CONS-FREE PROGRAMMING WITH IMMUTABLE FUNCTIONS

CYNTHIA KOP

Department of Computer Science, Copenhagen University ⋆
e-mail address: kop@di.ku.dk

Abstract. We investigate the power of non-determinism in purely functional programming languages with higher-order types. Specifically, we set out to characterise the hierarchy

$$\mathsf{NP} \subsetneq \mathsf{NEXP} \subsetneq \mathsf{NEXP}^{(2)} \subsetneq \cdots \subsetneq \mathsf{NEXP}^{(k)} \subsetneq \cdots$$

solely in terms of higher-typed, purely functional programs. Although the work is incomplete, we present an initial approach using *cons-free programs with immutable functions*.

1. Introduction

In [3], Jones introduces *cons-free programming*: working with a small functional programming language, cons-free programs are defined to be *read-only*: recursive data cannot be created or altered (beyond taking sub-expressions), only read from the input. By imposing further restrictions on data order and recursion style, classes of cons-free programs turn out to characterise various classes in the time and space hierarchies of computational complexity. However, this concerns only *deterministic* classes.

It is tantalising to consider the non-deterministic classes such as NP: is there a way to characterise these using cons-free programs? Unfortunately, merely adding non-determinism to Jones' language does not suffice: cons-free programs with data order 0 characterise P whether or not a non-deterministic choice operator is included in the language [1], and for higher data orders K adding such an operator increases the expressivity from $\mathsf{EXP}^K\mathsf{TIME}$ to $\mathsf{ELEMENTARY}$ [4]. Thus, additional language features or limitations are needed.

In this work, we explore cons-free programs with *immutable functions*, where data of higher type may be created but not manipulated afterwards. We present some initial ideas to obtain both a characterisation of NP by terminating cons-free programs with immutable functions, and a generalisation towards all classes in the hierarchy

$$\mathsf{NP} \subsetneq \mathsf{NEXP} \subsetneq \mathsf{NEXP}^{(2)} \subsetneq \cdots \subsetneq \mathsf{NEXP}^{(k)} \subsetneq \cdots$$

This is a work in progress; the core results have not yet been fully proven correct, and definitions may be tweaked. The goal is not only to find a characterisation of the hierarchy above, but also to identify the difficulties on the way. It is not unlikely that this could help to find other interesting characterisations, and highlight properties of non-determinism.

⋆ Supported by the Marie Skłodowska-Curie action "HORIP", program H2020-MSCA-IF-2014, 658162.

2. Preliminaries

A more elaborate presentation of the subjects discussed in this section is available in [4].

2.1. Turing Machines and complexity. We assume familiarity with standard notions of Turing Machines and complexity classes (see, e.g., [5, 2]); here, we fix notation.

Turing Machines (TMs) are triples (A, S, T) of finite sets of *tape symbols*, *states* and *transitions*, where $A \supseteq \{0, 1, \llcorner\!\lrcorner\}$, $S \supseteq \{\mathtt{start}, \mathtt{accept}, \mathtt{reject}\}$ and T contains tuples (i, r, w, d, j) with $i \in S \setminus \{\mathtt{accept}, \mathtt{reject}\}$ (the *original state*), $r \in A$ (the *read symbol*), $w \in A$ (the *written symbol*), $d \in \{\mathtt{L}, \mathtt{R}\}$ (the *direction*), and $j \in S$ (the *result state*). Every TM in this paper has a single, right-infinite tape. A TM *accepts* a decision problem $X \subseteq \{0, 1\}^+$ if for any $x \in \{0, 1\}^+$: $x \in X$ iff there is an evaluation starting on the tape $\llcorner\!\lrcorner x_1 \dots x_n \llcorner\!\lrcorner\llcorner\!\lrcorner \dots$ which ends in the $\mathtt{accept}$ state. For $h : \mathbb{N} \longrightarrow \mathbb{N}$ a function, a TM $\mathcal{M}$ *runs in time* $\lambda n.h(n)$ if for $x \in \{0, 1\}^n$: if $\mathcal{M}$ accepts x, then this can be done in at most $h(n)$ steps.

Let $h : \mathbb{N} \to \mathbb{N}$ be a function. Then, $\mathrm{NTIME}\,(h(n))$ is the set of all $X \subseteq \{0, 1\}^+$ such that there exist $a > 0$ and a TM running in time $\lambda n.a \cdot h(n)$ that accepts X.

For $K, n \geq 0$, let $\exp_2^0(n) = n$ and $\exp_2^{K+1}(n) = \exp_2^K(2^n) = 2^{\exp_2^K(n)}$. For $K \geq 0$, define $\mathsf{NEXP}^{(K)} \triangleq \bigcup_{a,b \in \mathbb{N}} \mathrm{NTIME}\left(\exp_2^K(an^b)\right)$. Let $\mathsf{ELEMENTARY} \triangleq \bigcup_{K \in \mathbb{N}} \mathsf{NEXP}^{(K)}$.

2.2. Non-deterministic programs. We consider functional programs with simple types and call-by-value evaluation. Data constructors (denoted $\mathtt{c}$) are at least $\mathtt{true}, \mathtt{false} : \mathtt{bool}$, $[] : \mathtt{list}$ and $:: : \mathtt{bool} \Rightarrow \mathtt{list} \Rightarrow \mathtt{list}$ (denoted infix), although others are allowed. There is no pre-defined integer datatype. Notions of *data* and *values* are given by the grammar to the right, where $\mathtt{f}$ indicates a function symbol defined by one or more clauses. The language supports $\mathtt{if\ then\ else}$ statements, but has no $\mathtt{let}$ construct. For a program $\mathtt{p}$ with *main function* $\mathtt{f}_1 : \iota_1 \Rightarrow \dots \Rightarrow \iota_M \Rightarrow \kappa$—where κ and each ι_i have type order 0—and *input data* $d_1, \dots, d_M$, $\mathtt{p}$ has *result value* b if there is an evaluation $\mathtt{f}_1\ d_1 \cdots d_M \to b$.

$$
\begin{array}{rcl}
d, b \in \mathtt{Data} & ::= & \mathtt{c}\ d_1 \cdots d_m \mid (d, b) \\
v, w \in \mathtt{Value} & ::= & d \mid (v, w) \mid \mathtt{f}\ v_1 \cdots v_n \\
 & & (n < \mathtt{arity}_{\mathtt{p}}(\mathtt{f}))
\end{array}
$$

There is also a non-deterministic choice construct: $\mathtt{choose}\ s_1 \cdots s_m$ may evaluate to a value v if some s_i does. Thus, a program can have multiple result values on the same input.

A program $\mathtt{p}$ with main function $\mathtt{f}_1 : \mathtt{list} \Rightarrow \mathtt{bool}$ *accepts* a decision problem X if for all $x = x_1 \dots x_n \in \{0, 1\}^*$: $x \in X$ iff $\mathtt{p}$ has result value $\mathtt{true}$ on input $\overline{x_1}{::} \dots {::}\overline{x_n}{::}[]$, where $\overline{1} = \mathtt{true}$ and $\overline{0} = \mathtt{false}$. It is not necessary for $\mathtt{true}$ to be the *only* result value.

2.3. Cons-free programs. A clause $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ is *cons-free* if for all sub-expressions t of s: if $t = \mathtt{c}\ t_1 \cdots t_m$ with $\mathtt{c}$ a data constructor, then $t \in \mathtt{Data}$ or t also occurs as a sub-expression of some ℓ_i. A program is cons-free if all its clauses are.

Intuitively, in a cons-free program no new recursive data can be created: all data encountered during evaluation occur inside the input, or as part of some clause.

Example 1. The clauses for $\mathtt{last}$ below are cons-free; the clauses for $\mathtt{flip}$ are not.

$$
\begin{array}{ll}
\mathtt{last}\ (x{::}[]) = x & \mathtt{flip}\ [] = [] \\
\mathtt{last}\ (x{::}y{::}zs) = \mathtt{last}\ (y{::}zs) & \mathtt{flip}\ (\mathtt{true}{::}xs) = \mathtt{false}{::}(\mathtt{flip}\ xs) \\
 & \mathtt{flip}\ (\mathtt{false}{::}xs) = \mathtt{true}{::}(\mathtt{flip}\ xs)
\end{array}
$$

2.4. **Counting.** Cons-free programs neither have an integer data type, nor a way to construct unbounded recursive data (e.g., we cannot build `0`, `s 0`, `s (s 0)` etc.). It *is*, however, possible to design cons-free programs operating on certain bounded classes of numbers: by representing numbers as values using the input data. For example, given a list cs of length n as input:

(1) numbers $i \in \{0, \dots, n\}$ can be represented as lists of length i (sub-expressions of cs);
(2) numbers $i \in \{0, \dots, 4 \cdot (n+1)^2 - 1\}$ can be represented as tuples $(l_1, l_2, l_3) : \mathtt{list} \times \mathtt{list} \times \mathtt{list}$: writing $i = k_1 \cdot (n+1)^2 + k_2 \cdot (n+1) + k_3$, the number i is represented by a tuple $(l_1, \dots, l_3)$ such that each l_i has length k_i;
(3) numbers $i \in \{0, \dots, 2^{4 \cdots (n+1)^2} - 1\}$ can be represented by values $v : (\mathtt{list} \times \mathtt{list} \times \mathtt{list}) \Rightarrow \mathtt{bool}$: writing $i_0 \dots i_{4 \cdot (n+1)^2 - 1}$ for the bitvector corresponding to i, it is represented by any value v such that $v\ [j] \to \mathtt{true}$ iff $i_j = 1$, where $[j]$ is the representation of $j \in \{0, \dots, 4 \cdot (n+1)^2 - 1\}$ as a tuple from point (2).

Building on (3), numbers in $\{0, \dots, \exp_2^K(an^b)\}$ can be represented by values of type order K. It is not hard to construct cons-free rules to calculate successor and predecessor functions, and to test whether a number representation corresponds to 0.

Jones [3] uses these number representations and counting functions to write a cons-free program with data order K which simulates a given TM running in at most $\exp_2^K(an^b)$ steps. However, this program relies heavily on the machine being deterministic.

3. Characterising NP

To characterise non-deterministic classes we start by countering this problem: we present a cons-free program which determines the final state of a non-deterministic TM running in $\lambda n.h(n)$ steps, given number representations for $i \in \{0, \dots, h(n)\}$ and counting functions.

For a machine (A,S,T), let C be a fixed number such that for every $i \in S$ and $r \in A$ there are at most C different triples (w, d, j) such that $(i, r, w, d, j) \in T$. Let T' be a set of tuples (k, i, r, w, d, j) such that (a) $T = \{(i, r, w, d, j) \mid (k, i, r, w, d, j) \text{ occurs in } T'\}$ and (b) each combination (k, i, r) occurs at most once in T'. Now, our simulation uses the data constructors: $\mathtt{true} : \mathtt{bool}$, $\mathtt{false} : \mathtt{bool}$, $[] : \mathtt{list}$ and $:: : \mathtt{bool} \Rightarrow \mathtt{list} \Rightarrow \mathtt{list}$ noted in Section 2.2; $\mathtt{a} : \mathtt{symbol}$ for $a \in A$ (writing `B` for the blank symbol), $\mathtt{L}, \mathtt{R} : \mathtt{direc}$ and $\mathtt{s} : \mathtt{state}$ for $s \in S$; $\mathtt{action} : \mathtt{symbol} \Rightarrow \mathtt{direc} \Rightarrow \mathtt{state} \Rightarrow \mathtt{trans}$; $\mathtt{x1} : \mathtt{option}, \dots, \mathtt{xC} : \mathtt{option}$; and $\mathtt{end} : \mathtt{state} \Rightarrow \mathtt{trans}$. The rules to simulate the machine are given in Figure 1.

If h is a polynomial, this program has data order 1 following (2) of Section 2.4. Thus, non-deterministic cons-free programs with data order 1 can accept any decision problem in NP. However, since even deterministic such programs can accept all problems in EXP $\supseteq$ NP, this is not surprising. What *is* noteworthy is how we use the higher-order value: there is just one functional variable, which, once it has been created, is passed around but never altered. This is unlike the values representing numbers, where we modify a functional value by taking its "successor" or "predecessor". This observation leads to the following definition:

Definition 2. A program has *immutable functions* if for all clauses $\mathtt{f}\ \ell_1 \cdots \ell_k = s$: (a) the clause uses at most one variable with a type of order > 0, and (b) if there is such a variable, then s contains no other sub-expressions with type order > 0.

Every program where all variables have type order 0 automatically has immutable functions. The program of Figure 1 also has immutable functions, provided the number representations $[n]$ all have type order 0. We thus conclude:

$\mathtt{run}\ cs = \mathtt{test}\ (\mathtt{state}\ cs\ (\mathtt{rndf}\ cs\ [h(|cs|)])\ [h(|cs|)])$
$\mathtt{test}\ \mathtt{accept} = \mathtt{true}$
$\mathtt{test}\ s = \mathtt{false}$ for all $s \in S \setminus \{\mathtt{accept}\}$

$\mathtt{rndf}\ cs\ [n] = \mathtt{rnf}\ cs\ (\mathtt{choose}\ \mathtt{x1} \cdots \mathtt{xC})\ [n]$
$\mathtt{rnf}\ cs\ x\ [n] = \mathtt{if}\ [n = 0]\ \mathtt{then}\ x\ \mathtt{else}\ \mathtt{cmp}\ x\ (\mathtt{rndf}\ cs\ [n-1])\ [n]$
$\mathtt{cmp}\ x\ H\ [n]\ [i] = \mathtt{if}\ [n = i]\ \mathtt{then}\ x\ \mathtt{else}\ F\ i$

$\mathtt{transition}\ \mathtt{i}\ \mathtt{r}\ \mathtt{ch}_k = \mathtt{action}\ \mathtt{w}\ \mathtt{d}\ \mathtt{j}$ for all $(k,i,r,w,d,j) \in T'$
$\mathtt{transition}\ \mathtt{i}\ x\ y = \mathtt{end}\ \mathtt{i}$ for $i \in \{\mathtt{accept}, \mathtt{reject}\}$
$\mathtt{transition}\ \mathtt{i}\ \mathtt{r}\ \mathtt{ch}_k = \mathtt{end}\ \mathtt{reject}$ for all (k,i,r) s.t. $i \notin \{\mathtt{accept}, \mathtt{reject}\}$ and there are no (w,d,j) with $(k,i,r,w,d,j) \in T'$
$\mathtt{transat}\ cs\ H\ [n] = \mathtt{transition}\ (\mathtt{state}\ cs\ H\ [n])\ (\mathtt{tapesymb}\ cs\ H\ [n])\ (H\ [n])$

$\mathtt{get1}\ (\mathtt{action}\ x\ y\ z) = x$ $\quad$ $\mathtt{get1}\ (\mathtt{end}\ x) = \mathtt{B}$
$\mathtt{get2}\ (\mathtt{action}\ x\ y\ z) = y$ $\quad$ $\mathtt{get2}\ (\mathtt{end}\ x) = \mathtt{R}$
$\mathtt{get3}\ (\mathtt{action}\ x\ y\ z) = z$ $\quad$ $\mathtt{get3}\ (\mathtt{end}\ x) = x$

$\mathtt{state}\ cs\ H\ [n] = \mathtt{if}\ [n = 0]\ \mathtt{then}\ \mathtt{start}\ \mathtt{else}\ \mathtt{get3}\ (\mathtt{transat}\ cs\ H\ [n-1])$

$\mathtt{tapesymb}\ cs\ H\ [n] = \mathtt{tape}\ cs\ H\ [n]\ (\mathtt{pos}\ cs\ H\ [n])$

$\mathtt{pos}\ cs\ H\ [n] = \mathtt{if}\ [n = 0]\ \mathtt{then}\ [0]$
$\qquad \mathtt{else}\ \mathtt{adjust}\ cs\ (\mathtt{pos}\ cs\ H\ [n-1])\ (\mathtt{get2}\ (\mathtt{transat}\ cs\ H\ [n-1]))$
$\mathtt{adjust}\ cs\ [p]\ \mathtt{L} = [p-1]$
$\mathtt{adjust}\ cs\ [p]\ \mathtt{R} = [p+1]$

$\mathtt{tape}\ cs\ H\ [n]\ [p] = \mathtt{if}\ [n = 0]\ \mathtt{then}\ \mathtt{inputtape}\ cs\ [p]$
$\qquad \mathtt{else}\ \mathtt{tapehelp}\ cs\ H\ [n]\ [p]\ (\mathtt{pos}\ cs\ H\ [n-1])$
$\mathtt{tapehelp}\ cs\ H\ [n]\ [p]\ [i] = \mathtt{if}\ [p = i]\ \mathtt{then}\ \mathtt{get1}\ (\mathtt{transat}\ cs\ H\ [n-1])$
$\qquad \mathtt{else}\ \mathtt{tape}\ cs\ H\ [n-1]\ [p]\ [i]$

$\mathtt{inputtape}\ cs\ [p] = \mathtt{if}\ [p = 0]\ \mathtt{then}\ \mathtt{B}\ \mathtt{else}\ \mathtt{nth}\ cs\ [p-1]$
$\mathtt{nth}\ []\ [p] = \mathtt{B}$
$\mathtt{nth}\ (x{::}xs)\ [p] = \mathtt{if}\ [p = 0]\ \mathtt{then}\ \mathtt{bit}\ x\ \mathtt{else}\ \mathtt{nth}\ xs\ [p-1]$
$\mathtt{bit}\ \mathtt{true} = \mathtt{1}$
$\mathtt{bit}\ \mathtt{false} = \mathtt{0}$

Figure 1: Simulating a Non-deterministic Turing Machine (A, S, T')

Lemma 3. *Every decision problem in* NP *is accepted by a terminating cons-free program with immutable functions.*

Proof. If $X \in \mathsf{NP}$, then there are a, b and a TM $\mathcal{M}$ such that for all n and $x \in \{0,1\}^n$: $x \in X$ iff there is an evaluation of $\mathcal{M}$ which accepts x in at most $a \cdot n^b$ steps. Using $\mathcal{M}$ to build the program of Figure 1, $\mathtt{run}\ \overline{x} \to \mathtt{true}$ iff $\mathcal{M}$ accepts x, iff $x \in X$. □

For terminating cons-free programs with immutable functions to *characterise* NP, it remains to be seen that every decision problem accepted by such a program is in NP. That is, for a fixed program `p` we must design a (non-deterministic) algorithm operating in polynomial time which returns `YES` for input data d iff `p` has `true` as a result value on input d.

Towards this purpose, we first alter the program slightly:

Lemma 4. *If* $[\![\mathtt{p}]\!](d_1,\dots,d_M) \mapsto b$*, then* $[\![\mathtt{p}']\!](d_1,\dots,d_m) \mapsto b$*, where* $\mathtt{p}'$ *is obtained from* $\mathtt{p}$ *by replacing all sub-expressions* $s\ t_1 \cdots t_n$ *in the right-hand side of a clause where* s *is not an application by* $s \circ t_1 \cdots t_n$*. Here,*

- $(\mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3) \circ t_1 \cdots t_n = \mathtt{if}\ s_1\ \mathtt{then}\ (s_2 \circ t_1 \cdots t_n)\ \mathtt{else}\ (s_3 \circ t_1 \cdots t_n)$
- $(\mathtt{choose}\ s_1 \cdots s_m) \circ t_1 \cdots t_n = \mathtt{choose}\ (s_1 \circ t_1 \cdots t_n) \cdots (s_m \circ t_1 \cdots t_n)$
- $(a\ s_1 \cdots s_i) \circ t_1 \cdots t_n = a\ s_1 \cdots s_i\ t_1 \cdots t_n$ *for* $a \in \mathcal{V} \cup \mathcal{C} \cup \mathcal{D}$*.*

In addition, there is a transformation which preserves and reflects $[\![\mathtt{p}]\!](d_1,\dots,d_M) \mapsto b$ *such that all argument types of defined symbols and all clauses have type order* ≤ 1 *and there are no clauses* $\mathtt{f}\ \ell_1 \cdots \ell_k = x$ *with* x *a variable of functional type.*

Idea. The $\mathtt{if}/\mathtt{choose}$ change trivially works, the type order change uses the restriction on data order as detailed in [4, Lemma 1] and functional variables can be avoided because, by immutability, all clauses for $\mathtt{f}$ must have a variable as their right-hand side. □

We implicitly assume that the transformations of Lemma 4 have been done.

To reason on values in the algorithm, we define a semantical way to describe them.

Definition 5. For a fixed cons-free program $\mathtt{p}$ and input data expressions $d_1,\dots,d_M$, let $\mathcal{B}$ be the set of data expressions which occur either as a sub-expression of some d_i, or as sub-expression of the right-hand side of some clause. For ι a sort (basic type), let $[\![\iota]\!]_\mathcal{B} := \{b \in \mathcal{B} \mid b : \iota\}$. Also let $[\![\sigma \times \tau]\!]_\mathcal{B} := [\![\sigma]\!]_\mathcal{B} \times [\![\tau]\!]_\mathcal{B}$, and if κ is not an arrow type, $[\![\sigma_1 \Rightarrow \dots \Rightarrow \sigma_n \Rightarrow \kappa]\!]_\mathcal{B} := \{(e_1,\dots,e_n,o) \mid \forall 1 \leq i \leq n[e_i \in [\![\sigma_i]\!]_\mathcal{B}] \wedge o \in [\![\kappa]\!]_\mathcal{B}\}$

By relating values of type σ to elements of $[\![\sigma]\!]_\mathcal{B}$, we can prove that Algorithm 6 below returns b if and only if b is a result value of $\mathtt{p}$ for input $d_1,\dots,d_M$.

Algorithm 6. Let $\mathtt{p}$ be a fixed, terminating cons-free program with immutable functions.

Input: data expressions $d_1 : \kappa_1,\dots,d_M : \kappa_M$.

For every function symbol $\mathtt{f}$ with type $\sigma_1 \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa$ and arity k (the number of arguments to $\mathtt{f}$ in clauses), every $k \leq i \leq m$ and all $e_1 \in [\![\sigma_1]\!]_\mathcal{B},\dots,e_i \in [\![\sigma_i]\!]_\mathcal{B}, o \in [\![\sigma_{i+1} \Rightarrow \dots \Rightarrow \sigma_m \Rightarrow \kappa]\!]_\mathcal{B}$, note down a "statement" $\vdash \mathtt{f}\ e_1 \cdots e_i \mapsto o$. For all clauses $\mathtt{f}\ \ell_1 \cdots \ell_k = s$, also note down statements $\eta \vdash t \to o$ for every sub-expression $t : \sigma$ of $s \circ x_{k+1} \cdots x_i$, $o \in [\![\sigma]\!]_\mathcal{B}$ and η mapping all $y : \tau \in Var(\mathtt{f}\ \ell_1 \cdots \ell_k\ x_{k+1} \cdots x_i)$ to some element of $[\![\tau]\!]_\mathcal{B}$.

Treating η as a substitution, mark statements $\eta \vdash t \mapsto o$ confirmed if $t\eta = o$.

Now repeat the following steps, until no further changes are made:

1. Mark statements $\vdash \mathtt{f}\ e_1 \cdots e_i \mapsto o$ confirmed if $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ is the first clause that matches $\mathtt{f}\ e_1 \cdots e_k$ and $\eta \vdash s \circ x_{k+1} \cdots x_i \mapsto o$ is marked confirmed, where η is the "substitution" such that $(\mathtt{f}\ \ell_1 \cdots \ell_k\ x_{k+1} \cdots x_i)\eta = \mathtt{f}\ e_1 \cdots e_i$.
2. Mark statements $\eta \vdash x\ s_1 \cdots s_m \mapsto o$ with x a variable confirmed if there is $(e_1,\dots, e_m, o) \in \eta(x)$ s.t. $\eta \vdash s_i \mapsto e_i$ for all i. (By immutability, $x\ s_1 \cdots s_m$ has base type.)
3. Mark statements $\eta \vdash (s_1, s_2) \mapsto (o_1, o_2)$ confirmed if both $\eta \vdash s_i \mapsto o_i$ are confirmed.
4. Mark statements $\eta \vdash \mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3 \mapsto o$ confirmed if (a) $\eta \vdash s_1 \mapsto \mathtt{true}$ and $\eta \vdash s_2 \mapsto o$ are both confirmed, or (b) $\eta \vdash s_1 \mapsto \mathtt{false}$ and $\eta \vdash s_3 \mapsto o$ are both confirmed.
5. Mark statements $\eta \vdash \mathtt{choose}\ s_1 \cdots s_m \mapsto o$ confirmed if some $\eta \vdash s_i \mapsto o$ is confirmed.
6. Mark statements $\eta \vdash \mathtt{f}\ s_1 \cdots s_n \mapsto o$ confirmed if there are $e_1,\dots,e_n$ with $\vdash s_i \mapsto e_i$ confirmed for $1 \leq i \leq n$ and either (a) $n \geq \mathtt{arity}_\mathtt{p}(\mathtt{f})$ and $\mathtt{f}\ e_1 \cdots e_n \mapsto o$ is confirmed, or (b) $n < \mathtt{arity}_\mathtt{p}(\mathtt{f})$ and $\mathtt{f}\ e_1 \cdots e_m \mapsto u$ is confirmed for all $(e_{n+1},\dots,e_m,u) \in o$.

Output: return the set of all b such that $\mathtt{f}_1\ d_1 \cdots d_M \mapsto b$ is confirmed.

This algorithm has exponential complexity since, for σ of order 1, the cardinality of $[\![\sigma]\!]_{\mathcal{B}}$ is exponential in the input size (the number of constructors in $d_1, \dots, d_M$). However, since a program with immutable functions cannot effectively use values with type order > 1—so can be transformed to give all values and clauses type order 1 or 0—and the size of each $e \in [\![\sigma]\!]_{\mathcal{B}}$ is polynomial, the following non-deterministic algorithm runs in polynomial time:

Algorithm 7. Let $S := \{\kappa \mid \kappa$ is a type of order 0 which is used as argument type of some $\mathtt{f}\}$. Let $T := \max\{\mathsf{Card}([\![\kappa]\!]_{\mathcal{B}}) \mid \kappa \in S\}$, and let $N := \langle$number of function symbols$\rangle \cdot T^{2 \cdot \langle\text{greatest arity}\rangle \cdot \langle\text{greatest clause depth}\rangle + 1}$. For every clause $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ of base type, and every sub-expression of s which has a higher type σ and is not a variable, generate N elements of $[\![\sigma]\!]_{\mathcal{B}}$. Let $\Xi := \bigcup_{\kappa \in S} [\![\kappa]\!]_{\mathcal{B}} \cup \{$ the functional "values" thus generated $\}$.

Now run Algorithm 6, but only consider statements with all e_i and o in Ξ.

Proposition 8. $\mathtt{p}$ *has result value b iff there is an evaluation of Algorithm 7 which returns a set containing b.*

Proof Idea. We can safely assume that if $\mathtt{f}\ b_1 \cdots b_m \to d$, it is derived in the same way each time it is used. Therefore, in any derivation, at most $T^{\langle\text{greatest arity}+1\rangle}$ distinct values are created to be passed around; the formation of each value may require $\langle$number of function symbols$\rangle \cdot T^{\langle\text{greatest arity} \cdot \langle\text{greatest clause depth}\rangle - 1}$ additional helper values. □

By Lemma 3 and Proposition 8, we have: terminating cons-free programs with immutable functions characterise NP. This also holds for the limitation to any data order ≥ 1.

4. Beyond NP

Unlike Jones, we do not obtain a hierarchy of characterisations for increasing data orders. However, we *can* obtain a hierarchical result by extending the definition of *immutable*:

Definition 9. A program has *order n immutable functions* if for all clauses $\mathtt{f}\ \ell_1 \cdots \ell_k = s$: (a) the clause uses at most one variable with a type of order $\geq n$, and (b) if there is such a variable, then s contains no other sub-expressions with type order $\geq n$.

Proposition 10. *Terminating cons-free programs with order $K + 1$ immutable functions characterise* $\mathsf{NEXP}^{(K)}$.

Proof Idea. An easy adaptation from the proofs of Lema 3 and Proposition 8. □

5. Conclusion and discussion

If Propositions 8 and 10 hold, we have obtained a characterisation of the hierarchy $\mathsf{NP} \subsetneq \mathsf{NEXP} \subsetneq \mathsf{NEXP}^{(2)} \subsetneq \cdots \subsetneq \mathsf{NEXP}^{(K)} \subsetneq \cdots$ in primarily syntactic terms.

Arguably, this is a rather inelegant characterisation, both because of the termination requirement and because the definition of immutability itself is somewhat arcane; it is not a direct translation of the intuition that functional values, once created, may not be altered.

The difficulty is that non-determinism is very powerful, and easily raises expressivity too far when not contained. This is evidenced in [4] where adding non-determinism to cons-free programs of data order $K \geq 1$ raises the characterised class from $\mathsf{EXP}^{(K)}$ to $\mathsf{ELEMENTARY}$. (In [4], we did not use immutability; alternatively restricting the clauses to disallow partial application of functional variable resulted in the original hierarchy

$\mathsf{P} \subsetneq \mathsf{EXP}^{(1)} \subsetneq \mathsf{EXP}^{(2)} \subsetneq \cdots$.) In our setting, we must be careful that the allowances made to *build* the initial function cannot be exploited to manipulate exponentially many distinct values. For example, if we drop the termination requirement, we could identify the lowest number $i < 2^n$ such that $P(i)$ holds for any polytime-decidable property P, as follows:

$\mathtt{bit\_of\_lowest}\ [n]\ [j] = \mathtt{f}\ [n]\ [j]$ $\qquad$ $\mathtt{nul}\ [j] = \mathtt{false}$

$\mathtt{f}\ [n] = \mathtt{choose}\ \mathtt{nul}\ (\mathtt{succtest}\ (\mathtt{f}\ [n]))$ $\qquad$ $\mathtt{succ}\ F\ [n]\ [j] = \dots$

$\mathtt{succtest}\ F\ [j] = \mathtt{if}\ \mathtt{prop}\ F\ \mathtt{then}\ F\ [j]\ \mathtt{else}\ \mathtt{succ}\ F\ [n]\ [j]$

Here, the clauses for $\mathtt{succ}\ F\ [n]\ [j]$ result in `true` if $b_j = 1$ when representing the successor of F as a bitvector $b_1 \dots b_n$, and in `false` otherwise. It is unlikely that the corresponding decision problem is in NP. Similar problems may arise if we allow multiple higher-order variables, although we do not yet have an example illustrating this problem.

In the future, we intend to complete the proofs, and study these restrictions further. Even if this does not lead to an elegant characterisation of the $\mathsf{NEXP}^{(K)}$ hierarchy, it is likely to give further insights in the power of non-determinism in higher-order cons-free programs.

References

[1] G. Bonfante. Some programming languages for logspace and ptime. In *AMAST*, pages 66–80, 2006.
[2] N. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1997.
[3] N. Jones. The expressive power of higher-order types or, life without CONS. *Journal of Functional Programming*, 11(1):55–94, 2001.
[4] C. Kop and J. Simonsen. The power of non-determinism in higher-order implicit complexity. In *ESOP*, pages 668–695, 2017.
[5] C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.