A Type-Theoretic Characterization of Polynomial Time Computable Functions

Master Thesis in Mathematics at the Radboud Universiteit Nijmegen

Rik de Kort

supervised by
Dr. Herman Geuvers

September 7, 2016
Word of thanks

I’d like to thank my supervisor, Herman Geuvers, for being incredibly patient and giving me the time and support I needed. I’d like to thank my friends and family, in particular my mother, for keeping me (mostly) sane during this time. Thank you, the reader, for taking the time to give my thesis a look.

Contents

1 Introduction ........................................ 2

2 Preliminaries .................................... 3

3 A simple cost model for the weak $\lambda$-calculus ........... 6
   3.1 Simulating Turing Machines ......................... 10
   3.2 Simulating Lambda Terms .......................... 18
   3.3 Space cost modification and extension to full $\lambda$-calculus . . . . . . . 23

4 The Type System DIAL$_{lin}$ and Main Result ............... 24

5 Completeness ................................... 31

6 Soundness .................................... 32
   6.1 Realizability ...................................... 32
   6.2 Size Realizability .................................. 44
   6.3 Putting it all together ............................ 53

7 Modifications and Extensions .......................... 55
   7.1 The linear premise for $\Rightarrow$ .................. 55
   7.2 Combining regular and size realizability .......... 58
   7.3 More general Church and Scott datatypes ......... 64

8 Conclusions and related work ....................... 67
1 Introduction

Together with Turing machines, Church’s λ-calculus was the first formal system capturing the notions of “computation”, and “algorithm”. It was quickly proven that λ-calculus and Turing machines are equivalent models of computation: any function that can be computed in one can also be computed in the other. The Church-Turing thesis extends this result to the statement that any model capturing the notions of “computation” and “algorithm” will be equivalent to Turing machines and the λ-calculus.

Despite their computational equivalence, both models are quite dissimilar: Turing machines admit simple, intuitive complexity measures, while the λ-calculus doesn’t, and the λ-calculus admits simple, intuitive type systems, while Turing machines don’t.

The issue of finding a complexity measure for the simple untyped λ-calculus is a very easily posed problem, but where do we start? The first question that needs to be answered is: what do we consider a “good” complexity measure? Given the Church-Turing thesis in computing theory, and the many links between computing theory and complexity theory, it seems natural to expect a complexity-theoretic version of the Church-Turing thesis. We will call this statement the Polynomial Invariance Thesis.

*Reasonable machines can simulate each other within a polynomially-bounded overhead in time and a constant-factor overhead in space.*

This angle has been thoroughly explored in the literature, in the work of Lévy, Lamping, and many others [25, 15, 16, 14, 2, 13]. Most of the complexity measures for the simple, untyped λ-calculus rely on operations with graphs, and aren’t very intuitive. It is difficult to combine them with other interesting possibilities in the λ-calculus such as the addition of type systems. So the question becomes, what is an intuitive complexity measure for the λ-calculus that satisfies the Polynomial Invariance Thesis?

An obvious candidate is the number of β-reduction steps done in normalizing a given λ-term. It was only recently proven [1] that this is indeed a “good” complexity measure. Preceding this result, however, Dal Lago and Martini [10] defined an intuitive complexity measure for the untyped λ-calculus with weak call-by-value reduction that satisfies the polynomial invariance thesis. The cost model takes as measure the size difference between terms before and after a reduction step, which intuitively corresponds to the amount of work one must do to calculate the result of the β-step. This model has the advantage of admitting a mostly straightforward proof.

Implicit computational complexity theory aims to characterize complexity classes by bounding the types of algorithms used rather than explicitly bounding the runtime or the amount of space used. One of the earliest results is due to Cobham [6], giving a functional characterization of polytime functions. Consequently Leivant [17, 18] developed a “nicer” way to characterize polytime
functions, and went on to produce an extensive series together with Marion [21, 20, 22, 23, 24], exploring the tiered recursion introduced in [19]. A similar idea is explored by Bellantoni and Cook in [3].

These ideas were consequently combined with the work of Martini and Dal Lago by Brunel and Terui [5] in a type-theoretic characterization of polynomial-time computable functions. More specifically, they outline a type system, called DIAL$_{lin}$, in which functions are polytime if and only if the corresponding $\lambda$-terms have a certain type.

In this Master Thesis, we will explore the complexity measure defined by Dal Lago and Martini, before moving onto the work of Brunel and Terui. We will stick relatively close to the referred articles, especially in the beginning. We will also use similar notation: in the first few section, we typically denote $\lambda$-terms by $M$ and $N$, whereas in the second part we will prefer $t, u$.

2 Preliminaries

The cost model we will be using for the most part of this thesis is based on the weak call-by-value $\lambda$-calculus. This variation was introduced by Plotkin [26]. It has the Church-Rosser property, which is essential in establishing our cost model.

**Definition 2.1 ($\lambda$-calculus).** $\lambda$-terms are given by the following grammar.

$$M ::= x \mid \lambda x. M \mid MM$$

where $x$ is a variable. The set of all $\lambda$-terms is denoted $\Lambda$.

For terms $\lambda x. M$ ($\lambda$-abstractions) we call $M$ the body of the term. occurrences of $x$ in $M$ are said to be bound in this case. If an occurrence of $x$ is not bound, we call it free. Generally we will say variables are bound or free, by which we mean that all their occurrences are bounded or free, respectively.

For any term $M$, $FV(M)$ denotes the set of free variables in $M$. This can be formally defined using recursion over the term structure.

$$FV(x) = x \quad FV(\lambda x. M) = FV(M) \setminus \{x\} \quad FV(MN) = FV(M) \cup FV(N)$$

If we have multiple $\lambda$-abstractions after eachother, we only write the first $\lambda$ followed by the list of variables being abstracted over.

$$\lambda x.\lambda y.x y = \lambda y. x y$$

*Values* are $\lambda$-terms given by the following grammar.

$$V ::= x \mid \lambda x. M$$

where $x$ is a variable and $M$ is a $\lambda$-term.
We can substitute terms for variables. \( M[N/x] \) denotes the term \( M \) with \( N \) substituted for \( x \). This can be formally defined by recursion over the term structure.

\[
x[V/x] = V, \quad y[V/x] = y
\]

\[
(MN)[V/x] = M[V/x]N[V/x], \quad (\lambda y.M)[V/x] = \lambda y.(M[V/x])
\]

where \( y \neq x \), and \( y \) does not occur in \( V \).

\( \beta \)-reduction is denoted by \( \to_\beta \) and is given by following rules.

\[
(\lambda x.M)V \to_\beta M[V/x] \quad (\beta\text{-step})
\]

\[
N \to_\beta N' \quad MN \to_\beta MN' \quad \text{(right compatibility)}
\]

\[
M \to_\beta M' \quad MN \to_\beta M'N \quad \text{(left compatibility)}
\]

where \( V \) must be a value.

A term \( M \) is in \textit{normal form} if there is no term \( N \) such that \( M \to_\beta N \), that is, \( M \) does not reduce any further. It is stated here without proof that if a term has a reduction path that ends in a normal form, then any other path reaching a normal form will reach the same normal form. Hence we write \( \llbracket M \rrbracket \) for the normal form of \( M \), if it exists.

Oftentimes we shall just write \( \to \) rather than \( \to_\beta \). It should be clear from the context what we mean.

Note that in this definition, we never reduce the body of a \( \lambda \)-term. This is what we mean when we say we are using \textit{weak} reduction. We also have \textit{strong} reduction where we allow reductions to occur in the body of a \( \lambda \)-term. This is done by introducing the following rule.

\[
M \to_\beta M' \quad \lambda x.M \to_\beta \lambda x.M' \quad \text{(\( \lambda \)-compatibility)}
\]

Also note that, since values don’t reduce anymore, we have to fully reduce an argument before applying it to a function. This is what we mean when we say we are using \textit{call-by-value} reduction. We can have arbitrary reduction by replacing the \( \beta \)-step rule by

\[
(\lambda x.M)N \to_\beta M[N/x] \quad (\beta\text{-step})
\]

where we have replaced the \( V \) by a \( N \).

Strong arbitrary reduction also has unique normal forms. We write \( \llbracket M \rrbracket_\beta \) if \( M \) for the normal form of \( M \), if it exists.

Unless otherwise stated, we will be using weak call-by-value reduction. Exceptions are Sections 6.2 and 7.2.
Definition 2.2 (Equivalence and equality). We write $M \equiv N$ if $M$ is the same as $N$ up to renaming of variables.

We write $M =_\beta N$ for the reflexive symmetric transitive closure of $\rightarrow_\beta$ in the strong sense.

Definition 2.3 (I). We denote

$$I = \lambda x. x$$

Definition 2.4 (Lengths and weights). 1. The length $|M|$ of a term $M$ is given by counting the number of symbols in $M$. This can be formally defined recursively.

$$|x| = 1, \quad |MN| = |M| + |N|, \quad |\lambda x. M| = 1 + |M|$$

2. We define the length $|\alpha|$ of a finite sequence of natural numbers in the usual way.

$$|\langle \rangle| = 0, |\langle n \rangle| = |\alpha| + 1$$

3. For a finite sequence of natural numbers $\alpha$, we define the weight of $\alpha$ to be

$$||\alpha|| = \sum_{n \in \alpha} n$$

In this definition, the 1 in $|\lambda x. M| = 1 + |M|$ is arbitrary. We also could’ve chosen some other number greater than 0.

Definition 2.5 (Finite strings). Let $S = \{s_0, \ldots, s_k\}$ be a finite set. The set of finite strings over $S$, denoted $S^*$, are finite sequences of symbols in $S$. We denote by $\Sigma$ the strings over $\{0, 1\}$. The empty string is denoted $\epsilon$. For a string $u$ the reverse of the string is notated $u^r$. We will denote the concatenation of two strings $u, v$ by $uv$. Concatenating to a single character $s$ a string $u$ is denoted $su$. $us$ for concatenating a single character to a string is similar.

All of these can be formally defined using recursion.
3 A simple cost model for the weak $\lambda$-calculus

We will introduce a simple cost model for the weak call-by-value $\lambda$-calculus. This model was first defined by Dal Lago and Martini in [10] and has the advantage of being “self-sustaining”: no references to graph reductions or the like are made. We define the cost of a $\lambda$-term and show that this is well-defined before showing that this cost model satisfies the polynomial invariance thesis.

Intuitively, the cost of a single $\beta$-step corresponds to the size-increase of the term: shuffling around parts of a term can be done pretty quickly, but when the size of the term increases, we have some actual copying to do and this takes time.

**Definition 3.1 (Cost model).** We define a relation $\alpha \overset{\text{uni21A0}}{\Rightarrow} \alpha$, where $\alpha$ is a finite sequence of natural numbers as follows, in an operational semantics style.

\[
\begin{align*}
\text{(reflect)} & \quad M \overset{\text{uni21A0}}{\Rightarrow} M \\
\text{(apply)} & \quad M \overset{\alpha \beta}{\Rightarrow} N \quad N \overset{\beta}{\Rightarrow} K \\
\text{(reduce)} & \quad M \overset{\text{reflect}}{\Rightarrow} N 
\end{align*}
\]

Here $\alpha \beta$ denotes the concatenation of $\alpha$ and $\beta$.

Let $M$ be a term with normal form $N$ such that $M \overset{\alpha}{\Rightarrow} N$. We define the cost of $M$ as

\[
\text{cost}(M) = |M| + ||\alpha||
\]

If $M$ does not have a normal form, we set

\[
\text{cost}(M) = \infty
\]

We note that it is not strictly necessary to work with lists of natural numbers, and instead can use the following rules.

\[
\begin{align*}
M \overset{\text{reflect}}{\Rightarrow} M \\
M \overset{\text{reduce}}{\Rightarrow} N \\
M \overset{\text{apply}}{\Rightarrow} K
\end{align*}
\]

However, the lists are helpful in proving the cost model well-defined, so we will keep using them.

We will now prove well-definedness, meaning there cannot exist a term $M$ such that $M \overset{\alpha}{\Rightarrow} N$ and $M \overset{\beta}{\Rightarrow} N$, but $||\alpha|| \neq ||\beta||$. We start with the Diamond Property, a statement of the strong Church-Rosser property of the weak call-by-value $\lambda$-calculus with respect to our cost model. Well-definedness is then a relatively easy consequence.
Lemma 3.2 (Diamond Property). If $M \xrightarrow{(n)} N$ and $M \xrightarrow{(l)} L$, then either $N \equiv L$ or there exists $P$ such that $N \xrightarrow{(l)} P$ and $L \xrightarrow{(n)} P$. In diagram form:

![Diagram](image)

Proof. By induction on the structure of $M$.

- If $M$ is a variable, $M$ does not reduce at all. Hence $M \equiv N \equiv L$ and the proposition holds.

- If $M$ is an abstraction, again $M$ does not reduce at all. Hence $M \equiv N \equiv L$ and the proposition holds.

- $M = QR$ for some terms $Q$ and $R$. There are several options for obtaining $M \xrightarrow{(n)} N$ and $M \xrightarrow{(l)} L$.

  - If $Q = \lambda x.T$ and $R$ a value, then we must have $N \equiv L \equiv T[R/x]$.
  - If the above case does not occur, only one of $Q, R$ can have been reduced to obtain $M \xrightarrow{(n)} N$ or $M \xrightarrow{(l)} L$: every reduction step increases the length of the associated list by one, the list has length 1, and we cannot do parallel reduction. This gives four options:
    
    * $N = TR$ and $L = UR$ (both in first). Here, we must have $Q \xrightarrow{(n)} T$ and $Q \xrightarrow{(l)} U$. By induction hypothesis, we have $T \equiv U$, or $W$ such that $T \xrightarrow{(n)} W$ and $U \xrightarrow{(n)} W$. In the former case, we have $N \equiv L$.
      
      In the latter, setting $P = WR$, we then have that $N = TR \xrightarrow{(n)} WR = P$ by right compatibility, and $L = UR \xrightarrow{(n)} WR = P$ by left compatibility, which is what we wanted.
    
    * $N = TR$ and $L = QU$ (first in $N$, second in $L$). Here, we must have $Q \xrightarrow{(n)} T$ and $R \xrightarrow{(l)} U$. Setting $P = TU$, we then obtain that $N = TR \xrightarrow{(l)} TU = P$ and $L = QU \xrightarrow{(n)} TU = P$, which is what we wanted.
    
    * $N = QT$ and $L = UR$ (second in $N$, first in $L$). This case is symmetric to the previous one.
    
    * $N = QT$ and $L = QU$ (both in second). This case is similar to the first case.

\[\boxdot\]
We will now show we can extend this diamond property to reductions of arbitrary length.

**Proposition 3.3** (Extended Diamond Property). If $M \xrightarrow{\alpha} N$ and $M \xrightarrow{\beta} L$, there exist $\gamma, \delta, K$ such that $N \xrightarrow{\gamma} K$, $L \xrightarrow{\delta} K$, $||\alpha\gamma|| = ||\beta\delta||$, $|\gamma| = |\beta|$, and $|\delta| = |\alpha|$.

**Proof.** The basic idea is to tile the diamonds from the Diamond Property to tie $\alpha$ and $\beta$ back together, as it were. Formally, the argument goes by induction on the lengths of $\alpha$ and $\beta$.

If $\alpha = \langle \rangle$, we must have $M = N$, since we cannot obtain the empty list any other way. Then $K = L$, $\gamma = \beta$, and $\delta = \langle \rangle$ satisfy the requirements since $N \xrightarrow{\beta} K$.

If $\beta = \langle \rangle$, similarly we can take $K = N$, $\gamma = \langle \rangle$, and $\delta = \alpha$.

Suppose we have $\alpha = \langle a \rangle \alpha'$, $\beta = \langle b \rangle \beta'$. Then there exists $N'$ such that $M \xrightarrow{\langle a \rangle} N'$. Similarly, there exists $L'$ such that $M \xrightarrow{\langle b \rangle} L'$. Applying the Diamond Property, we see that either $N' = L'$, in which case the induction hypothesis immediately yields the conclusion, or $N' \neq L'$, in which case we obtain the following diagram.

![Diagram](image)

For the sake of readability, we say that a square commutes if and only if its sides satisfy a relation like the one we’re trying to prove. We’re currently trying to prove the outermost square commutes.

Square I is just a visualisation of the second case of the Diamond Property, and thus commutes.

Square II commutes by induction hypothesis: clearly $\alpha'$ is shorter than $\alpha$ and $\langle b \rangle$ is shorter than $\beta$. Hence $N''$, $\gamma'$, and $\eta$ exist, and satisfy $||\alpha'\gamma'|| = ||\langle b \rangle \eta||$ and $|\eta| = |\alpha'|$.

Square III commutes similarly.

Square IV commutes again by induction hypothesis. Since $|\eta| = |\alpha'|$, $\eta$ is shorter than $\alpha$, and similarly $\zeta$ is shorter than $\beta$, so we may apply the induction hypothesis to obtain $K$, $\gamma''$, and $\delta''$ satisfying $||\eta\gamma''|| = ||\zeta\delta''||$. 

8
Setting $\gamma = \gamma'\gamma''$ and $\delta = \delta'\delta''$, some diagram chasing will give the required result.

Well-definedness follows immediately from the Extended Diamond Property, but if we inspect the proof of this property, we see that something slightly stronger holds: if we have $N$ in normal form and $M \xrightarrow{\alpha} N$ and $M \xrightarrow{\beta} N$, then $\alpha$ is a permutation of $\beta$. In a sense, we are firing the same $\beta$-redexes, just in a different order. This is made precise by Roth in [27].

**Theorem 3.4** (Well-definedness of cost). For $N$ a normal form, if $M \xrightarrow{\alpha} N$ and $M \xrightarrow{\beta} N$, then $\|\alpha\| = \|\beta\|$.

**Proof.** Using the proposition we obtain $P$, $\gamma$, and $\delta$ such that $N \xrightarrow{\gamma} P$, $N \xrightarrow{\delta} P$, and $\|\alpha\| = \|\beta\|$. $N$ is a normal form, so $P = N$, which means $\delta = \gamma = \{\}$, so $\|\alpha\| = \|\beta\|$. \qed

In the original paper [10] this notion of cost is said to be a notion of time-cost, with no further explanation as to why this would be the case. Aren’t we looking at the size of the lambda plus the “extra” size it takes to do our reduction steps? Why is this not a space cost? This becomes clear if we look at the fundamental difference between time and space: *space can be reused, time cannot*. Once we perform a reduction step, our cost will always go up, even if we performed a much more expensive reduction step before. Hence our cost model is much more like time than it is like space.

In the next section, we will prove that using this cost model, we can simulate Turing machines with polynomial time overhead, and Turing machines can simulate weak call-by-value reduction with polynomial time overhead. We will say a bit about space cost later.
3.1 Simulating Turing Machines

At this point we will slightly deviate from [10]: we will use a slightly different formalism for Turing machines in an effort to do away with some of the notational complexity in that paper.

**Definition 3.5** (Turing Machines). A *Turing Machine* is a tuple $M = (S, Q, q_0, \delta, q_f)$ for which the following conditions hold.

- $S = \{s_0, \ldots, s_n\}$ is a finite set called the *alphabet*
- $s_0$ in $S$ is the *blank symbol*. Sometimes we write $B$ instead of $s_0$.
- $Q = \{q_0, \ldots, q_f\}$ is a finite set of *states*
- $q_0$ in $Q$ is the *initial state*.
- $q_f$ in $Q$ is the *final state*.
- $\delta$ is a partial function from $Q \times S$ to $Q \times S \times \{L, R\}$, for which $\delta(q, x)$ is defined if and only if $q \neq q_f$.

The Turing Machine $M$ will operate in the usual way: there will be a tape with cells for writing a symbol, extending infinitely far to the right. The given *input*, which is finite, is written on the tape, leaving the first cell blank. The *tape head* is a device which can move alongside the tape, read the symbols in the cells, and replace those symbols. $M$ operates as follows: the tape head starts in state $q_0$. At every step, it reads the symbol $x$ in the current cell, then takes action according to the definition of $\delta$. That is, if $\delta(q, x) = (q', x', L)$, the tape head will go to state $q'$, write $x'$ in the cell (erasing $x$), and move one cell to the left. If $\delta(q, x) = (q', x', R)$, the same thing happens, but the tape head moves one cell to the right. This process is then repeated. If $\delta(q, x)$ is not defined, the tape head stops doing anything. The Turing machine is said to *terminate* in this case.

If at any point the tape head runs off the tape (to the left), the Turing machine is said to have *crashed*. We don’t have to worry about this here since crashing Turing machines aren’t all that interesting.
We will now show how to encode a Turing machine in the weak call-by-value λ-calculus, starting with the finite sets and strings. We will see a recursion operator, and then use this to manipulate strings. Finally, we will see how to simulate the transition function of a Turing machine before putting this together to obtain the final simulation result.

**Definition 3.6** (Encoding of finite sets and finite strings over finite sets). Given a set \( S = \{s_0, \ldots, s_k\} \) we encode an element \( s_i \) of \( S \) as:

\[
^r s_i^S = \lambda x_0 x_1 \ldots x_k. x_i
\]

We encode finite strings over \( S \) by

\[
^r (\cdot)^{S^*} = \lambda x_0 \ldots x_k y. y \\
^r s_i u^{S^*} = \lambda x_0 \ldots x_k y. x_i^r u^{S^*}
\]

This string encoding scheme is due to Scott, defined in unpublished lecture notes for a class taught in 1963. A published definition can be found in [29].

We often omit the superscript from our representation notation if it’s clear what we’re representing.

The following lemma shows we can do recursion without incurring too great a cost.

**Lemma 3.7** (Recursion Lemma). There exists a term \( H \) such that for any value \( N \), \( H N \overset{*}{\rightarrow} N(\lambda z. H N z) \), where \( \|\alpha\| = O(|N|) \).

**Proof.** Let \( M = \lambda x f f(\lambda z. x x f z) \), and set \( H = M M \). Note that \( M \) is a value. Let \( N \) be any term.

\[
H N \equiv (\lambda x f f(\lambda z. x x f z)) M N \quad \text{definition}
\]

\[
\overset{(a)}{\rightarrow} (\lambda f. f(\lambda z. M M f z) N
\]

\[
\overset{(b)}{\Rightarrow} N(\lambda z. M M N z)
\]

\[
\equiv N(\lambda z. H N z) \quad \text{definition}
\]

Equivalence corresponds to a “free” reduction step, so the first and last step don’t add to the cost.

\( a \) equals \(|M| - 3 \) since there is an extra instance of \( M \) but this is accompanied by a removal of a \( \lambda \) and the two occurrences of \( x \).

Similarly, \( b \) equals \(|N| - 3 \).

In total, \( H N \overset{*}{\rightarrow} N(\lambda z. H N z) \) with \( \|\alpha\| = |M| + |N| - 6 \) which is \( O(|N|) \) since \( M \) is fixed.

\( H \) is Turing’s fixpoint combinator in call-by-value form, which “immobilises” the \( x x f \)-part of Turing’s fixpoint combinator by doing an \( \eta \)-expansion.
We can extend a string with a symbol, concatenate two strings, and concatenate the reverse of a string with another.

**Lemma 3.8** (String operations). For any set $S$ there exist terms $\text{ext}$, $\text{cat}$, and $\text{rcat}$ such that for any $s_i \in S$, $u, v \in S^*$ we have:

\[
\begin{align*}
\text{ext} & \colon s \mapsto s^* \mapsto u \mapsto s^* \mapsto u^* \\
\text{cat} & \colon u \mapsto v \mapsto u \mapsto v \\
\text{rcat} & \colon u \mapsto v \mapsto u \mapsto v
\end{align*}
\]

where $||\alpha|| = O(1)$, $||\beta|| = O(|s|)$, and $||\gamma|| = O(|u|)$.

**Proof.** We assume $S = \{s_0, \ldots, s_k\}$. We define

\[
\text{ext} \equiv \lambda s u.s M_0 \ldots M_k u
\]

where $M_i \equiv \lambda u x_0 \ldots x_k y. x_i u$.

\[
\text{cat} \equiv H(\lambda x u v. u N_0 \ldots N_k I v)
\]

where $N_i \equiv \lambda u v. (\lambda g x_0 \ldots x_k y. x_i g)(x u v)$.

\[
\text{rcat} \equiv H(\lambda x u v. P_0 \ldots P_k I v)
\]

where $P_i \equiv \lambda u v. x(u(\lambda x_0 \ldots x_k y. x_i v))$.

In total, this gives us a finite sequence with weight $k + 4$, which is $O(1)$.

To prove the statements for $\text{cat}$ and $\text{rcat}$, we induct over the length of $u$. We abbreviate $N_i[\lambda z. \text{cat} z / x]$ to $N'_i$.

\[
\text{cat} \colon u \mapsto \lambda u v. (\lambda u v. u N'_0 \ldots N'_k I v) \mapsto u \mapsto v
\]

Recursion Lemma, $||\alpha|| = O(1)$

\[
\begin{align*}
\text{ext} & \colon s \mapsto s^* \mapsto u \mapsto s^* \mapsto u^* \\
\text{cat} & \colon u \mapsto v \mapsto u \mapsto v \\
\text{rcat} & \colon u \mapsto v \mapsto u \mapsto v
\end{align*}
\]

terms only get smaller

\[
\begin{align*}
\alpha & \colon M_i \mapsto u^3 \\
(1) & \colon \lambda x_0 \ldots x_k y. x_i u \\
\equiv & \colon s_i u^3
\end{align*}
\]

definition

\[
\begin{align*}
\beta & \colon I' \mapsto u^3 \\
(1) & \colon v^3
\end{align*}
\]

smaller terms
Clearly, the weight of this reduction sequence is $O(1)$.

$$
\text{cat}^{r} s_{i} u^{r} v^{3} \rightarrow^{\alpha} (\lambda u. u N_{0}^{i} \ldots N_{k}^{i} I v)^{r} a_{i} u^{r} v^{3}
$$

recursion lemma, $\|\alpha\| = O(1)$

$$
\xrightarrow{(1,1)} s_{i} u^{r} N_{0}^{i} \ldots N_{k}^{i} I v^{3}
$$

$$
\xrightarrow{\beta} (\lambda u. (\lambda g. x_{0} \ldots x_{k} y. x_{i} g) (\lambda z. \text{cat} z) u v) u^{r} v^{3}
$$

definition of $\text{\text{\`a} u^{r}}$,

$\|\beta\| = O(1)$

$$
\xrightarrow{(1,1)} (\lambda g x_{0} \ldots x_{k} y. x_{i} g) ((\lambda z. \text{cat} z)^{r} u^{r} v^{3})
$$

$$
\xrightarrow{(1)} (\lambda g x_{0} \ldots x_{k} y. x_{i} g) (\text{cat}^{r} u^{r} v^{3})
$$

$$
\xrightarrow{\gamma} (\lambda g x_{0} \ldots x_{k} y. x_{i} g) (\text{cat}^{r} u^{r} v^{3})
$$

induction hypothesis

$$
\xrightarrow{(1)} \lambda x_{0} \ldots x_{k} y. x_{i}^{r} u v^{3}
$$

$\equiv \text{cat}^{r} u^{r} v^{3}$

definition

With the exception of the induction hypothesis step, the weight of the steps is constant. Now the induction is over the length of $u$, so this proves that $\text{cat}^{r} u^{r} v^{3} \rightarrow^{\beta} u v^{3}$ with $\|\alpha\| = O(|u|)$.

The proof for $\text{rcat}$ is similar. We write $P_{i}^{r}$ for $P_{i}^{r} \left[ \lambda z. \text{rcat} z / x \right]$.

$$
\text{rcat}^{r} ()^{r} v^{3} \rightarrow (\lambda u. u P_{0}^{r} \ldots P_{k}^{r} I v)^{r} ()^{r} v^{3}
$$

recursion lemma, $\|\alpha\| = O(1)$

$$
\xrightarrow{(1,1)} ()^{r} P_{0}^{r} \ldots P_{k}^{r} I^{r} v^{3}
$$

$$
\xrightarrow{\beta} I^{r} v^{3}
$$

$\|\beta\| = k + 1 = O(1)$

$$
\xrightarrow{(1)} v^{3}
$$

Clearly, the weight of this reduction sequence is $O(1)$.

$$
\text{rcat}^{r} s_{i} u^{r} v^{3} \rightarrow (\lambda u. u P_{0}^{r} \ldots P_{k}^{r} I v)^{r} a_{i} u^{r} v^{3}
$$

recursion lemma, $\|\alpha\| = O(1)$

$$
\xrightarrow{(1,1)} s_{i} u^{r} P_{0}^{r} \ldots P_{k}^{r} I^{r} v^{3}
$$

$$
\xrightarrow{\beta} (\lambda u. (\lambda z. \text{rcat} z) u (\lambda x_{0} \ldots x_{k} y. x_{i} v)) u^{r} v^{3}
$$

definition of $\text{\text{\`a} u^{r}}$,

$\|\beta\| = O(1)$

$$
\xrightarrow{(1,1)} (\lambda z. \text{rcat} z)^{r} u^{r} (\lambda x_{0} \ldots x_{k} y. x_{i} v^{3})
$$

$$
\xrightarrow{\gamma} (\lambda z. \text{rcat} z)^{r} u^{r} s_{i} v^{3}
$$

definition of $\text{\text{\`a} u^{r}}$

$$
\xrightarrow{(1)} \text{rcat}^{r} u^{r} s_{i} v^{3}
$$

$$
\xrightarrow{\gamma} u^{r} s_{i} v^{3} = (s_{i} u)^{r} v^{3}
$$

induction hypothesis
With the exception of the induction hypothesis step, the weight of the steps is constant. The induction is over the length of $u$, so this proves that $\text{rcat}^\nu u^\nu v^\nu \xrightarrow{\alpha} u^\nu v^\nu$ with $\|\alpha\| = O(|u|)$.

Using strings, we can encode configurations of the Turing machine.

**Definition 3.9 (Configurations and encoding of configurations).** A configuration of a Turing machine consists of

- The current content of the tape.
- The position of the tape head on the tape.
- The state the tape head is in.

We encode a configuration as follows:

$$\langle u, s, v, q \rangle \equiv \lambda x.x^\nu u^\nu s^\nu v^\nu q$$

where $u$ is the string of symbols on the tape preceding the current position, $s$ is the symbol in the current position, $v$ is the string of symbols on the tape succeeding the current position, ending with the last non-blank symbol, and $q$ is the current state.

Our goal is now to define three $\lambda$-terms. One for handling input, which takes a string and produces the initial configuration, one that models the transition function, and one that takes a final configuration and produces the output string. We shall tackle the easier two first.

**Proposition 3.10.** For every Turing machine $M = (S, Q, q_0, \delta, q_f)$ there exist terms $\text{init}, \text{fin}$ such that

$$\text{init}^\nu u^\nu \xrightarrow{\alpha} \langle \rangle, B, u, q_0^\nu$$

$$\text{fin}^\nu u, s, v, q_f^\nu \xrightarrow{\beta} u^\nu v^\nu$$

where $\|\alpha\| = O(1), \|\beta\| = O(|uv|)$.

**Proof.** Define

$$\text{init} \equiv \lambda u x x^\nu e^\nu B^\nu u^\nu q_0^\nu$$

$$\text{fin} \equiv \lambda c e (\lambda usvq. \text{rcat}(\text{extav}))$$

Using the properties we proved about $\text{rcat}$ and $\text{ext}$, it is clear that these terms satisfy the required properties.
Proposition 3.11. For every Turing machine $M = (S,Q,q_0,\delta,q_f)$ there exists a term $T$ such that for any configuration $C$:

- If $D$ is a final configuration reachable from $C$ in $n$ steps, then $T^nC \xrightarrow{\alpha} D$ with $\|\alpha\| = O(n)$.
- If there is no such configuration, $T^nC$ diverges.

Proof. Again we assume that $S = \{s_0, \ldots, s_k\}$.

We first define $|Q| (k+1)$ terms $N_{i,j}$ which represent the action $M$ takes in state $q_i$ when reading the symbol $s_j$.

$$N_{i,j} \begin{cases} \lambda uvx.xu' s_j' v' q_f' & \text{if } q_i = q_f \\ \lambda uvx(uP_0 \ldots P_k(\text{ext}' s_i' v)' q_k') & \text{if } \delta(q_i, s_j) = (q_k, s_l, L) \\ \lambda uvx(vR_0 \ldots R_k R(\text{ext}' s_i' u)' q_k') & \text{if } \delta(q_i, s_j) = (q_k, s_l, R) \end{cases}$$

where

$$P_i \equiv \lambda uvqx.xu' s_i' vq$$
$$R_i \equiv \lambda uvqx.xu' s_i' vq$$
$$R \equiv \lambda uvqx.xu' B^{\gamma} \epsilon q$$

We now only need to select the appropriate $N_{i,j}$, and iterate this process.

$$T \equiv H(\lambda xc.(\lambda usvq.qM_0 \ldots M_{|Q|} usv))$$

where

$$M_i \equiv \lambda usv. sN_{i,0} \ldots N_{i,k}uv$$

Since we are using deterministic Turing machines, we can speak of the next configuration $D$ reachable from a configuration $C$. To prove the proposition it suffices to show that $T^nC \xrightarrow{\alpha} T^nD$ where $\alpha$ does not depend on the number of steps $M$ takes from configuration $D$ onwards. Then if $M$ does not terminate, $T^nC$ diverges, because it will always find the next configuration. If it does, the result follows by induction on the number of steps until termination.

We write $C = \gamma u, s_j, v, q_i$, $M'_i \equiv M_i[\lambda z.Tz/x]$, $N'_{i,j} \equiv N_{i,j}[\lambda z.Tz/x]$ for all $i, j$.

$$T^\gamma u, s_j, v, q_i \xrightarrow{\alpha} (\lambda xc(\lambda usvq.qM_0' \ldots M_{|Q|} usv))u, s_j, v, q_i$$

recursion lemma, $\|\alpha\| = O(1)$

$$\xrightarrow{(1)} u, s_j, v, q_i(\lambda usvq.M_0' \ldots M_{|Q|} usv)$$
$$\xrightarrow{(1,1,1,1)} q_i.M_0' \ldots M_{|Q|}' u^\gamma \gamma s_j \gamma v$$
$$\xrightarrow{\delta} (\lambda usv.sN'_{i,0} \ldots N'_{i,k} uv)u^\gamma \gamma s_j \gamma v$$

$\|\gamma\| = O(1)$

$$\xrightarrow{(1,1,1)} s_j.N'_{i,0} \ldots N'_{i,k} u^\gamma \gamma v$$

$$\delta N'_{i,j} u^\gamma \gamma v / v$$

$\|\delta\| = O(1)$
Clearly, all of this is $O(1)$. We distinguish three cases.

If $q_i = Q_f$, $C$ is a final configuration, so $TC$ should reduce to $C$ in constant time. Now we have:

$$N_{i,j} \cdot u^{\gamma \tau} v^\gamma \xrightarrow{\{\}^0} \lambda x. u^{\gamma \tau} s_j^{\gamma \tau} v^\gamma q_f^\gamma$$
with $\text{definition of } N_{i,j}$

which is what we wanted.

If $\delta(q_i, s_j) = (q_f, s_j', L)$, we can replace $u$ by $us_l$, since if $u$ were the empty string, $M$ would crash: $u$ being empty means the tape head is at the very left edge of the tape. We now wish to prove that $T^f us_l, s_j, v, q_i^\gamma \xrightarrow{\alpha} T^f u, s_l, s_j'v, q_i^\gamma$ with $|\alpha| = O(1)$.

$$N_{i,j} \cdot (us_l)^{\gamma \tau} v^\gamma \xrightarrow{\{\}^0} (\lambda z. T z)((us_l)^{\gamma \tau} P_0 \ldots P_k(\text{ext}^{s_j^{\gamma \tau} v^\gamma} q_i^\gamma)) \xrightarrow{\{\}^0} (\lambda z. T z)((us_l)^{\gamma \tau} P_0 \ldots P_k s_j^{\gamma \tau} v^\gamma q_i^\gamma) \quad \text{with } |\alpha| = O(1)$$

This proves what we wanted.

If $\delta(q_i, s_j) = (q_f, s_j', R)$, and $v = \{\}$, we wish to prove that $T^f u, s_j, \{\}, q_i^\gamma \xrightarrow{\alpha} T^f us_j', B, \{\}, q_i^\gamma$ with $|\alpha| = O(1)$.

$$N_{i,j} \cdot u^{\gamma \tau} \varepsilon \xrightarrow{\{\}^0} (\lambda z. T z)((\varepsilon^c R_0 \ldots R_k R(\text{ext} s_j^{\gamma \tau} u^\gamma) q_i^\gamma) \quad \text{with } |\alpha| = O(1)$$

Here $|\alpha| = O(1)$, so this proves what we wanted.
If \( \delta(q_i, s_j) = (q_i', s_j', R) \), and \( v \neq \{ \} \), write \( s_lv \) instead of \( v \). We wish to prove that \( T^u, s_j, s_lv, q_i \overset{\alpha}{\Rightarrow} T^us_j', s_l, v, q_i' \) with \( ||\alpha|| = O(1) \).

\[
N_{i,j}^u \overset{\gamma}{\Rightarrow} (\lambda z.Tz)(\gamma')\\ \\
\overset{\alpha}{\Rightarrow} T(R_i'^\gamma'(us_{j'}^\gamma')'q_{i'}')\\ \\
\overset{\{1,1\}}{\Rightarrow} T(\lambda x.x'^\gamma'(us_{j'}^\gamma')'s_l'^\gamma'v'^\gamma'q_{i'}')\\ \\
\overset{\{ \} }{\Rightarrow} T('us_{j'}, s_l, v, q_{i'}')
\]

Similar to above, \( ||\alpha|| = O(1) \)

Which is what we wanted. This concludes the proof.

Proving that \( \lambda \)-terms simulate Turing machines with polynomial overhead is now relatively easy.

**Theorem 3.12** (Simulation of Turing Machines by \( \lambda \)-Terms). For every Turing machine \( M \) there exists a \( \lambda \)-term \( \bar{M} \) such that for every string \( x \) the following hold.

If \( \bar{M} \) on input \( x \) terminates with output \( y \) in \( n \) steps, we have:

\( \bar{M}^x \overset{\alpha}{\Rightarrow} y \)

with \( ||\alpha|| = O(n) \).

If \( M \) on input \( x \) doesn’t terminate, \( M^x \) diverges.

**Proof.** Define

\( \bar{M} \equiv \lambda x. \text{fin}(T(\text{init}x)) \)

The result follows easily from the Propositions 3.10 and 3.11.

Note that this is only half of the polynomial invariance thesis! We also need to show that Turing machines can simulate \( \lambda \)-terms with polynomial overhead.
3.2 Simulating Lambda Terms

We will show that there exists a Turing machine which takes as input (the code of) a \( \lambda \)-term, and reduces the \( \lambda \)-term to the (code of the) weak call-by-value normal form in time polynomial in the complexity of the \( \lambda \)-term. This proof is uniform in the sense that there is one Turing machine which works for all \( \lambda \)-terms. The previous section can be turned into a uniform proof as well: we simply need to fix some type of effective encoding for Turing machines in the \( \lambda \)-calculus, and build a term that, given a code of a Turing machine, builds the term simulating the machine.

We start by introducing our codes of \( \lambda \)-terms.

**Definition 3.13 (Coding \( \lambda \)-terms).** We encode \( \lambda \)-terms using the symbols \( \lambda, @, 0, 1 \). We shall write \( M# \) for the code of \( M \). \( M# \) is defined as follows:

- \( x# = \# \) if \( x \) is free.
- \( x# = \#s \) is \( x \) is bound and has De Bruijn index \( s \) (in binary).
- \( (\lambda x.M)# = \lambda M# \)
- \( (MN)# = @M# N# \)

De Bruijn indices work as follows: in \( \lambda x.M_0 \), \( x \) initially gets De Bruijn index 0. Then if \( M_0 \) contains some \( \lambda y.M_1 \), occurrences of \( x \) in \( M_1 \) will have De Bruijn index 1. If \( M_1 \) contains some \( \lambda z.M_2 \), occurrences of \( x \) in \( M_2 \) will have De Bruijn index 2, occurrences of \( y \) in \( M_2 \) will have De Bruijn index 1, and occurrences of \( z \) in \( M_2 \) will have De Bruijn index 0, and so forth.

We write \( \#M \) for the number of symbols in \( M# \).

Intuitively, the De Bruijn index of a variable corresponds to its “distance” to its binding \( \lambda \), measured in the number of \( \lambda \)-abstractions occuring in between.

In this definition, all free variables have the same (empty) code. This will not matter for the proof, since free variables don’t assume values during reduction.

**Example 1.** We will put some terms and their codes next to eachother. Note that we can retrieve the original structure from the code.

<table>
<thead>
<tr>
<th>Term</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda x.x y )</td>
<td>( \lambda @ # # # # )</td>
</tr>
<tr>
<td>( \lambda x.y(\lambda z.x z) )</td>
<td>( \lambda \lambda @ @ # # # # )</td>
</tr>
<tr>
<td>( (\lambda x.y)(\lambda y.x y(\lambda z.x z)) )</td>
<td>( @ @ # # # # # # # # # # # # )</td>
</tr>
</tbody>
</table>
Due to our use of De Bruijn-indices, \( |M| \) might be larger than \(|M|\).

**Lemma 3.14.** For every term \( M \), \( |M| = O(|M| \log |M|) \). Furthermore, this bound is tight: there exists a sequence \( \{M_n\}_{n \in \mathbb{N}} \) such that \( |M_n| = \Theta(n) \), but \(|M| = \Theta(|M_n| \log |M_n|)\).

**Proof.** Fix \( M \). There are at most \( 2|M| - 1 \) applications and abstractions in \( M \), since an abstraction requires a body and an application requires something to apply. The number of different variables is bounded by \(|M|\). The length of the De Bruijn index for a variable in \( M \) is bounded by \( |\log |M|| \), so the code of a variable has length at most \( 1 + |\log |M|| \). Hence the length of the encoding of all variables together is bounded by \(|M|(1 + |\log |M||)\).

Taking these two observations together, we get \( |M| \leq 2|M| - 1 + |M|(1 + |\log |M||) = O(|M| \log |M|)\).

For the second part, we define

\[
M_n = \lambda xy \ldots y.x \ldots x
\]

where \( y \) occurs \( n \) times and \( x \) occurs \( n + 1 \) times in the body.

The code of \( M_n \) is:

\[
\lambda \lambda \ldots \lambda \# u \ldots \# u \# u
\]

where \( \lambda \) occurs \( n + 1 \) times, and \( \# \) occurs \( n \) times, \( \# \) occurs \( n + 1 \) times, \( u \) being the De Bruijn index of \( x \), equal to \( n \) in binary, occurring \( n + 1 \) times.

Now \(|M| = (n + 1) + (n + 1) = 2n + 2 = \Theta(n)\), but \(|M| = (n + 1) + n + (n + 1) + (n + 1)\log n = 3n + 2 + \log n + n[\log n] = \Theta(n \log n)\).

Simulation of \( \lambda \)-terms by Turing machines requires less technical computations. The main difficulty of the proof is seeing how we can detect redexes to fire, because the weak call-by-value \( \lambda \)-calculus does not reduce under a \( \lambda \) and fully reduces an argument before firing the corresponding \( \beta \)-redex.

**Theorem 3.15** (Simulation of \( \lambda \)-terms by Turing machines). There exists a Turing machine \( R \) such that for any \( \lambda \)-term \( M \):

- If \( \text{cost}(M) = \infty \), then \( R \) does not terminate on input \( M^\# \).
- If \( M \) has (weak call-by-value) normal form \( N \) with \( \text{cost}(M) = n \), then \( R \) will terminate on input \( M^\# \) in a number of steps polynomial in \( n \).

**Proof.** \( R \) will have nine tapes: \textit{current}, \textit{preredex}, \textit{functional}, \textit{argument}, \textit{postredex}, \textit{reduct}, \textit{stackterm}, \textit{stackredex}, and \textit{counter}. \textit{Current} will be the input and output tape.

We will first describe \( R \) informally: on input \( M^\# \), \( R \) goes through the input looking for a redex, using \textit{stackterm} and \textit{stackredex} to keep track of term structure. Everything encountered before finding a redex is copied into \textit{preredex}. Once a redex has been found, \( R \) copies the functional part into \textit{functional} and the argument part into \textit{argument}. Everything after goes into \textit{postredex}. With the

19
help of counter for dealing with the De Bruijn indices, \( R \) copies the argument into the right places of functional. Then the contents of preredex, functional, and postredex are copied (concatenated) to current, before repeating the entire procedure again. If no redex is found, \( R \) terminates.

Every such cycle performs a reduction step, so if the reduction of \( M \) terminates, so will \( R \) on input \( M^\# \), and if it doesn’t, neither will \( R \). What we need to prove is that we can perform this cycle, and that we can do so effectively. More specifically, every cycle must correspond to a weak call-by-value normalisation step with some cost \( k \), and the number of steps \( R \) takes in a cycle must not be too large compared to \( k \) in order to obtain polynomial overhead.

We will first describe the use of stackterm and stackredex, and how they are used to keep track of term structure. As the name already hints at, we will treat these tapes as stacks. The fundamental observation is that any argument of an application must end it with a variable, and if we have a variable, then we have just seen an argument of an application. Similarly, the body of an abstraction must end with either a lone variable (\( \lambda x.x \)) or the second argument of an application (\( \lambda x.xx \)) (which also ends with a variable).

stackterm will be used to keep track of overall term structure, while stackredex will be used to ensure we read the functional and argument parts of a redex correctly. They operate in the same way.

We use three symbols on the stacks: \( F\@, S\@, B\lambda \). They are manipulated according to the following rules.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Read</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( @ )</td>
<td>Push</td>
<td>( F@ ).</td>
</tr>
<tr>
<td>( \lambda )</td>
<td>Push</td>
<td>( B\lambda ).</td>
</tr>
<tr>
<td>( \triangleright )</td>
<td>Pop</td>
<td>( S@, B\lambda ), looking for ( F@ ). If ( F@ ) found, pop ( F@ ), push ( S@ ). If no ( F@ ) is found, the stack stays empty.</td>
</tr>
<tr>
<td>0</td>
<td>No action.</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>No action.</td>
<td></td>
</tr>
</tbody>
</table>

These symbols have the following meaning.

- If there’s an \( F\@ \), we are reading the first argument of the corresponding application.
- If there’s an \( S\@ \) on the stack, we are reading the second argument of the corresponding application. Note that we only obtain \( S\@ \) by replacing \( F\@ \), so this is well-defined.
- If there’s a \( B\lambda \) on the stack, we are inside the body of a \( \lambda \)-term.

Having read (the code of) a well-formed \( \lambda \)-term, the stack will be empty, and the final relevant symbol with respect to the stack will have been \( \triangleright \). This is proven by induction on the term structure.
• If \( M \) is a variable, then the stack is empty to being with.

• If \( M \) is an abstraction, \( R \) first pushes a \( B_\lambda \) before reading the body of the term. This ends with reading \( \beta \). By induction hypothesis, the popping of \( S_\@ \) and \( B_\lambda \) that occurs as a result of reading the \( \beta \) doesn’t encounter a \( F_\@ \) (since if we read the body on its own, the stack ends up being empty), so the initial \( B_\lambda \) has to get popped as well.

• If \( M \) is an application, we will first push a \( F_\@ \), then read the first argument. This ends with reading \( \beta \). By induction hypothesis, we don’t encounter a \( F_\@ \) other than the one we pushed initially (otherwise the stack wouldn’t be empty after reading the first argument on its own). Hence the \( F_\@ \) is replaced by \( S_\@ \) and we start reading the second argument. Similar to the case for a \( \lambda \)-abstraction, this \( S_\@ \) will get popped once we’re done reading the second argument.

This procedure will be more intuitive if we see an example. Consider \( R \) reading the term \( (\lambda x. xy)((\lambda xy.xy)(\lambda z.xz)) \). As we have already seen, this has code \( @\lambda @ \bullet 0 \bullet \lambda @ \bullet 1 \bullet 0 \lambda @ \bullet 10 \bullet 0 \). The following table shows what happens with the stack as we read the term. The position of the tape head is indicated by a line under the symbol currently being read.

<table>
<thead>
<tr>
<th>Tape</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>( @\lambda @ \bullet 0 \bullet \lambda @ \bullet 1 \bullet 0 \lambda @ \bullet 10 \bullet 0 )</td>
<td>( \epsilon )</td>
</tr>
<tr>
<td>( @\lambda @ \bullet 0 \bullet \lambda @ \bullet 1 \bullet 0 \lambda @ \bullet 10 \bullet 0 )</td>
<td>( F_@ B_\lambda )</td>
</tr>
<tr>
<td>( @\lambda @ \bullet 0 \bullet \lambda @ \bullet 1 \bullet 0 \lambda @ \bullet 10 \bullet 0 )</td>
<td>( F_@ B_\lambda F_@ )</td>
</tr>
<tr>
<td>( @\lambda @ \bullet 0 \bullet \lambda @ \bullet 1 \bullet 0 \lambda @ \bullet 10 \bullet 0 )</td>
<td>( F_@ B_\lambda S_@ )</td>
</tr>
<tr>
<td>( @\lambda @ \bullet 0 \bullet \lambda @ \bullet 1 \bullet 0 \lambda @ \bullet 10 \bullet 0 )</td>
<td>( F_@ B_\lambda S_@ )</td>
</tr>
<tr>
<td>( @\lambda @ \bullet 0 \bullet \lambda @ \bullet 1 \bullet 0 \lambda @ \bullet 10 \bullet 0 )</td>
<td>( S_@ )</td>
</tr>
<tr>
<td>( @\lambda @ \bullet 0 \bullet \lambda @ \bullet 1 \bullet 0 \lambda @ \bullet 10 \bullet 0 )</td>
<td>( S_@ \lambda )</td>
</tr>
</tbody>
</table>

We now precisely describe how \( R \) operates.

1. It walks through the term, copying everything into \textit{preredex}, and keeping track of the term structure using \textit{stackterm}, until it reaches an \( @ \) followed by a \( \lambda \), at which point it will start copying what’s read into \textit{functional} rather than \textit{preredex}, but only if there isn’t a \( B_\lambda \) on \textit{stackterm}. In this way, we won’t do any reductions in the body of a \( \lambda \)-term.

At this point, it also starts keeping track of term structure using \textit{stackredex} (note: we do not stop using \textit{stackterm}). Once we encounter the argument of the redex (once \textit{stackredex} is empty), it checks if it’s a value (which happens if and only if the first symbol is \( \beta \) or \( \lambda \)). If not, the contents of \textit{functional} are concatenated to the contents of \textit{preredex} and \( R \) continues as normal. If \( R \) finishes reading the term without encountering a redex, it terminates.
If we do encounter a value, we keep using stackredex (which was empty after reading the functional part) to copy the argument into argument. Having done this (when stackredex is empty again), R will copy the rest of the term into postredex.

2. Assuming we found a redex, R writes 0 onto counter, and starts walking through functional while keeping track of term structure using stackredex (which was empty after reading the argument). If R reads a λ, the value on counter is incremented, and will be decremented if a Bλ is popped from the stack. In this way, counter will always contain the De Bruijn index of the variable we wish to replace by the contents of argument.

If R reads a ▶, it compares the number following it (if any) with the contents of counter. If this is equal, R replaces the occurrence of ▶, including the number, by the contents of argument.

3. Once the complete functional is read, our beta reduction step is complete, so R empties current and copies preredex, functional, and postredex (in that order) to current.

4. R returns the tape head to the first position and goes back to step 1.

To analyze complexity, suppose we have (the code of) a term $N$ on the tape.

Step 1 takes at most $O\left(\frac{\|N\|^2}{\|N\|}\right)$ steps: we might have to go back and forth a couple times to check for $B\lambda$ on stackterm, which has length at most $\|N\|$ and we encounter an @ followed by a λ at most $\|N\|$ times.

For the complexity of step 2, say the corresponding reduction step is $N \rightarrow N'$, having weight $n$. We know that the argument of the redex being fired has length $O(\|N\|)$, so the copying and replacing step will take time $O(n \|N\|)$. Then copying the new term onto current will take $O(\|N'\|)$.

Hence each reduction step will take $O(\|N\|^2 + n\|N\| + \|N'\|)$ time.

Note that cost $(M)$ gives an upper bound of the size of any terms $N, N'$ in the reduction path, and it gives an upper bound on the weight of any reduction step, so each reduction step will take $O(\text{cost}(M)^2)$.

Finally, cost $(M)$ also gives an upper bound on the number of reduction steps. Hence, the running time of $R$ is $O\left(\frac{\text{cost}(M)^3}{\text{cost}(M)}\right)$, which is polynomial in cost$(M)$.

\[\square\]
3.3 Space cost modification and extension to full $\lambda$-calculus

We have already seen this cost model defines a time cost rather than a space cost. The obvious question, then, is whether this kind of cost model can be modified to give a good (polynomially invariant) definition of the space cost of a $\lambda$-term. This seems unlikely. The basic idea would be to modify the list $\alpha$ in some fashion, or the definition of $|\alpha|$.

An initial attempt might be to take, instead of the sum of numbers in $\alpha$, the maximum. This corresponds to our idea of “reusing” space, since if a certain number is a maximum, we can freely add numbers lower than the maximum without increasing the cost. This model has the shortcoming that it only looks at single reduction steps: we might have some sequence of reduction steps, each small, but adding up to a big increase in term size. However, this big increase is not reflected in taking the maximum over $\alpha$, and this is not what we want. The following remark gives an example.

Remark 1. Let $M = \lambda xyzw. wxxyyzz$, $N = \lambda xzyuw. I$ and $K = \lambda x.xxx$. Consider the term $K(MIIIN)$.

Reducing this term to normal form, we first obtain 3 size-increases of length 1 when firing the redexes corresponding to the three $I$’s. Then two steps which reduce the length of the term by firing the redexes involving $N$. At this point we have $KI$, which results in a size-increase of 2, obtaining $IIIII$. Reducing this to normal form only makes terms smaller.

The maximum over this $\alpha$ is clearly 2, though we had combined size-increase of 3 initially.

So taking the maximum over $\alpha$ doesn’t seem to work so well. Perhaps we need to change the numbers in $\alpha$ to not reflect the size difference, but simply the absolute size. Then the maximum does capture our intuitive notion of space. The following remark shows that this doesn’t work out either: the weak call-by-value calculus does not behave well with respect to the length of the term.

Remark 2. Let $M = (II)((\lambda x.xxx)I)$, having length 17. $M$ reduces in one step to $I((\lambda x.xxx)I)$, of length 14, and also in one step to $(II)(III)$, having length 20. These both reduce (via the diamond property) to $I(III)$, which has length 16. Observe that from here on out, the term will only get smaller (until it reduces to $I$, having length 4). Hence the maximum length of terms in the reduction path can change when a different reduction strategy is adopted.

One could get around this by fixing a reduction strategy (say, leftmost-outermost), and then following the proof given above in order to arrive at a space complexity measure for the weak call-by-value $\lambda$-calculus (or probably even the regular $\lambda$-calculus). However, if we look at implementing $\beta$-reduction on a machine, things become more complicated.

In implementing reduction, we would like to be very efficient with space. If we fire a redex like $(\lambda xy.xxx)V$, we don’t create two instances of $V$, but rather “link” both $x$’s to $V$, so we only have one copy of $V$. The occurrences share $V$, in a sense. This greatly reduces the space we use, though it also creates other
problems. If we enter a value for \( y \), it might do different things to the first and second instance of \( V \). This means we want to unshare \( V \).

This sharing and unsharing can lead to orphaned terms, which aren’t used in the reduction anymore, but exist in the memory, taking up space. We want to remove these, so we have to do some kind of garbage collection in order to keep the amount of space used minimal.

If we look at reduction like this, the “length of the \( \lambda \)-term”-model doesn’t correspond to this idea of space cost at all. Rather, space cost depends on the inner structure of the term and what happens with different copies of the same term, and this model doesn’t capture these subtleties.

A second natural question would be if this cost model is extensible to the full \( \lambda \)-calculus. The well-definedness of the cost model strongly relies on the fact that the weak call-by-value \( \lambda \)-calculus is strongly confluent. It thus seems unlikely that we can even define something similar for the full \( \lambda \)-calculus. The following remark shows that this issue is of the worst possible nature.

Remark 3. Let \( \omega = \lambda x.x x \) and \( \Omega = \omega \omega \). Then the term \( M = (\lambda xy.y)\Omega \) admits both a one-step reduction to normal form and an infinite reduction path in the full \( \lambda \)-calculus. However, in the weak call-by-value \( \lambda \)-calculus, \( M \) only admits an infinite reduction sequence because \( \Omega \) doesn’t reduce to a value.

Good results have been attained for the full \( \lambda \)-calculus with a fixed reduction strategy: Accattoli and Dal Lago [11] recently proved that under leftmost-outermost reduction, the number of \( \beta \)-steps taken is a good cost model (satisfying the polynomial invariance thesis) for the full \( \lambda \)-calculus. Interestingly, Roth [27] consequently proved using similar techniques that in the weak call-by-value \( \lambda \)-calculus, the number of \( \beta \)-steps it takes to reduce a term to weak call-by-value normal form (which, as Lemma 3.2 shows, is the same for any reduction path) is a good cost model.

The proofs are a lot more elaborate than the rather straightforward proof of polynomial invariance seen here, so we will not discuss them further. We will however briefly return to the cost model of Accattoli and Dal Lago in Section 7.2.

4 The Type System \( \text{DIAL}_{\text{lin}} \) and Main Result

In the following we put the cost model to use. We specify a type system for the weak call-by-value \( \lambda \)-calculus, and prove that it admits a characterization of polynomial time functions. That is, there exists a type such that all terms of that type are in \( \text{FP} \), and all functions in \( \text{FP} \) have a term with such a type. This was first done by Aloïs Brunel and Kazushige Terui in [5]. Another article by the same authors, giving a slightly different proof can be found in [4]. The trick is to limit the possible recursion, as the following remark shows.

Remark 4. Suppose we wish to define \( f(n) = 2^n \) using recursion. In the standard way, we define \( 2^0 = 1 \), and \( 2^{n+1} = 2 \cdot 2^n \). This requires us to use multiplication, which in turn is recursively defined using addition, which in turn is recursively
defined using successor. So this definition of $2^n$ has a “recursion depth” of 3, while addition has a “recursion depth” of 1. We could remedy this a little by using $2^{n+1} = 2^n + 2^n$, but now we need to use our argument twice. There is something inherently different about the recursion in the definition of $f(n) = 2^n$ as opposed to that of $f(n) = n + k$ for some number $k$.

This has been formalized in [19], where a classification of $\mathbb{P}$ using this notion of stratified recursion has been developed. Other works on the subject include [17, 18, 21, 20, 22, 23, 24, 3].

The recursion depth issue is avoided by using “tiering”: separating different levels of recursion and preventing recursion of too great a depth. The multiple argument use issue is avoided by being careful about how we use variables.

Restrictions on variables and recursion depth are achieved using linear logic. Succinctly put, linear logic is “resource aware”. In classical logic, the statement $A \Rightarrow A \land A$ is derivable. If we inspect the derivation, we see that we need to use the hypothesized proof of the premise not once, but twice. In linear logic, on the contrary, we are very careful about these types of derivations. In particular, we keep track of when we use a formula or a variable in a non-linear fashion, i.e. if we duplicate it.

Brunel and Terui capture these concepts using a system of linear logic called $\text{DIAL}_{\text{lin}}$. $\text{DIAL}$ stands for “dual intuitionistic affine logic”, the $\text{lin}$ subscript is to indicate we use linear logic. It is a dual type system in the sense that we will have two contexts, linear and non-linear, which says something about how the variables are used (avoiding the multiple use issue) and an affine type system in the sense that we have two types of formulas, linear and non-linear, which says something about what the variables are (avoiding the tiering issue).
Definition 4.1 (DIAL\(_{\text{lin}}\)). Let \(\alpha, \beta, \ldots\) be a given set of propositional variables.

Formulas come in two “flavours”, linear and non-linear. We denote linear formulas with the letters \(L, M\), and non-linear formulas with the letters \(A, B\). They are generated by the following grammars:

\[
L, M ::= \alpha \mid \forall \alpha L \mid \mu \alpha L(^*) \mid L \rightarrow M
\]

\[
A, B ::= \alpha \mid \forall \alpha A \mid L \rightarrow B \mid A \Rightarrow B
\]

(^*) For \(\mu \alpha L\) we also require that \(\alpha\) occurs only positively in \(L\), defined below.

A **typing** is a statement of the form \(t : A\) of \(t : L\), with \(t\) a \(\lambda\)-term. In the former case, we say that \(t\) has non-linear type, and in the latter case we say that \(t\) has linear type.

A **judgement** is a statement of the form \(\Gamma; \Delta \vdash t : A\), with \(\Gamma\) a list non-linear typings for variables, and \(\Delta\) a list of linear typings for variables. We assume the variables typed in \(\Gamma\) and \(\Delta\) are distinct.
We say that some term $t$ has type $A$ (respectively $L$) if $\vdash t : A$ (respectively $\vdash t : L$) is derivable using the typing rules from Figure 1.

**Definition 4.2** (Positive/negative occurrence of variables). Let $\alpha$ be a propositional formula and $L$ a linear formula containing $\alpha$. We define the occurrences of $\alpha$ in $L$ to be positive or negative by induction on the structure of $L$.

- $L = \alpha$: $\alpha$ occurs positively in $\alpha$.
- $L = \forall \beta M$: $\alpha$ occurs positively in $L$ if and only if it does so in $M$. $\alpha$ occurs negatively in $L$ if and only if it does so in $M$.
- $L = \mu \beta M$: $\alpha$ occurs positively in $L$ if and only if it does so in $M$. $\alpha$ occurs negatively in $L$ if and only if it does so in $M$.
- $L = M \rightarrow K$: $\alpha$ occurs positively in $L$ if and only if it occurs positively in $K$ or negatively in $M$. It occurs negatively in $L$ if and only if it occurs negatively in $K$ and positively in $M$.

A variable can occur both positively and negatively in the same formula: $\alpha$ occurs both positively and negatively in $\alpha \rightarrow \alpha$.

Positive or negative occurrences of $\alpha$ give some information about the number of implications in which $\alpha$ occurs in the premise. $\alpha$ occurs only negatively in $\alpha \rightarrow \beta$, but occurs only positively in $\alpha \rightarrow \beta$, and again only negatively in $(\alpha \rightarrow \beta) \rightarrow \beta$. However, this information is not complete: $\alpha$ occurs both positively and negatively in $\alpha \rightarrow \alpha$, so again both positively and negatively in $(\alpha \rightarrow \alpha) \rightarrow \beta$.

The restriction on $\mu \alpha L$ that $\alpha$ only occurs positively in $L$ is important in showing the soundness of our characterization.

In essence, there are two types of linearity in DIAL$_{\text{lin}}$. The first is linearity of formulas, as defined by the grammar. The second is linearity of variable usage, as defined by the type system. Putting a variable in the linear context entails a promise to only use that variable once, whereas non-linear use of a variable means we’re allowed to use it more than once. This is realized by only allowing contraction to take place on variables in the non-linear context, as well as requiring the argument of an application involving a non-linear function to contain no linear variables. The distinction between $\rightarrow$ and $\Rightarrow$ then is that in the former, we only use the argument in a linear fashion, whereas in the second we can use it however many times we want.

From the definition of DIAL$_{\text{lin}}$, it’s clear that linear formulas are those containing no $\Rightarrow$. There are some specific features involving linear formulas, namely that we only allow linear formulas to be substituted for variables in rules $\forall e$, $\mu i$, and $\mu e$ (see Figure 1), and the premise of a linear implication must be linear.

These first three restrictions limit the amount of recursion possible in DIAL$_{\text{lin}}$ in general. A consequence is that the exponential function is not typable in
DIAL\textsubscript{lin}, and in fact all functions in DIAL\textsubscript{lin} are polynomial time computable in a certain sense. This raises some obvious questions about the result we’re trying to prove, which will be answered later, together with a formal proof of the former fact.

The restriction that a linear implication must have a linear premise is a bit unpleasant: after all, the linear arrow (or lollipop) says something about how the variables are used. Related to this issue is the fact that the linear context can only contain linear variables, which is also not something we would initially think to require. In this version of DIAL\textsubscript{lin}, linear use and linearity of variables are very much intertwined. Note that we can still obtain linear variables in the non-linear context via the use of (Derel). This intertwinedness is a bit annoying, but the proof of soundness seems to force our hand. We will explore options to get rid of it in Section 7.1.

Very important (but also quite standard) is the following property.

\textbf{Theorem 4.3} (Subject Reduction Property for DIAL\textsubscript{lin}). If $\Gamma; \Delta \vdash t : A$ and $t \rightarrow_{\beta} t'$, then $\Gamma; \Delta \vdash t' : A$

\textit{Proof.} By straightforward induction on the definition of $\rightarrow_{\beta}$. \hfill $\square$

We use two different types of numerals and words: Church-style, and Scott-style. Church-style numerals and words have some iterative structure in their definition, while Scott-style numerals and words are more linear. We’ve already seen Scott-style words over arbitrary alphabets. Here we will stick with the binary alphabet. It will also prove useful to have an explicit encoding of Scott-style bits. We use $\epsilon$ to denote an undefined bit.

\textbf{Definition 4.4} (Numerals, words, and bits). All words occur over a binary alphabet.

\begin{align*}
n_C &= \lambda f x. f^n x \text{ where } f^0 x = x, f^{n+1} x = f(f^n x).

w_C &= \lambda f_0 f_1 x. f_i ((f_{i_0} \ldots (f_{i_n} x) \ldots)) \text{ where } i_j = 0 \text{ if the } j^{\text{th}} \text{ character in } w \text{ is } 0 \text{ and } i_j = 1 \text{ if the } j^{\text{th}} \text{ character in } w \text{ is } 1.

0_S &= \lambda x y, \quad (n + 1)_S = \lambda x y. (n_S).

\epsilon_S &= \lambda x_0 x_1 y, \quad (0w)_S = \lambda x_0 x_1 y. x_0(w_S), \quad (1w)_S = \lambda x_0 x_1 y. x_1(w_S).

0_B &= \lambda f_0 f_1 x. f_0, \quad 1_B = \lambda f_0 f_1 x, \quad \epsilon_B = \lambda f_0 f_1 x x.
\end{align*}

We will first need to see that these are valid (typable) in DIAL\textsubscript{lin}.

\textbf{Proposition 4.5} (Validity of numerals and words). The following hold:

\begin{itemize}
  \item $\vdash n_C : N_C$ where $N_C \equiv \forall \alpha (\alpha \rightarrow \alpha) \Rightarrow (\alpha \rightarrow \alpha)$
  \item $\vdash w_C : W_C$ where $W_C \equiv \forall \alpha (\alpha \rightarrow \alpha) \Rightarrow (\alpha \rightarrow \alpha) \Rightarrow (\alpha \rightarrow \alpha)$
  \item $\vdash n_S : N_S$ where $N_S \equiv \mu \beta \forall \alpha (\beta \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$
  \item $\vdash w_S : W_S$ where $W_S \equiv \mu \beta \forall \alpha (\beta \rightarrow \alpha) \rightarrow (\beta \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$
  \item $\vdash i_B : B_S$ where $B_S \equiv \forall \alpha (\alpha \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha)$
\end{itemize}
Proof. \(\vdash n_C : N_C\): We first prove that \(f : \alpha \rightarrow \alpha; \ x : \alpha \vdash f^n x : \alpha\) for all \(n\), using induction.

\[
\frac{\vdash x : \alpha \vdash x : \alpha}{\vdash f : \alpha \rightarrow \alpha; \ x : \alpha \vdash f^n x : \alpha} \quad \text{(Weak 2)}
\]

\[
\frac{\vdash f : \alpha \rightarrow \alpha; \ x : \alpha \vdash f : \alpha \rightarrow \alpha}{\vdash f : \alpha \rightarrow \alpha; \ x : \alpha \vdash f^n x : \alpha} \quad \text{Induction Hypothesis}
\]

The result is now straightforward:

\[
\frac{\vdash f : \alpha \rightarrow \alpha; \ x : \alpha \vdash f^n x : \alpha}{\vdash \lambda x. f^n x : (\alpha \rightarrow \alpha) \Rightarrow (\alpha \rightarrow \alpha)} \quad \forall_i
\]

\(\vdash w_C : W_C\):
This proof is similar to the one for \(\vdash n_C : N_C\), with the exception that we have \(f_0\) and \(f_1\) rather than just one \(f\). Instead of proving \(f : \alpha \rightarrow \alpha; \ x : \alpha \vdash f^n x : \alpha\) for all \(n\), we must prove

\[
\vdash f_0 : \alpha \rightarrow \alpha, f_1 : \alpha \rightarrow \alpha; \ x : \alpha \vdash f_{i_0}(f_{i_1}(\ldots(f_{i_n}x)\ldots)) : \alpha
\]

for all possible \(i_0, \ldots, i_n\) and all \(n\). This is done in the same way as above.

\(\vdash n_S : N_S\):
We proceed by induction on \(n\).

\[
\frac{\vdash y : \alpha \vdash y : \alpha}{\vdash x : N_S \vdash y : \alpha} \quad \text{(Ax 2)}
\]

\[
\frac{\vdash x : N_S \vdash y : \alpha}{\vdash \lambda y. x : N_S \rightarrow \alpha \rightarrow \alpha} \quad \text{(Weak 1)}
\]

\[
\frac{\vdash \lambda y. y : N_S \rightarrow \alpha \rightarrow \alpha}{\vdash \lambda x. y : \forall (N_S \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)} \quad \forall_i
\]

\[
\frac{\vdash \lambda x. y : \forall (N_S \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)}{\vdash \lambda x. y : \mu \forall (\beta \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)} \quad \mu_i
\]

\[
\frac{\vdash x : N_S \rightarrow \alpha}{\vdash x : N_S \rightarrow \alpha} \quad \text{(Ax 2)}
\]

\[
\frac{\vdash y : \alpha \vdash x : N_S \rightarrow \alpha}{\vdash \lambda y. x : N_S \rightarrow \alpha \rightarrow \alpha} \quad \text{Induction Hypothesis}
\]

\[
\frac{\vdash \lambda y. x : N_S \rightarrow \alpha \rightarrow \alpha}{\vdash \lambda x. x : (N_S \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)} \quad \forall_i
\]

\[
\frac{\vdash \lambda x. x : (N_S \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)}{\vdash \lambda x. x : \mu \forall (\beta \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)} \quad \mu_i
\]
\( \vdash w_S : W_S : \)

Like the case \( \vdash w_C : W_C \), this is similar to the proof for \( \vdash n_S : N_S \). We simply mimic the proof, inducting on the length of the word, and add an extra \( \rightarrow i \) rule.

\( \vdash i_b : B_S : \)

We do the proof for \( \epsilon_b \equiv \lambda f_0 f_1 x. x \). The other two cases are similar.

\[
\begin{align*}
& \frac{\vdash x : \alpha \vdash x : \alpha}{\vdash f_0 : \alpha, f_1 : \alpha, x : \alpha \vdash x : \alpha} \quad \text{(Ax2)} \\
& \frac{\vdash f_0 : \alpha, f_1 : \alpha, x : \alpha \vdash x : \alpha}{\vdash f_0 : \alpha, f_1 : \alpha \vdash \lambda x.x : \alpha \to \alpha} \quad \text{\( \rightarrow i \)} \\
& \frac{\vdash f_0 : \alpha \vdash \lambda f_1 x.x : \alpha \to \alpha \to \alpha}{\vdash \lambda f_0 f_1 x.x : \alpha \to \alpha \to \alpha \to \alpha \quad \text{\( \rightarrow i \)}}
\end{align*}
\]

Now we can state our main result, characterizing all polynomial time computable functions using a type in \text{DIAL}_{\text{lin}}.

**Theorem 4.6 (Main Result).** A function \( f : \Sigma \to \Sigma \) (or \( f : N \to N \)) is in \text{FP} if and only if there exists a \( \lambda \)-term \( t \) of type \( W_C \Rightarrow W_S \) (respectively \( N_C \Rightarrow N_S \)) satisfying

\[ f(x) = y \iff t(x_C) =_\beta y_S \]

That is, a function is polynomial if and only if it’s of type Church \( \Rightarrow \) Scott.

Note here the \( =_\beta \). This “hides” a calculation. By the subject reduction property, we know \( t(x_C) \) is of type \( w_S \), and so is \( y_S \). However, while \( y_S \) is in normal form, \( t(x_C) \) isn’t. In order to see these two are equal (or not), we need to find the word which is represented by \( t(x_C) \), and this requires us to do some work.

This theorem is valid for both numerals and binary words. It can also be shown that it holds for words over an arbitrary alphabet, but this yields no additional insight. In light of this result we also might consider numerals to be words over a unary alphabet.

As is common, we will break down the proof of this “if and only if”-statements in two parts. The “only if”-part, completeness, is fairly simple, whereas the “if”-part, soundness, requires more work.

It is also possible to define and type Scott-style and Church-style trees in \text{DIAL}_{\text{lin}}, and thus state the main result for trees. Surprisingly, while the proof of soundness holds for these data types, the proof of completeness runs into some difficulties. We will see more in this in Section 7.3.
5 Completeness

Completeness is the statement that any polynomial time computable function from words to words is representable in $\text{DIAL}_{\text{lin}}$. The proof of this statement will be considerably easier if we don’t have to deal with adding numerals to each other. The following remark shows that this can be done.

**Remark 5.** For a given polynomial $p(n) = p_0 + p_1 n + p_2 n^2 + \ldots + p_d n^d$ we have:

$$p(n) = p_0 + p_1 n + p_2 n^2 + \ldots + p_d n^d \leq (p_0 + p_1 + \ldots + p_d) n^d$$

Hence if we have a polynomial time Turing Machine, we can assume the upper bound on its number of steps is given by a polynomial of the form $an^b$.

**Theorem 5.1** (FP-completeness). Suppose $f : \Sigma \to \Sigma$ is computable in polynomial time. Then there exists a term $t$ of type $\text{W}_C \Rightarrow \text{W}_S$ such that:

$$f(x) = y \iff [t(x_C)] =_\beta y_S$$

**Proof.** Unfortunately, we cannot simply inspect the proof of Theorem 3.12 and see that the simulating term has the correct type in the case the Turing machine takes a polynomial amount of steps, because $\text{DIAL}_{\text{lin}}$ does not admit arbitrary recursion.

However, when we have a fixed bound on the number of steps, we can use Church-numerals to enumerate the transition function: $nCfx \beta$-reduces to $f^n x$. We can then proceed as follows: the actual simulation of the Turing machine is done entirely using Scott-words, as before. We need the following ingredients:

- A way to go from the input and the given upper bound (which is a polynomial) to the Church-numeral for the upper bound for this input.
- A way to go from Church-words to Scott-words: the input is given in Church form, but we want to do the simulation with the tape represented using Scott-words.
- Replacing recursion by iteration, in the definition of $T$ and the string operators. Since the given Turing machine is polynomial time, we have upper bounds on the number of iterations, so we can use Church-numerals.

For the first item, consider the term $\lambda wfx.wffx$. Applied a Church-word of length $n$, this is $\beta$-equivalent to the Church-numeral $n_C$. We can multiply two Church-numerals by applying them to each other (that is, $(nm)_C$ is $\beta$-equivalent to $n_Cm_C$), and we can do this a for any $n$ a fixed number of times to arrive at a monomial. By remark 5 this suffices.

For the second item, define $M_0 \equiv \lambda w x_0 x_1 y. x_0 w$, $M_1 \equiv \lambda w x_0 x_1 y. x_1 w$. A straightforward calculation shows that $w_CM_0M_1 \epsilon S$ reduces to $w_N$.

For the third item, it suffices to replace every occurrence of $H$ by the relevant Church-numeral. Since the Church-numeral is polynomial in size, this does not yield any additional difficulty.

\[\square\]
6 Soundness

Soundness is a lot more intricate than completeness. The basic idea is to go show that everything is polynomial in $\text{DIAL}_{\text{lin}}$. The proof goes through the possible typing rules and see that a combinatorial explosion can’t take place. Formally, to every term $t$ of type $A$ in $\text{DIAL}_{\text{lin}}$ we can associate a polynomial $p$ such that the resources used by $t$ (be it cost of reducing $t$ to weak call-by-value normal form, or something else) are bounded by $p$. Existence of such a polynomial can be proven by induction on the derivation.

6.1 Realizability

Theorem 6.1 (FP-soundness). Let $t$ be a term of type $W_C \Rightarrow W_S$.

Let $f : \Sigma \rightarrow \Sigma$ be defined by

$$f(x) = y \iff [t(x_C)] =_\beta y_S$$

Then $f$ is polynomial time computable.

The proof will proceed in roughly three parts: first we introduce the specific flavour of polynomial we’re going to use, then we introduce our realizability relation, which associates polynomials with terms and types, and finally we prove Adequacy: every typable term has a polynomial majorizing the resources used by that term.

In reality our polynomials are simply additive terms in multiple variables, but we will call them higher order polynomials all the same.

Definition 6.2 (Higher order polynomials). Consider the simple types defined by the following grammar.

$$\sigma, \tau ::= o \mid \sigma \rightarrow \sigma$$

$o$ is called the base type. We assume a set of variables $V(\sigma)$ for every simple type $\sigma$.

A higher order polynomial of type $\sigma$, denoted $p : \sigma$, is built as follows:

- If $x \in V(\sigma)$, then $x : \sigma$.
- If $n \in \mathbb{N}$, then $n : o$.
- We have a term $+ : o \rightarrow o \rightarrow o$.
- If $p : \sigma \rightarrow \tau$ and $q : \sigma$, then $pq : \tau$.
- If $x \in V(\sigma)$ and $p : \tau$, then $\lambda x.p : \sigma \rightarrow \tau$.

We identify higher order polynomials up to $\alpha\beta\eta$-equivalence (so we may do reduction steps, rename variables, and add or drop abstractions), as well as simple arithmetic equivalences (where we interpret the term $+$ as addition).

We often write $p(q_1, \ldots, q_n)$ for $pq_1 \ldots q_n$, as well as $kp$ for $p + p + \ldots + p$ ($k$ times) if $k \in \mathbb{N}$ and $p : o$.

The complete set of higher order polynomials is denoted $\Pi$. 

32
We note that higher order polynomials are monotone: we only use addition and non-negative numbers in their definition.

Addition of two higher order polynomials, both not of type $o$ is not defined yet. It is not necessary for DIAL$_{lin}$ in this form and is related to the requirement that the premise of $\rightarrow o$ be linear: in the proof of Adequacy, we need to add the majorizer for the premise to the majorizer for the conclusion. We will see a definition of addition of two arbitrary higher order polynomials in Section 7.1.

We can also define a lowering operator, which will bring a term of higher order type down to base type. This is simply done by setting all variables equal to 0. By monotonicity, this gives a lower bound on the value of a higher order polynomial.

**Definition 6.3** (Lowering operator $\downarrow$). First define $0_o = 0$, $0_{\sigma \rightarrow \tau} = \lambda x.0_{\tau}$.

If $p : o$, then $\downarrow p = p$.

If $p : \sigma \rightarrow \tau$, then $\downarrow p = \downarrow (p0_{\sigma})$.

The complete set of higher order polynomials is denoted by $\Pi$.

Note that for a higher order polynomial $p$ of type $\tau$, $\downarrow p$ is a higher order polynomial of type $o$, namely the application of some zeroes corresponding to the type $\tau$ to $p$.

The idea is to capture the non-linearity of a type using these higher order polynomials: the type of the polynomial corresponds to the non-linear structure in the term. More specifically, we have a map $o$ from formulas in DIAL$_{lin}$ to simple types.

$$o(L) = o, \; o(L \rightarrow A) = o(A), \; o(A \Rightarrow B) = o(A) \rightarrow o(B), \; o(\forall \alpha A) = o(A)$$

Using the higher order polynomials we will introduce a realizability relation $t, p \vdash_\eta A$ where (assuming variables in $\eta$) $t$ is a term of type $A$, and $p$ provides an upper bound on potential number of $\beta$-steps to be taken upon applying $t$ to arguments. This allows us to conclude that the function computed by $t$ is polynomial time computable.

Before we can introduce realizability properly, we will need to define saturated sets. They help us deal with free variables and provide some very useful properties. The following notation will be very useful.

**Definition 6.4** (Notation).

- Finite sequences are denoted by a bar: $\bar{x}, \bar{u}, \bar{y}, \ldots$.
- The $i$-th element of a sequence $\bar{x}$ (or $\bar{u}, \bar{y}, \ldots$) is denoted $x_i$ (respectively $u_i, y_i, \ldots$).
- In a $\lambda$-term, a finite sequence corresponds to the application of all its elements in order, that is $\bar{x} \equiv x_1 x_2 \ldots x_n$.
- We write $M\langle y_1/x_1, \ldots, y_n/x_n \rangle$ or $M\langle \bar{y}/\bar{x} \rangle$ for $(\lambda x_1 \ldots x_n.M)y_1 \ldots y_n$.
- $\theta, \kappa$ denote lists of the form $y_1/x_1, \ldots, y_n/x_n$.

Intuitively, angled brackets correspond to a kind of “pre-substitution”. If we perform some $\beta$ steps on a term $M\langle y_1/x_1, \ldots, y_n/x_n \rangle$, we obtain $M[y_1/x_1, \ldots, y_n/x_n]$. 

33
Definition 6.5 (Saturated sets). Let $\tau$ be a higher order polynomial type. A set $X \subseteq \Lambda \times \Pi$ is called saturated set (satset) of type $\tau$ if $X \neq \emptyset$, and for all $(t, p) \in X$ we have:

- **Bound:** $\text{cost}(t) \leq p$.
- **Monotonicity:** For all $n \in \mathbb{N}$: $(t, p + n) \in X$.
- **Exchanging:** If $t = t_0(\theta, v_1/y_1, v_2/y_2, \kappa)\bar{u}$, then $(t_0(\theta, v_2/y_2, v_1/y_1, \kappa)\bar{u}, p) \in X$.
- **Weakening:** If $t = t_0(\theta)\bar{u}$, $w$ is a term that normalizes, and $z \notin \text{FV}(t_0)$, then $(t_0(\theta, w/z)\bar{u}, p + \text{cost}(w) + 2) \in X$.
- **Contraction:** If $t = t_0(\theta, v/y_1, v/y_2, \kappa)\bar{u}$, then $(t_0[z/y_1, z/y_2](\theta, v/z, \kappa)\bar{u}, p) \in X$.
- **Concatenation:** If $t = (t_0(\theta))(t_1(\kappa))\bar{u}$, then $(t_0t_1(\theta, \kappa)\bar{u}, p) \in X$.
- **Identity:** If $t = t_0\bar{u}$, then $(x(t_0/x)\bar{u}, p + 3) \in X$.

$p$ is a higher order polynomial majorizing the cost of reducing $t$ to (weak call-by-value) normal form, as the first condition implies.

Since we will see variations on the theme of realizability later on, it is useful to build a bit of insight in the definition of satsets and how it corresponds to our cost model. This will give us a starting point for giving the appropriate definitions in other contexts.

Proposition 6.6. There exists a saturated set $X_0$ of type $\alpha$, given by

$$X_0 = \{(t, n) : t \Downarrow \text{ and } \text{cost}(t) \leq n\}$$

Furthermore, $X_0$ is the greatest saturated set of type $\alpha$.

Proof. $(\lambda x.x, 2) \in X_0$, so $X_0 \neq \emptyset$. Suppose $(t, n) \in X_0$, then $t \Downarrow$ and $\text{cost}(t) \leq n$.

- **Bound:** satisfied due to the definition.
- **Monotonicity:** let $m \in \mathbb{N}$. Then $\text{cost}(t) \leq n \leq n + m$, so $(t, n + m) \in X_0$.
- **Exchanging:** exchanging the order of two redexes doesn’t change the cost of reducing the term: since we do not reduce in the body of a $\lambda$-term, it is “immobilized” until we have fired both redexes.
- **Weakening:** the length of the term is increased by 1 (because of the extra $\lambda$), plus $|W|$. Then the cost of reducing the term increases by the cost for reducing $W$ to normal form, since we can only have values as arguments, as well as an extra step for firing the new redex with $W$ as argument. In total this gives us an extra cost of $2 + \text{cost}(W)$, since $|W|$ is contained in $\text{cost}(W)$. 34
• Contraction: we reduce the length by $1 + |v|$, because we remove a $\lambda$ and the corresponding argument in contracting $y_1, y_2$. The cost for firing the new redex does not exceed the cost of firing the two original redexes since the size increase is majorized by the sum of the size increases.

• Concatenation: this is just interchanging the order of the $\lambda$’s and the arguments.

• Identity: the new term is $(\lambda x.x)t_0u$, which gives a size increase of 2, and a cost of 1 for firing the redex.

Hence $X_0$ is a saturated set of type $o$. To see that it is the greatest, let $Y$ be a saturated set of type $o$, and let $(t, n) \in Y$. We have $t$ terminating, and by condition bound, $\text{cost}(t) \leq n$, so $(t, n) \in X_0$.

**Definition 6.7.** A valuation $\eta$ maps a propositional variable $\alpha$ to a saturated set of type $o$.

We write $\eta\{\alpha \mapsto X\}$ for the valuation which maps $\beta$ to $\eta(\beta)$ for all $\beta \neq \alpha$, and $\alpha$ to $X$.

Valuations assign values to free variables in formulas.

Finally we are ready to define realizability.

**Definition 6.8 (Realizability).** We define the relation $t, p \vdash_\eta A$ where $t$ is a term called the realizer of $A$, $p$ a higher order polynomial of type $o(A)$, called the majorizer of $t$, $\eta$ a valuation, and $A$ a type.

We write $\bar{A}_\eta = \{(t, p) : t, p \vdash_\eta A\}$.

$\vdash_\eta$ is inductively defined on $A$.

• $t, p \vdash_\eta \alpha$ for a propositional variable $\alpha$ if and only if $(t, p) \in \eta(\alpha)$.

• $t, p \vdash_\eta L \rightarrow A$ if and only if $\text{cost}(t) \leq \downarrow p$ and for all $u, m$, $u, m \vdash_\eta L \Longrightarrow tu, p + m \vdash_\eta A$.

• $t, p \vdash_\eta A \Rightarrow B$ if and only if $\text{cost}(t) \leq \downarrow p$ and for all $u, q$, $u, q \vdash_\eta A \Longrightarrow tu, p(q) \vdash_\eta B$.

• $t, p \vdash_\eta \forall \alpha A$ if and only if $t, p \vdash_{\eta(\alpha \rightarrow X)} A$ for every saturated set $X$ of type $\alpha$.

• $t, p \vdash_\eta \mu \alpha L$ if and only if $(t, p) \in X$ for every saturated set $X$ of type $\alpha$ satisfying $\bar{L}_{\eta(\alpha \rightarrow X)} \subseteq X$.

It is clear that realizability mimics the intended meaning of terms, giving explicit cost bounds in the process. The valuation subscript is necessary in order to deal with unbound propositional variables, but in the end we will only be interested in closed formulas, such as $W_C, W_S, B_S$.

As stated before, Adequacy is our main goal in this section. It shows that all terms typable in DIALadmit a polynomial majorizer, so in particular terms of type $W_C \Rightarrow W_S$ do.
**Theorem 6.9** (Adequacy Theorem). Suppose \( \bar{x} : \bar{C}, \bar{y} : \bar{M} \vdash t : A \) is derivable. Let \( \eta \) be any valuation. Then there exists a higher order polynomial \( p(\bar{x}) : o(A) \) where the variables \( \bar{x} \) are of types \( o(C) \) satisfying the following.

Suppose there exist realizers \( \bar{u}, \bar{s} \) for \( \bar{C} \) and \( \bar{M} \) with majorizers \( \bar{q}, \bar{m} \), respectively. Formally

\[
\bar{u}, \bar{q} \vdash_{\eta} \bar{C}, \bar{s}, \bar{m} \vdash_{\eta} \bar{M}
\]

Then \( t(\bar{u}/\bar{x}, \bar{s}/\bar{y}) \) realizes \( A \) with majorizer \( p(\bar{q}) + \bar{m} \). Formally

\[
t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash_{\eta} A
\]

This formulation differs slightly from that of Brunel and Terui \([5]\): they have an additional bounding condition. The proof they give requires this bounding condition for the cases \((\neg \alpha)\) and \((\Rightarrow \alpha)\), but it is not necessary, as we’ll see. It is also not mentioned in the other article \([4]\).

The following lemma will be useful in proving the Adequacy Theorem. It establishes a correspondence between saturated sets and realizability, a fact which will be used extensively in the proof of the Adequacy Theorem, as well as some other useful properties of realizability.

**Lemma 6.10** (Realizability Lemma). For all formulas \( A \), linear formulas \( L \), and valuations \( \eta \):

1. \( \bar{A}_{\eta} \) is a saturated set of type \( A \).
2. \( t, p \vdash_{\eta} A[L/\alpha] \iff t, p \vdash_{\eta(\alpha \Rightarrow L_{\eta})} A \).
3. \( t, p \vdash_{\eta} \forall \alpha A \iff t, p \vdash_{\eta} A[L/\alpha] \).
4. If \( \alpha \) occurs only positively in \( L \), \( f(X) = \bar{L}_{\eta(\alpha \Rightarrow X)} \) is monotone increasing in \( X \). If \( \alpha \) occurs only negatively in \( L \), \( f(X) \) is monotone decreasing.
5. \( \bar{\mu} \alpha \bar{L}_{\eta} \) is the least fixpoint of \( f(X) = \bar{L}_{\eta(\alpha \Rightarrow X)} \).
6. \( t, p \vdash_{\eta} \bar{\mu} \alpha L \iff t, p \vdash_{\eta} L[\bar{\mu} \alpha L/\alpha] \).

**Proof.**

1. By induction on \( A \) in the definition of realizability.

For \( A = \alpha \), we have \( t, p \vdash_{\eta} A \iff (t, p) \in \eta(A) \), so \( \bar{A}_{\eta} = \eta(A) \). By definition of valuation, \( \eta(A) \) is a saturated set.

For \( A = L \Rightarrow B \), we assume \( \bar{L}_{\eta}, \bar{B}_{\eta} \) are saturated.

- **Bound:** satisfied by the assumption that \( \text{cost}(t) \leq t \).
- **Monotonicity:** Let \( n \in \mathbb{N} \). For any \( u, m \in \bar{L}_{\eta} \), we have \( (tu, p + m + n) \in \bar{B}_{\eta} \) by induction hypothesis. Since \( m + n = n + m \) (we identified higher order polynomials up to arithmetic equivalences), this proves that \( t, p + n \vdash_{\eta} L \Rightarrow B \), so \( t, p + n \in \bar{A}_{\eta} \).
- **Exchange, weakening, contraction, concatenation, and identity:** similar to monotonicity.

36
For $A = B \Rightarrow C$, the proof is similar to the above.

For $A = \forall \alpha B$, by induction hypothesis, we have $\bar{B}_{\eta(\alpha \rightarrow X)}$ a saturated set for any saturated $X$ of type $\alpha$.

- **Bound**: $(t, p) \in \bar{B}_{\eta(\alpha \rightarrow X)}$ by assumption, so $\text{cost}(t) \leq \downarrow p$.
- **Monotonicity**: Let $n \in \mathbb{N}$. By induction hypothesis, for any saturated $X$ of type $\alpha$: $t, p + n \not\models_{\eta(\alpha \rightarrow X)} B$, which proves $(t, p + n) \in \bar{A}_\eta$ by definition of realizability.
- **Exchange, weakening, contraction, concatenation, and identity**: similar to monotonicity.

For $A = \mu \alpha L$:

- **Bound**: $(t, p) \in \bar{A}_\eta$, so for every saturated set $X$ of type $\alpha$ satisfying $\bar{L}_{\eta(\alpha \rightarrow X)} \subseteq X$, we have $(t, p) \in X$. In particular, this holds for $X_0$, so $(t, p) \in X_0$, which proves condition bound.
- **Monotonicity**: Let $n \in \mathbb{N}$. For every saturated set $X$ of type $\alpha$ satisfying $\bar{L}_{\eta(\alpha \rightarrow X)} \subseteq X$, we have $(t, p) \in X$. Then also $(t, p + n) \in X$, which proves condition monotonicity.
- **Exchange, weakening, contraction, concatenation, and identity**: similar to monotonicity.

2. By induction on $A$ in the definition of realizability.

For $A = \alpha$, we have $t, p \not\models_{\eta} A[L/\alpha] \iff (t, p) \in \bar{L}_\eta$ by item 1, since $A[L/\alpha] = L$. This happens if and only if $t, p \not\models_{\eta(\alpha \rightarrow X)} A$ by definition of realizability.

For $A = M \rightarrow B$, note $A[L/\alpha] = M[L/\alpha] \rightarrow B[L/\alpha]$. Then $t, p \not\models_{\eta} A[L/\alpha]$ if and only if for all $u, m$ with $u, m \not\models_{\eta} M[L/\alpha]$ we have $tu, p + m \not\models_{\eta} B[L/\alpha]$. Combining this with our induction hypothesis, we obtain $t, p \not\models_{\eta} A[L/\alpha]$ if and only if for all $u, m$ with $u, m \not\models_{\eta(\alpha \rightarrow X)} M$ we have $tu, p + m \not\models_{\eta(\alpha \rightarrow X)} B$, which is precisely the definition of $t, p \not\models_{\eta(\alpha \rightarrow X)} A$.

For $A = B \Rightarrow C$, the proof is similar to the above.

For $A = \forall \beta B$, note $A[L/\alpha] = \forall \beta(B[L/\alpha])$. Now $t, p \not\models_{\eta} A[L/\alpha]$ if and only if for all saturated sets $X$ of type $\alpha$ we have $t, p \not\models_{\eta(\beta \rightarrow X)} B[L/\alpha]$, which by induction hypothesis is equivalent to for all saturated sets $X$ of type $\alpha$ we have $t, p \not\models_{\eta(\beta \rightarrow X)} B$, which is precisely the definition of $t, p \not\models_{\eta(\alpha \rightarrow X)} B$.

For $A = \mu \beta M$, note $A[L/\alpha] = \mu \beta(M[L/\alpha])$. Now $t, p \not\models_{\eta} A[L/\alpha]$ if and only if for all saturated sets $X$ of type $\alpha$ satisfying $M[L/\alpha]_{\eta(\beta \rightarrow X)} \subseteq X$ we have $(t, p) \in X$. By item 1 and the induction hypothesis, for all saturated sets $X$ of type $\alpha$ satisfying $M_{\eta(\beta \rightarrow X, \alpha \rightarrow X)} \subseteq X$ we have $(t, p) \in X$. This is precisely the definition of $t, p \not\models_{\eta(\alpha \rightarrow X)} A$. 

37
3. Suppose \( t, p \models \eta \forall \alpha A \), and let \( L \) be a linear formula. Since \( \overline{L}_\eta \) is saturated, we have \( t, p \models \eta[\alpha \mapsto \overline{L}_\eta] A \) by definition of realizability. By item 2, it follows that \( t, p \models \eta A[L/\alpha] \).

4. By induction on \( L \).

If \( L = \alpha \), \( \alpha \) occurs positively in \( L \), so the second statement vacuously holds. For the first statement, note \( f(X) = \overline{L}_\eta[\alpha \mapsto X] = X \), which is clearly monotone increasing.

If \( L = \forall \beta M \), assume \( \alpha \) occurs only positively in \( L \). By definition of positive occurrences, \( \alpha \) occurs positively in \( M \). By induction hypothesis, \( g(X) = \overline{M}_{\eta'}[\alpha \mapsto X] \) is monotone increasing for any valuation \( \eta' \).

\( (t, p) \in \overline{L}_\eta[\alpha \mapsto X] \) if and only if \( (t, p) \in \overline{M}_{\eta'}[\beta \mapsto \alpha \mapsto Y] \) for any set \( Y \) of type \( \alpha \), so \( f(X) \) is monotone increasing as well. A similar argument works for the negative occurrence case.

If \( L = \mu \beta M \), the proof is similar to the one for \( \forall \beta M \).

If \( L = M \rightarrow N \), assume \( \alpha \) occurs only positively in \( L \). Then \( \alpha \) occurs only negatively in \( M \), and only positively in \( N \). Now suppose we have \( X \in Y \).

By induction hypothesis, \( \overline{M}_{\eta} \subseteq \overline{M}_{\eta'} \subseteq \overline{N}_{\eta} \subseteq \overline{N}_{\eta'} \).

Thus \( (t, p) \in \overline{L}_\eta \) if and only if \( (t, p) \in \overline{M}_{\eta'} \subseteq \overline{N}_{\eta} \).

A similar argument works for the negative occurrence case.

5. Because \( \alpha \) occurs only positively in \( L \), \( f \) is monotone increasing. We say a saturated set \( X \) of type \( \alpha \) is a prefixpoint if \( f(X) \subseteq X \). \( \mu \alpha L \) is the infimum of all prefixpoints of \( f \), since \( (t, p) \in \mu \alpha L \) if and only if \( (t, p) \in X \) for all saturated sets \( X \) of type \( \alpha \) satisfying \( f(X) \subseteq X \).

Since \( f \) is monotone, \( f(\mu \alpha L) \subseteq f(X) \subseteq X \) for all prefixpoints \( X \). \( \mu \alpha L \) is an infimum of these, so \( f(\mu \alpha L) \subseteq \mu \alpha L \).

Since \( f \) is monotone, \( f(f(\mu \alpha L)) \subseteq f(\mu \alpha L) \), so \( f(\mu \alpha L) \) is a prefixpoint. Hence \( \mu \alpha L \subseteq f(\mu \alpha L) \).

This proves \( f(\mu \alpha L) = \mu \alpha L \), \( \mu \alpha L \) is a fixpoint of \( f \). It is the least, since any fixpoint of \( f \) is also a fixpoint.

6. By items 2 and 5.

\[
(t, p) \models \eta L[\mu \alpha L/\alpha] \iff (t, p) \models \eta[\alpha \mapsto \overline{L}_\eta] L \\
\quad \iff (t, p) \in \overline{L}_\eta[\alpha \mapsto \overline{\mu \alpha L}]
\]

\[\square\]
We can now prove the Adequacy theorem. As stated before, we go along the typing derivation and explicitly give a polynomial that works as a majorizer. The most interesting steps are those involving $\to$ and $\Rightarrow$. The ones involving $\mu$ are also a bit interesting in and of themselves, but the fundamental observation about their correctness was proved in the Realizability Lemma.

**Proof of the Adequacy Theorem.** We induct on the derivation of $\bar{x} : \bar{C}, \bar{y} : \bar{M} \vdash t : A$.

We will drop the valuation subscript if it doesn’t matter for the proof.

We write $\Gamma = \bar{x} : \bar{C}, \Delta = \bar{y} : \bar{M}$. Terms in $\bar{u}$ will be substituted for those in $\bar{x}$, and likewise for those $\bar{s}$ and $\bar{y}$. $p, q$ will always denote higher order polynomials, while $m$ will always denote a natural number. The lengths of $\bar{x}$ and $\bar{y}$ are denoted by $l_x$ and $l_y$, respectively.

It is obvious how to add a list of natural numbers to a higher order polynomial. We will simply write $p + \bar{m}$ for $p + m_1 + m_2 + \ldots + m_l$.

These conventions extend to subscripts and accents.

Case (ax1).

We show $p(x) = x + 3$ is a majorizer. Suppose we have $u, q$ such that $u, q \vdash A$. Using Lemma 6.10, condition identity for saturated sets implies that $x(u/x), q + 3 \vdash A$, so $p$ suffices.

Case (ax2).

We have $; x : L \vdash x : L$, with $t = x$. We show $p = 3$ is a majorizer. Suppose we have $s, m \vdash L$. Condition identity for saturated sets implies $x(s/x), m + 3 \vdash L$, and $m + 3 = p + m$.

Case ($\mu_e$).

\[
\begin{align*}
\Gamma; \Delta \vdash t : \mu \alpha L & \\
\Gamma; \Delta \vdash t : L[\mu \alpha L/\alpha]
\end{align*}
\]

Suppose we have $\bar{u}, \bar{q} \vdash \bar{C}$ and $\bar{s}, \bar{m} \vdash \bar{M}$.

By induction hypothesis, $t(\bar{a}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash \mu \alpha L$. By item 5 of Lemma 6.10 we obtain $t(\bar{a}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash L[\mu \alpha L/\alpha]$.

Case ($\mu_i$).

\[
\begin{align*}
\Gamma; \Delta \vdash t : L[\mu \alpha L/\alpha] & \\
\Gamma; \Delta \vdash t : \mu \alpha L
\end{align*}
\]

Similar to the above, using item 5 of Lemma 6.10.
Case ($\forall_\alpha$).
\[
\frac{\Gamma; \Delta \vdash t : \forall \alpha A}{\Gamma; \Delta \vdash t : A[L/\alpha]}
\]
Direct from induction hypothesis and item 3 of Lemma 6.10.

Case ($\forall_i$).
\[
\frac{\Gamma; \Delta \vdash t : A \quad \alpha \notin \text{FV}(\Gamma; \Delta)}{\Gamma; \Delta \vdash t : \forall \alpha A}
\]
From the induction hypothesis we obtain a majorizer $p$ of the premise. We show that $p$ is also a suitable majorizer for the conclusion.

Suppose $\bar{u}, \bar{q} \vdash_{\eta} C$ and $\bar{s}, \bar{m} \vdash_{\eta} M$, then $t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash_{\eta} A$ for any valuation $\eta$.

By definition of realizability, $t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash_{\eta} \forall \alpha A$ if and only if for every saturated set $X$ of type $o$, $t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash_{\eta(\alpha \Rightarrow X)} A$. This is true since $\eta(\alpha \Rightarrow X)$ is still a valuation.

Case ($\Rightarrow_\alpha$).
\[
\frac{\Gamma_1; \Delta \vdash t_1 : A \Rightarrow B \quad \Gamma_2; \Delta \vdash t_2 : A}{\Gamma_1, \Gamma_2; \Delta \vdash t_1 t_2 : B}
\]
From the induction hypothesis we obtain majorizers $p_1, p_2$ of the premise. $p_1$ is of type $o(A \Rightarrow B) = o(A) \rightarrow o(B)$ and $p_2$ is of type $o(A)$. We show that $p' = p_1(p_2)$ is a suitable majorizer.

Suppose $\bar{u}_1, \bar{q}_1 \vdash C_1$ and $\bar{u}_2, \bar{q}_2 \vdash C_2$, and $\bar{s}, \bar{m} \vdash M$.
Then $t_1(\bar{u}_1/\bar{x}_1, \bar{s}/\bar{y}), p_1(z, \bar{q}) + \bar{m} \vdash A \Rightarrow B$, and $t_2(\bar{u}_2/\bar{x}_2), p_2(\bar{q}_2) \vdash A$.

Applying the definition of realizability for $A \Rightarrow B$, we obtain
\[
t_1(\bar{u}_1/\bar{x}_1, \bar{s}/\bar{y})t_2(\bar{u}_2/\bar{x}_2), p_1(p_2(\bar{x}_2), \bar{x}_1) \vdash B
\]
Applying conditions monotonicity, concatenation, and exchange yields
\[
t_1 t_2(\bar{u}_1/\bar{x}_1, \bar{s}/\bar{y}), p'(\bar{x}_1, \bar{x}_2) \vdash B
\]

Case ($\Rightarrow_i$).
\[
\frac{\Gamma, z : A; \Delta \vdash t : B}{\Gamma; \Delta \vdash \lambda z.t : A \Rightarrow B}
\]
From the induction hypothesis, we obtain a majorizer of the premise. We will define $p' = \lambda z.p(z)$ and show $p'$ is a majorizer of the conclusion.

Suppose $\bar{u}, \bar{q} \vdash C$ and $\bar{w}, \bar{r} \vdash A$, and $\bar{s}, \bar{m} \vdash M$. Unfolding the definition of realizability for $A \Rightarrow B$, we have to prove two things.

\[
\text{cost}((\lambda z.t)(\bar{u}/\bar{x}, \bar{s}/\bar{y})) \leq p'(\bar{q}) + \bar{m}
\]

40
We show

\[ (\lambda z.t)(\bar{u}/\bar{x}, \bar{s}/\bar{y})w, p'(r, \bar{q}) \vdash B \]

The first item holds: \( \text{cost}((\lambda z.t)(\bar{u}/\bar{x}, \bar{s}/\bar{y})) \leq \text{cost}(t(\bar{u}/\bar{x}, \bar{s}/\bar{y}, w/z)) \) for any term \( w \), because the reducing the \( u_t, s_t \) to normal form and firing the corresponding \( \beta \)-redexes will always happen, but then the first reduction will stop, whereas the second will also reduce \( w \) to normal form and then fire the corresponding redex, after which we might get even more reduction steps.

The second item follows using conditions monotonicity and exchange for the premise.

Case \((\to_r)\).

\[
\frac{\Gamma_1; \Delta_1 \vdash t_1 : L \to B \quad \Gamma_2; \Delta_2 \vdash t_2 : L}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t_1t_2 : B}
\]

From the induction hypothesis, we obtain majorizers \( p_1, p_2 \) of the premises. We will show \( p = p_1 + p_2 \) is a suitable majorizer for the conclusion. Note that this is well-defined because \( p_2 \) is of type \( o(L) = o \).

Suppose \( \bar{u}_i, q_i \models C_i \) and \( \bar{s}_i, m_i \models M_i \) for \( i = 1, 2 \). By the definition of realizability for \( L \to B \) we obtain

\[
t_1(\bar{u}_1/\bar{x}_1, \bar{m}_1/\bar{y}_1)t_2(\bar{u}_2/\bar{x}_2, \bar{m}_2/\bar{y}_2), p_1(\bar{q}_1) + m_1 + p_2(\bar{q}_2) + m_2 \vdash B
\]

Applying conditions concatenation and exchange gives us

\[
t_1t_2(\bar{u}_1/\bar{x}_1, \bar{u}_2/\bar{x}_2, \bar{s}_1/\bar{y}_1, \bar{s}_2/\bar{y}_2), p(\bar{q}_1, \bar{q}_2) + \bar{m}_1 + \bar{m}_2 \vdash B
\]

By induction hypothesis, there exist polynomials \( p_1, p_2 \), such that, if we have \( \bar{u}_1, q_1 \models C_1 \) and \( \bar{s}_1, m_1 \models M \), then \( t_1(\bar{u}_1/\bar{x}_1, \bar{m}_1/\bar{y}_1), p_1(\bar{q}_1) + m_1 \vdash L \to B \), and if we have \( \bar{u}_2, q_2 \models C_2 \) and \( \bar{s}_2, m_2 \models M \), then \( t_2(\bar{u}_2/\bar{x}_2, \bar{m}_2/\bar{y}_2), p_2(\bar{q}_2) + m_2 \vdash L \). We show \( p(\bar{x}_1, \bar{x}_2) = p_1(\bar{x}_1) + p_2(\bar{x}_2) \) is a suitable majorizer.

Case \((\to_l)\).

\[
\frac{\Gamma; \Delta, z : L \vdash t : B}{\Gamma; \Delta \vdash \lambda z.t : L \to B}
\]

The proof for this is the same as for the case \((\Rightarrow)\), where we leave \( z \) out of the majorizer.

Case (Contr).

\[
\frac{\Gamma, z_0 : A, z_1 : A; \Delta \vdash t : B}{\Gamma, z : A; \Delta \vdash t[z/z_0, z/z_1] : B}
\]

From the induction hypothesis we obtain a majorizer \( p \) of the premise. We show that \( p' = \lambda z.p(z, z) \) is a suitable majorizer of the conclusion.

Suppose \( \bar{u}, \bar{q} \models C \) and \( w, r \models A \), and \( \bar{s}/\bar{y} \models M \). Then

\[
t(\bar{u}/\bar{x}, w/z_0, w/z_1, \bar{s}/\bar{y}), p(\bar{q}, r, r) + \bar{m}
\]
Applying condition weakening yields
\[ t[z/z_0, z/z_1](\bar{u}/\bar{x}, w/z, \bar{s}/\bar{y}, p'(\bar{q}, r) + \bar{m} \vdash B \]
which is what we wanted.

**Case (Derel).**

\[ \Gamma; \Delta, x : L \vdash t : B \]
\[ \Gamma, z : L; \Delta \vdash t[z/x] : B \]

From the induction hypothesis, we obtain a majorizer \( p \) for the premise. It’s easily seen \( p' = \lambda z.p + z \) that is a suitable majorizer for the conclusion.

**Case (Weak).**

\[ \Gamma; \Delta \vdash t : B \]
\[ \Gamma, \Gamma'; \Delta, \Delta' \vdash t : B \]

From the induction hypothesis, we obtain a majorizer \( p \) of the premise. We show \( p' = \lambda \bar{x}'.p + \sum_l \downarrow (x_l' + 2) + 2 \cdot l_{\bar{y}'} \) is a suitable majorizer for the conclusion.

Suppose \( x, \bar{q} \vdash C \) and \( \bar{x}', \bar{q}' \vdash C' \), and \( s, \bar{m} \vdash M \) and \( \bar{s}', \bar{m}' \vdash M' \). By induction hypothesis
\[ t(\bar{u}/\bar{x}, \bar{s}/\bar{y}, p(\bar{q}) + \bar{m} \vdash B \]
Applying condition weakening \( l_{x'} + l_{\bar{y}'} \) times as well as condition exchange yields
\[ t(\bar{u}/\bar{x}, \bar{u}' / x', \bar{s}/\bar{y}, \bar{s}' / \bar{y}', p'(\bar{q}, \bar{q}')) + \bar{m} + \bar{m}' \vdash B \]
which is what we wanted.

\[ \square \]

The proof for the cases \((\neg_1)\) and \((\Rightarrow_1)\) are considerably more complex in the original paper [3]. This isn’t necessary, as the above proof shows.

For a term \( t \) which is typable in \( \text{DIAL}_{\text{lin}} \), the Adequacy theorem gives a polynomially sized bound on cost(\( t \)), which immediately follows from condition bound. However, we are not interested in the cost of reducing a fixed term, as this has no impact. We are interested in functions, that is, terms of type \( t : A \Rightarrow B \) or \( t : A \rightarrow B \).

**Corollary 6.11.** Let \( t \) be a term with \( \vdash t : A \Rightarrow B \) (respectively \( A \rightarrow B \)) for some types \( A, B \). Let \( \Lambda_A = \{ u : \vdash u : A \} \), \( \Lambda_B = \{ u : \vdash u : B \} \) the sets of all \( \lambda \)-terms of types \( A \) and \( B \), respectively. \( t \) defines a map \( \Lambda_A \rightarrow \Lambda_B \) by \( t(x) = [tx] \).

Let \( \Sigma_A = \{ u^\# : u \in \Lambda_A \} \), \( \Sigma_B = \{ u^\# : u \in \Lambda_B \} \).

Suppose every term \( x \in \Lambda_A \) admits a majorizer \( q_x \) which is polynomial in \( |x| \) (not necessarily requiring \( |x| \) as an argument). Then there exists a polynomial time computable function \( f : \Sigma_A \rightarrow \Sigma_B \) such that the following diagram commutes:

\[ \begin{array}{ccc}
\Sigma_A & \xrightarrow{f} & \Sigma_B \\
\downarrow_{=} & & \downarrow_{=} \\
\Lambda_A & \xrightarrow{t} & \Lambda_B 
\end{array} \]
Proof. \( f \) is defined as follows: on input \( x^\# \in \Sigma_A \), \( f \) calculates \((tx)^\#\) and reduces this to weak call-by-value normal form. Clearly \( f(x^\#) = t(x)^\# \), so the diagram commutes. We have to prove that \( f \) is polynomial time computable. Calculating \((tx)^\#\) from \( x^\# \) can be done in linear time, since \((tx)^\# = \@_{t^\#} x^\#\), and \( t \) is fixed. The Adequacy Theorem gives us a (higher order) polynomial \( p \) such that \( t,p \in \mathcal{A} \Rightarrow B \). We by assumption, we also have a polynomial \( q \) such that \( x,q \in \mathcal{A} \Rightarrow B \). Using the definition of realizability for \( A \Rightarrow B \) and applying condition bound, we obtain cost \((tx) \leq \downarrow p(q_x)\).

Now \( |x^\#| \) (length as a string) is polynomial in \(|x| \) (length as a \( \lambda \)-term), so \(|x| \) is also polynomial in \(|x^\#|\). Then \( p(q_x) \) is polynomial in \(|x^\#|\), so cost\((tx)\) is. Applying Theorem [3.15](simulation of \( \lambda \)-terms by Turing machines) yields that the cost of reducing \((tx)^\#\) to weak call-by-value normal form is polynomial in \(|x^\#|\).

Hence both operations take polynomial time in the length of \( x^\# \) and \( f \) is polynomial time computable.

The assumption that our majorizer is polynomial in \(|x| \) is fairly mild: both Church and Scott words satisfy this assumption.

Our hope now would be that we can mimic a similar argument with the encodings of words in Church-style and Scott-style, slightly changing the definition of the function \( t \).

Plot Twist

Unfortunately, we can’t. \( t \mapsto t^\# \) translates a \( \lambda \)-term on a syntactic level: essentially we’re copying the structure of a \( \lambda \)-term onto the tape of a Turing machine. Our encodings operate on a more semantic level: a \( \lambda \)-term represents a specific word, and we’re interested in that word rather than the structure of the \( \lambda \)-term. The arrow with our encoding in the commuting diagram is going the wrong way!

In the following we write \( t \) for a term of type \( WC \Rightarrow WS \), \( x \) for a term of type \( WC \), \( w \) for the word represented by \( x \) and \( f : \Sigma \to \Sigma \) for the function corresponding to \( t \).

This semantic encoding has as a consequence that different \( \lambda \)-terms can represent the same word: indeed, even the term \( tx \) already represents a word in Scott-style without having to do any reductions. In order to show that the function \( f \) is polynomial time computable, we have to show that we can extract all the bits from \( tx \) in polynomial time.

The main difficulty here is that weak call-by-value reduction is not strong enough to do this in one go: while we can show that we can access the \( i \)-th bit of \( tx \) in time polynomial in \( i \) and the length of the word represented by \( tx \) (Proposition [6.24]), weak call-by-value cannot detect the length of the word represented by \( tx \). Hence we cannot prove that the length of the word represented by \( tx \) is polynomial with our current theorems. This is however essential in proving that we can extract all the bits from \( tx \) in polynomial time, so we need a different approach.
The next section adapts the realizability tools from the previous section in order to show that the length of the word represented by $tx$ is polynomial. The proof of soundness then proceeds as follows: on input $w$, we calculate $x#$. Using our weak call-by-value bit extraction procedure we obtain polynomial time computable functions $f_0, f_1, \ldots$ such that $f_i(x#)$ yields $b_{2i}$, where $b$ is the $i$-th bit of the word represented by $tx$. We can easily translate this to the actual $i$-th bit. Since the length of the word represented by $tx$ is polynomial, we need to calculate only polynomially many $f_i(x#)$, so the entire procedure is polynomial and calculates the word represented by $tx$.

6.2 Size Realizability

We cannot talk about the length of words directly if we wish to use a form of realizability, since realizability talks about $\lambda$-terms. Instead, we use the number of applications in a term as a measure of size.

**Definition 6.12** (Applicative size). For a term $t$, the **applicative size** of $t$, denoted $\#t$, is the number of applications in $t$. Formally

$$
\#x = 0, \#(tu) = \#t + \#u + 1, \#(\lambda x.t) = \#t
$$

The first observation we make is that the length of a word is equal to the number of applications in the Scott representation.

**Proposition 6.13.** Let $w$ be a word of length $n$. Then $\#_S w = n$.

**Proof.** By induction on the length of $w$, denoted by $n$.

If $w = \epsilon$, then $w_S = \lambda f_0 f_1 x.x$, which has applicative size 0.

If $w = 0w'$, then $w_S = \lambda f_0 f_1 x.f_0(w'_S)$. By induction hypothesis $\#_{\beta} w'_S = n - 1$.

Using the definition of applicative size, it follows that $\#_{\beta} w_S = n$.

The case $w = 1w'$ is symmetric. \[\square\]

For arbitrary terms of type $W_S$, we need the number of applications in the $\beta$-normal form (which gives the representation as in the definition), rather than the number of applications in the current term. That means the quantity we seek to bound is $\#_{[t]_{\beta}}$. $\beta$-normal form here is found by extending the possible reductions with $\lambda$-compatibility as seen in the preliminaries. This means we keep using a call-by-value strategy.

The definition of size-saturated sets, size-realizability, and proof of the size-adequacy theorem follow a similar pattern to what we’ve already seen before. The main new trick is to introduce an inert object which will help us deal with realizability: it allows us to immobilize applications.

**Definition 6.14.** We write $\boxvoid$ for the **inert variable**. This is a fixed variable that is never assigned a value.

This isn’t so much a definition as it is a promise: we promise to not use the variable $\boxvoid$ such that it can assume a value. In particular, we won’t be using it as the variable in a $\lambda$-abstraction. Using $\boxvoid$, we can define size-saturated sets.
Definition 6.15 (Size-saturated sets). Let $\tau$ be a higher order polynomial type. A set $X \subseteq \Lambda \times \Pi$ is called size-saturated of type $\tau$ if $X \neq \emptyset$, and for all $(t, p) \in X$ we have:

- **Size Bound:** $\#[t]_\beta \leq p$.
- **Monotonicity:** For all $n \in \mathbb{N}$: $(t, p + n) \in X$.
- **Exchanging:** If $t = t_0(\theta, v_1, v_2, \kappa)\bar{u}$, then $(t_0(\theta, v_2, v_1, \kappa)\bar{u}, p) \in X$.
- **Size Weakening:** If $t = t_0\bar{u}$, $w$ is a term that normalizes, and $z \notin \text{FV}(t_0)$, then $(t_0(w/z)\bar{u}, p) \in X$.
- **Contraction:** If $t = t_0\bar{u}$, then $(x(t_0/x)\bar{u}, p) \in X$.
- **Concatenation:** if $t = (t_0)\bar{u}$, then $(t_0(t_1)\bar{u}, p) \in X$.
- **Size Identity:** If $t = t_0\bar{u}$, then $(t_0\bar{u}, p) \in X$.
- **Size Variable:** $(\Box, 0, \tau) \in X$.

Note that the size variable condition contradicts condition bound in our original realizability relation. This is the reason we don’t define size-realizability and regular realizability together in one go. It’s possible to work around this, but it’d make our definitions and proofs messier. We will also see some variations on realizability later on, so it’s good to have multiple examples to build some intuition.

As with saturated sets, there exists a canonical example.

Proposition 6.16. There exists a size-saturated set $X^*_0$ of type $\alpha$, given by

$$X^*_0 = \{(t, n) : t \downarrow \text{ and } \#[t]_\beta \leq n\}$$

Furthermore, $X^*_0$ is the greatest saturated set of type $\alpha$.

Proof. $(\lambda x. x, 0) \in X^*_0$, so $X^*_0 \neq \emptyset$. Suppose $(t, n) \in X^*_0$, then $t \downarrow$ and $\text{cost}(t) \leq n$.

- **Size Bound:** satisfied due to the definition.
- **Monotonicity:** let $m \in \mathbb{N}$. Then $\#t \leq n \leq n + m$, so $(t, n + m) \in X_0$.
- **Exchanging:** since $\lambda$-calculus is confluent, and exchanging the order of two redexes doesn’t matter with a call-by-value strategy, the $\beta$-normal form doesn’t change.
- **Size Weakening:** we add an extra application, but we are interested in the number of applications in the $\beta$-normal form, where $w$ and $z$ will clearly not be present.
• Contraction: the $\beta$-normal form of the term doesn’t change, so the applicative size stays the same.

• Concatenation: by a similar argument as for exchanging, the $\beta$-normal form doesn’t change.

• Identity: the new term is $(\lambda x.x)t_0\bar{u}$, which has the same normal form as $t$.

• Size Variable: $\#[\Box]_\beta = 0$, so $(\Box, 0) \in X^s_\emptyset$.

Hence $X^s_\emptyset$ is a size-saturated set of type $o$. To see that it is the greatest, let $Y$ be a size-saturated set of type $o$, and let $(t, n) \in Y$. By condition size bound, we have $t$ terminating, and $\#[t]_\beta \leq n$, so $(t, n) \in X^s_\emptyset$.

**Definition 6.17.** A size-valuation $\eta$ maps a propositional variable $\alpha$ to a size-saturated set $\eta(\alpha)$ of type $o$.

We write $\eta(\alpha \mapsto X)$ for the valuation which maps $\beta$ to $\eta(\beta)$ for all $\beta \neq \alpha$, and $\alpha$ to $X$.

**Definition 6.18** (Size-realizability). We define the relation $t, p \vdash^s_\eta A$ where $t$ is a term (including the inert variable $\Box$) called the size-realizer of $A$, $p: o(A)$ a closed higher order polynomial called the size-majorizer of $t$, $\eta$ a size-valuation, and $A$ a type.

We write $\bar{A}^s_\eta = \{ (t, p) : t, p \vdash^s_\eta A \}$. $\vdash^s_\eta$ is inductively defined on $A$.

• $t, p \vdash^s_\eta \alpha$ for a propositional variable $\alpha$ if and only if $(t, p) \in \eta(\alpha)$.

• $t, p \vdash^s_\eta L \rightarrow A$ if $t = \Box$, or $u, m \vdash^s_\eta L$ implies $tu, p + m \vdash^s_\eta A$ for all $u, m$.

• $t, p \vdash^s_\eta A \Rightarrow B$ if $t = \Box$, or $u, q \vdash^s_\eta A$ implies $tu, p(q) \vdash^s_\eta B$ for all $u, q$.

• $t, p \vdash^s_\eta \forall \alpha A$ if $t, p \vdash^s_\eta \forall (\alpha \mapsto X)$ $A$ for every size-saturated set $X$ of type $o$.

• $t, p \vdash^s_\eta \mu \alpha L$ if $(t, p) \in X$ for every size-saturated set $X$ of type $o$ satisfying $\bar{L}^s_{\eta(\alpha \mapsto X)} \subseteq X$.

Like before, the following lemma will give some insight in the definition and will be helpful in the proof of the Size Adequacy Theorem.

**Lemma 6.19.** For all formulas $A$, linear formulas $L$, and size-valuations $\eta$:

1. $\bar{A}^s_\eta$ is a size-saturated set of type $o(A)$.

2. $t, p \vdash^s_\eta A[L/\alpha] \iff t, p \vdash^s_{\eta(\alpha \mapsto \bar{L}^s_\eta)} A$.

3. $t, p \vdash^s_\eta \forall \alpha A \iff t, p \vdash^s_\eta A[L/\alpha]$.

4. If $\alpha$ occurs only positively in $L$, $f(X) = \bar{L}^s_{\eta(\alpha \mapsto X)}$ is monotone increasing.

If $\alpha$ occurs only negatively in $L$, $f(X) = \bar{L}^s_{\eta(\alpha \mapsto X)}$ is monotone decreasing.
5. $\mu A L_\eta$ is the least fixpoint of $f(X) = \hat{L}_{\eta(\alpha \rightarrow X)}^\eta$.

6. $t, p \models^*_{\eta} \mu A L \iff t, p \models^*_{\eta} L[\mu A L/\alpha]$.

Proof. 1. By induction on the structure of $A$ in the definition of size realizability.

For $A = \alpha$, we have $t, p \models^*_{\eta} A \iff (t, p) \in \eta(A)$, so $\overline{A}^\eta = \eta(A)$, which is size-saturated.

For $A = L \rightarrow B$, by induction hypothesis $\hat{L}_{\eta}^\eta, \hat{B}_{\eta}^\eta$ are size-saturated.

- Size bound: if $t = \square$, we are done since $\#[\square]_\beta = 0$. If not, note that $(\square, 0) \in \hat{L}_{\eta}^\eta$ by induction hypothesis, so $(\square, p) \in \hat{B}_{\eta}^\eta$. Then $\#[\square]_\beta \leq \downarrow p$ by condition size bound for $\hat{B}_{\eta}^\eta$. Since $\square$ is inert, we have $\#[t]_\beta \leq \#[\square]_\beta$, so condition size bound is satisfied.

- Monotonicity: let $n \in \mathbb{N}$. If $t = \square$, we are done because $p$ can be chosen arbitrarily. If not, let $(u, m) \in \hat{L}_{\eta}^\eta$. By definition of realizability, $(tu, p + m) \in \hat{B}_{\eta}^\eta$, so by condition monotonicity, $(tu, p + m + n) \in \hat{B}_{\eta}^\eta$. This proves $(t, p + n) \in \overline{A}^\eta$.

- Exchanging, size weakening, contraction, concatenation, and size identity: similar to monotonicity, with the note that if $t = \square$, the statements are vacuously true.

- Size variable: satisfied since $\square, p \models^*_{\eta} L \rightarrow B$ with $p$ arbitrary.

For $A = B \Rightarrow C$, by induction hypothesis $\hat{B}_{\eta}^\eta, \hat{C}_{\eta}^\eta$ are size-saturated.

- Size bound: if $t = \square$, we are done since $\#[\square]_\beta = 0$. If not, note that $(\square, 0) \in \hat{L}_{\eta}^\eta$ by induction hypothesis, so $(\square, p(0)) \in \hat{B}_{\eta}^\eta$. Then $\#[\square]_\beta \leq \downarrow p \downarrow p(0)$ by condition size bound for $\hat{B}_{\eta}^\eta$ and definition of $\downarrow$. Since $\square$ is inert, we have $\#[t]_\beta \leq \#[\square]_\beta$, so condition size bound is satisfied.

- Monotonicity: let $n \in \mathbb{N}$. If $t = \square$, we are done because $p$ can be chosen arbitrarily. Suppose then $(u, q) \in \hat{L}_{\eta}^\eta$. By definition of realizability, $(tu, p(q)) \in \hat{B}_{\eta}^\eta$, so by condition monotonicity, $(tu, p(q) + n) \in \hat{B}_{\eta}^\eta$. This proves $(t, p + n) \in \overline{A}^\eta$.

- Exchanging, size weakening, contraction, concatenation, and size identity: similar to monotonicity, with the note that if $t = \square$, the statements are vacuously true.

- Size variable is satisfied since $\square, p \models^*_{\eta} B \Rightarrow C$ with $p$ arbitrary.

For $A = \forall \alpha B$, by induction hypothesis $\hat{B}_{\eta(\alpha \rightarrow X)}^\eta$ is size-saturated for any size-saturated set $X$ of type $\alpha$.

- Size bound: $\#[t]_\beta \leq \downarrow p$ by condition size bound for $\hat{B}_{\eta(\alpha \rightarrow X)}^\eta$.
2. By induction on the structure of $A$ in the definition of size realizability.

We note the statement is trivial if $t = \bot$. We write $\eta' = \eta(\alpha \mapsto \hat{L}_\eta^s)$ for any size valuation $\eta$.

For $A = \alpha$, we have $t, p \vdash^s_\eta A[L/\alpha] \iff t, p \vdash^s_\eta L \iff (t, p) \in \hat{L}_\eta^s \iff (t, p) \in \eta'(\alpha) \iff t, p \vdash^s_{\eta'} \alpha \iff t, p \vdash^s_{\eta'} A$.

For $A = M \rightarrow B$, by induction hypothesis, $t, p \vdash^s_\eta M[L/\alpha] \iff t, p \vdash^s_\eta M$, and $t, p \vdash^s_\eta B[L/\alpha] \iff t, p \vdash^s_\eta B$. Note $A[L/\alpha] = M[L/\alpha] \rightarrow B[L/\alpha]$.

$t, p \vdash^s_\eta A[L/\alpha]$ if and only if $u, m \vdash^s_\eta M[L/\alpha] \iff tu, p + m \vdash^s_\eta B[L/\alpha]$ for all $u, m$. Applying the induction hypothesis, this holds if and only if $u, m \vdash^s_\eta M \iff tu, p + m \vdash^s_{\eta'} B$ for all $u, m$. This proves that $t, p \vdash^s_\eta A[L/\alpha] \iff t, p \vdash^s_{\eta'} A$.

For $A = B \Rightarrow C$, the proof is almost exactly the same as for $M \rightarrow B$, but with $p(q)$ instead of $p + m$.

For $A = \forall \beta B$, by induction hypothesis $t, p \vdash^s_\eta B[L/\alpha] \iff t, p \vdash^s_{\eta'} B$. Note that $A[L/\alpha] = \forall \beta B[L/\alpha]$, and $\eta$ is arbitrary. In particular, we can replace $\eta$ by $\eta(\beta \mapsto X)$ in the induction hypothesis.

$t, p \vdash^s_\eta A[L/\alpha]$ if and only if for every size-saturated set $X$ of type $o$ $t, p \vdash^s_{\eta(\beta \mapsto X)} B[L/\alpha]$, if and only if for every size-saturated set of type $X$ of type $o$, $t, p \vdash^s_{\eta(X \rightarrow \beta)} B$, if and only if $t, p \vdash^s_{\eta'} A$.

For $A = \mu \alpha L$, by induction hypothesis $\hat{L}_\eta^s \subseteq X^n \subseteq X^n$. This proves $(t, p + n) \in \hat{A}_\eta^n$.

- Monotonicity: let $n \in \mathbb{N}$. For any size-saturated set $X$ of type $o$, by condition monotonicity for $\hat{L}_\eta^s$, $(t, p + n) \in \hat{L}_\eta^s$. This proves $(t, p + n) \in \hat{A}_\eta^n$.

- Exchanging, size weakening, contraction, concatenation, size identity, and size variable: similar to monotonicity.

For $A = \mu \alpha L$, by induction hypothesis $\hat{L}_\eta^s$ is size-saturated for any size-saturated set $X$ of type $o$.

- Size bound: $X^n$ is the largest size-saturated of type $o$, so we have $\hat{L}_\eta^s \subseteq X^n$. This proves $(t, p) \in X^n$, which satisfies condition size bound.

- Monotonicity: let $n \in \mathbb{N}$. For every size saturated set $X$ of type $o$ such that $\hat{L}_\eta^s \subseteq X$ we have $(t, p) \in X$. By condition monotonicity for $X$ then $(t, p + n) \in X$, which proves $(t, p + n) \in \hat{A}_\eta^n$.

- Exchanging, size weakening, contraction, concatenation, size identity, and size variable: similar to monotonicity.

2. By induction on the structure of $A$ in the definition of size realizability.

We write $A[L/\alpha] = \hat{L}_\eta^s$ for any size valuation $\eta$.

For $A = \alpha$, we have $t, p \vdash^s_\eta A[L/\alpha] \iff t, p \vdash^s_\eta L \iff (t, p) \in \hat{L}_\eta^s \iff (t, p) \in \eta'(\alpha) \iff t, p \vdash^s_{\eta'} \alpha \iff t, p \vdash^s_{\eta'} A$.

For $A = M \rightarrow B$, by induction hypothesis, $t, p \vdash^s_\eta M[L/\alpha] \iff t, p \vdash^s_\eta M$, and $t, p \vdash^s_\eta B[L/\alpha] \iff t, p \vdash^s_\eta B$. Note $A[L/\alpha] = M[L/\alpha] \rightarrow B[L/\alpha]$.

$t, p \vdash^s_\eta M[L/\alpha]$ if and only if $u, m \vdash^s_\eta M[L/\alpha] \iff tu, p + m \vdash^s_\eta B[L/\alpha]$ for all $u, m$. Applying the induction hypothesis, this holds if and only if $u, m \vdash^s_\eta M \iff tu, p + m \vdash^s_{\eta'} B$ for all $u, m$. This proves that $t, p \vdash^s_\eta A[L/\alpha] \iff t, p \vdash^s_{\eta'} A$.

For $A = B \Rightarrow C$, the proof is almost exactly the same as for $M \rightarrow B$, but with $p(q)$ instead of $p + m$.

For $A = \forall \beta B$, by induction hypothesis $t, p \vdash^s_\eta B[L/\alpha] \iff t, p \vdash^s_{\eta'} B$. Note that $A[L/\alpha] = \forall \beta B[L/\alpha]$, and $\eta$ is arbitrary. In particular, we can replace $\eta$ by $\eta(\beta \mapsto X)$ in the induction hypothesis.

$t, p \vdash^s_\eta B[L/\alpha]$ if and only if for every size-saturated set $X$ of type $o$ $t, p \vdash^s_{\eta(\beta \mapsto X)} B[L/\alpha]$, if and only if for every size-saturated set of type $X$ of type $o$, $t, p \vdash^s_{\eta(X \rightarrow \beta)} B$, if and only if $t, p \vdash^s_{\eta'} A$.

For $A = \mu \alpha L$, by induction hypothesis $t, p \vdash^s_\eta M[L/\alpha] \iff t, p \vdash^s_{\eta'} M$. Again, note that $A[L/\alpha] = \mu \alpha M[L/\alpha]$, and $\eta$ can be replaced by $\eta(\beta \mapsto X)$ in the induction hypothesis.

48
We also need $M[L/\alpha]^s_\eta \subseteq X \iff \overline{M} \subseteq X$ for all size-saturated sets $X$ of type $o$. This follows using item 1 and the induction hypothesis.

$t, p \vdash^s A[L/\alpha] \iff t, p \vdash^s_{\eta[\beta \to X]} M[L/\alpha]$ for every size-saturated set $X$ of type $o$ satisfying $\overline{M[L/\alpha]^s_\eta} \subseteq X$, if and only if $t, p \vdash^s_{\eta[\beta \to X]} M$ for every size-saturated set $X$ of type $o$ satisfying $\overline{M} \subseteq X$, if and only if $t, p \vdash^s A$.

3. $t, p \vdash^s \forall \alpha A$ if and only if $t, p \vdash^s_{\eta[\alpha \to X]} A$ for all size-saturated sets $X$ of type $o$. By item 1, $\overline{L}_n^s$ is such a set, so $t, p \vdash^s_{\eta[\alpha \to \overline{L}_n^s]} A$, which by item 2 is equivalent to $t, p \vdash^s A[L/\alpha]$.

4. Similar to the Realizability Lemma.

5. $\alpha$ occurs only positively in $L$, so $f$ is monotone. Call a size-saturated set $X$ a prefixpoint of $f$ if $f(X) \subseteq X$. Then $\mu \alpha L^s$ is the infimum of all prefixpoints of $f$, since $(t, p) \in \mu \alpha L^s$ if and only if $(t, p) \in X$ for all size-saturated sets $X$ such that $f(X) \subseteq X$.

Since $f$ is monotone, this proves $f(\mu \alpha L^s) \subseteq f(X) \subseteq X$ for all saturated sets $X$, but $\mu \alpha L^s$ in particular, giving $f(\mu \alpha L^s) \subseteq \mu \alpha L^s$.

Again since $f$ is monotone, $f(f(\mu \alpha L^s)) \subseteq f(\mu \alpha L^s)$. So $f(\mu \alpha L^s)$ is a prefixpoint of $f$, yielding $\mu \alpha L^s \subseteq f(\mu \alpha L^s)$.

So $f(\mu \alpha L^s) = \mu \alpha L^s$. It is also the least fixpoint, since any fixpoint must be a prefixpoint as well.

6. $t, p \vdash^s L[\mu \alpha L/\alpha] \iff t, p \vdash^s_{\eta[\alpha \to \mu \alpha L]} L$ by item 2, which is equivalent to $(t, p) \in \overline{L}^s_\eta[\alpha \to \mu \alpha L] = f(\mu \alpha L) = \mu \alpha L$, which is equivalent to $t, p \vdash^s \mu \alpha L$.

We can now state and prove the Size Adequacy Theorem.

**Theorem 6.20 (Size-Adequacy Theorem).** Suppose $\bar{x} : \bar{C}, \bar{y} : \bar{M} \vdash t : A$ is derivable. Let $\eta$ be any size-valuation. Then there exists a higher order polynomial $p(\bar{x}) : o(A)$ where the variables $\bar{x}$ are of types $o(\bar{C})$ satisfying the following.

Suppose there exist size-realizers $\bar{u}, \bar{s}$ for $\bar{C}$ and $\bar{M}$ with size-majorizers $\bar{q}, \bar{m}$, respectively. Formally

$$\bar{u}, \bar{q} \vdash^s \bar{C}, \bar{s}, \bar{m} \vdash^s \bar{M}$$

Then $t(\bar{u}/\bar{x}, \bar{s}/\bar{y})$ size-realizes $A$ with size-majorizer $p(\bar{q}) + \bar{m}$. Formally

$$t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash^s A$$

**Proof.** We induct on the derivation of $\bar{x} : \bar{C}, \bar{y} : \bar{M} \vdash t : A$.

Like in the proof of the Adequacy Theorem (Theorem 6.9), we will drop the valuation subscript if it doesn’t matter for the proof. Additionally, we will drop the $s$ superscript from the size realizability relation.
We write $\Gamma = \bar{x} : \bar{C}, \Delta = \bar{y} : \bar{M}$. Terms in $\bar{u}$ will be substituted for those in $\bar{x}$, and likewise for those $\bar{s}$ and $\bar{y}$. $p,q$ will always denote higher order polynomials, while $m$ will always denote a natural number. The lengths of $\bar{x}$ and $\bar{y}$ are denoted by $l_x$ and $l_y$, respectively.

These conventions extend to subscripts and accents.

Case (ax1).

$\Gamma; \Delta \vdash x : A$

We claim $p(x) = x$ suffices. Suppose $u,q \vdash A$, then condition size identity guarantees $x(u/x), q \vdash A$, and $p(q) = q$.

Case (ax2).

$\vdash y : L \vdash y : L$

We claim $p = 0$ suffices. Suppose $s,m \vdash L$, then condition size identity guarantees $x(s/y), m \vdash A$, and $p + m = m$.

Case ($\mu_i$).

$\frac{\Gamma; \Delta \vdash t : L[\mu\alpha L/\alpha]}{\Gamma; \Delta \vdash t : \mu\alpha L}$

By induction hypothesis, we have a size-majorizer $p(\bar{x})$ such that for $\bar{u}, \bar{q} \vdash \bar{C}$ and $\bar{s}, \bar{m} \vdash \bar{M}$ we have $t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash L[\mu\alpha L/\alpha]$. Using item 5 of Lemma 6.19 this gives $t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash \mu\alpha L$, so $p(\bar{x})$ suffices as a size-majorizer.

Case ($\mu_e$).

$\frac{\Gamma; \Delta \vdash t : \mu\alpha L}{\Gamma; \Delta \vdash t : L[\mu\alpha L/L]}$

Similar to the above, again using item 5 of Lemma 6.19

Case ($\forall_e$).

$\frac{\Gamma; \Delta \vdash t : \forall\alpha A}{\Gamma; \Delta \vdash t : A[L/\alpha]}$

By induction hypothesis, we have a size-majorizer $p(\bar{x})$ such that for $\bar{u}, \bar{q} \vdash \bar{C}$ and $\bar{s}, \bar{m} \vdash \bar{M}$ we have

$t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash \forall\alpha A$

Using item 3 of Lemma 6.19 this gives

$t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash A[L/\alpha]$

so $p(\bar{x})$ suffices as a size-majorizer.
Case $(\forall_i)$.  
\[ \Gamma; \Delta \vdash t : A \quad \alpha \notin \text{FV}(\Gamma; \Delta) \]
\[ \Gamma; \Delta \vdash \forall \alpha A \]

By induction hypothesis, we have a size-majorizer $p(\bar{x})$ such that for $\bar{u}, \bar{q} \vdash \bar{C}$ and $\bar{s}, \bar{m} \vdash \bar{M}$ we have $t(\bar{u}/\bar{x}, \bar{s}/\bar{y})p(\bar{q}) + \bar{m} \vdash A$. We show $p(\bar{x})$ is a suitable size-majorizer.

Unfolding definition of size-realizability for $\forall \alpha A$, we have to show that for every saturated set $X$ of type $\alpha$ we have $t(\bar{u}/\bar{x}, \bar{s}/\bar{y})p(\bar{q}) + \bar{m} \vdash A(\alpha \rightarrow X)$. Now $\eta$ in the induction hypothesis is an arbitrary valuation, so we can replace it by $\eta(\alpha \rightarrow X)$ and get the required result.

Case $(\Rightarrow_e)$.  
\[ \Gamma_1; \Delta \vdash t_1 : A \Rightarrow B \quad \Gamma_2; \vdash t_2 : A \]
\[ \Gamma_1, \Gamma_2; \Delta \vdash t_1 t_2 : B \]

By induction hypothesis, we have size-majorizers $p_1(\bar{x}_1, z), p_2(\bar{x}_2)$ such that for $\bar{u}_1, \bar{q}_1 \vdash \bar{C}_1$ and $\bar{s}_1, \bar{m}_1 \vdash \bar{M}_1$ we have $t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y})p_1(\bar{q}_1, z) + \bar{m}_1 \vdash A \Rightarrow B$, and for $\bar{u}_2, \bar{q}_2 \vdash \bar{C}_2$ we have $t_2(\bar{u}_2/\bar{x}_2)p_2(\bar{q}_2) + \bar{m} \vdash A$.

Unfolding the definition of size-realizability for $A \Rightarrow B$, we obtain either $t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y}) = \Box$, or $t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y})t_2(\bar{u}_2/\bar{x}_2)p_1(\bar{q}_1, z) + \bar{m}_1 \vdash B$.

In the first case, this means we must have $\bar{x}_1, \bar{y}$ empty. The corresponding premise is $\vdash \Box : A \Rightarrow B$. However, this is not derivable in DIAL in which $\Box$ is a variable (this is easily shown using induction), so this case simply does not occur.

In the second case, using conditions concatenation and exchange yields $p(\bar{x}_1, \bar{x}_2) = p_1(\bar{x}_1, p_2(\bar{x}_2))$ is a suitable size-majorizer.

Case $(\Rightarrow_i)$.  
\[ \Gamma_1, z : A; \Delta \vdash t : B \]
\[ \Gamma; \Delta \vdash \lambda z.t : A \Rightarrow B \]

By induction hypothesis, we have a size-majorizer $p(\bar{x}, z)$ such that for $\bar{u}, \bar{q} \vdash \bar{C}, w, r \vdash A$, and $\bar{s}, \bar{m} \vdash \bar{M}$ we have $t(\bar{u}/\bar{x}, w/z, \bar{s}/\bar{y})p(\bar{q}, r) + \bar{m} \vdash B$. We show $p$ is a suitable size-majorizer.

Unfolding the definition of size-realizability for $A \Rightarrow B$, it suffices to prove that $(\lambda z.t)(\bar{u}/\bar{x}, \bar{s}/\bar{y})w, p(\bar{q}, r) + \bar{m} \vdash B$. Noting $(\lambda z.t)(\bar{u}/\bar{x}, \bar{s}/\bar{y})w = t(\bar{u}/\bar{x}, \bar{s}/\bar{y}, w/z)$ and using condition exchange yields the required result.

Case $(\neg_o)$.
\[ \Gamma_1; \Delta \vdash t_1 : L \rightarrow B \quad \Gamma_2; \Delta_2 \vdash t_2 : L \]
\[ \Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t_1 t_2 : B \]

By induction hypothesis, we have size-majorizers $p_1(\bar{x}_1), p_2(\bar{x}_2)$ such that for $\bar{u}_1, \bar{q}_1 \vdash \bar{C}_1$ and $\bar{s}_1, \bar{m}_1 \vdash \bar{M}_1$ we have $t_1(\bar{u}_1, \bar{x}_1, \bar{s}_1/\bar{y}_1)p_1(\bar{q}_1) + \bar{m}_1 \vdash A \rightarrow B$, and for $\bar{u}_2, \bar{q}_2 \vdash \bar{C}_2$ and $\bar{s}_2, \bar{m}_2 \vdash \bar{M}_2$ we have $t_2(\bar{u}_2, \bar{x}_2, \bar{s}_2/\bar{y}_2)p_2(\bar{q}_2) + \bar{m}_2 \vdash L$.

Unfolding the definition of size-realizability for $A \rightarrow B$, we obtain either $t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y}_1) = \Box$, or $t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y}_1)t_2(\bar{u}_2, \bar{x}_2, \bar{s}_2/\bar{y}_2)p_1(\bar{q}_1) + \bar{m}_1 + p_2(\bar{q}_2) + \bar{m}_2 \vdash B$.

51
The first case doesn’t occur, similar to the proof for $(\Rightarrow)$. In the second case, using conditions concatenation and exchange yields that $p(x_1, x_2) = p(x_1) + p(x_2)$ is a suitable size-majorizer. Note that this is well-defined because $p(x_2)$ is of type $o(L) = o$.

Case $(\neg t)$.

\[
\Gamma; \Delta, z : L \vdash t : B \\
\Gamma; \Delta \vdash \lambda z.t : L \leadsto B
\]

By induction hypothesis, we have a size-majorizer $p(x)$ such that for $\bar{u}, \bar{q} \vdash \bar{C}$ and $\bar{s}, \bar{m} \vdash \bar{M}, v, n \vdash L$ we have $t(\bar{u}/x, \bar{s}/\bar{y}, v/\bar{z}), p(\bar{q}) + \bar{m} + n \vdash B$. We show $p$ is a suitable size-majorizer.

Unfolding the definition of realizability for $L \leadsto B$, we prove that

\[
(\lambda z.t)(\bar{u}/x, \bar{s}/\bar{y})v, p(\bar{q}) + \bar{m} + n \vdash B
\]

This certainly holds using the fact that $(\lambda z.t)(\bar{u}/x, \bar{s}/\bar{y})v = t(\bar{u}/x, \bar{s}/\bar{y}, v/\bar{z})$ and using condition exchange.

Case (Contr).

\[
\Gamma, z_0 : A, z_1 : A; \Delta \vdash t : B \\
\Gamma, z : A; \Delta \vdash t[z/z_0, z/z_1] : B
\]

By induction hypothesis, we have a size-majorizer $p(x, y)$ such that for $\bar{u}, \bar{q} \vdash \bar{C}$ and $\bar{s}, \bar{m} \vdash \bar{M}, w, r \vdash A$ we have

\[
t(\bar{u}/x, w/z_0, w/z_1, \bar{s}/\bar{y}), p(\bar{q}, r, r) + \bar{m} \vdash B
\]

Using condition contraction, we derive

\[
t[z/z_0, z/z_1](\bar{u}/x, w/z, \bar{s}/\bar{y}), p(\bar{q}, r, r) + \bar{m} \vdash B
\]

which shows that $p'(x, z) = p(x, z, z)$ is a suitable size-majorizer.

Case (Derel).

\[
\Gamma; \Delta, x : L \vdash t : B \\
\Gamma, z : L; \Delta \vdash t[z/x] : B
\]

By induction hypothesis we have a size-majorizer $p(x)$ such that for $\bar{u}, \bar{q} \vdash \bar{C}$ and $\bar{s}, \bar{m} \vdash \bar{M}$, and $w, n \vdash L$ we have $t(\bar{u}/x, \bar{s}/\bar{y}, w/x), p(\bar{q}) + \bar{m} + n \vdash B$.

Clearly, $p'(x, z) = p(x) + z$ is a suitable majorizer. This is well defined because $z$ is of type $o(L) = o$.

Case (Weak).

\[
\Gamma; \Delta \vdash t : A \\
\Gamma, \Gamma'; \Delta, \Delta' \vdash t : A
\]

By induction hypothesis we have a size-majorizer $p(x)$ such that for $\bar{u}, \bar{q} \vdash \bar{C}$ and $\bar{s}, \bar{m} \vdash \bar{M}$ we have

\[
t(\bar{u}/x, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash A
\]
Suppose \( \bar{u}', q' \vdash \bar{C}' \) and \( s', \bar{m}' \vdash \bar{M}' \), then by applying condition weakening \( l_x + l_y \) times and condition monotonicity \( l_y \) times, we obtain

\[
t(\bar{u}/\bar{x}, \bar{u}'/\bar{x}', \bar{s}/\bar{y}, \bar{s}'/\bar{y}') \cdot p(q) + \bar{m} + \bar{m}' \vdash A
\]

So \( p' = \lambda \bar{x}'.p(x) \) is a suitable size-majorizer.

\[\square\]

**Corollary 6.21.** Let \( t \) be a term typable in \( \text{DIAL}_{\text{lin}} \). Then there exists a higher order polynomial \( p \) such that \( \#[t] \leq p \).

**Corollary 6.22.** The exponential function, either on Church or on Scott numerals, is not typable in \( \text{DIAL}_{\text{lin}} \).

**Proof.** The applicative size of \( 2^n \) in both Scott and Church representation is \( O(2^n) \).

### 6.3 Putting it all together

We first observe Church representations have (regular) majorizers linear in the length of the word.

**Proposition 6.23.** Let \( w \) be a word of length \( n \). Then for any valuation \( \eta \), \( w_C, q_n \vdash \eta W_C \) where \( q_n(z_0, z_1) : o \to o \to o \) is linear in \( n \).

Note that the multiplication by \( n \) is defined since we consider \( n \) to be fixed.

**Proof.** By induction on \( n \), inspecting the proof of the Adequacy Theorem. In case \( n = 0 \), we have a constant size majorizer. Then every application of \( f_0 \) or \( f_1 \) adds \( z_0 + 3 \) or \( z_1 + 3 \) to the majorizer. Hence the majorizer is linear in \( n \).

The following proposition shows that we can access the \( i \)-th bit of a Scott word with cost linear in \( i \) and the cost of reducing the word.

**Proposition 6.24.** For every \( i \in \mathbb{N} \), there exists a term \( \text{bit}_i \) of type \( W_S \to B_S \) such that for every closed term \( t \in W_S \) representing a Scott word \( w \) of length \( n \), we have \( [\text{bit}_i t] = j_b \) if the \( i \)-th bit of \( w \) is \( j \), and \( [\text{bit}_i t] = \epsilon_b \) if \( i \geq n \).

Furthermore, \( \text{cost}(\text{bit}_i t) \) is linear in \( i \) and \( \text{cost}(t) \).

**Proof.** First note we have a predecessor function, given by

\[
pred \equiv \lambda z. z II \epsilon_S
\]

\( \text{pred} \) is of type \( W_S \to W_S \), and \( [\text{pred}(0w)_S] = w_S \), \( [\text{pred}(1w)_S] = w_S \), and \( [\text{pred}(\epsilon)] = \epsilon \).

We define

\[
q \equiv \lambda x. x(\lambda y.0_b)(\lambda y.1_b)(\epsilon_b)
\]

which is of type \( W_S \to B \). Finally,

\[
\text{bit}_i \equiv \lambda x. q \text{pred}^i(x)
\]

53
To see bit, does what it’s supposed to do, we first need to see that on a term \( \lambda z.t \) representing a word \( w \), \( \text{pred}(\lambda f_0.t) \) reduces to a term representing the predecessor of \( w \).

\( \text{pred} \lambda f_0.t \) reduces in one step to \( (\lambda f_0.t)I \epsilon_S \). \( I \) is a value, so this reduces in one step to \( t[I/f_0]I \epsilon_S \), “opening up” \( t \) for weak \( \beta \)-reduction. In this way, we obtain two more \( \lambda \)-abstractions as the word unfolds. That is, we obtain \( I(t') \) or \( \epsilon_S \) where \( t' \) is the predecessor of the word \( w \). We obtain \( \epsilon_S \) if and only if \( w = \epsilon_S \).

On a closed term \( u \) of type \( W_S \), \( q.u \) will always evaluate to one of \( 0_B, 1_B, \epsilon_B \). To see this, let \( \gamma \) be a fresh propositional variable and \( z_0, z_1, z_2 \) be variables of type \( \gamma \). Define

\[
q_\gamma = \lambda x. x(\lambda y. z_0)(\lambda y. z_1) z_2
\]

This term has type \( W_S \to \gamma \). For any closed term \( u : W_S \), we have \( q_\gamma u : \gamma \), so by the subject reduction property (Theorem 4.3), \( q(\gamma, u) : \gamma \).

\( \lbrack q, u \rbrack \) cannot be an abstraction, since \( \gamma \) is a propositional variable and need not be of the form \( L \to B \) or \( A \Rightarrow B \).

\( \lbrack q, u \rbrack \) cannot be an application, since the only possible head variables are \( z_0, z_1, z_2 \), which already have type \( \gamma \).

Hence \( \lbrack q, u \rbrack \) is equal to one of \( z_0, z_1, z_2 \). Substituting \( \gamma = B_S \) and \( z_0 = 0_B \), \( z_1 = 1_B \), \( z_2 = \epsilon_B \) yields the required result.

We can show \( \text{bit}_i t \) yields the \( i \)-th bit of \( w \) by induction on \( i \). The above shows that the reduction does not get stuck, even with weak call-by-value reduction.

Since \( \text{bit}_i \) is typable in \( \text{DIAL}_{\text{lin}} \), using the Adequacy Theorem we obtain a polynomial \( p \) such that for any valuation \( \eta \), \( \text{bit}_i, p \vdash_\eta W_S \Rightarrow B_S \).

We can be a bit more precise with this: note \( o(W_S \Rightarrow W_S) = o \). By the Adequacy Theorem, there exists a natural number \( m \) such that \( \text{pred}, m \vdash_\eta W_S \Rightarrow W_S \). This implies that for any \( w, n \vdash_\eta W_S \), \( \text{pred} w, m + n \vdash_\eta W_S \). Induction over \( i \) yields \( \text{pred}^i w, im + n \vdash_\eta W_S \).

Note \( o(W_S \Rightarrow B) = o \). By the Adequacy Theorem, there exists a natural number \( m' \) such that \( q, m' \vdash_\eta W_S \Rightarrow B_S \). Combining this with the previous result, we obtain \( q \text{pred}^i w, im + m' + n \vdash_\eta B_S \).

Inspecting the proof of the Adequacy Theorem, we find a natural number \( m'' \) such that \( \text{bit}_i, im + m'' \vdash_\eta W_S \Rightarrow B_S \). This proves \( \text{cost}(\text{bit}_i t) \) is linear in \( i \) and \( \text{cost}(t) \).

Finally we are ready to prove soundness.

**Proof of Soundness (Theorem 6.1).** Let \( t \) be a term of type \( W_C \Rightarrow W_S \) and \( \eta \) any valuation.

By the Adequacy Theorem, there exists a higher order polynomial \( p \) of type \( o(W_C) \to o \) such that \( t, p \vdash_\eta W_C \Rightarrow W_S \).

Using Proposition 6.24, we obtain \( m, m' \) natural numbers such that

\[
\lambda w. \text{bit}_i(tw), im + m' + p(w) \vdash_\eta W_C \Rightarrow B_S
\]
By Proposition 6.23, using the definition of realizability for $W_C \Rightarrow B_S$, and applying condition bound for saturated sets, there exists a polynomial $q(n)$ such that for any word $w \in \{0,1\}^*$ of length $n$, and any $i \in \mathbb{N}$ we have

$$\text{cost(bit}_i(tw_C)) \leq im + m' + p(q(n))$$

Applying Theorem 3.15 (simulation of $\lambda$-reduction on a Turing machine), this proves the existence of Turing machines $M_i$ such that on input $w$, $M_i$ calculates the $i$-th bit of $[tw_C]_\beta$, and does so in time bounded by some polynomial $P(i,n)$, which is monotone in $i$ (since the cost for reducing bit$_i(tw_C)$ is). Note that we can generalize this to a Turing machine $M$ which takes $i$ as first part of the input and then calculates $M_i$ on the rest of the input: the only difference is adding or removing iterations of pred.

By the Size Adequacy Theorem, we obtain a polynomial $r(x)$ such that for any word $w \in \{0,1\}^*$ of length $n$, $\#[tw_C]_\beta \leq r(n)$. Using Proposition 6.13 we conclude that the word $f(w)$ represented by $[tw_C]$ has length at most $r(n)$.

In order to obtain $f(w)$, we can now calculate bit$_0(tw_C)$, bit$_1(tw_C)$, ..., until some $m$ where bit$_m(tw_C) = \epsilon_B$, marking the end of $f(w)$. This requires us to use $M$ at most $r(n)$ times, never with an input $i$ exceeding $r(n)$. The total cost on input $w \in \{0,1\}^*$ of length $n$, then, is bounded by $r(n)P(r(n),n)$. Clearly, this is polynomial, so $f$ is polynomial time computable.

Taking all of these results together proves our type-theoretic characterization of polynomial time functions to be correct: a function is polynomial time computable if and only if it can be typed in $\text{DIAL}_{lin}$ as $W_C \Rightarrow W_S$.

7 Modifications and Extensions

The proof of our main result leaves something to be desired. There are two main issues: the first is that the linear premise of a linear implication must be linear, which is unsatisfactory because the linear implication says something about the use of a variable, rather than the variable itself. The second is that we have to prove soundness in two large steps: regular realizability and size realizability, which is unsatisfactory because it’s not elegant.

In this section, we will explore ways to deal with these problems.

7.1 The linear premise for $\rightarrow$

Brunel and Terui mention this as an unpleasant restriction, and remark that the realizability argument forces it. In particular, it’s related to the addition of two higher order polynomials not being defined. In order to deal with this, we need to change the typing rules for $\text{DIAL}_{lin}$ and find some way to define addition of higher order polynomials. Then we can revisit the Realizability Lemma and the Adequacy Theorem to see if our changes hold up. We will go over these changes relatively quickly.

The current rules for $\rightarrow$ are the following.
It seems easy enough to adapt the \((\Rightarrow i)\)-rule: simply replace the typing \(z : L\) by \(z : A\) and we’re done. The problem is that the typing \(z : L\) takes place in the linear context, which does not admit non-linear variables. One possible solution is to place the new typing \(z : A\) in the non-linear context, obtaining the following rule.

\[
\frac{\Gamma, z : A; \Delta \vdash t : B}{\Gamma; \Delta \vdash \lambda z. t : A \rightarrow B}
\]

However, this is exactly the \((\Rightarrow)\)-rule with \(\Rightarrow\) replaced by \(\rightarrow\)! The idea behind the distinction between \(\Rightarrow\) and \(\rightarrow\) is to separate linear and non-linear use of a function variable, and this rule does not reflect that distinction.

The goal then must be to introduce non-linear variables in the linear context. The use of linear variables is forced by the \((ax2)\)-rule, so we have to change this, and then adapt the \((\Rightarrow e)\) - and \((\Rightarrow i)\)-rules accordingly. We do not need to place additional restrictions on the adjusted \((\Rightarrow e)\)-rule like for \((\Rightarrow i)\), since the argument is only used linearly.

\[
\frac{\vdash y : A \rightarrow y : A}{(ax2)'}
\]

\[
\frac{\Gamma_1; \Delta_1 \vdash t_1 : A \rightarrow B \quad \Gamma_2; \Delta_2 \vdash t_2 : A}{\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t_1 t_2 : B} \quad (\Rightarrow e)'
\]

\[
\frac{\Gamma; \Delta, z : A \vdash t : B}{\Gamma; \Delta \vdash \lambda z. t : A \rightarrow B} \quad (\Rightarrow i)'
\]

Having set up this new type system, which we will call DIAL\(_{lin}'\), we turn can turn towards realizability. The first change we need is immediately clear: the higher order polynomials which majorize terms have variables corresponding to variables in the non-linear context, and not to those in the linear context. Now we need to have variables for terms in the linear context as well, else we cannot prove Adequacy for \((ax2)\)'.

The second change lies in the majorizer for \(A \rightarrow B\). In the original realizability semantics, the majorizer for a term of type \(L \rightarrow B\) had the same type as a term majorizing \(B\). However, in \(A \rightarrow B\) the majorizer will also depend on \(A\). In particular, since the a term \(\lambda z. t : A \rightarrow B\) has to use the variable \(z\) of type \(A\) linearly, this cost needs to be additive. In order to prove Adequacy for \((\Rightarrow e)\)' we would like to use majorizer \(p_1 + p_2\) where \(p_1\) is a majorizer for the function and \(p_2\) a majorizer for the argument. Since the argument is non-linear, this means we need to add two higher order polynomials of non-base type, which hasn’t so far been defined.

**Definition 7.1** (Addition of higher order polynomials). Let \(p_1 = \lambda \bar{x}_1.p_1(\bar{x}_1), p_2 = \lambda \bar{x}_2.p_2(\bar{x}_2)\) be higher order polynomials of type \(r_1, r_2\), respectively. We define
the sum of \( p_1 \) and \( p_2 \) by

\[
p_1 + p_2 = \lambda \bar{x}_1 \bar{x}_2. p_1(\bar{x}_1) + p_2(\bar{x}_2)
\]

which is of type \( \tau_1 \rightarrow \tau_2 \).

This definition is motivated by the following observation: suppose we want to add polynomials \( 3x + y \) and \( 4x \). The knee-jerk response is to say this sum is \( 7x + y \). However, since we identified higher order polynomials up to \( \alpha \)-conversion, we can also say the sum should be \( 3x + 5y \). Merging arguments thus is problematic, and picking a way to do so would be arbitrary. Furthermore, it does not reflect the way we wish to use addition. In proving Adequacy of \( (\neg\epsilon)' \) we wish for the variables in the addition to remain separate.

Addition is not commutative, because higher order polynomials are \( \lambda \)-terms, and \( \lambda xy.x \) is a different term than \( \lambda yx.x \). However, this has no serious implications; we only need to be a bit careful with the order at times.

Clearly, this implies that \( o(A \rightarrow B) = o(A) \rightarrow o(B) \). One remark here is that the correspondence between the non-linear structure of a term and it’s polynomial majorizer breaks down: linear variables are going to appear in the polynomial majorizer while not contributing to the non-linear structure. A way to solve this would be to introduce an additive type, but this method works fine.

Next we adjust the definition of realizability for \( A \rightarrow B \):

\[
\bullet \ t, p \vdash_{\eta} A \rightarrow B \text{ if and only if } \text{cost}(t) \leq p \text{ and for any } u, q \vdash_{\eta} A \text{ we have } tu, p + q \vdash_{\eta} B.
\]

That covers the definitions. The proof of Lemma 6.10 remains the same, replacing the \( m \) by \( q \) in the cases corresponding to \( A \rightarrow B \). We will need to slightly adjust the statement and proof of the Adequacy Theorem.

We will keep the distinction between linear and non-linear arguments of polynomial majorizers clear by separating them by a semicolon.

**Theorem 7.2 (Adequacy Theorem Revisited).** Suppose \( \bar{x} : \bar{C}; \bar{y} : \bar{D} \vdash t : A \) is derivable in \( \text{DIAL}_{lin}' \).

Let \( \eta \) be any valuation. Then there exists a higher order polynomial \( p(\bar{x}; \bar{y}) \) of type \( o(A) \) where there variables \( \bar{x}, \bar{y} \) are of types \( \bar{C}, \bar{D} \) satisfying the following statement: If \( \bar{u}, \bar{q} \vdash_{\eta} \bar{C} \) and \( \bar{s}, \bar{r} \vdash_{\eta} \bar{D} \), then

\[
t(\bar{u}/\bar{x}, \bar{s}/\bar{y}); p(\bar{q}; \bar{r}) \vdash_{\eta} A,
\]

**Proof.** We adopt the same notation as in the proof of the Adequacy Theorem. The proof is the same for all cases using the new notation except \( (ax2)' \), \( (\neg\epsilon)' \), and \( (\neg\epsilon)' \).

Case \( (ax2)' \).

\[
; y : A \vdash y : A
\]

We take as majorizer \( p(\; y) = y + 3 \). The proof is the same as for \( (ax1) \).
By induction hypothesis, we have majorizers \( p_1, p_2 \) of \( t_1 \) and \( t_2 \). We take \( p = p_1 + p_2 \) as majorizer.

Suppose \( \bar{u}_i, \bar{q}_i \vdash \bar{C}_i \) and \( \bar{s}_i, \bar{r}_i \vdash \bar{D}_i \) for \( i = 1, 2 \). By induction hypothesis \( t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y}_1), p_1(\bar{q}_1; \bar{r}_1) \vdash A \to B \). Unfolding the definition of realizability for \( A \to B \) gives

\[
t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y}_1)t_2(\bar{u}_2/\bar{x}_2, \bar{s}_2/\bar{y}_2), p_1(\bar{q}_1; \bar{r}_1) + p_2(\bar{q}_2; \bar{r}_2) \vdash B
\]

Applying conditions concatenation and exchange yields the required result.

Similarly, we can prove the Size Adequacy Theorem revisited, in order to arrive at a revisited type-theoretic characterization of polytime functions.

**Theorem 7.3.** A function \( f : \Sigma \to \Sigma \) is polynomial time computable if and only if there exists a term \( t \) of type \( WC \Rightarrow WS \) in \( DIAL'_{lin} \) such that for all words \( w \)

\[
tw_C =_\beta (f(w))_S
\]

### 7.2 Combining regular and size realizability

The following result is due to Accattoli and Dal Lago [1].

**Definition 7.4** (Cost of strong reduction). Let \( t \) be a \( \lambda \)-term. If \( t \) has a normal form, we denote by \( \text{cost}_\beta(t) \) the number of \( \beta \)-steps it takes to reduce \( t \) to normal form using a leftmost-outermost reduction strategy. If \( t \) does not have a normal form, we define \( \text{cost}_\beta(t) = \infty \).

**Theorem 7.5.** \( \text{cost}_\beta \) satisfies the Polynomial Invariance Thesis.

This result is interesting for our purposes because leftmost-outermost reduction finds a \( \beta \)-normal form, if one exists. This variation on reduction is strong enough to detect the length of the output word, and, in fact, to calculate it in one go! We will use the now familiar realizability construction to see if this works. In proving \( \beta \)-Adequacy, we will see the inert variable trick again: this time there is no conflict in condition variable and condition bound since \( \text{cost}_\beta(\mathbf{0}) = 0 \).

We will introduce \( \beta \)-saturated sets and \( \beta \)-realizability, prove \( \beta \)-Adequacy, and finally give an alternative proof of soundness.
Definition 7.6 (β-saturated sets). A β-saturated set of simple type τ is a nonempty set of pairs (t, p) with t a λ-term and p: τ a higher order polynomial, such that if (t, p) ∈ X, the following conditions hold:

- β-bound: cost_β(t) ≤ p.
- Monotonicity: For any n ∈ ℕ, (t, p + n) ∈ X.
- Exchanging: If t = t₀(θ, v₁/y₁, v₂/y₂, κ)ū, then (t₀(θ, v₂/y₂, v₁/y₁, κ)ū, p) ∈ X.
- β-weakening: If t = t₀(θ)ū and w is a term that normalizes, then (t₀[θ, w/z]ū, p + cost_β(w) + 1) ∈ X.
- Contraction: If t = t₀(θ, v/y₁, v/y₂, κ)ū, then (t₀[z/y₁, z/y₂](θ, v/z, κ)ū, p) ∈ X.
- Concatenation: If t = (t₀(θ))(t₁(κ))ū, then (t₀t₁(θ, κ)ū, p) ∈ X.
- β-identity: If t = t₀ū, then (x(t₀/x)ū, p + 1) ∈ X.
- β-variable: (ι, 0) ∈ X.

Note these conditions overlap with regular realizability, except for those cases where the rule is prefixed with a β.

Again, there exists a greatest β-saturated set of base type.

Proposition 7.7. Let X₀ = \{(t, p) : cost_β(t) ≤ p\}. X₀ is the greatest saturated set of type o.

Proof. Straightforward using the properties of leftmost-outermost reduction. □

Definition 7.8 (β-valuation). A β-valuation maps a propositional variable α to a β-saturated set of base type o.

Definition 7.9 (β-realizability). We define a relation t, p ⊨_η A where t is a λ-term called the (β-)realizer, p is a closed higher order polynomial of type o(A) called the (β-)majorizer, η is a (β-)valuation, and A is a formula. The relation is defined inductively on A.

We write A_η = \{(t, p) : (t, p) ⊨_η A\}.

- t, p ⊨_η α for a propositional variable α if and only if (t, p) ∈ η(α).
- t, p ⊨_η L → A if t = ι, or cost_β(t) ≤ p and u, m ⊨_η L implies tu, p + m ⊨_η A for all u, m.
- t, p ⊨_η A → B if t = ι, or cost_β(t) ≤ p and u, q ⊨_η A implies tu, p(q) ⊨_η B for all u, q.
- t, p ⊨_η ∀αA if t, p ⊨_η(α→X) A for every saturated set X of type o.
• \( t, p \models_{\beta} \mu \alpha L \) if \((t, p) \in X\) for every saturated set \( X \) of type \( o \) satisfying \( \overline{L}_{\eta(\alpha \mapsto X)} \subseteq X \).

We will drop the \( \beta \)-prefixes and -superscripts if it’s clear what we mean.

**Lemma 7.10 (\( \beta \)-realizability lemma).** For all formulas \( A \), terms \( t \), closed higher order polynomials \( p \), and valuations \( \eta \)

1. \( \overline{A}_\eta \) is a saturated set of type \( A \).
2. \( t, p \models_\eta A[L/\alpha] \iff t, p \models_{\eta(\alpha \mapsto \overline{L}_\eta)} A \).
3. \( t, p \models_\eta \forall \alpha A \iff t, p \models_\eta A[L/\alpha] \).
4. \( \overline{\mu \alpha L}_\eta \) is the least fixpoint of \( f(X) = \overline{L}_{\eta(\alpha \mapsto X)} \).
5. \( t, p \models_\eta \overline{\mu \alpha L} \iff t, p \models_\eta \overline{L}[\mu \alpha L/\alpha] \).

**Proof.**

1. We will only prove the case \( A \models L \rightarrow B \). The rest is similar, as in the proof of the original Realizability Lemma. By induction hypothesis, \( \overline{L}_\eta, \overline{B}_\eta \) are \( \beta \)-saturated sets. Suppose \((t, p) \in \overline{A}_\eta \).

   • Condition \( \beta \)-bound: in case \( t = \top \), \( \text{cost}_\beta(t) = 0 \), so this is satisfied. If not, it is satisfied by the requirement that \( \text{cost}_\beta(t) \leq \downarrow p \) in the definition of \( t, p \models_\eta L \rightarrow B \).

   • Condition monotonicity is satisfied as follows: trivial for \( t = \top \). Suppose \( n \in \mathbb{N} \). Then clearly, \( \text{cost}_\beta(t) \leq \downarrow p + n \). Suppose \( u, m \models_\eta L \). By definition of realizability of \( t, p \models_\eta L \rightarrow B \), \( t, u, p + m \models_\eta B \). By induction hypothesis and condition monotonicity, \( t, u, p + n + m \models_\eta B \). This proves \( t, p + n \models_\eta L \rightarrow B \).

   • Conditions exchange, \( \beta \)-weakening, contraction, concatenation, and \( \beta \)-identity are similar.

   • Condition \( \beta \)-variable is satisfied because we have \( \top, p \models_{\beta} A \) for any \( p \).

2. Again by induction. We will only prove the case \( A \models M \rightarrow B \). Note \( A[L/\alpha] = M[L/\alpha] \rightarrow B[L/\alpha] \). Write \( \eta' = \eta(\alpha \mapsto \overline{L}_\eta) \).

   \( t, p \models_{\eta'} A \) if and only if \( \text{cost}_\beta(t) \leq \downarrow p \) and for every \( u, m \models_{\eta'} M \) we have \( t, u, p + m \models_{\eta'} B \). By induction hypothesis, this is equivalent to \( \text{cost}_\beta(t) \leq \downarrow p \) and for every \( u, m \models_\eta M[L/\alpha] \) we have \( t, u, p + m \models_\eta B[L/\alpha] \). By definition of realizability of \( \rightarrow \), this is equivalent to \( t, p \models_\eta A[L/\alpha] \).

3. Follows easily from 1 and 2.

4. Same as in the Realizability Lemma.

5. Same as in the Realizability Lemma.
Theorem 7.11 (β-Adequacy Theorem). Suppose \( \bar{x} : \bar{C} ; \bar{y} : M \vdash t : A \) in DIAL_{lin}. Then there exists a higher order polynomial \( p \) of type \( o(A) \) such that for any valuation \( \eta \), the following property holds.

If \( \bar{u}, \bar{q} \Vdash^\beta_\eta \bar{C} \), and \( \bar{s}, \bar{m} \Vdash^\beta_\eta \bar{M} \), then

\[
t, p(\bar{q}) + \bar{m} \Vdash^\beta_\eta A
\]

Proof. We write \( \Gamma = \bar{x} : \bar{C}, \Delta = \bar{y} : \bar{M} \). Terms in \( \bar{u} \) will be substituted for those in \( \bar{x} \), and likewise for those \( \bar{s} \) and \( \bar{y} \). \( p, q \) will always denote higher order polynomials, while \( m \) will always denote a natural number. The lengths of \( \bar{x} \) and \( \bar{y} \) are denoted by \( l_x \) and \( l_y \), respectively.

These conventions extend to subscripts and accents.

Case (ax1).

\[
\bar{x} : A; \bar{x} : A
\]

\( p(x) = x + 1 \) is a suitable majorizer. Suppose \( u, q \Vdash A \), then we have to prove that \( x(u/x), p(q) \Vdash A \), which follows easily using condition β-identity.

Case (ax2).

\[
\bar{y} : L; \bar{y} : L
\]

\( p = 1 \) is a suitable majorizer. Suppose \( s, m \Vdash L \), then we have to prove that \( y(s/y), p + m \Vdash L \), which follows easily using condition β-identity.

Case (µe).

\[
\begin{align*}
\Gamma; \Delta \vdash t : \mu \alpha L \\
\Gamma; \Delta \vdash t : L[\mu \alpha L/L]
\end{align*}
\]

By induction hypothesis, we have a majorizer \( p \) of the premise. Using item 5 of the β-Realizability Lemma, it follows that \( p \) is also a majorizer of the conclusion.

Case (µi).

\[
\begin{align*}
\Gamma; \Delta \vdash t : \mu \alpha L \\
\Gamma; \Delta \vdash t : L[\mu \alpha L/\alpha]
\end{align*}
\]

By induction hypothesis, we have a majorizer \( p \) of the premise. Using item 5 of the β-Realizability Lemma, it follows that \( p \) is also a majorizer of the conclusion.

Case (∀e).

\[
\begin{align*}
\Gamma; \Delta \vdash t : \forall \alpha A \\
\Gamma; \Delta \vdash t : A[L/\alpha]
\end{align*}
\]

By induction hypothesis, we have a majorizer \( p \) of the premise. Using item 4 of the β-Realizability Lemma, it follows that \( p \) is also a majorizer of the conclusion.
of realizability for $\forall \alpha A$. Showing
that cost $p$ choose $\beta$ applying the definition of realizability for $\Rightarrow$ suppose we have $\bar{u}, \bar{q} \vdash_\eta C$ and $\bar{s}, \bar{m} \vdash_\eta \bar{M}$. Then we wish to prove that
$$t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash_\eta \forall \alpha A$$

Unfolding the definition of $\beta$-realizability for $\forall \alpha A$, let $X$ be any saturated set of type $\alpha$. Then $t(\bar{u}/\bar{x}, \bar{s}/\bar{y})p(\bar{q}) + \bar{m} \vdash_\eta_{\alpha\Rightarrow X} A$ holds since $p$ is a majorizer irrespective of the valuation.

Case ($\Rightarrow$).
$$\Gamma_1; \Delta \vdash t_1 : A \quad \Gamma_2; \vdash t_2 : A$$
$$\Gamma_1, \Gamma_2; \Delta \vdash t_1 t_2 : B$$

By induction hypothesis, we have majorizers $p_1, p_2$ for the premises. Now suppose we have $\bar{u}_i, \bar{q}_i \vdash C_i$ and $\bar{s}_i, \bar{m}_i \vdash \bar{M}_i$ for $i = 1, 2$. Then we wish to prove that $p_1(\bar{x}_1, p_2(\bar{x}_2)) + \downarrow p_2(\bar{x}_2)$ is a suitable majorizer, which follows easily by applying the definition of realizability for $\Rightarrow$ (noting, as in the proof of size realizability, that $t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y}_1)$ = $\square$ can’t occur) and using conditions concatenation and exchange for $\beta$-saturated sets.

Case ($\Rightarrow$).
$$\Gamma, z : A; \Delta \vdash t : B$$
$$\Gamma; \Delta \vdash \lambda z.t : A \Rightarrow B$$

By induction hypothesis, we have a majorizer $p$ for the premise. Suppose we have $\bar{u}, \bar{q} \vdash C$, and $\bar{s}, \bar{m} \vdash \bar{M}$, and $w, r \vdash A$, then $t(\bar{u}/\bar{q}, w/z, \bar{s}/\bar{y}), p(\bar{q}, r) + \bar{m} \vdash B$.

We wish to find $p'$ such that $(\lambda z.t)(\bar{u}/\bar{q}, \bar{s}/\bar{y}), p'(\bar{q}) + \bar{m} \vdash A \Rightarrow B$. We will choose $p'(\bar{x}) = \lambda z.p(\bar{x}, z)$.

Write $t' = (\lambda z.t)(\bar{u}/\bar{q}, \bar{s}/\bar{y})$. Unfolding the definition of realizability, we prove that cost$_\beta(t') \leq$ cost$_\beta(p'(\bar{q}) + \bar{m})$ and for $w, r \vdash A$ we have $t'w, p'(\bar{q}, r) + \bar{m} \vdash B$.

This follows by using the inert variable trick and the induction hypothesis: cost$_\beta(t') \leq$ cost$_\beta(t(\bar{u}/\bar{q}, \bar{s}/\bar{y}, \square)/z)) \leq$ (p(\bar{q}, 0) + \bar{m})$. Noting $t'w = t(\bar{u}/\bar{q}, \bar{s}/\bar{y}, w/z)$ gives us the second part via the induction hypothesis.

Case ($\Rightarrow$).
$$\Gamma_1; \Delta_1 \vdash t_1 : L \Rightarrow B \quad \Gamma_2; \Delta_2 \vdash t_2 : L$$
$$\Gamma_1, \Gamma_2; \Delta_1, \Delta_2 \vdash t_1 t_2 : B$$

By induction hypothesis, we have majorizers $p_1, p_2$ of the premise. We will show $p = p_1 + p_2$ suffices as a majorizer of the conclusion. Unfolding the definition of realizability for $L \Rightarrow B$ (noting, as in the proof of size realizability, that $t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y}_1) = \square$ can’t occur), we obtain
$$t_1(\bar{u}_1/\bar{x}_1, \bar{s}_1/\bar{y}_1)t_2(\bar{s}_2/\bar{y}_2), p_1(\bar{q}_1) + p_2 + \bar{m}_1 + \bar{m}_2 \vdash B$$
We apply conditions concatenation and exchange to obtain the result.

Case \(\lnot i\).

\[
\Gamma; \Delta, z : L \vdash t : B \\
\Gamma; \Delta \vdash \lambda z. t : L \to B
\]

This is similar to \(\Rightarrow i\). By induction hypothesis, we have a majorizer \(p\) of the premise. Then \(p'(\bar{x}) = p(\bar{x})\) works as a majorizer for the conclusion.

Case (Constr).

\[
\Gamma, x : A, y : A; \Delta \vdash t : B \\
\Gamma, z : A; \Delta \vdash t[z/x, z/y] : B
\]

By induction hypothesis, we have a majorizer \(p(\bar{x}, x, y)\) of the premise. Using condition contraction, \(\lambda z.p(\bar{x}, z, z)\) suffices as a majorizer for the conclusion.

Case (Derel).

\[
\Gamma; \Delta, x : L \vdash t : B \\
\Gamma, z : L; \Delta \vdash t[z/x] : B
\]

By induction hypothesis, we have a majorizer \(p\) of the premise. Then \(o(L) = o\), so we can define \(\lambda x.p + x\) as majorizer for the conclusion.

Case (Weak).

\[
\Gamma; \Delta \vdash t : A \\
\Gamma, \Gamma'; \Delta, \Delta' \vdash t : A
\]

By induction hypothesis, we have a majorizer \(p\) of the premise. Suppose \(\bar{u}, \bar{q} \vdash \bar{C}\) and \(\bar{s}, \bar{m} \vdash \bar{M}\), then

\[
t(\bar{u}/\bar{x}, \bar{s}/\bar{y}), p(\bar{q}) + \bar{m} \vdash A
\]

Applying condition \(\beta\)-weakening \(l_x + l_y\) times, and noting that \(\text{cost}(u'_i) \leq q'_i\), we obtain

\[
t(\bar{u}/\bar{x}, \bar{s}/\bar{y}, \bar{u}'/\bar{x}', \bar{s}'/\bar{y}'), p(\bar{q}) + \downarrow \bar{q} + \bar{m} + \bar{m}' + l_x + l_y \vdash A
\]

Condition exchange yields that \(\lambda \bar{x}', p + \downarrow \bar{x}' + l_x + l_y\) is a suitable majorizer for the conclusion.

Using the \(\beta\)-Adequacy, we can give an alternative proof of Soundness of DIAL lin (Theorem 6.1).

**Theorem 7.12 (Soundness).** Suppose \(\vdash t : W_C \Rightarrow W_S\). Then there exists a polynomial time computable function \(f : \Sigma \to \Sigma\) such that for every word \(w\)

\[
(f(w))_S = \beta tw_C
\]
Proof. Inspecting the proof of $\beta$-Adequacy, we see that for every length $n$ there exists a majorizer $q_n$, polynomial in $n$, which satisfies $w_C, q_n \vdash W_C$ for any word $w$ of length $n$.

From $\beta$-Adequacy, we obtain a higher order polynomial $p$ such that for any word $w$ of length $n$, cost$_{\beta}(tw_C) \leq p(q_n)$. Hence the cost of reducing $tw_C$ to full, strong $\beta$-normal form is polynomial in $n$. The full normal form gives us all the bits of the resulting Scott word.

$f$ is then defined as follows: on input $w$, it puts (the code of) $tw_C$ on the tape, then carries out the reduction (which can be done in time polynomial in $p(q_n)$, which is again polynomial), and from this extracts the word. Clearly, this $f$ is polynomial time computable. \hfill $\square$

### 7.3 More general Church and Scott datatypes

Up until now we have worked with Church and Scott words. It is not difficult to see that the same proof works for Church and Scott numerals. A natural question to ask is: what of other Church-style and Scott-style encoded data structures? What of lists, or trees? These will turn out to work fine as well, so that we can say polynomial function in general have type Church $\Rightarrow$ Scott.

We will use binary trees over a finite alphabet. It is also possible to do this for binary trees over the natural numbers, or even finitely branching trees over the natural numbers, but binary trees over a finite alphabet prove the most instructive, and the generalization is straightforward. Importantly, with a finite alphabet, we can take the size of the input tree to be the number of nodes and leaves in the tree. Should we use numbers, then we also have to take into account the size of these.

**Definition 7.13** (Church and Scott lists and trees). Let $S = \{s_0, \ldots, s_k\}$ be a finite set. A finite sequence $(s_{i_0}, \ldots, s_{i_n})$ is encoded Church-Style by

$$s_C = \lambda f_0 f_1 \ldots f_k x. f_{i_0}(\ldots(f_{i_n}x)\ldots)$$

It is encoded Scott-style as a string $s_S$. Elements of $S$ are encoded Scott-style (see Definition 3.6).

A leaf $Leaf$ has Church representation $Leaf_C = \lambda l. l$, and a fork $Fork L a R$ with left and right trees $L, R$ and datum $a$, respectively, has Church representation

$$(Fork L a R)_C = \lambda f. f(Lf)a(Rlf)$$

A leaf $Leaf$ has Scott representation $Leaf_S = \lambda l. l$, and a fork $Fork L a R$ with left and right trees $L, R$ and datum $a$, respectively, has Scott representation

$$(Fork L a R)_S = \lambda f. fLaR$$

Of course these need to be typable in DIAL$_{\text{lin}}$.

**Proposition 7.14.** Let $s$ be any finite sequence, and $T$ any tree with data type $A$. Then the following typings are derivable in DIAL$_{\text{lin}}$. 

---

64
• \( s_C : S_C \) where \( S_C \equiv \forall \alpha (\alpha \rightarrow \alpha) \Rightarrow \ldots \Rightarrow (\alpha \rightarrow \alpha) \) in which \((\alpha \rightarrow \alpha)\) occurs \(k+2\) times.

• \( s_S : S_S \) where \( S_S \equiv \mu \beta \forall \alpha (\beta \rightarrow \alpha) \Rightarrow \ldots \Rightarrow (\beta \rightarrow \alpha) \Rightarrow (\alpha \rightarrow \alpha) \) in which \((\beta \rightarrow \alpha)\) occurs \(k+1\) times.

• \( (s_i)_S : S'_S \) where \( S'_S \equiv \forall \alpha (\alpha \rightarrow \ldots \rightarrow \alpha) \) where \( \rightarrow \) occurs \(k+1\) times.

• \( T_C : Tree^A_C \) where \( Tree^A_C \equiv \forall \alpha (\alpha \rightarrow A \rightarrow \alpha) \Rightarrow \alpha \rightarrow \alpha. \)

• \( T_S : Tree^A_S \) where \( Tree^A_S \equiv \mu \beta \forall \alpha (\beta \rightarrow A \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha. \)

**Proof.** Straightforward.

We can follow the steps from Section 5 and Section 6.3 to prove completeness and soundness. It is clear that these work for sequences. Trees are a little more complicated. While the proof of soundness transfers easily due to Adequacy holding generally for terms typable in \( \text{DIAL}_{\text{lin}} \), the proof of completeness is a bit rougher because it very specifically uses the properties of Church and Scott words and numerals. In particular, we need to be able to extract a size bound from a Church-style tree. In order to do this, we need to be able to add two Church numerals together.

**Proposition 7.15.** There exists a term \( \text{plus}_C \) such that for any \( n, m \in \mathbb{N} \)

\[
\text{plus}_C n C m C = \beta (n + m)_C
\]

and \( \text{cost}_S(\text{plus}_C n C m C) \) is constant. Furthermore, \( \text{plus}_C \) has type \( N_C \rightarrow N_C \rightarrow N_C \) in \( \text{DIAL}_{\text{lin}} \).

**Proof.** We define

\[
\text{plus}_C = \lambda mnfx. mf(nfx)
\]

To see this is typable, we first prove \( f : \alpha \rightarrow \alpha; m : N_C \vdash mf : \alpha \rightarrow \alpha. \)

\[
\frac{f : \alpha \rightarrow \alpha; \vdash f : \alpha \rightarrow \alpha}{\vdash m : N_C \vdash m : N_C} \quad (\text{ax1})
\]

\[
\frac{m : N_C \vdash m : N_C}{\vdash m : N_C \vdash \alpha \rightarrow (\alpha \rightarrow \alpha)} \quad (\forall e)
\]

\[
\frac{f : \alpha \rightarrow \alpha; m : N_C \vdash mf : \alpha \rightarrow \alpha}{\vdash m : N_C \vdash m : (\alpha \rightarrow (\alpha \rightarrow \alpha))} \quad (\Rightarrow e)
\]

Clearly, this also holds for \( n \) instead of \( m \), which allows us to prove prove \( f : \alpha \rightarrow \alpha; n : N_C, x : \alpha \vdash nfx : \alpha \)

\[
\frac{f : \alpha \rightarrow \alpha; n : N_C \vdash nfx : \alpha}{\vdash x : \alpha \vdash x : \alpha} \quad (\text{ax2})
\]

\[
\frac{\vdash x : \alpha \vdash x : \alpha}{\vdash f : \alpha \rightarrow \alpha; n : N_C, x : \alpha \vdash nfx : \alpha} \quad (\Rightarrow e)
\]

Now we can provide the rest of the type derivation.
\( f : \alpha \rightarrow \alpha; \quad n : N_C, x : \alpha \vdash nf : \alpha \quad f : \alpha \rightarrow \alpha; \quad m : N_C \vdash mf : \alpha \rightarrow \alpha \)

\( \vdash m : N_C \vdash \lambda fx.mf(nfx) : \alpha \rightarrow \alpha \)
\( \vdash m : N_C, n : N_C \vdash \lambda fx.mf(nfx) : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \)
\( \vdash m : N_C, n : N_C \vdash \lambda fx.mf(nfx) : N_C \rightarrow N_C \)  
\( \vdash \lambda mnfx.mf(nfx) : N_C \rightarrow N_C \rightarrow N_C \)

Seeing \( \text{plus}_C \) really defines addition is simple. Note that \( m_Cf^\beta = \lambda x.f^m x \), and \( n_Cf = f^n x \), so \( m_Cf(n_Cfx)^\beta = f^m(f^n x) = f^{m+n} x \), which is what we wanted.

\[ \text{cost}_\beta(\text{plus}_Cn_cm_C) \]  

is equal to 6: first two steps for firing the redexes corresponding to \( n_C \) and \( m_C \), then reducing \( m_Cf(n_Cfx) \) which takes two steps, and finally reducing \( n_Cfx \), which takes another two steps. This doesn’t seem to make sense if we consider the inductive definition of addition: we’d expect something linear or polynomial in \( n \) and \( m \). Note however that we only really need to do work if we enter values for \( f \) and \( x \): this addition is more akin to a simple typographical operation than actually doing full-blown addition. \( \square \)

Using \( \text{plus}_C \), we can show how to extract the Church numeral for the size of a binary Church tree from the tree itself.

**Proposition 7.16.** For any data type \( A \), there exists a term \( T_{\text{bound}} \) typable in \( \text{DIAL}_{\text{lin}} \) such that for any tree \( T \) with data type \( A \) with \( n \) items, \( T_{\text{bound}} T \) reduces to \( n_C \) and \( \text{cost}_\beta(T_{\text{bound}} t) \) is polynomial in \( n \).

**Proof.** We write \( F = \lambda LAR.\text{plus}_C(\text{plus}_CLR)1_C \), and define

\[ T_{\text{bound}} = \lambda T. T0_CF \]

Correctness is proven by induction on the tree structure. We prove that for any tree \( T \) of size \( n \), \( T0_CF = \beta n_C \), and that \( \text{cost}_\beta(T0_CF) \) is polynomial in \( n \). The result then easily follows.

\[ (\lambda f.l)0_CF = \beta 0_C \]

Clearly this takes a polynomial number of steps in \( n \): the tree size is 1, and the number of steps is 2.

Write \( n, m \) for the size of the left and right trees, respectively. Then the size of \( T \) with left and right trees \( L, R \) is given by \( n + m + 1 \).

\[ (\lambda f.(Llf)a(Rlf))0_CF = \beta F(L0_CF)a(R0_CF) \]
\[ = \beta F(n_C)a(m_C) \]
\[ = \beta \text{plus}_C(\text{plus}_Cn_cm_C)1_C \]
\[ = \beta (n + m + 1)_C \]
Where we have used the induction hypothesis and the above proposition. The first step has cost 2, the second step has cost polynomial in $n$ and $m$ by induction hypothesis, the third step has cost 3, and the fourth step has cost 12. In total we add a constant amount of work, so the final cost of this reduction is polynomial in the size of $T$.

It is straightforward to obtain the Scott representation of a tree from the Church representation of that tree, so we can safely conclude the following theorem.

**Theorem 7.17** (Main Result for trees). Let $f$ be a function from binary trees over a finite alphabet $A$ to binary trees over $A$. Then $f$ is polynomial time computable if and only if there exists a term $t$ with $\vdash t : \text{Tree}_C^A \Rightarrow \text{Tree}_S^A$ satisfying

$$f(T) = S \iff tT_C =_\beta S_S$$

That is, a function on trees is polynomial if and only if it can be represented by a term of type Church $\Rightarrow$ Scott.

### 8 Conclusions and related work

We’ve seen a type-theoretic characterization of polynomial time computable functions: polynomial time computable functions can be typed Church $\Rightarrow$ Scott. This characterization was developed by Brunel and Terui [5], using a cost model for the weak call-by-value $\lambda$-calculus due to Martini and Dal Lago [10]. We gave an alternate proof of the soundness of this characterization in Section 7.2, based on a result by Accattoli and Dal Lago [1].

Related work falls in roughly two categories: $\lambda$-calculus-oriented and implicit computational complexity-oriented. The $\lambda$-calculus-oriented work primarily concerns itself with cost measures for the $\lambda$-calculus. This is interesting for implicit computational complexity theory because it is relatively easy to add types to the $\lambda$-calculus, thus characterizing functions. While cost models for the weak call-by-value $\lambda$-calculus and full $\lambda$-calculus (with leftmost-outermost reduction) are known, it might also be useful to study the $\lambda$-calculus with parallel reductions or any other variations.

Another area of interest here is complexity of functional programming languages. $\lambda$-calculus acts as a sort of “primordial language” for these, and complexity measures can give new insight in the implementation of functional programming languages on computers. Related to Section 3 in particular are [25, 15, 16, 2, 1, 27].

Implicit computational complexity theory is a fast-growing field with many interesting characteristics. We already mentioned work by Cobham, Leivant, Marion, Bellantoni, and Cook on tiering and characterizing complexity classes using this idea [6, 17, 18, 19, 21, 20, 22, 23, 24, 3].

One obvious question to ask after seeing the main result of this thesis is if we can give similar characterizations of other complexity classes. To our knowledge
there are no such characterizations involving type systems. However, there do exist characterizations in a much similar vein (using tiering and linear logic) of \( \mathbb{P} \) and \( \text{FP} \) by previously mentioned authors [21, 3], of \( \text{PSPACE} \) by Gaboardi, Marion, and others, as well as previous mentioned authors [11, 22, 23], of \( \text{NP} \) by previously mentioned authors [12], of \( \text{LOGSPACE} \) by Schopp [28], and of \( \text{ALOGTIME} \) by previously mentioned authors [24]. This is list far from exhaustive and more results and papers may readily be found by a simple internet search.

We've also seen several different variations on the realizability relation introduced in Subsection 6.1. This is studied in detail by Dal Lago and Hofmann [7, 8, 9], particularly with an eye on developing tools for proofs of implicit complexity characterizations (most proofs so far being quite ad-hoc). The realizability relation from Subsection 6.1 differs on some crucial points: it is not categorical, and the resource monoid used is only partial rather than complete.

Interestingly, our type system is ill-suited for characterizing other complexity classes, such as \( \text{EXP} \) or \( \text{LOGTIME} \), and most implicit characterizations of complexity classes only characterize one class at a time. It would be interesting to see if one can give a classification which distinguishes between, say, exponential and polynomial time functions (obviously a classification distinguishing between polytime and nondeterministic polytime functions would be quite a result!). To our knowledge this has not been done.

All-in-all, it is a very exciting time for people interested in complexity theory.
References


