$

If $M \not\equiv x$ then an s-chain of x in Π is defined as a chain of x in Π, otherwise an s-chain of x in Π is every sequence $z\boldsymbol{c}$, where $\boldsymbol{c}$ is a chain of z in Σ.
An s-ancestor *is defined analogously to ancestor, but with respect to s-chains.*

The following lemma follows easily from the previous definition.

Lemma 5. *Let $\Pi \triangleright \Gamma \vdash M : \sigma$. Then $\forall x :\ n_o(x, M) \leq n_e(x, \Pi) \leq n_a(x, \Pi)$.*

The notion of effective ancestor will be used in Section 4. The Generation Lemma connects the shape of a term with its possible typings, and will be useful in the sequel. Let Π and Π' be two derivations in STA, proving the same conclusion: $\Pi \rightsquigarrow \Pi'$ denotes the fact that Π' is obtained from Π by commuting some rule applications, by erasing m rule applications, by inserting $n \leq m$ applications of rule (w), for some $n, m \geq 0$, and by renaming some variables.

Lemma 6 (Generation Lemma)

1. *$\Gamma \vdash \lambda x.M : \sigma$ implies $\sigma \equiv !^i(\forall \boldsymbol{\alpha}.\tau \multimap A)$, for some τ, A and $i, |\boldsymbol{\alpha}| \geq 0$;*
2. *$\Pi \triangleright \Gamma \vdash \lambda x.P : \sigma \multimap A$ implies $\Pi \rightsquigarrow \Pi'$, whose last rule is $(\multimap R)$.*
3. *$\Pi \triangleright \Gamma, x : \forall \boldsymbol{\alpha}.\tau \multimap A \vdash M\{xQ\} : \sigma$ implies that Π is composed by a subderivation Σ, followed by a sequence δ of rules not containing rule (sp), and the last rule of Σ is:*

$$\frac{\Gamma' \vdash Q' : \tau' \quad \Gamma'', z : A' \vdash P\{z\} : \sigma'}{\Gamma', \Gamma'', t : \tau' \multimap A' \vdash P\{z\}[tQ'/z] : \sigma'}\ (\multimap L)$$

where $\tau' \equiv \tau[\boldsymbol{B}/\boldsymbol{\alpha}]$, $A' \equiv A[\boldsymbol{B}/\boldsymbol{\alpha}]$, t is the only s-ancestor of x in Π, for some $P, Q', \Gamma', \Gamma'', \boldsymbol{B}, \boldsymbol{\alpha}, \sigma'$.
4. *$\Pi \triangleright \Gamma \vdash M :!\sigma$ implies $\Pi \rightsquigarrow \Pi'$ where Π' is composed by a subderivation, ending with the rule (sp) proving $!\Gamma' \vdash M :!\sigma$, followed by a sequence of rules (w), $(\multimap L)$, (m),$(\forall L)$, (cut), all dealing with variables not occurring in M.*
5. *$\Pi \triangleright !\Gamma \vdash M :!\sigma$ implies $\Pi \rightsquigarrow \Pi'$, whose last rule is (sp).*
6. *$\Pi \triangleright \Gamma \vdash \lambda x.M : \forall \alpha.A$ implies $\Pi \rightsquigarrow \Pi'$ where the last rule of Π' is $(\forall R)$.*

3.1 Subject Reduction

STA enjoys the subject reduction property. The proof is based on the fact that, while formally the (cut) rule in it is linear, the duplication cut is a derived rule. To prove this, we need a further lemma.

Lemma 7. *$\Pi \triangleright \Gamma, x :!^n \forall \boldsymbol{\alpha}.A \vdash M : \sigma$ where $A \not\equiv \forall \boldsymbol{\beta}.B$ and y is an ancestor of x in Π imply that y has been introduced with type $\forall \boldsymbol{\alpha'}.A[\boldsymbol{B}/\boldsymbol{\alpha''}]$, for some $\boldsymbol{\alpha'}, \boldsymbol{\alpha''}$ (possible empty) such that $\boldsymbol{\alpha} = \boldsymbol{\alpha''}, \boldsymbol{\alpha'}$*

Lemma 8. *The following rule is derivable in STA:*

$$\frac{\Pi \triangleright !^n \Gamma \vdash M :!^n B \quad \Sigma \triangleright x :!^n B, \Delta \vdash N : \sigma \quad !^n \Gamma \# \Delta}{!^n \Gamma, \Delta \vdash N[M/x] : \sigma}\ (dup)$$

Proof. In case $n = 0$ the proof is obvious, since (dup) coincides with the (cut) rule. Otherwise let $B \equiv \forall \boldsymbol{\alpha}.A$. By Lemma 6.5, Π can be transformed into a derivation Π'', which is $\Pi' \triangleright \Gamma \vdash M : A$ followed by n applications of rule (sp). Let the ancestors of x in Σ be $x_1, ..., x_m$. By Lemma 7, x_j has been introduced with type $\forall \boldsymbol{\alpha'_j}.A[\boldsymbol{C_j}/\boldsymbol{\alpha''_j}]$, for some $\boldsymbol{C_j}, \boldsymbol{\alpha'_j}, \boldsymbol{\alpha''_j}$ $(1 \leq j \leq m)$. By Lemma 3, there are disjoint derivations $\Pi'_j \triangleright \Gamma_j \vdash M_j : \forall \boldsymbol{\alpha'_j}.A[\boldsymbol{C_j}/\boldsymbol{\alpha''_j}]$, where M_j and Γ_j are fresh copies of M and Γ $(1 \leq j \leq m)$. Then, for every x_j introduced by an (Id) rule, replace such rule by Π'_j. For every x_j introduced by a $(\multimap L)$ rule as:

$$\frac{\Gamma' \vdash R : \tau[\boldsymbol{C_j}/\boldsymbol{\alpha''_j}] \quad \Delta', z : B'[\boldsymbol{C_j}/\boldsymbol{\alpha''_j}] \vdash T : \sigma}{\Gamma', x_j : (\tau \multimap B')[\boldsymbol{C_j}/\boldsymbol{\alpha''_j}], \Delta' \vdash T[x_j R/z] : \sigma}\ (\multimap L)$$

where $\forall \boldsymbol{\alpha'_j}.A[\boldsymbol{C_j}/\boldsymbol{\alpha''_j}] \equiv (\tau \multimap B')[\boldsymbol{C_j}/\boldsymbol{\alpha''_j}]$, $|\boldsymbol{\alpha'_j}| = 0$, after this insert the rule:

$$\frac{\Pi'_j \quad \Gamma', x_j : A[\boldsymbol{C_j}/\boldsymbol{\alpha''_j}], \Delta' \vdash T[x_j R/z] : \sigma}{\Gamma', \Delta', \Gamma_j \vdash T[M_j R/z] : \sigma}\ (cut).$$

For every x_j introduced by a (w) rule, just erase the rule. Moreover arrange the context and the subject in all rules according with these modifications. Then replace every application of a rule (m), when applied on variables in a chain of x in Σ, by a sequence of rules (m) applied to the free variables of the corresponding copy of M. So the resulting derivation Φ proves $!^n \Gamma, \Delta \vdash N[M/y] : \sigma$. □

Note that the rule (dup) cannot represent a $(S\ cut)$ since by Lemma 6.4 a derivation where both the context and the type are modal always corresponds to a !-box. The above lemma allow us to freely use (dup) rule in what follows.

Theorem 9 (Subject Reduction). *If $\Gamma \vdash M : \sigma$ and $M \to_\beta M'$ then $\Gamma \vdash M' : \sigma$*

Proof. By induction on the derivation. The only interesting case is when the last rule is (cut), creating the redex $(\lambda y.P)Q$ reduced in M.

$$\frac{\Psi \triangleright \Gamma \vdash \lambda y.P : A \quad \Sigma \triangleright \Delta, x : A \vdash N\{xQ\} : \sigma \quad \Gamma \# \Delta}{\Gamma, \Delta \vdash N\{xQ\}[\lambda y.P/x] : \sigma}\ (cut).$$

By Lemma 6.1, and by the constraint on the (cut) rule, $A \equiv \forall \boldsymbol{\alpha}.\tau \multimap B$. By Lemma 6.3, Σ is composed by a subderivation Σ_2, followed by a sequence δ of rules not containing (sp). Σ_2 is:

$$\frac{\Theta_1 \triangleright \Delta'' \vdash Q' : \tau' \quad \Theta_2 \triangleright \Delta', z : B' \vdash N'\{z\} : \sigma'}{\Delta', t : \tau' \multimap B', \Delta'' \vdash N'\{z\}[tQ'/z] : \sigma'}$$

where $\tau' = \tau[\boldsymbol{C}/\boldsymbol{\alpha}]$, $B' = B[\boldsymbol{C}/\boldsymbol{\alpha}]$, t is the only s-ancestor of x, for some $\Delta', \Delta'', N', Q', \sigma', \boldsymbol{C}$. By Lemma 6.2,6, $\Psi \rightsquigarrow \Psi'$ ending as:

$$\dfrac{\dfrac{\Gamma, y : \tau \vdash P : B}{\Gamma \vdash \lambda y.P : \tau \multimap B}\ (\multimap R)}{\Gamma \vdash \lambda y.P : \forall \boldsymbol{\alpha}.\tau \multimap B}\ (\forall R)^*$$

hence by Lemma 3.ii), there is a derivation $\Phi \triangleright \Gamma, y : \tau' \vdash P : B'$, since $\boldsymbol{\alpha}$ are not free in Γ.

Let $\tau' \equiv !^n A'$ $(n \geq 0)$ for some A'. By Lemma 6.4,5, Θ_1 can be transformed in a derivation composed by a subderivation $\Theta_4 \triangleright !^n \Delta''' \vdash Q' :!^n A'$, followed by a sequence δ' of applications of rules (w), $(\multimap L)$,(m), $(\forall L)$, (cut), all dealing with variables not occurring in Q. So we can apply the derived rule (dup) obtaining:

$$\dfrac{\Theta_4 :!^n \Delta''' \vdash Q' :!^n A' \quad \Phi \triangleright \Gamma, y :!^n A' \vdash P : B'}{\Gamma, !^n \Delta''' \vdash P[Q'/y] : B'}$$

Then, by applying δ' we have $\Theta_3 \triangleright \Gamma, \Delta'' \vdash P[Q'/y] : B'$. Hence,

$$\dfrac{\Theta_3 \quad \Theta_2}{\Gamma, \Delta', \Delta'' \vdash N'\{z\}[P[Q'/y]/z] : \sigma}\ (cut)$$

and by applying δ the desired derivation can be built. □

The proof of subject reduction gives evidence to the fact that β-reduction, while corresponding to the cut elimination in case the bound variable occurs at most once, is reflected into a global transformation of the derivation, in case a duplication of the argument is necessary.

4 Complexity

In this section we will show that STA is correct and complete for polynomial complexity. Namely, in Subsection 4.1, we will prove that, if a term M can be typed in STA by a derivation Π, then it reduces to normal form in a number of β-reduction steps which is bounded by $|M|^{\mathtt{d}(\Pi)+1}$, where $|M|$ is the number of symbols of M and $\mathtt{d}(\Pi)$ is the number of nested applications of rule (sp) in Π. So working with terms typed by derivations of fixed degree assures us to keep only a polynomial number of computation steps. Then we show that this polystep result extends to a polytime result. In the Subsection 4.2, we prove that STA is complete both for PTIME and FPTIME, using a representation of Turing machine working in polynomial time by λ-terms, typable in STA through derivations obeying suitable constraints. The key idea is that data can be represented by terms typable by linear types using derivations of degree 0. Obviously programs can duplicate their data, so a derivation typing an application of a program to its data can have degree greater than 0, but this degree depends only on the program, so it does not affect the complexity measure.

4.1 Complexity of Reductions in STA

SLL enjoys a polynomial time bound, namely the cut-elimination procedure is polynomial in the size of the proof. This result holds obviously also for STA. But we need something more, namely to relate the polynomial bound to the size of the λ-terms. So we prove this result by defining measures of both terms and proofs, which are an adaptation of those given by Lafont, but they do not take into account the subderivations that do not contribute to the term formation (which we will call "erasing").

Definition 10.

- *The size* $|M|$ *of a term* M *is defined as* $|x| = 1$, $|\lambda x.M| = |M| + 1$, $|MN| = |M| + |N| + 1$. *The size* $|\Pi|$ *of a proof* Π *is the number of rules in* Π.
- *In the rules* $(\multimap L)$ *and* (cut), *the subderivation proving* $\Gamma \vdash M : \tau$ *and* $\Gamma \vdash M : A$ *respectively is* erasing *if* $x \notin FV(N)$ *(using notation of Table 2).*
- *The* rank *of a rule* (m), *as defined in Table 2, is the number* $k \leq n$ *of variables* x_i *such that* $x_i \in FV(M)$ $(1 \leq i \leq n)$. *Let* r *be the the maximum rank of a rule* (m) *in* Π, *not considering erasing subderivations. The rank* $\mathtt{rk}(\Pi)$ *of* Π *is the maximum between* 1 *and* r.
- *The degree* $\mathtt{d}(\Pi)$ *of* Π *is the maximum nesting of applications of rule* (sp) *in* Π, *not considering erasing subderivations;*
- *The weight* $\mathtt{W}(\Pi, r)$ *of* Π *with respect to* r *is defined inductively as follows.*
  - *If the last applied rule is* (Id) *then* $\mathtt{W}(\Pi, r) = 1$.
  - *If the last applied rule is* $(\multimap R)$ *with premise a derivation* Σ, *then* $\mathtt{W}(\Pi, r) = \mathtt{W}(\Sigma, r) + 1$.
  - *If the last applied rule is* (sp) *with premise a derivation* Σ, *then* $\mathtt{W}(\Pi, r) = r\mathtt{W}(\Sigma, r)$.
  - *If the last applied rule is:*

$$\frac{\Sigma \triangleright \Gamma \vdash M : A \quad \Phi \triangleright x : A, \Delta \vdash N : \sigma}{\Gamma, \Delta \vdash N[M/x] : \sigma}\ (cut)$$

    then $\mathtt{W}(\Pi, r) = \mathtt{W}(\Sigma, r) + \mathtt{W}(\Phi, r) - 1$ *if* $x \in FV(N)$, $\mathtt{W}(\Pi, r) = \mathtt{W}(\Phi, r)$ *otherwise.*
  - *In every other case* $\mathtt{W}(\Pi, r)$ *is the sum of the weights of the premises with respect to* r, *not counting erasing subderivations.*

Lemma 11. *Let* $\Pi \triangleright \Gamma \vdash M : \sigma$. *Then:*
1. $\mathtt{rk}(\Pi) \leq |M| \leq |\Pi|$. 2. $\mathtt{W}(\Pi, 1) \leq |M|$. 3. $\mathtt{W}(\Pi, r) \leq r^{\mathtt{d}(\Pi)}\mathtt{W}(\Pi, 1)$
4. $x :!^q A \in \Gamma$ *implies* $n_o(x, M) \leq n_e(x, \Pi) \leq \mathtt{rk}(\Pi)^q$.
5. $\Pi \rightsquigarrow \Pi'$ *implies* $\mathtt{W}(\Pi', r) \leq \mathtt{W}(\Pi, r)$.

The normalization of proofs is based on the notion of substitution. The following lemma extends the weight definition to the derived rule (dup) by taking into account the weight of the involved proofs.

Lemma 12. *Let* Φ *be a derivation ending with the rule* (dup), *with premises* Σ *and* Π. *Then, if* $r \geq \mathtt{rk}(\Phi)$, $\mathtt{W}(\Phi, r) \leq \mathtt{W}(\Sigma, r) + \mathtt{W}(\Pi, r)$.

Proof. It suffices to verify how the weights are modified in the proof of Lemma 8, using Lemma 11. We will use exactly the same notations as in the lemma.

The transformation of Π in Π'' leaves the weight unchanged. In particular $\mathtt{W}(\Pi, r) = r^n \mathtt{W}(\Pi', r)$ and $\mathtt{W}(\Pi'_j, r) = \mathtt{W}(\Pi', r)$.

By Lemma 6.4 and Lemma 6.5 Σ can be transformed in a derivation Σ' followed by $k \leq n$ applications of rule (sp) and by a sequence of rules dealing with variables not occurring in N. In particular $\mathtt{W}(\Sigma, r) = r^k \mathtt{W}(\Sigma', r)$.

Then, for every ancestor x_j in Σ the replacement by Π'_j increases the weight $\mathtt{W}(\Sigma', r)$ of at most a quantity $\mathtt{W}(\Pi', r)$. By definition of weight we are interested only in effective ancestors, hence by Lemma 11.4:

$\mathtt{W}(\Phi, r) = r^k(\mathtt{W}(\Sigma', r) + n_e(x, \Sigma')\mathtt{W}(\Pi', r)) \leq r^k \mathtt{W}(\Sigma', r) + r^k r^{n-k} \mathtt{W}(\Pi', r)$
$\leq r^k \mathtt{W}(\Sigma', r) + r^n \mathtt{W}(\Pi', r) = \mathtt{W}(\Sigma, r) + \mathtt{W}(\Pi, r).$ □

Now we can show that the weight decreases when a β-reduction is performed.

Lemma 13. *Let $\Pi \triangleright \Gamma \vdash M : \sigma$ and $M \to_\beta M'$. There is a derivation $\Pi' \triangleright \Gamma \vdash M' : \sigma$, with $\mathtt{rk}(\Pi) \geq \mathtt{rk}(\Pi')$, such that if $r \geq \mathtt{rk}(\Pi')$ then $\mathtt{W}(\Pi', r) < \mathtt{W}(\Pi, r)$.*

Proof. It suffices to verify how the weights are modified in the proof of Theorem 9, using Lemma 11. We will use exactly the same notations as in the theorem. Note that if a derivation $\tilde{\Phi}$ is composed by a derivation $\tilde{\Phi}_1$, followed either by the sequence δ or δ', then either $\mathtt{W}(\tilde{\Phi}, r) = \mathtt{W}(\tilde{\Phi}_1, r) + c$ or $\mathtt{W}(\tilde{\Phi}, r) = \mathtt{W}(\tilde{\Phi}_1, r)$, where c is a constant depending only on δ, since δ does not contain rule (sp). Then looking at the definition of weight and to the theorem we can state the following (in)equalities:

$\mathtt{W}(\Pi', r) = \mathtt{W}(\Theta_3, r) + \mathtt{W}(\Theta_2, r) + c - 1 \leq \mathtt{W}(\Phi, r) + \mathtt{W}(\Theta_4, r) + \mathtt{W}(\Theta_2, r) + c - 1$
$= \mathtt{W}(\Phi, r) + \mathtt{W}(\Theta_1, r) + \mathtt{W}(\Theta_2, r) + c - 1 = \mathtt{W}(\Phi, r) + \mathtt{W}(\Sigma, r) - 1$
$< \mathtt{W}(\Psi, r) + \mathtt{W}(\Sigma, r) - 1 = \mathtt{W}(\Pi, r).$ □

Finally the desired results can be obtained.

Theorem 14 (Strong Polystep Soundness). *Let $\Pi \triangleright \Gamma \vdash M : \sigma$, and M β-reduces to M' in m steps. Then:*

$$(1) \quad m \leq |M|^{\mathtt{d}(\Pi)+1} \qquad (2) \quad |M'| \leq |M|^{\mathtt{d}(\Pi)+1}$$

Proof. (1) By repeatedly using Lemma 13 and by Lemma 11.3, since $|M| \geq \mathtt{rk}(\Pi)$. (2) By repeatedly using Lemma 13 and by Lemma 11.2. □

Theorem 15 (Polytime Soundness). *Let $\Pi \triangleright \Gamma \vdash M : \sigma$, then M can be evaluated to normal form on a Turing Machine in time $O(|M|^{3(\mathtt{d}(\Pi)+1)})$.*

Proof. Clearly, as pointed in [10], a β reduction step $N \to_\beta N'$ can be simulated in time $O(|N|^2)$ on a Turing Machine. Let $M \equiv M_0 \to_\beta M_1 \to_\beta \cdots \to_\beta M_n$ be a reduction of M to normal form M_n. By Theorem 14.2 $|M_i| \leq |M|^{\mathtt{d}(\Pi)+1}$ for $0 \leq i \leq n$, hence each step in the reduction takes time $O(|M|^{2(\mathtt{d}(\Pi)+1)})$. Furthermore since by Theorem 14.1 n is $O(|M|^{\mathtt{d}(\Pi)+1})$, the conclusion follows. □

Clearly Theorem 15 implies that a strong polytime soundness holds, considering Turing Machine with an oracle for strategies.

4.2 Polynomial Time Completeness

In order to prove polynomial time completeness of STA we need to encode Turing Machines (TM) configurations, transitions between configurations and iterators. We achieve such results considering the usual notion of lambda definability, given in [11], generalized to different kinds of data. We encode input data in the usual way, TM configurations and transitions following the lines of [7] and iterators as usual by Church numerals. We stress that we allow a liberal typing, the only constraint we impose in order to respect Theorem 14 is that each input data must be typable through derivations with degree 0.

Some syntactic sugar. To keep notation concise we add some syntactic sugar. Let $M \circ N$ stand for $\lambda z.M(Nz)$, $M_1 \circ M_2 \circ \cdots \circ M_n$ stand for $\lambda z.M_1(M_2(\cdots(M_n z)))$. As usual I stands for $\lambda x.x$, and $\vdash I : \forall\alpha.\alpha \multimap \alpha$.

Tensor product is definable by second order as $\sigma \otimes \tau \doteq \forall\alpha.(\sigma \multimap \tau \multimap \alpha) \multimap \alpha$. The constructors and destructors for this data type are $\langle M, N\rangle \doteq \lambda x.xMN$, $\texttt{let } z \texttt{ be } x, y \texttt{ in } N \doteq z(\lambda x.\lambda y.N)$. n-ary tensor product can be easily defined through the binary one and we use σ^n to denote $\sigma \otimes \cdots \otimes \sigma$ n-times.

Note that, since STA is an affine system, tensor product enjoys some properties of the additive conjunction, as to allow the projectors.

Polynomials To encode natural numbers we will use Church numerals, i.e. $\underline{n} \doteq \lambda s.\lambda z.s^n(z)$. Successor, addition and multiplication are λ-definable in the usual way: $\underline{succ} \doteq \lambda psz.s(psz)$, $\underline{add} \doteq \lambda pqsz.ps(qsz)$ and $\underline{mul} \doteq \lambda pqs.p(qs)$. Note that the above terms are not typable by using the usual type for natural numbers $\mathbf{N} \doteq \forall\alpha.!(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$. For this reason we define *indexed types*, one for each $i \in \mathbb{N}$: $\mathbf{N}_i \doteq \forall\alpha.!^i(\alpha \multimap \alpha) \multimap \alpha \multimap \alpha$ in particular we have $\mathbf{N}_1 \equiv \mathbf{N}$. Clearly for each Church numerals $\underline{n}$ and for each $i > 0 \in \mathbb{N}$ it holds $\vdash \underline{n} : \mathbf{N}_i$. Using indexed type it is easy to derive the following typing: $\vdash \underline{succ} : \mathbf{N}_i \multimap \mathbf{N}_{i+1}$, $\vdash \underline{add} : \mathbf{N}_i \multimap \mathbf{N}_j \multimap \mathbf{N}_{max(i,j)+1}$, $\vdash \underline{mul} : \mathbf{N}_j \multimap !^j\mathbf{N}_i \multimap \mathbf{N}_{i+j}$.

Note that Church numerals behave as iterators only on terms typable with type $\mu \multimap \mu$ for some linear type μ. For this reason the above terms cannot be iterated. Nevertheless they can be composed. In fact, if we want to multiply two natural numbers we can use $\underline{mul}$ typed as $\vdash \underline{mul} : \mathbf{N} \multimap !\mathbf{N} \multimap \mathbf{N}_2$ while if we want to multiply three natural numbers, through the term $\lambda xyz.\underline{mul}(\underline{mul}\, x\, y)z$, the two occurrences of $\underline{mul}$ in it need to be typed by different types, for example $\mathbf{N} \multimap !\mathbf{N} \multimap \mathbf{N}_2$ the innermost one and $\mathbf{N}_2 \multimap !!\mathbf{N} \multimap \mathbf{N}_3$ the outermost. Such change of type, in particular on the number of modalities, does not depend on the values of data.

Lemma 16. *Let P be a polynomial in the variable X and $deg(P)$ its degree. Then there is a term $\underline{P}$ λ-defining P typable as $x :!^{deg(P)}\mathbf{N} \vdash \underline{P} : \mathbf{N}_{2deg(P)+1}$.*

Proof. Consider P in Horner normal form, i.e., $P = a_0 + X(a_1 + X(\cdots(a_{n-1} + Xa_n)\cdots)$. By induction on $deg(P)$ we show something stronger, i.e., for $i > 0$, it is derivable $x_0 : \mathbf{N}_i, x_1 :!^i\mathbf{N}_i, \ldots, x_n :!^{i(deg(P^*)-1)}\mathbf{N}_i \vdash \underline{P^*} : \mathbf{N}_{i(deg(P^*))+deg(P^*)+1}$ where $P^* = a_0 + X_0(a_1 + X_1(\cdots(a_{n-1} + X_n a_n)\cdots)$ so conclusion follows using (m) rule and taking $i = 1$.

Base case is trivial, so consider $P^* = a_0 + X_0(P')$. By induction hypothesis $x_1 : \mathbf{N}_i, \ldots, x_n :!^{i(deg(P')-1)}\mathbf{N}_i \vdash \underline{P'} : \mathbf{N}_{i(deg(P'))+deg(P')+1}$.
Take $\underline{P^*} \equiv \underline{add}(\underline{a_0}, \underline{mul}(x_0, \underline{P'}))$, clearly we have $x_0 : \mathbf{N}_i, x_1 :!^i\mathbf{N}_i, \ldots, x_n :!^{i(deg(P')-1)+i}\mathbf{N}_i \vdash \underline{P^*} : \mathbf{N}_{i(deg(P')+1)+deg(P')+1+1}$. Since $deg(P^*) = deg(P') + 1$: it follows $x_0 : \mathbf{N}_i, x_1 :!^i\mathbf{N}_i, \ldots, x_n :!^{i(deg(P^*)-1)}\mathbf{N}_i \vdash \underline{P^*} : \mathbf{N}_{i(deg(P^*))+deg(P^*)+1}$.
Now by taking $i = 1$ and repeatedly applying (m) rule we conclude $x :!^{deg(P)}\mathbf{N} \vdash \underline{P} \equiv \underline{P^*}[x/x_1, \cdots, x/x_n] : \mathbf{N}_{2deg(P)+1}$ □

Booleans. Let us encode booleans, as usual, by: $\mathbf{0} \doteq \lambda xy.x$, $\mathbf{1} \doteq \lambda xy.y$ and `if` x `then` M `else` $N \doteq xMN$. They can be typed in STA using as type for booleans: $\mathbf{B} \doteq \forall\alpha.\alpha \multimap \alpha \multimap \alpha$. In particular it holds $\vdash b : \mathbf{B}$ for $b \in \{\mathbf{0}, \mathbf{1}\}$. Moreover assuming $\Gamma \vdash M : \sigma$ and $\Delta \vdash N : \sigma$ and $\Gamma \# \Delta$ it follows $\Gamma, \Delta, x : \mathbf{B} \vdash$ `if` x `then` M `else` $N : \sigma$.

With the help of the conditional we can define all the usual boolean functions $\vdash \underline{And} : \mathbf{B} \multimap \mathbf{B} \multimap \mathbf{B}$, $\vdash \underline{Or} : \mathbf{B} \multimap \mathbf{B} \multimap \mathbf{B}$, $\vdash \underline{Not} : \mathbf{B} \multimap \mathbf{B}$. In particular we have contraction on booleans: $\vdash \underline{Cnt} \doteq \lambda b.$ `if` b `then` $\langle \mathbf{0}, \mathbf{0}\rangle$ `else` $\langle \mathbf{1}, \mathbf{1}\rangle : \mathbf{B} \multimap \mathbf{B} \otimes \mathbf{B}$. The above functions and weakening are useful to prove the following.

Lemma 17. *Each boolean total function $f : \mathbb{B}^n \to \mathbb{B}^m$, where $n, m \geq 1$, can be λ-defined by a term $\underline{f}$ typable in STA as $\vdash \underline{f} : \mathbf{B}^n \multimap \mathbf{B}^m$.*

Strings String of booleans can be encoded by: $[\,] \doteq \lambda cz.z$ and $[b_0, b_1, \ldots, b_n] \doteq \lambda cz.cb_0(\cdots(cb_nz)\cdots)$ where $b_i \in \{\mathbf{0}, \mathbf{1}\}$. Boolean strings are typable in STA with the indexed type $\mathbf{S}_i \doteq \forall\alpha.!^i(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap (\alpha \multimap \alpha)$. In particular for each $n, i > 0 \in \mathbb{N}$ it holds $b_0 : \mathbf{B}, ..., b_n : \mathbf{B} \vdash [b_0, \ldots, b_n] : \mathbf{S}_i$. The term $\mathbf{len} \doteq \lambda cs.c(\lambda xy.sy)$ λ-defines the function returning the length of an input string and is typable in STA with typing $\vdash \mathbf{len} : \mathbf{S}_i \multimap \mathbf{N}_i$.

Turing Machine. We can λ-define Turing Machine configurations by terms of the shape $\lambda c.\langle cb_0^l \circ \cdots \circ cb_n^l, cb_0^r \circ \cdots \circ cb_m^r, Q\rangle$ where $cb_0^l \circ \cdots \circ cb_n^l$ and $cb_0^r \circ \cdots \circ cb_m^r$ represent respectively the left and the right part of the tape, while $Q \equiv \langle b_1, \cdots, b_n\rangle$ is a n-ary tensor product of boolean values representing the current state. We assume without loss of generality that by convention the left part of the tape is represented in a reversed order, that the alphabet is composed of only the two symbols $\mathbf{0}$ and $\mathbf{1}$, that the scanned symbol is the first symbol of the right part and that states are divided in accepting and rejecting. The indexed type of a Turing Machine configuration in STA is $\mathbf{TM}_i \doteq \forall\alpha.!^i(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap ((\alpha \multimap \alpha)^2 \otimes \mathbf{B}^q)$. In fact, $\vdash \lambda c.\langle cb_0^l \circ \cdots \circ cb_n^l, cb_0^r \circ \cdots \circ cb_m^r, Q\rangle : \mathbf{TM}_i$. The term $\mathbf{Init} \doteq \lambda tc.\langle \lambda z.z, \lambda z.t(c\mathbf{0})z, \underline{q_0}\rangle$ λ-defines the function that, taking as input $\underline{Q}$, defining a polynomial Q, gives as output a Turing Machine with tape of length Q filled by $\mathbf{0}$'s in the initial state q_0 and with the head at the beginning of the tape. As expected $\vdash \mathbf{Init} : \mathbf{N}_i \multimap \mathbf{TM}_i$. In what follows ID_i will denote:

$$\forall\alpha.!^i(\mathbf{B} \multimap \alpha \multimap \alpha) \multimap ((\alpha \multimap \alpha)^2 \otimes (\mathbf{B} \multimap \alpha \multimap \alpha) \otimes \mathbf{B} \otimes (\mathbf{B} \multimap \alpha \multimap \alpha) \otimes \mathbf{B} \otimes \mathbf{B}^q)$$

Following [7], the work of showing that Turing Machine transitions are λ-definable can be decomposed in two steps. Firstly we have a term

$$\mathbf{Dec} \doteq \lambda sc.\ \texttt{let}\ s(F(c))\ \texttt{be}\ l, r, q\ \texttt{in}\ \texttt{let}\ l\langle I, \lambda x.I, \mathbf{0}\rangle \\ \texttt{be}\ t_l, c_l, b_0^l\ \texttt{in}\ \texttt{let}\ r\langle I, \lambda x.I, \mathbf{0}\rangle\ \texttt{be}\ t_r, c_r, b_0^r\ \texttt{in}\ \langle t_l, t_r, c_l, b_0^l, c_r, b_0^r, q\rangle$$

where $F(c) \doteq \lambda bz.\ \mathtt{let}\ z\ \mathtt{be}\ g,h,i\ \mathtt{in}\ \langle hi \circ g, c, b\rangle$. Such a term is typable as $\vdash \mathbf{Dec} : \mathbf{TM}_i \multimap ID_i$ and its behaviour is to decompose a configuration as:

$$\mathbf{Dec}(\lambda c.\langle cb_0^l \circ \cdots \circ cb_n^l, cb_0^r \circ \cdots \circ cb_m^r, Q\rangle) \to_\beta^* \\ \lambda c.\langle cb_1^l \circ \cdots \circ cb_n^l, cb_1^r \circ \cdots \circ cb_m^r, c, b_0^l, c, b_0^r, Q\rangle$$

Then we can define a term

$$\mathbf{Com} \doteq \lambda sc.\ \mathtt{let}\ sc\ \mathtt{be}\ l,r,c_l,b_l,c_r,b_r,q\ \mathtt{in} \\ \mathtt{let}\ \underline{\delta}\langle b_r,q\rangle\ \mathtt{be}\ b',q',m\ \mathtt{in}\ (\mathtt{if}\ m\ \mathtt{then}\ \mathbf{R}\ \mathtt{else}\ \mathbf{L})b'q'\langle l,r,c_l,b_l,c_r\rangle$$

where $\mathbf{R} \doteq \lambda b'q's.\ \mathtt{let}\ s\ \mathtt{be}\ l,r,c_l,b_l,c_r\ \mathtt{in}\ \langle c_r b' \circ c_l b_l \circ l, r, q'\rangle$ and $\mathbf{L} \doteq \lambda b'q's.\ \mathtt{let}\ s\ \mathtt{be}\ l,r,c_l,b_l,c_r\ \mathtt{in}\ \langle l, c_l b_l \circ c_r b' \circ r, q'\rangle$. Such a term is typable as $\vdash \mathbf{Com} : ID_i \multimap \mathbf{TM}_i$ and depending on the δ transition function, it combines the symbols returning a configuration as:

$$\mathbf{Com}\ (\lambda c.\langle cb_1^l \circ \cdots \circ cb_n^l, cb_1^r \circ \cdots \circ cb_m^r, c, b_0^l, c, b_0^r, Q\rangle) \\ \to_\beta^* \lambda c.\langle cb' \circ cb_0^l \circ cb_1^l \circ \cdots \circ cb_n^l, cb_1^r \circ \cdots \circ cb_m^r, Q'\rangle \ \text{ if } \delta(b_0^r, Q) = (b', Q', \text{Right}) \\ \text{or} \\ \to_\beta^* \lambda c.\langle cb_1^l \circ \cdots \circ cb_n^l, cb_0^l \circ cb' \circ cb_1^r \circ \cdots \circ cb_m^r, Q'\rangle \ \text{ if } \delta(b_0^r, Q) = (b', Q', \text{Left})$$

Checking the typing and the behavior for **Dec** and **Com** is boring but easy.

By combining the above terms we obtain the term $\mathbf{Tr} \doteq \mathbf{Com} \circ \mathbf{Dec}$ which λ-defines a Turing Machine transition and as expected admits the typing $\vdash \mathbf{Tr} : \mathbf{TM}_i \multimap \mathbf{TM}_i$.

Let the open term $\mathbf{T}(b)$ be $\lambda sc.\ \mathtt{let}\ sc\ \mathtt{be}\ l,r,c_l,b_l,c_r,b_r,q\ \mathtt{in}\ \mathbf{R}bq\langle l,r,c_l,b_l,c_r\rangle$ where $\mathbf{R}$ is defined as above. Such term is typable as $b : \mathbf{B} \vdash \mathbf{T}(b) : ID_i \multimap \mathbf{TM}_i$, and it is useful to define the term $\mathbf{In} \doteq \lambda sm.s(\lambda b.\mathbf{T}(b) \circ \mathbf{Dec})m$ that, when supplied by a boolean string and a Turing Machine, writes the input string on the tape of the Turing Machine. Such a term is typable as $\vdash \mathbf{In} : \mathbf{S} \multimap \mathbf{TM}_i \multimap \mathbf{TM}_i$.

The term $\mathbf{Ext} \doteq \lambda s.\ \mathtt{let}\ s(\lambda b.\lambda c.c)\ \mathtt{be}\ l,r,q\ \mathtt{in}\ \underline{f}(q)$ permits to extract the information that the machine is in an accepting or non-accepting state. Since Lemma 17 assures the existence of $\underline{f}$, it holds $\vdash \mathbf{Ext} : \mathbf{TM}_i \multimap \mathbf{B}$. Now we can finally show that STA is complete for PTIME.

Theorem 18 (PTIME Completeness). *Let a decision problem $\mathfrak{P}$ be decided in* polynomial time *P, where $deg(P) = m$, and in* polynomial space *Q, where $deg(Q) = l$, by a Turing Machine $\mathcal{M}$. Then it is λ-definable by a term $\underline{M}$ typable in STA as $s :!^{max(l,m,1)+1}\mathbf{S} \vdash \underline{M} : \mathbf{B}$.*

Proof. By Lemma 16: $s_p :!^m\mathbf{S} \vdash P[\mathbf{len}s_p/x] : \mathbf{N}_{2m+1}$ and $s_q :!^l\mathbf{S} \vdash Q[\mathbf{len}s_q/x] : \mathbf{N}_{2l+1}$. Furthermore by composition: $s : \mathbf{S}, q : \mathbf{N}_{2l+1}, p : \mathbf{N}_{2m+1} \vdash \mathbf{Ext}(p\mathbf{Tr}(\mathbf{In}s(\mathbf{Init}(q)))) : \mathbf{B}$ so by (cut) and some applications of (m) rule the conclusion follows. □

Without loss of generality we can assume that a Turing machine stops on accepting states with the head at the begin of the tape. Hence the term $\mathbf{Ext_F} \doteq \lambda sc.\ \mathtt{let}\ sc\ \mathtt{be}\ l,r,q\ \mathtt{in}\ r$ extracts the result from the tape. It is easy to verify that the typing $\vdash \mathbf{Ext_F} : \mathbf{TM}_i \multimap \mathbf{S}_i$ holds. So we can conclude that STA is also complete for FPTIME.

Theorem 19 (FPTIME Completeness). *Let a function $\mathcal{F}$ be computed in* polynomial time *P, where $deg(P) = m$, and in* polynomial space *Q, where $deg(Q) = l$, by a Turing Machine $\mathcal{M}$. Then it is λ-definable by a term $\underline{M}$ typable in STA as $!^{max(l,m,1)+1}\mathbf{S} \vdash \underline{M} : \mathbf{S}_{2m+1}$.*

In the work of Lafont [2], in the proof of PTIME completeness of SLL, an important role is played by the notions of generic and homogeneous proofs, being respectively proofs without multiplexors (rule (m)) and with multiplexors of a fixed rank. Lafont uses homogeneous proofs for representing data and generic proofs for representing programs. We do not use this classification for proving the completeness of STA.

References

1. Girard, J.Y.: Light Linear Logic. Information and Computation 143(2), 175–204 (1998)
2. Lafont, Y.: Soft linear logic and polynomial time. Theoretical Computer Science 318(1-2), 163–180 (2004)
3. Baillot, P., Mogbil, V.: Soft lambda-calculus: a language for polynomial time computation. In: Walukiewicz, I. (ed.) FOSSACS 2004. LNCS, vol. 2987, pp. 27–41. Springer, Heidelberg (2004)
4. Baillot, P., Terui, K.: Light types for polynomial time computation in lambda-calculus. In: Proceedings of LICS 2004, pp. 266–275. IEEE Computer Society Press, Los Alamitos (2004)
5. Asperti, A.: Light Affine Logic. In: Proceedings of LICS 1998, pp. 300–308. IEEE Computer Society Press, Los Alamitos (1998)
6. Asperti, A., Roversi, L.: Intuitionistic Light Affine Logic. ACM Transactions on Computational Logic 3(1), 137–175 (2002)
7. Mairson, H.G., Terui, K.: On the Computational Complexity of Cut-Elimination in Linear Logic. In: Blundo, C., Laneve, C. (eds.) ICTCS 2003. LNCS, vol. 2841, pp. 23–36. Springer, Heidelberg (2003)
8. Ronchi Della Rocca, S., Roversi, L.: Lambda calculus and Intuitionistic Linear Logic. Studia Logica 59(3) (1997)
9. Coppola, P., Dal Lago, U., Ronchi Della Rocca, S.: Elementary Affine Logic and the Call by Value Lambda Calculus. In: Urzyczyn, P. (ed.) TLCA 2005. LNCS, vol. 3461, pp. 131–145. Springer, Heidelberg (2005)
10. Terui, K.: Light Affine Lambda Calculus and Polytime Strong Normalization. In: Proceedings of LICS 2001, pp. 209–220. IEEE Computer Society Press, Los Alamitos (2001)
11. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. Elsevier/North-Holland, Amsterdam, London, New York (1984)