Polynomial Time in the Parametric Lambda Calculus

Brian F. Redmond

Department of Computing, Mathematics and Statistical Sciences
Grande Prairie Regional College
10726 – 106 Avenue, Grande Prairie, AB, T8V 4C4, Canada
bredmond@GPRC.ab.ca

Abstract

In this paper we investigate Implicit Computational Complexity via the parametric lambda calculus of Ronchi Della Rocca and Paolini [13]. We show that a particular instantiation of the set of input values leads to a characterization of polynomial time computations in a similar way to Lafont's Soft Linear Logic [9]. This characterization is manifestly type-free and does not require any ad hoc extensions to the pure lambda calculus. Moreover, there is a natural extension to nondeterminism with the addition of explicit products.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Parametric Lambda Calculus, Polynomial Time Complexity, Combinators, Nondeterminism and Explicit Products, Implicit Computational Complexity

Digital Object Identifier 10.4230/LIPIcs.TLCA.2015.288

1 Introduction

There is an inherent problem in studying computational complexity within the lambda calculus in that a single beta reduction step can effectively square the size of a term, thus leading to an exponential size term after only a linear number of steps. It therefore seems unreasonable to simply count the number of beta reduction steps as a measure of the computational complexity of reduction. Indeed, there have been studies, most recently in [1], which give more reasonable measures of the complexity of a given λ-term using explicit substitutions and notions of sharing. These delicate issues are entirely avoided here as all beta reduction steps, except for a constant number (which does not depend on the size of the input), do not increase the size of the lambda term. Therefore, we feel justified in taking a very simplistic approach to measuring the complexity of reduction of a given lambda term. We define a simple "by-value" operational semantics and define the complexity of reduction as the size of its corresponding evaluation tree, which we show is polynomial in the size of the input binary word. We claim that any such reduction can be simulated on a Turing machine with polynomial overhead.

Most studies of complexity within the (pure) lambda calculus rely on typing restrictions to ensure that terms are strongly normalizing. For example, it is well known that terms in the simply typed lambda calculus are strongly normalizing and, moreover, that the class of representable numerical functions is precisely the class of *extended polynomials* [15]. Adding an (impredicative) operation of abstraction on types, as in Girard's system **F** [6], greatly increases the class of representable functions to the class of functions provably total in second order Peano arithmetic [7]. Nevertheless, the system remains strongly normalizing. A system somewhere in the middle of these two extremes is obtained by stratifying type abstraction into a finite number of levels. Indeed, in this case, the class of representable functions is precisely the class of super-elementary functions, i.e. the class of $\mathcal{E}_4$ in Grzegorczyk's subrecursive hierarchy [10]. Finally, there are also well known studies on the complexity of beta equivalence in the simply typed lambda calculus [16], as well as related results when restricting to low functional orders [14].

Another fruitful approach to investigating complexity in the lambda calculus is through linear logic and related type systems. In these studies, it is not typically type abstraction that is limited, but instead duplication is controlled via the modified exponentials ! and § (see [5] and [2], for example). These type systems were derived from their corresponding systems of (light) linear logic and proof nets, and illustrate the computational power of duplication in the lambda calculus. Unfortunately, type checking and type inference are undecidable in the presence of unrestricted polymorphism [4].

In this paper we take an entirely different approach to studying complexity within the lambda calculus via the parametric lambda calculus [13]. With a frugal choice of so-called input values, we show that strong normalization is guaranteed, yet the system remains expressive enough to capture polynomial time computations. Moreover, there is a natural extension to nondeterminism with the addition of explicit products. In contrast to the above studies, however, this approach does not rely on typing restrictions as we work in an entirely type-free setting. Nevertheless, we believe a system of intersection types can be introduced post hoc if desired.

This work is closely related to the author's work on Bounded Combinatory Logic [12]. In that work, the usual Curry combinators B, C, K, W are introduced, but the duplication combinator W has one of its arguments restricted to a proper subset of combinators, namely the BCK-combinators. This ensures that only *affine linear* terms are duplicated, and leads to a simple characterization of polynomial time computations. The "moral" analogue in the lambda calculus corresponds to a particular instantiation of the parametric lambda calculus, which is the focus of the current paper. However, the systems are not equivalent and in fact use completely different reduction strategies and encodings. For this reason, we chose to present this work independently and investigate the relationship between the two system in future work.

2 An Instance of the Parametric Lambda Calculus

One of the aims of the parametric lambda calculus is to study in a uniform way various systems of the pure lambda calculus, in particular its call-by-name and call-by-value versions [13]. This is done by restricting beta reduction to subsets of lambda terms, called input values, that satisfy certain closure conditions. These closure conditions guarantee important properties like confluence are satisfied. In this paper we study various instantiations of the parametric lambda calculus in the context of Implicit Computational Complexity (ICC). We refer the reader to [13] for much of the notation used as well as some of the basic definitions in the lambda calculus. However, all nonstandard notation, terminology, and definitions will be explicitly stated.

The set of lambda terms, Λ, is defined in the usual manner. We assume a countably infinite set of variables, Var, and define the set of lambda terms Λ as follows:

$$M, N ::= x \mid (MN) \mid (\lambda x.M)$$

where $x \in \text{Var}$. For notational convenience we tacitly assume the standard conventions regarding parentheses and for contracting multiple lambda abstractions. We use the symbol $\equiv$ to denote the syntactical identity of terms up to α-congruence, and $M[N/x]$ denotes the capture-free substitution of N for x in M. Finally, we need the notion of a context $C[.]$ as defined in [13], Def. 1.1.11.

Let M be a generic lambda term. For each subterm of M of the form $\lambda x.P$, let $\#(\lambda x.P)$ denote the number of times x occurs free in P. The **size**[1] of a lambda term M, denoted $s(M)$, is defined by induction on M as follows: $s(x) = 1$ if x is a variable, $s(\lambda x.P) = 1 + s(P)$ and $s(PQ) = s(P) + s(Q)$. Recall from [13] that a subset Δ of Λ is called a set of **input values** if it satisfies the following three conditions:

1. $\mathrm{Var} \subseteq \Delta$
2. If P and Q are in Δ, then so is $P[Q/x]$, for each $x \in \mathrm{Var}$
3. If $M \in \Delta$ and $M \rightarrow_\Delta N$, then $N \in \Delta$

In condition 3, the symbol $\rightarrow_\Delta$ denotes a one-step Δ-reduction [13].

In this paper we introduce a set of input values, denoted Φ, consisting of "pseudo-affine" terms. This is made precise in the following definition: Let Φ denote the smallest subset of Λ satisfying the following closure properties:

- $x \in \mathrm{Var}$ implies $x \in \Phi$,
- If $M, N \in \Phi$, then $(MN) \in \Phi$,
- If $M \in \Phi$ and $x \in \mathrm{Var}$ and x occurs free at most once in M, then $(\lambda x.M) \in \Phi$.

We shall often refer to Φ as the set of **player** terms and to $\Lambda \backslash \Phi$ as the set of **opponent** terms. For example, the closed terms $I \equiv \lambda x.x, B \equiv \lambda xyz.x(yz), C \equiv \lambda xyz.xzy$, and $K \equiv \lambda xy.x$ are all player terms, but $D \equiv \lambda x.xx$ is an opponent term. Note that *free* variables may appear more than once in a player term. This follows terminology introduced in [12].

The one-step Φ-reduction $\rightarrow_\Phi$ is defined as the contextual closure of the following rule:

$$(\lambda x.M)N \rightarrow_\Phi M[N/x] \quad \text{if and only if} \quad N \in \Phi$$

Let $\rightarrow_\Phi^*$ denote the reflexive and transitive closure of $\rightarrow_\Phi$. A redex $R \equiv (\lambda x.P)Q$ with $(\lambda x.P)$ an opponent term and $Q \in \Phi$ is called an **opponent redex**. A redex $R \equiv (\lambda x.P)Q \in \Phi$ is called a **player redex**. This terminology extends to their respective reductions as well.

Define the **degree** of a term M, denoted $d(M)$, as follows: $d(M) = 0$ if $M \in \Phi$ and $d(MN) = d(M) + d(N)$ if $MN \in \Lambda \backslash \Phi$ and $d(\lambda x.P) = 1 + d(P)$ if $\lambda x.P \in \Lambda \backslash \Phi$. For example, the degree of the term $\lambda x.(\lambda y.yy)x$ is 2.

▶ **Theorem 1** (confluence). *The subset Φ as defined above is a set of input values. Therefore, the reduction relation $\rightarrow_\Phi^*$ is confluent by Theorem 1.2.5 in [13].*[2]

Proof. Condition 1 is obvious. For condition 2, we use induction on the structure of P:

- If $P \equiv y$, then $P[Q/x]$ is either $Q \in \Phi$ if $y \equiv x$, or $y \in \Phi$ if $y \not\equiv x$.
- If $P \equiv P_1P_2 \in \Phi$, then $P_1 \in \Phi$ and $P_2 \in \Phi$. By the induction hypothesis, we have $P_1[Q/x], P_2[Q/x] \in \Phi$. Thus, $P[Q/x] \equiv P_1[Q/x]P_2[Q/x] \in \Phi$.
- If $P \equiv \lambda z.M \in \Phi$, then $M \in \Phi$ and z occurs free at most once in M. If $z \equiv x$, then $P[Q/x] \equiv P \in \Phi$. Otherwise, by α-congruence, we may assume that $z \notin \mathrm{FV}(Q)$, so z occurs free at most once in $M[Q/x]$. By the induction hypothesis we have $M[Q/x] \in \Phi$. Thus, $P[Q/x] \equiv \lambda z.M[Q/x] \in \Phi$.

Finally, for condition 3, we proceed by induction on the structure of M:

- If M is a variable, then the result is vacuously true.

[1] This is called *length* in [8].
[2] The set $\mathrm{Var} \cup \Phi^0$, where Φ^0 denotes the set of closed terms in Φ, also forms a set of input values. However, as we shall see in Section 3.1, pseudo-affine terms provide more flexibility when defining programs.

- If $M \equiv P_1P_2 \in \Phi$ and the Φ-redex/reduction is entirely in P_i ($i \in \{1,2\}$), then $P_i \to_\Phi P'_i$. By the induction hypothesis $P'_i \in \Phi$, so either $N \equiv P'_1P_2 \in \Phi$ or $N \equiv P_1P'_2 \in \Phi$. On the other hand, if $M \equiv (\lambda x.P)Q$ and $(\lambda x.P)Q \to_\Phi P[Q/x] \equiv N$, then $N \in \Phi$ by condition 2.
- If $M \equiv \lambda x.P \in \Phi$, then x occurs free in $P \in \Phi$ at most once, and $P \to_\Phi P'$. By the induction hypothesis, $P' \in \Phi$. Moreover, since the redex contracted in $P \in \Phi$ must be a player redex, the number of times x occurs free in P' is no more than 1. Thus, $\lambda x.P' \in \Phi$.

Therefore, Φ is a valid set of input values. ◀

Note that any term in the $\lambda\Phi$-calculus has the form: $\lambda x_1 \cdots x_n.\zeta M_1 \cdots M_m$ ($n, m \geq 0$) where ζ is either a variable or a Φ-redex or a head block. (A head block is a term $(\lambda x.P)Q$, where $Q \notin \Phi$.) We say that a term is in Φ-normal form if it has the form $\lambda x_1 \cdots x_n.\zeta M_1 \cdots M_m$, where M_i is in Φ-normal form ($1 \leq i \leq m$) and ζ is either a variable or a head block $(\lambda x.P)Q$, where both P and Q are in Φ-normal form. Note that a lambda term is in Φ-normal form iff it contains no Φ-redexes.

▶ **Lemma 2.** *If P is any term and Q is a player term, then $d(P) = d(P[Q/x])$.*

Proof. First note that if P is a player term, then so is $P[Q/x]$ and both have degree 0. So assume P is an opponent term and argue by induction on P. If $P \equiv P_1P_2$, then by the induction hypothesis, $d(P_1[Q/x]) = d(P_1)$ and $d(P_2[Q/x]) = d(P_2)$. Thus, $d(P[Q/x]) = d(P_1[Q/x]P_2[Q/x]) = d(P_1[Q/x]) + d(P_2[Q/x]) = d(P_1) + d(P_2) = d(P)$. If $P \equiv \lambda z.P_1$, then $d(P) = 1 + d(P_1)$. If $z \equiv x$, then $P[Q/x] \equiv P$ and the result follows. If $z \not\equiv x$, then $P[Q/x] \equiv \lambda z.P_1[Q/x]$ where we may assume by α-congruence that $z \notin \mathrm{FV}(Q)$. Then by the induction hypothesis, we have $d(P_1) = d(P_1[Q/x])$. Thus, $d(P) = 1 + d(P_1) = 1 + d(P_1[Q/x]) = d(P[Q/x])$. ◀

▶ **Theorem 3** (strong normalization)**.** *The $\lambda\Phi$-calculus is strongly normalizing.*

Proof. Let M be an arbitrary term. Let $R \equiv (\lambda x.P)Q$ be a redex in M and let M change to M' by contracting R. If R is an opponent redex, then by induction on M we show that $d(M') < d(M)$:

- If $M \equiv N_1N_2$, and R is contained in N_i, then let N_i change to N'_i by contracting R. By the induction hypothesis, $d(N'_i) < d(N_i)$. Thus, $d(M') < d(M)$. If $M \equiv R$, then $M' \equiv P[Q/x]$. Then $d(M') = d(P) < 1 + d(P) = d(M)$, where the first equality follows from Lemma 2.
- If $M \equiv \lambda z.N$, then R must be contained in N. Let N change to N' by contracting R, so $M' \equiv \lambda z.N'$. By the induction hypothesis, we have $d(N') < d(N)$. Thus, $d(M') \leq 1 + d(N') < 1 + d(N) = d(M)$.

On the other hand, if R is a player redex, then $s(M') < s(M)$ and $d(M') \leq d(M)$. Therefore, in between each of the at most $d(M)$ opponent reductions, there can be at most a finite number of player reductions, and reduction always terminates. ◀

It is useful to have a rough estimate on the complexity of reduction. Note that since each opponent term can at most square the size of the term, the size of any reduct is bounded by $s(M)^{2^d}$, where $d = d(M)$. Therefore, in between each of the at most $d = d(M)$ opponent reductions, there can be at most $s(M)^{2^d}$ player reductions. This leads to normalization in *double exponential time*, $s(M)^{2^{O(d)}}$.

There is some flexibility in what to take as the set of **output values**, Θ (see [13]). At the very least, the set of Φ-normal forms, denoted Φ-NF, should be contained in Θ. For the purposes of this paper, it suffices to assume that $\Theta = \Phi$-NF. In the next section we shall represent polynomial time algorithms by programs in the $\lambda\Phi$-calculus.

3 The Parametric Lambda Calculus and Polynomial Time

To begin we must represent some basic data structures like booleans and boolean strings in the $\lambda\Phi$-calculus, but we shall not require them to be encoded as player terms, so they are not necessarily input values[3]. Other data structures will be introduced as needed.

Booleans

Booleans are represented by the set $\{True, False\}$ and the conditional $Cond$, where:

$$True \equiv \lambda xy.x, \qquad False \equiv \lambda xy.y, \qquad Cond \equiv I$$

Note that $CondTrueMN \rightarrow^*_\Phi M$ and $CondFalseMN \rightarrow^*_\Phi N$ for any $M, N \in \Phi$, and that the terms $True, False, Cond$ belong to $\Phi \cap \Theta$.

Boolean Strings

Boolean strings $w = b_1 b_2 \cdots b_n \in \{0,1\}^*$, are represented using a Church-style encoding as follows:

$$W \equiv \lambda fx.fa_1(fa_2(\cdots fa_{n-1}(fa_n x)\cdots)), \qquad a_i \equiv \begin{cases} zero \equiv \lambda xyz.y, & b_i = 0 \\ one \equiv \lambda xyz.z, & b_i = 1 \end{cases}$$

For example, the boolean string $1011 \in \{0,1\}^*$ is encoded by the opponent term:

$$\lambda fx.fone(fzero(fone(fonex)))$$

Note that $d(W) \leq 1$ for all string encodings W; this fact will be important in the proof of Theorem 5. Note that boolean strings are in Φ-normal form, but they are not necessarily input values. Thus, boolean strings are not in general duplicable in our setting. For this reason a slightly more general notion of representability is required:

▶ **Definition 4.** A predicate $A \subseteq \{0,1\}^*$ is representable in the $\lambda\Phi$-calculus if there exists a context $C[.]$ such that, for all $w \in \{0,1\}^*$, $C[W] \rightarrow^*_\Phi Bool$, where W is the encoding of w (as defined above) and $Bool \equiv True$ if $w \in A$ and $Bool \equiv False$ if $w \notin A$. In this case, the context $C[.]$ is said to represent the predicate in the $\lambda\Phi$-calculus.

We now prove one of the main results of this paper:

▶ **Theorem 5** (soundness). *Let $C[.]$ represent a predicate in the $\lambda\Phi$-calculus. Then, for all $w \in \{0,1\}^*$, the term $C[W]$, where W is the encoding of w, reduces to $True$ or $False$ in time polynomial in the size/length, denoted $|w|$, of w.*

Proof. We define a simple call-by-value operational semantics that proves judgements of the

[3] An alternative system in which all of the required basic data structures are input values will be sketched in the conclusion.

form $M \Downarrow_{\mathbf{V}} N$, where M is any lambda term and N is a term in Φ-normal form:

$$\frac{(M_i \Downarrow_{\mathbf{V}} N_i)_{i\leq m}}{xM_1\cdots M_m \Downarrow_{\mathbf{V}} xN_1\cdots N_m}\ (var)$$

$$\frac{M \Downarrow_{\mathbf{V}} N}{\lambda x.M \Downarrow_{\mathbf{V}} \lambda x.N}\ (abs)$$

$$\frac{Q \Downarrow_{\mathbf{V}} Q' \quad Q' \in \Phi \quad P[Q'/x]M_1\cdots M_m \Downarrow_{\mathbf{V}} N}{(\lambda x.P)QM_1\cdots M_m \Downarrow_{\mathbf{V}} N}\ (head)$$

$$\frac{Q \Downarrow_{\mathbf{V}} Q' \quad Q' \notin \Phi \quad P \Downarrow_{\mathbf{V}} P' \quad (M_i \Downarrow_{\mathbf{V}} N_i)_{i\leq m}}{(\lambda x.P)QM_1\cdots M_m \Downarrow_{\mathbf{V}} (\lambda x.P')Q'N_1\cdots N_m}\ (block)$$

An easy induction on the evaluation tree shows that if $M \Downarrow_{\mathbf{V}} N$, then $M \to_{\Phi}^* N$, where N is in Φ-normal form, and the length of this reduction sequence is bounded by the size of the evaluation tree.

Our runtime bound is therefore obtained by bounding the size of the (rooted) evaluation tree – i.e. the total number of rules in the evaluation tree for the judgement $M \Downarrow_{\mathbf{V}} N$. Let n denote the total number of $(head)$-rules contained in such a tree. We show by induction on the height of the tree that the size of the tree is bounded by $(n+1)h$, where h denotes the maximum size of any reduct of M. To this end, we define a partial order on the vertices (proof rules) of the evaluation tree such that $u \leq v$ iff the unique path from the root to v passes through u. Consider the subtree obtained by removing the *right* branch of each of the $p \geq 0$ occurences of minimal (with respect to the above partial order) $(head)$-rules. This subtree has size bounded by $s(M) \leq h$ as each rule in the subtree decreases the size of the term. By the induction hypothesis, each of the removed branches, which end with judgements of the form $P[Q'/x]M_1\cdots M_m \Downarrow_{\mathbf{V}} N$, has size bounded by $(n_i+1)h$, where n_i $(1 \leq i \leq p)$ denotes the number of $(head)$-rules in branch i. Therefore, the total size of the evaluation tree is bounded by $h + \sum_{i\leq p}(n_i+1)h = (1+\sum_{i\leq p} n_i + p)h = (1+n)h$, as claimed.

Therefore, it follows by the discussion immediately following the proof of Theorem 3 that the size of the evaluation tree is bounded by $s(C[W])^{2^{O(d)}}$, where d is the degree of $C[W]$, as both n are h are so bounded. And since $s(C[W]) = O(|w|)$ and d remains fixed, the bound is in fact polynomial in $|w|$ (albeit with possibly large degree). Therefore, our reduction machine, as defined by the operational semantics above, runs in polynomial time. ◀

3.1 Polynomial Time Completeness

In this section we shall use the notation $M^n N \equiv N$, if $n = 0$, and $M^{n+1}N \equiv M(M^n N)$, if $n > 0$.

We begin by defining a context $\mathit{Iter}_{P,M}[.]$ which is used to iterate a player term $M \in \Phi$ a polynomial $P(n)$ number of times. More precisely, we claim:

$$\mathit{Iter}_{P,M}[W] \to_{\Phi}^* \lambda y.M^{P(n)}y \in \Phi \qquad (1)$$

where $n = |w|$. Recall that any polynomial with natural number coefficients can be represented in *Horner normal form*. For example, the polynomial $2n^3 + 4n^2 + 3n + 5$ is represented in Horner normal form as $(((2)n+4)n+3)n+5$. Given a polynomial $P(n)$, which is either a constant a_0 or has the form $P(n) = P_1(n)n + a_0$, where $P_1(n)$ is in Horner normal form, the context $\mathit{Iter}_{P,M}[.]$ is defined inductively on the structure of P as follows:

$$\begin{aligned} \mathit{Iter}_{a_0,M}[.] &\equiv \lambda y.M^{a_0}y \\ \mathit{Iter}_{P,M}[.] &\equiv \lambda y.M^{a_0}([.](K\,\mathit{Iter}_{P_1,M}[.])y) \qquad (K \equiv \lambda xy.x) \end{aligned}$$

We argue by induction on the structure of the polynomial that this context satisfies reduction (1). Indeed, the base case is clear. Otherwise we suppose $P(n) = (P_1(n))n + a_0$. By the induction hypothesis we have that $\mathit{Iter}_{P_1,M}[W] \rightarrow^*_\Phi \lambda y.M^{P_1(n)}y \in \Phi$, where $n = |w|$. Then:

$$\begin{aligned}
\mathit{Iter}_{P,M}[W] &\equiv \lambda x.M^{a_0}(W(K\mathit{Iter}_{P_1,M}[W])x) \\
&\rightarrow^*_\Phi \lambda x.M^{a_0}(W(K\lambda y.M^{P_1(n)}y)x) \\
&\rightarrow^*_\Phi \lambda x.M^{a_0}((\lambda y.M^{P_1(n)}y)^n x) \\
&\rightarrow^*_\Phi \lambda x.M^{a_0}(M^{(P_1(n))n}x) \\
&\equiv \lambda x.M^{(P_1(n))n+a_0}x \\
&\equiv \lambda x.M^{P(n)}x \in \Phi
\end{aligned}$$

This finishes the induction. Therefore, for any player term N, we have:

$$\mathit{Iter}_{P,M}[W]N \rightarrow^*_\Phi M^{P(n)}N \in \Phi$$

Note, in particular, that $\lambda f.\mathit{Iter}_{P,f}[W] \rightarrow^*_\Phi \lambda fx.f^{P(n)}x$, which is the Church representation of $P(n)$. We shall use this iteration combinator to iterate the transition function of a space-bounded Turing machine a polynomial number of times. But first we need the following preliminaries.

Affine Tensor Products

Given player terms $N_1, \ldots, N_m \in \Phi$, we write $N_1 \otimes \cdots \otimes N_m$ for the term $\lambda x.xN_1 \cdots N_m$ and Prj_i for the term $\lambda f.f(\lambda x_1 \ldots x_m.x_i)$ for each $1 \leq i \leq m$. Note that $N_1 \otimes \cdots \otimes N_m, Prj_i \in \Phi$ and satisfy: $Prj_i N_1 \otimes \cdots \otimes N_m \rightarrow^*_\Phi N_i$ for each $1 \leq i \leq m$. Occasionally we shall also use the notation $\lambda x_1 \otimes \cdots \otimes x_m.M$ for the term $\lambda f.f(\lambda x_1 \ldots x_m.M)$. This satisfies $(\lambda x_1 \otimes \cdots \otimes x_m.M)(N_1 \otimes \cdots \otimes N_m) \rightarrow^*_\Phi M[N_1/x_1] \cdots [N_m/x_m]$ for any terms $M, N_1, \ldots, N_m \in \Phi$. For example, Prj_i could be written instead as $\lambda x_1 \otimes \cdots \otimes x_m.x_i$ for each $1 \leq i \leq m$.

Lists

Lists are encoded using player terms for the constructors *nil* and *cons* as follows:

$$\mathit{nil} \equiv \lambda xy.y \qquad H :: T \equiv \lambda x.xHT \equiv H \otimes T$$

Note that $\mathit{nil} \in \Phi$ and $H :: T \in \Phi$ iff $H, T \in \Phi$. We shall assume that "::" associates to the right.

Space-Bounded Turing Machines

Suppose we are given a Turing machine with k states, tape alphabet $\{\sqcup, 0, 1\}$, and input alphabet $\{0, 1\}$, where $\sqcup$ is a special symbol for "blank". These three symbols are encoded, respectively, by the terms $\mathit{blank} \equiv \lambda xyz.x$, $\mathit{zero} \equiv \lambda xyz.y$ and $\mathit{one} \equiv \lambda xyz.z$. For convenience, we shall assume that the set of states always contains (distinct) special *accepting* and *rejecting* states, and that the machine cannot change states once it reaches one of these two terminating states. Moreover, we assume that the tape is sufficiently large so that either the accepting or rejecting state is always reached before the machine encounters either end of the tape.

A configuration of the Turing machine is encoded as an affine triple tensor product, $S \otimes L \otimes R$, where S encodes the current state, L encodes the left part of the tape (in reverse order), and R encodes the right part of the tape. The head of the Turing machine is always assumed to be positioned on the head of R. The left and right parts of the tape are encoded as lists of the tape alphabet. The k states S_i are encoded as follows. Suppose, for example, in state i, the machine's instructions, upon reading the symbol on the head, are:

$$\begin{array}{lcl} \sqcup & \mapsto & (s_{j1}, 0, \mathit{move\_right}) \\ 0 & \mapsto & (s_{j2}, 1, \mathit{move\_left}) \\ 1 & \mapsto & (s_{j3}, \sqcup, \mathit{move\_left}) \end{array}$$

Then state S_i is encoded as follows:

$$S_i \equiv \lambda x_1 \dots x_k, h_1 \otimes t_1, h_2 \otimes t_2 . h_2(F_1)(F_2)(F_3)x_1 \cdots x_k h_1 t_1 t_2$$

$$\begin{array}{lcl} F_1 & \equiv & \lambda x_1 \dots x_k h_1 t_1 t_2 . x_{j1} \otimes (\mathit{zero} :: h_1 :: t_1) \otimes t_2 \\ F_2 & \equiv & \lambda x_1 \dots x_k h_1 t_1 t_2 . x_{j2} \otimes t_1 \otimes (h_1 :: \mathit{one} :: t_2) \\ F_3 & \equiv & \lambda x_1 \dots x_k h_1 t_1 t_2 . x_{j3} \otimes t_1 \otimes (h_1 :: \mathit{blank} :: t_2) \end{array}$$

The transition function is then defined as $T \equiv \lambda z \otimes l \otimes r. z S_1 \cdots S_k l r$ and satisfies $T(S_i \otimes L \otimes R) \to^*_\Phi S_j \otimes L' \otimes R'$, where S_j is the new current state and L' and R' are encodings of the updated left and right parts of the tape after one iteration of the machine. The special accepting and rejecting states simply remain in the same state and write *one* or *zero*, respectively, on the head of the tape. Finally, observe that there is a term *out* (encoded via appropriate projections) that returns the head of the right tape from a given configuration of the machine. Note that all the terms in this encoding belong to Φ.

▶ **Theorem 6** (completeness). *If a predicate is computable by a Turing machine in polynomial time $P(n)$ and polynomial space $Q(n)$*[4]*, then it is representable in the $\lambda\Phi$-calculus by a context $C[.]$.*

Proof. Observe that there is a player term, denoted *pad*, which satisfies $\mathit{pad}(S \otimes L \otimes R) \to^*_\Phi S \otimes (\mathit{blank} :: L) \otimes (\mathit{blank} :: R)$. Then $\mathit{Iter}_{Q,\mathit{pad}}[W](S_1 \otimes \mathit{nil} \otimes \mathit{nil})$ pads out the tape sufficiently for the full computation of the machine and puts it in the initial state S_1. Next, apply a player context *write* such that $\mathit{write}[W](S \otimes L \otimes R) \to^*_\Phi S \otimes L \otimes (W(\lambda xy.(x :: y))R)$, which writes the binary string w onto the right part of the tape. Now apply $\mathit{Iter}_{P,T}[W]$, where T is the encoding of the transition function of the machine (as described above), to iterate the transition function $P(|w|)$ times, which suffices for the machine to reach a terminating state. Finally, use the player term *out* (mentioned above) to return the head of the right tape R and apply it to the arguments $I, \mathit{False}, \mathit{True}$ to get a boolean value which indicates whether the machine is in an accepting or a rejecting state. ◀

4 Various Extensions

In this section we investigate a few natural extensions of the $\lambda\Phi$-calculus. The first extends the set of input values to include the so-called Φ-valuable terms (defined below). This larger set of input values allows for further flexibility in defining programs, but at the expense of strong normalization. The second extension adds explicit products to the language for the purpose of characterizing nondeterministic polynomial computations.

[4] Of course, the explicit space bound $Q(n)$ is not necessary here since one may simply take $Q(n) = P(n)$. The explicit accounting of both time and space resources in the encoding of TMs is similar to that given in [9].

4.1 Φ-valuability

In order to reduce a redex $(\lambda x.P)Q$ in the $\lambda\Phi$-calculus, it is first necessary to reduce Q to an input value. However, suppose we relax this condition as follows. Recall that a term M is called **Φ-valuable** if there is an $N \in \Phi$ such that $M \to_{\Phi}^{*} N$. Let Φ_v denote the set of Φ-valuable terms.

▶ **Theorem 7.** *Φ_v forms a set of input values such that $\Phi \subset \Phi_v$. Therefore, the reduction relation $\to_{\Phi_v}^{*}$ is confluent by Theorem 1.2.5 in [13].*

Proof. Condition 1 is immediate. For condition 2, let $P, Q \in \Phi_v$, so there are terms $P', Q' \in \Phi$ such that $P \to_{\Phi}^{*} P'$ and $Q \to_{\Phi}^{*} Q'$. Then $P[Q/x] \to_{\Phi}^{*} P'[Q'/x]$ (by Lemmas 1.2.21 and 1.2.22 in [13]). But then $P'[Q'/x]$ is in Φ by the substitution closure of Φ. Hence $P[Q/x] \in \Phi_v$. For condition 3, let $M \equiv C[(\lambda x.P)Q]$ and $N \equiv C[P[Q/x]]$. Then $Q \in \Phi_v$ in order for this reduction to happen. So there is a term $Q' \in \Phi$ such that $Q \to_{\Phi}^{*} Q'$. Then $M \to_{\Phi}^{*} C[(\lambda x.P)Q'] \to_{\Phi} C[P[Q'/x]]$. But since $M \in \Phi_v$, there is a term $M' \in \Phi$ such that $M \to_{\Phi}^{*} M'$. So by the confluence of $\to_{\Phi}^{*}$ there is a term $M'' \in \Phi$ such that $C[P[Q'/x]] \to_{\Phi}^{*} M''$. But then $N \to_{\Phi}^{*} C[P[Q'/x]] \to_{\Phi}^{*} M''$, which shows that $N \in \Phi_v$. ◀

The $\lambda\Phi_v$-calculus is *not* strongly normalizing. For example, consider the term $D \equiv \lambda y.KI(yy)I$, which is Φ-valuable since it reduces to $\lambda yx.x \in \Phi$. However, $DD \to_{\Phi_v} KI(DD)I \to_{\Phi_v} KI(KI(DD)I)I \to_{\Phi_v} \cdots$, which leads to an infinite reduction sequence. Nevertheless, every term in the $\lambda\Phi_v$-calculus has a (unique) normal form:

▶ **Theorem 8** (weak normalization)**.** *Every term in the $\lambda\Phi_v$-calculus has a unique normal form.*

Proof. Let M be a term. If there are no Φ_v-redexes, then M is a Φ_v-nf and we are done. Otherwise, let $(\lambda x.P)Q$ be a Φ_v-redex in M. Then $Q \in \Phi_v$, and thus $Q \to_{\Phi}^{*} Q'$, where $Q' \in \Phi$. Consider the Φ-reduction of M: $M \equiv C[(\lambda x.P)Q] \to_{\Phi}^{*} C[(\lambda x.P)Q'] \to_{\Phi} C[P[Q'/x]]$. If $C[P[Q'/x]]$ is a Φ_v-nf, then we are done. Otherwise, repeat this procedure and extend the Φ-reduction of M. This process must terminate since there are no infinite Φ-reductions. Therefore, M is weakly normalizable. Uniqueness comes from the confluence property (Theorem 7). ◀

Of course, by including the Φ-valuable terms in the set of input values, we have not obtained any new complexity results. However, Theorem 7 is an interesting general result about the parametric lambda calculus that is true of any set of input values and may have future applications. Theorem 8 is less applicable as it requires the additional fact that the $\lambda\Phi$-calculus is strongly normalizing.

4.2 Explicit Products and Nondeterminism

In this section we study nondeterminism in the parametric lambda calculus with the use of explicit products. Of course, one could simply add a new term constructor $M + N$ (*sum*) together with nondeterministic projections, but the resulting system would not be confluent (by construction). Here we present an alternative approach based on the idea of a polynomial verifier.

Let $\Lambda_{\times}$ denote the set of lambda calculus with explicit products:

$$M, N ::= x \mid MN \mid \lambda x.M \mid \langle M, N\rangle \mid \pi_1 M \mid \pi_2 M$$

where $x \in \mathrm{Var}$. The notions of substitution and α-equivalence are extended in the obvious manner. Here, the term $\langle M, N\rangle$ is called an **explicit product** and $\pi_i M$ are called **projections**. For a given set of input values $\Delta \subseteq \Lambda_\times$, we extend $\to_\Delta$ as the contextual closure of the following rules:

$$\begin{array}{lll}(\lambda x.M)N & \to_\Delta & M[N/x] \quad \text{iff } N \in \Delta \\ \pi_1\langle M, N\rangle & \to_\Delta & M \\ \pi_2\langle M, N\rangle & \to_\Delta & N\end{array}$$

As before, we let $\to^*_\Delta$ denote the reflexive and transitive closure of $\to_\Delta$. The following is a straightforward generalization of Theorem 1.2.5 in [13].

▶ **Theorem 9.** *The $\lambda\Delta$-calculus with explicit products and $\to^*_\Delta$ as defined above is confluent.*

Proof. The definitions of deterministic parallel reduction $\hookrightarrow_\Delta$ and nondeterminsitic parallel reduction $\Rightarrow_\Delta$, Definition 1.2.19 in [13], are extended with the following clauses:

5. $M \hookrightarrow_\Delta M'$, $N \hookrightarrow_\Delta N'$ imply $\langle M, N\rangle \hookrightarrow_\Delta \langle M', N'\rangle$;
6. $M \hookrightarrow_\Delta M'$ and $M \not\equiv \langle M_1, M_2\rangle$ imply $\pi_i M \hookrightarrow_\Delta \pi_i M'$, for $i \in \{1, 2\}$;
7. $M_1 \hookrightarrow_\Delta M_1'$, $M_2 \hookrightarrow_\Delta M_2'$ imply $\pi_i\langle M_1, M_2\rangle \hookrightarrow_\Delta M_i'$, for $i \in \{1, 2\}$.

5. $M \Rightarrow_\Delta M'$, $N \Rightarrow_\Delta N'$ imply $\langle M, N\rangle \Rightarrow_\Delta \langle M', N'\rangle$;
6. $M \Rightarrow_\Delta M'$ implies $\pi_i M \Rightarrow_\Delta \pi_i M'$, for $i \in \{1, 2\}$;
7. $M_1 \Rightarrow_\Delta M_1'$, $M_2 \Rightarrow_\Delta M_2'$ imply $\pi_i\langle M_1, M_2\rangle \Rightarrow_\Delta M_i'$, for $i \in \{1, 2\}$.

Then Lemmas 1.2.21, 1.2.22, Property 1.2.23 and Lemma 1.2.24 in [13] all have straightforward generalizations by checking the extra cases. The details are left to the reader. Finally, the proof of Lemma 1.2.25 and the rest of the proof of Theorem 1.2.5 are unchanged. ◀

We expand the set of input/player values to include affine linear terms with explicit products: Let $\Phi_\times$ be defined the smallest subset of $\Lambda_\times$ satisfying the following closure properties:

- $x \in \mathrm{Var}$ implies $x \in \Phi_\times$,
- If $M, N \in \Phi_\times$, then $(MN) \in \Phi_\times$,
- If $M \in \Phi_\times$, then $\pi_i M \in \Phi_\times$ for $i \in \{1, 2\}$,
- If $M, N \in \Phi_\times$, then $\langle M, N\rangle \in \Phi_\times$,
- If $M \in \Phi_\times$ and $x \in \mathrm{Var}$ and x occurs free at most once in M, then $(\lambda x.M) \in \Phi_\times$.

Note that if $M \in \Phi_\times$ and is closed, then M is affine linear even with (additive) explicit products. One could define a more general notion of input values based on the notion of *slice* (as defined in [11], for example) which includes terms like $\lambda x.\langle x, x\rangle$ and is strongly normalizing. However, such a system would require a lazy reduction strategy for explicit products as well as pointers to avoid an exponential explosion in the size of a term (cf. [12]). We don't believe this added complication is necessary here.[5]

The following two theorems are straightforward generalizations of Theorems 1 and 3, so the proofs have been omitted.

▶ **Theorem 10.** *$\Phi_\times$ forms a set of input values such that $\Phi \subset \Phi_\times$. Therefore, the reduction relation $\to^*_{\Phi_\times}$ is confluent by Theorem 9.*

▶ **Theorem 11.** *The $\lambda\Phi_\times$-calculus is strongly normalizing.*

[5] However, we do believe this more general set of input values is the starting point for a characterization of PSPACE (see [12]).

We extend the simple call-by-value operational semantics introduced in the proof of Theorem 5 with the following rules:

$$\frac{P_1 \Downarrow_{\mathbf{V}} P_1' \quad P_2 \Downarrow_{\mathbf{V}} P_2' \quad (M_i \Downarrow_{\mathbf{V}} N_i)_{i \leq m}}{\langle P_1, P_2\rangle M_1 \cdots M_m \Downarrow_{\mathbf{V}} \langle P_1', P_2'\rangle N_1 \cdots N_m}\ (pair)$$

$$\frac{Q \Downarrow_{\mathbf{V}} Q' \quad Q' \equiv \langle Q_1', Q_2'\rangle \quad Q_i' M_1 \cdots M_m \Downarrow_{\mathbf{V}} N}{\pi_i Q M_1 \cdots M_m \Downarrow_{\mathbf{V}} N}\ (proj_i)$$

$$\frac{Q \Downarrow_{\mathbf{V}} Q' \quad Q' \not\equiv \langle Q_1', Q_2'\rangle \quad (M_i \Downarrow_{\mathbf{V}} N_i)_{i \leq m}}{\pi_i Q M_1 \cdots M_m \Downarrow_{\mathbf{V}} \pi_i Q' N_1 \cdots N_m}\ (block_i)$$

Once again, an easy induction on the evaluation tree shows that if $M \Downarrow_{\mathbf{V}} N$, then $M \to^*_{\Phi_\times} N$, where N is in $\Phi_\times$-normal form, and the length of this reduction sequence is bounded by the size of the evaluation tree. On the other hand, by Theorem 10, if $M \to^*_{\Phi_\times} N$, where N is in $\Phi_\times$-normal form, then M evaluates to N according to the operational semantics defined above.

Additive Booleans

Explicit products allow for an alternative definition of booleans and conditional:

$$Proj_1 \equiv \lambda f.\pi_1 f \quad Proj_2 \equiv \lambda f.\pi_2 f \quad if\ b\ then\ M\ else\ N \equiv \lambda b.b\langle M, N\rangle$$

A term M is called **eventually true** if there exists a sequence of additive booleans $Proj_{i_1}, \ldots, Proj_{i_k}$ such that $M Proj_{i_1} \cdots Proj_{i_k} \to^*_{\Phi_\times} True$.

▶ **Definition 12.** A predicate $A \subseteq \{0,1\}^*$ is representable in the $\lambda\Phi_\times$-calculus if there is a context $C[.]$ such that, for all $w \in \{0,1\}^*$, $w \in A$ iff $C[W]$ is eventually true.

We have the following result:

▶ **Theorem 13.** *A predicate $A \subseteq \{0,1\}^*$ is representable in the $\lambda\Phi_\times$-calculus by a context $C[.]$ iff it is computable in nondeterministic polynomial time (NP).*

Proof. ($\Rightarrow$) Suppose a predicate A is representable by a context $C[.]$ in the $\lambda\Phi_\times$-calculus. If $w \in A$, then a straightforward generalization of Theorem 5 shows that, for any choice of projections $Proj_{i_1}, Proj_{i_2}, \ldots, Proj_{i_k}$, the term $C[W]Proj_{i_1}Proj_{i_2} \cdots Proj_{i_k}$ reduces in time bounded by a polynomial in $s(C[W]Proj_{i_1}Proj_{i_2} \cdots Proj_{i_k})$ to $True$.[6] Moreover, note that k must be bounded by a polynomial in $|w|$. Indeed, each projection input requires a head lambda abstraction. This head lambda abstraction cannot itself be a projection term because otherwise the normal form would have a (projection) block. And there can only be a polynomial in $|w|$ such head reductions. Therefore, the entire reduction is polynomial time in $|w|$ only.

($\Leftarrow$) Conversely, let A be a predicate computable on a nondeterministic Turing machine in polynomial time $P(n)$ and polynomial space $Q(n)$ (i.e. the maximum time and space used by any computational branch). The encoding of nondeterministic Turing machines is based on the encoding of deterministic Turing machines found in Section 3.1. However, for a nondeterministic machine, we assume a pair of transition functions T_l and T_r instead of just

[6] As noted above, we may assume, by Theorem 10, that this reduction sequence is determined by the operational semantics.

one. The following player term can be iterated $P(n)$ times using the iteration combinator from Section 3.1:

$$\mathit{Branch} \equiv \lambda fzb.b\langle \lambda xy.x(T_l y), \lambda xy.x(T_r y)\rangle fz$$

Let w be any binary word and let $n = |w|$. Let $\mathit{Initial}[W] \rightarrow^*_{\Phi_\times} \mathit{Config}$ initialize the machine by padding the tape out to size $Q(n)$, writing w on the initial segment of the tape, and putting it in the start state. Finally, let out be a term that reduces to True if a given configuration is accepting and reduces to False otherwise.

Let Z_k denote the normal form of $\mathit{Branch}^k \mathit{out}$, which has the form:

$$\begin{array}{lcll} Z_k & \equiv & \lambda zb.b\langle \lambda xy.x(T_l y), \lambda xy.x(T_r y)\rangle Z_{k-1} z, & k > 0 \\ Z_0 & \equiv & \mathit{out} & \end{array}$$

If $w \in A$, then there exists a sequence of $i_1, \ldots, i_k$, with $i_j \in \{1, 2\}$ and $k = P(n)$, specifying a path down the nondeterministic evaluation tree to an accepting leaf. This path is encoded by the series of projections $\mathit{Proj}_{i_1}, \ldots, \mathit{Proj}_{i_k}$ and verified as follows:

$$\begin{array}{ll} & \mathit{Iter}_{P,\mathit{Branch}}[W]\,\mathit{out}\,\mathit{Config}\,\mathit{Proj}_{i_1} \cdots \mathit{Proj}_{i_k} \\ \rightarrow^*_{\Phi_\times} & \mathit{Branch}^k\,\mathit{out}\,\mathit{Config}\,\mathit{Proj}_{i_1} \cdots \mathit{Proj}_{i_k} \\ \rightarrow^*_{\Phi_\times} & (\lambda b.b\langle \lambda xy.x(T_l y), \lambda xy.x(T_r y)\rangle Z_{k-1}\,\mathit{Config})\,\mathit{Proj}_{i_1} \cdots \mathit{Proj}_{i_k} \\ \rightarrow^*_{\Phi_\times} & \mathit{Proj}_{i_1}\langle \lambda xy.x(T_l y), \lambda xy.x(T_r y)\rangle Z_{k-1}\,\mathit{Config}\,\mathit{Proj}_{i_2} \cdots \mathit{Proj}_{i_k} \\ \rightarrow^*_{\Phi_\times} & Z_{k-1}\,\mathit{Config}_1\,\mathit{Proj}_{i_2} \cdots \mathit{Proj}_{i_k} \\ \rightarrow^*_{\Phi_\times} & \cdots \\ \rightarrow^*_{\Phi_\times} & Z_1\,\mathit{Config}_{k-1}\,\mathit{Proj}_{i_k} \\ \rightarrow^*_{\Phi_\times} & \mathit{out}\,\mathit{Config}_k \\ \rightarrow^*_{\Phi_\times} & \mathit{True} \end{array}$$

Moreover, this reduction proceeds according to the operational semantics defined above. Thus, $(\mathit{Iter}_{P,\mathit{Branch}}[W]\,\mathit{out}\,\mathit{Config})\,\mathit{Proj}_{i_1} \cdots \mathit{Proj}_{i_k} \Downarrow_{\mathbf{V}} \mathit{True}$. On the other hand, if $w \notin A$, then all such reductions reduce to *False*. ◀

5 Conclusion

In this paper we have demonstrated that a characterization of polynomial time computations can be obtained in the lambda calculus without requiring any typing information and/or ad hoc extensions to the language. Indeed, the characterization is obtained simply by restricting the set of input values to the so-called pseudo-affine terms. Moreover, a characterization of nondeterministic polynomial time is obtained with the addition of explicit products.

It would be interesting to investigate other (decidable) instantiations of the parametric lambda calculus in the context of Implicit Computational Complexity. For example, the choice to allow weakening in the language was made simply because it made the encoding of polynomial time TMs much more natural. However, if we change the final clause in the definition of Φ to specify that x occurs *exactly* once in P, we conjecture that this smaller set of input values also characterizes polynomial time. In this case, the encoding of polynomial bounded TMs might follow that in [11].

Finally, one unfortunate aspect of our characterization is that binary words (and Church numerals) are not in general input values. For this reason our definitions of representability use contexts instead, which is not standard. We consider two possibilites for dealing with this situation. First, one could use a player encoding of binary strings instead (in the style of the Barendregt numerals [3, 13]), together with a more natural definition of representability. However, it is not difficult to show that this leads to a system for linear time computations only. A second possibility is to use a system of *abstract binary numerals* (see [8]) instead of pure terms to represent binary strings. In this case, four new atomic constants, denoted $\epsilon, \sigma_1, \sigma_2$ and Z, are added to the system such that ϵ, σ_1 and σ_2 belong to the set of input values, but Z does not. Then any binary string $w \in \{0,1\}^*$ can be represented by an input value $\widehat{w}$ in the obvious way. Furthermore, we add the contextual closure of the following reduction rule:

$$Z\widehat{w} \to_\Phi \overline{w}$$

where $\overline{w}$ is the Church-style representation of w described in Section 3.[7] This *arithmetical extension* [8] of the $\lambda\Phi$-calculus leads to an *applied* system for polynomial time computations, similar to the pure system presented in this paper. It is an alternative if a standard notion of representability is desired.

Acknowledgements.

The author would like to thank Simona Ronchi Della Rocca for reading a preliminary draft of this paper and offering some useful suggestions for improvement. The author would also like to thank the anonymous referees for their valuable comments and suggestions regarding the presentation of the results of this paper.

References

1 Beniamino Accattoli and Ugo Dal Lago. Beta reduction is invariant, indeed. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS'14, Vienna, Austria, July 14–18, 2014*, page 8. ACM, 2014.

2 Patrick Baillot and Kazushige Terui. Light types for polynomial time computation in lambda-calculus. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 266–275. IEEE Computer Society, 2004.

3 Hendrik Pieter Barendregt. *The lambda calculus: its syntax and semantics.* Studies in logic and the foundations of mathematics. North-Holland, Amsterdam, New-York, Oxford, 1981.

4 J. Chrząszcz and A. Schubert. The role of polymorphism in the characterisation of complexity by soft types. (To appear).

5 Marco Gaboardi and Simona Ronchi Della Rocca. A soft type assignment system for *lambda*-calculus. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2007.

6 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'état, Université Paris 7, June 1972.

[7] Note that if Z is permitted to be an input value, then $\overline{w}$ must be as well. Otherwise, the system would not satisfy reduction closure (i.e. condition 3 in the definition of input values). This leads to (at least) elementary time complexity.

7 Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

8 J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, New York, NY, USA, 2 edition, 2008.

9 Yves Lafont. Soft linear logic and polynomial time. *Theor. Comput. Sci.*, 318(1-2):163–180, 2004.

10 Daniel Leivant. Finitely stratified polymorphism. *Inf. Comput.*, 93(1):93–113, 1991.

11 Harry G. Mairson and Kazushige Terui. On the computational complexity of cut-elimination in linear logic. In Carlo Blundo and Cosimo Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003, Bertinoro, Italy, October 13-15, 2003, Proceedings*, volume 2841 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2003.

12 B. Redmond. Bounded combinatory logic and lower complexity. (To appear).

13 Simona Ronchi Della Rocca and Luca Paolini. *The parametric lambda calculus: a metamodel for computation*. Texts in theoretical computer science. Springer-Verlag, New York, 2004.

14 Aleksy Schubert. The complexity of beta-reduction in low orders. In *TLCA*, pages 400–414, 2001.

15 H. Schwichtenberg. Definierbare Funktionen im λ-Kalkül mit Typen. *Arkhiv für mathematische Logik und Grundlagenforschung*, 17:113–114, 1976.

16 R. Statman. The typed lambda calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–82, 1979.