The Power of Non-Determinism in Higher-Order Implicit Complexity *

Characterising Complexity Classes using Non-deterministic Cons-free Programming

Cynthia Kop and Jakob Grue Simonsen

Department of Computer Science, University of Copenhagen (DIKU)
kop@di.ku.dk simonsen@di.ku.dk

Abstract. We investigate the power of non-determinism in purely functional programming languages with higher-order types. Specifically, we consider *cons-free* programs of varying data orders, equipped with explicit non-deterministic choice. Cons-freeness roughly means that data constructors cannot occur in function bodies and all manipulation of storage space thus has to happen indirectly using the call stack.

While cons-free programs have previously been used by several authors to characterise complexity classes, the work on *non-deterministic* programs has almost exclusively considered programs of data order 0. Previous work has shown that adding explicit non-determinism to cons-free programs taking data of order 0 does not increase expressivity; we prove that this—dramatically—is not the case for higher data orders: adding non-determinism to programs with data order at least 1 allows for a characterisation of the entire class of elementary-time decidable sets.

Keywords: implicit computational complexity, cons-free programming, EXPTIME hierarchy, non-deterministic programming, unary variables

1 Introduction

Implicit complexity is, roughly, the study of how to create bespoke programming languages that allow the programmer to write programs which are guaranteed to (a) *only* solve problems within a certain complexity class (e.g., the class of polynomial-time decidable sets of binary strings), and (b) to be able to solve *all* problems in this class. When equipped with an efficient execution engine, the programs of such a language may themselves be guaranteed to run within the complexity bounds of the class (e.g., run in polynomial time), and the plethora of means available for analysing programs devised by the programming language community means that methods from outside traditional complexity theory can conceivably be brought to bear on open problems in computational complexity.

* The authors are supported by the Marie Skłodowska-Curie action "HORIP", program H2020-MSCA-IF-2014, 658162 and by the Danish Council for Independent Research Sapere Aude grant "Complexity via Logic and Algebra" (COLA).

One successful approach to implicit complexity is to syntactically constrain the programmer's ability to create new data structures. In the seminal paper [12], Jones introduced *cons-free programming*: working with a small functional programming language, cons-free programs are *read-only*: recursive data cannot be created or altered (beyond taking sub-expressions), only read from input. By imposing further restrictions on *data order* (i.e., order 0 = integers, strings; order 1 = functions on data of order 0; etc.) and recursion scheme (e.g., full/tail/primitive recursion), classes of cons-free programs turn out to characterise various deterministic classes in the time and space hierarchies of computational complexity.

However, Jones' language is fully deterministic and, perhaps as a result, his characterisations concern only deterministic complexity classes. It is tantalising to consider the method in a non-deterministic setting instead: could adding non-deterministic choice directly to Jones' language suffice to increase the expressivity; for example, from P to NP?

The immediate answer is *no*: Bonfante showed [3] that adding a non-deterministic choice operator to cons-free programs with data order 0 makes no difference in expressivity: deterministic or not, such programs characterise P. However, the technical details are subtle and depend heavily on other features of the language: if the language is restricted to having primitive recursion, adding non-determinism does increase expressivity from L to NL [3].

While many authors consider the expressivity of allowing higher types, the interplay of higher types and non-determinism is not completely understood. For deterministic programs, Jones obtains several hierarchies of deterministic complexity classes by increasing the data order [12], but these hierarchies have at most an exponential increase in complexity between levels; given the expressivity added by non-determinism, it is *a priori* not evident that similarly "tame" hierarchies would arise in the non-deterministic setting.

The purpose of the present paper is to investigate the power of *higher-order* (cons-free) programming to characterise complexity classes. The main surprise is that while non-determinism does not add expressivity for first-order programs, the combination of second-order (or higher) programs and non-determinism characterises the full class of elementary-time decidable sets—and increasing the order beyond second-order programs does not further increase expressivity.

1.1 Overview and contributions

We define a purely functional programming language with non-deterministic choice and, following Jones [12], consider the restriction to *cons-free* programs.

Our results are summarised in Figure 1. For completeness, we have also included the results from [12]; although the language used there is slightly more syntactically restrictive than ours, the results easily generalise provided we limit interest to *deterministic* programs, where the `choose` operator is not used. As the technical machinations involved to procure the results for a language with full recursion are already intricate and lengthy, we have not yet considered the restriction to tail- or primitive recursion in the non-deterministic setting.

	data order 0	**data order 1**	**data order 2**	**data order 3**
cons-free deterministic	$\mathsf{P} = \mathsf{EXP}^0\mathsf{TIME}$	$\mathsf{EXP} = \mathsf{EXP}^1\mathsf{TIME}$	$\mathsf{EXP}^2\mathsf{TIME}$	$\mathsf{EXP}^3\mathsf{TIME}$
cons-free tail-recursive deterministic	$\mathsf{L} = \mathrm{EXP}^{-1}\mathrm{SPACE}$	$\mathsf{PSPACE} = \mathrm{EXP}^0\mathrm{SPACE}$	$\mathrm{EXP}^1\mathrm{SPACE}$	$\mathrm{EXP}^2\mathrm{SPACE}$
cons-free primitive recursive deterministic	$\mathsf{L} = \mathrm{EXP}^{-1}\mathrm{SPACE}$	$\mathsf{P} = \mathsf{EXP}^0\mathsf{TIME}$	$\mathsf{PSPACE} = \mathrm{EXP}^0\mathrm{SPACE}$	$\mathsf{EXP} = \mathsf{EXP}^1\mathsf{TIME}$

The characterisations obtained in [12], transposed to the more permissive language used here. This list (and the one below) should be imagined as extending infinitely to the right. The "limit" for all rows (i.e., all finite data orders allowed) characterises ELEMENTARY, the class of elementary-time decidable sets.

	data order 0	**data order 1**	**data order 2**	**data order 3**
cons-free	P	ELEMENTARY	ELEMENTARY	ELEMENTARY
cons-free unary variables	$\mathsf{P} = \mathsf{EXP}^0\mathsf{TIME}$	$\mathsf{EXP} = \mathsf{EXP}^1\mathsf{TIME}$	$\mathsf{EXP}^2\mathsf{TIME}$	$\mathsf{EXP}^3\mathsf{TIME}$

Characterisations obtained by allowing non-deterministic choice. As above, the "limit" where all data orders are allowed characterises ELEMENTARY (for both rows).

Fig. 1. Overview of the results discussed or obtained in this paper.

On a grander scale, our paper has two major contributions: (a) we show that previous observations about the increase in expressiveness when adding non-determinism change dramatically at higher types, and (b) we provide a characterisation of the EXPTIME hierarchy using a non-deterministic language. We believe that this may provide a basis for future characterisation of the non-deterministic classes between the members of this hierarchy.

Note that (a) is highly surprising: As evidenced by early work of Cook [5] merely adding full non-determinism to a restricted (i.e., non-Turing complete) computation model may result in it characterising a *deterministic* class of problems. This observation holds true for cons-free programs where non-determinism has been added as well, by Bonfante [3], by de Carvalho and Simonsen [7], and by Kop and Simonsen [14], all resulting in characterisations of deterministic classes such as P but not in characterisations of NP (unless $\mathsf{P} = \mathsf{NP}$). With the exception of [14], all of the above attempts at adding non-determinism have considered data order at most 0, and one would expect few changes when passing to higher data orders. This turns out to be patently false as simply increasing to data order 1 results in a characterisation of the elementary-time decidable sets.

1.2 Related work

The creation of programming languages that characterise complexity classes has been a research area since Cobham's work in the 1960ies, but saw rapid development only after similar advances in the related area of *descriptive complexity* (see,

e.g., [10]) in the 1980ies and Bellantoni and Cook's work on characterisations of P [1] using constraints on recursion in a purely functional language with programs reminiscent of classic recursion theoretic functions. Following Bellantoni and Cook, a number of authors obtained programming languages by constraints on recursion, and under a plethora of names (e.g., *safe*, *tiered* or *ramified* recursion, see [4,6] for overviews), and this area continues to be active. The main difference with our work is that we consider full recursion in all variables, but place syntactic constraints on the function bodies (both cons-freeness and unary variables). Also, as in traditional complexity theory we consider decision problems (i.e., what *sets* can be decided by programs), whereas much research in implicit complexity considers functional complexity (i.e., what *functions* can be computed).

Cons-free programs, combined with various limitations on recursion, were introduced by Jones [12], building on ground-breaking work by Goerdt [9,8], and have been studied by a number of authors and inspired similar results for *imperative* languages (for example [16,15]). The main difference with our work is that we consider full recursion with full non-determinism, but—in Section 7—impose an extra constraint (*unary variables*) not present in the previous literature.

Characterisation of non-deterministic complexity classes via programming languages remains a largely unexplored area. Bellantoni obtained a characterisation of NP in his dissertation [2] using similar approaches as [1], but at the cost of having a minimisation operator (as in recursion theory), a restriction later removed by Oitavem [17]. A general framework for implicitly characterising a larger hierarchy of non-deterministic classes remains an open problem.

2 A purely functional, non-deterministic, call-by-value programming language

We define a simple call-by-value programming language with explicit non-deterministic choice. This generalises Jones' toy language in [12] by supporting different types and pattern-matching as well as non-determinism. Using a more permissive language actually *simplifies* the proofs and examples, since we do not need to encode all data as boolean lists, and have fewer special cases.

2.1 Syntax

We consider programs defined by the syntax in Figure 2

$$
\begin{array}{rl}
\mathtt{p} \in \mathtt{Program} & ::= \rho_1\ \rho_2\ \ldots\ \rho_N \\
\rho \in \mathtt{Clause} & ::= \mathtt{f}\ \ell_1 \cdots \ell_k = s \\
\ell \in \mathtt{Pattern} & ::= x \mid \mathtt{c}\ \ell_1 \cdots \ell_m \\
s, t \in \mathtt{Expr} & ::= x \mid \mathtt{c} \mid \mathtt{f} \mid \mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3 \mid \mathtt{choose}\ s_1 \cdots s_n \mid (s,t) \mid s\ t \\
x, y \in \mathcal{V} & ::= \text{identifier} \\
\mathtt{c} \in \mathcal{C} & ::= \text{identifier disjoint from } \mathcal{V} \quad (\text{we assume } \{\mathtt{true}, \mathtt{false}\} \subseteq \mathcal{C}) \\
\mathtt{f}, \mathtt{g} \in \mathcal{D} & ::= \text{identifier disjoint from } \mathcal{V} \text{ and } \mathcal{C}
\end{array}
$$

Fig. 2. Syntax

We call elements of $\mathcal{V}$ *variables*, elements of $\mathcal{C}$ *data constructors* and elements of $\mathcal{D}$ *defined symbols*. The *root* of a clause $\mathtt{f}\ \ell_1 \cdots \ell_k = e$ is the defined symbol $\mathtt{f}$. The *main function* $\mathtt{f}_1$ of the program is the root of ρ_1. We denote $Var(s)$ for the set of variables occurring in an expression s. An expression s is *ground* if $Var(s) = \emptyset$. Application is left-associative, i.e., $s\ t\ u$ should be read $(s\ t)\ u$.

Definition 1. *For expressions s, t, we say that t* is a sub-expression of *s, notation $s \trianglerighteq t$, if this can be derived using the clauses:*

$$\begin{array}{rl} s \trianglerighteq t \text{ if } s = t \text{ or } s \rhd t & \\ (s_1, s_2) \rhd t \text{ if } s_1 \trianglerighteq t \text{ or } s_2 \trianglerighteq t & \mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3 \rhd t \text{ if } s_i \trianglerighteq t \text{ for some } i \\ s_1\ s_2 \rhd t \text{ if } s_1 \rhd t \text{ or } s_2 \trianglerighteq t & \mathtt{choose}\ s_1 \cdots s_n \rhd t \text{ if } s_i \trianglerighteq t \text{ for some } i \end{array}$$

Note: the head s of an application $s\ t$ is not *considered a sub-expression of $s\ t$.*

Note that the programs we consider do not have pre-defined data structures such as integers: these have to be encoded using inductive data structures in the usual fashion.

Example 1. Integers can be encoded as bitstrings of unbounded length: $\mathcal{C} \supseteq \{\mathtt{false}, \mathtt{true}, ::, []\}$. Here, :: is considered infix and right-associative, and [] denotes the end of the string. Using little endian, for example the number 6 is encoded by $\mathtt{false}::\mathtt{true}::\mathtt{true}::[]$ as well as $\mathtt{false}::\mathtt{true}::\mathtt{true}::\mathtt{false}::\mathtt{false}::[]$.

An example program using $\mathcal{C}$ is:

$$\begin{array}{ll} \mathtt{succ}\ [] = \mathtt{true}::[] & \mathtt{succ}\ (\mathtt{false}::xs) = \mathtt{true}::xs \\ & \mathtt{succ}\ (\mathtt{true}::xs) = \mathtt{true}::(\mathtt{succ}\ xs) \end{array}$$

Here, $\mathcal{D} = \{\mathtt{succ}\}$, and $xs \in \mathcal{V}$ and we for instance have $1::(\mathtt{succ}\ xs) \trianglerighteq xs$.

2.2 Typing

Programs have explicit simple types without polymorphism, with the usual definition of type order $o(\sigma)$; this is formally given in Figure 3.

$$\begin{array}{cc} \iota \in \mathcal{S} ::= \text{sort identifier} & o(\iota) = 0 \ \text{ for } \iota \in \mathcal{S} \\ \sigma, \tau \in \mathtt{Type} ::= \iota \mid \sigma \times \tau \mid \sigma \Rightarrow \tau & o(\sigma \times \tau) = \max(o(\sigma), o(\tau)) \\ & o(\sigma \Rightarrow \tau) = \max(o(\sigma) + 1, o(\tau)) \end{array}$$

Fig. 3. Types and type orders

The (finite) set $\mathcal{S}$ of sorts is used to type atomic data such as bits; we assume $\mathtt{bool} \in \mathcal{S}$. The function arrow $\Rightarrow$ is considered right-associative, so $\sigma \Rightarrow \tau \Rightarrow \pi$ should be read $\sigma \Rightarrow (\tau \Rightarrow \pi)$. Writing κ for either a sort or a pair type $\sigma \times \tau$, note that any type can be uniquely presented in the form $\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa$.

We will limit interest to *well-typed, well-formed* programs:

Definition 2. *A program* $\mathtt{p}$ *is* well-typed *if there is an assignment $\mathcal{F}$ from $\mathcal{C} \cup \mathcal{D}$ to the set of simple types such that:*

- *the main function $\mathtt{f}_1$ is assigned a type $\kappa_1 \Rightarrow \ldots \Rightarrow \kappa_M \Rightarrow \kappa$, with $o(\kappa_i) = 0$ for $1 \leq i \leq M$ and also $o(\kappa) = 0$*

- *data constructors* $\mathtt{c} \in \mathcal{C}$ *are assigned a type* $\kappa_1 \Rightarrow \ldots \Rightarrow \kappa_m \Rightarrow \iota$ *with* $\iota \in \mathcal{S}$ *and* $o(\kappa_i) = 0$ *for* $1 \leq i \leq m$
- *for all clauses* $\mathtt{f}\ \ell_1 \cdots \ell_k = s \in \mathtt{p}$*, the following hold:*
  - $\mathtt{f}\ \ell_1 \cdots \ell_k$ *is linear: no variable occurs more than once;*
  - $Var(s) \subseteq Var(\mathtt{f}\ \ell_1 \cdots \ell_k)$*;*
  - *there exist a type environment* Γ *mapping* $Var(\mathtt{f}\ \ell_1 \cdots \ell_k)$ *to simple types, and a simple type* σ*, such that both* $\mathtt{f}\ \ell_1 \cdots \ell_k : \sigma$ *and* $s : \sigma$ *using the rules in Figure 4; we call* σ *the type of the clause.*

$$\frac{}{a : \sigma}\ \text{if } a : \sigma \in \Gamma \cup \mathcal{F} \qquad \frac{s : \sigma \qquad t : \tau}{(s,t) : \sigma \times \tau} \qquad \frac{s : \sigma \Rightarrow \tau \qquad t : \sigma}{s\ t : \tau}$$

$$\frac{s_1 : \mathtt{bool} \qquad s_2 : \sigma \qquad s_3 : \sigma}{\mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3 : \sigma} \qquad \frac{s_1 : \sigma \qquad \ldots \qquad s_n : \sigma}{\mathtt{choose}\ s_1 \cdots s_n : \sigma}$$

Fig. 4. Typing (for fixed $\mathcal{F}$ and Γ, see Definition 2)

Example 2. The program of Example 1 is typed using $\mathcal{F} = \{\mathtt{false} : \mathtt{bool}, \mathtt{true} : \mathtt{bool}, [] : \mathtt{list}, :: : \mathtt{bool} \Rightarrow \mathtt{list} \Rightarrow \mathtt{list}, \mathtt{succ} : \mathtt{list} \Rightarrow \mathtt{list}\}$. As all argument and output types have order 0, the variable restrictions are satisfied and all clauses can be typed using $\Gamma = \{xs : \mathtt{list}\}$, the program is well-typed.

Definition 3. *A program* p *is* well-formed *if it is well-typed, and moreover:*

- *data constructors are always fully applied: for all data constructors* $\mathtt{c} : \kappa_1 \Rightarrow \ldots \Rightarrow \kappa_m \Rightarrow \iota \in \mathcal{F}$ *such that* $\mathtt{c}\ t_1 \cdots t_n$ *occurs as a sub-expression of the left- or right-hand side of any clause:* $n = m$*;*
- *the number of arguments to a given defined symbol is fixed: if* $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ *and* $\mathtt{f}\ \ell'_1 \cdots \ell'_n = t$ *are both in* p*, then* $k = n$*; we let* $\mathtt{arity}_{\mathtt{p}}(\mathtt{f})$ *denote* k*.*

Example 3. The program of Example 1 is well-formed, and $\mathtt{arity}_{\mathtt{p}}(\mathtt{succ}) = 1$.

However, the program would not be well-formed if the clauses below were added, because here the defined symbol or does not have a consistent arity.

$$\mathtt{id}\ x = x \qquad \mathtt{or}\ \mathtt{true}\ x = \mathtt{true} \qquad \mathtt{or}\ \mathtt{false} = \mathtt{id}\ x$$

Remark 1. Note that data constructors must (a) have a sort as output type (*not* a pair), and (b) occur only fully applied. This is entirely consistent with typical functional programming languages, where sorts and constructors tend to be declared with a grammar like the following:

$$sdec \in \mathtt{SortDec} ::= \mathtt{data}\ \iota = cdec_1 \mid \cdots \mid cdec_n$$
$$cdec \in \mathtt{ConstructorDec} ::= \mathtt{c}\ \sigma_1\ \cdots\ \sigma_m$$

In addition, we require that the arguments to data constructors have order 0. This is not standard in functional programming, but is the case in [12]. We limit interest to such constructors because, practically, these are the only ones which can be used in a *cons-free* program (as we will define in Section 3). It is simpler to define the notion of data that we will actually use from the start, rather than defining a more general notion and limiting it later.

Definition 4. *A program has* data order K *if all clauses can be typed using type environments Γ such that, for all $x : \sigma \in \Gamma$: $o(\sigma) \leq K$.*

Example 4. We consider a higher-order program, operating on the same data constructors as Example 1; however, now we encode numbers using *functions*:

$$\begin{array}{l}
\texttt{fsucc}\ F\ [] = \texttt{if}\ F\ []\ \texttt{then set}\ F\ \texttt{false else set}\ F\ []\ \texttt{true} \\
\texttt{fsucc}\ F\ xs = \texttt{if}\ F\ xs\ \texttt{then fsucc}\ (\texttt{set}\ F\ xs\ \texttt{false})\ (\texttt{tl}\ xs) \\
\qquad\qquad \texttt{else set}\ F\ xs\ \texttt{true} \\
\texttt{set}\ F\ val\ xs\ ys = \texttt{if eqlen}\ xs\ ys\ \texttt{then}\ val\ \texttt{else}\ F\ ys \\
\texttt{tl}\ (x::xs) = xs \qquad\qquad \texttt{eqlen}\ (x::xs)\ (y::ys) = \texttt{eqlen}\ xs\ ys \\
\texttt{eqlen}\ []\ [] = \texttt{true} \qquad \texttt{eqlen}\ xs\ ys = \texttt{false}
\end{array}$$

There is only one possible typing, which has $\texttt{fsucc} : (\texttt{list} \Rightarrow \texttt{bool}) \Rightarrow \texttt{list} \Rightarrow \texttt{list} \Rightarrow \texttt{bool}$ and therefore equips the `fsucc` clauses with type environments Γ containing $F : \texttt{list} \Rightarrow \texttt{bool}$—which has type order 1. As other variables have a type of order 0, this program has data order 1.

To explain the program: we use boolean lists as *unary* numbers of a limited size; assuming that (a) F represents a bitstring of length $N + 1$, and (b) *lst* has length N, the successor of F (modulo wrapping) is obtained by $\texttt{fsucc}\ F\ lst$.

In a program with data order K, we can assume wlog. that the defined symbols are assigned a type of order $\leq K + 1$, since there are no clauses for symbols of a higher type order (patterns of a higher type must be variables or tuples thereof). Moreover, we may assume wlog. that the output type of clauses respects K. Formally:

Definition 5. *A program is* proper *for data order K if for all $\texttt{f} \in \mathcal{D}$ with $\texttt{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa \in \mathcal{F}$ and arity $k \leq m$ we have: (a) $o(\sigma_i) \leq K$ for $1 \leq i \leq m$, and (b) $o(\sigma_{k+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa) \leq K$.*

Example 5. The following program has data order 0, but is not proper for it:

$$\begin{array}{l}
\texttt{start}\ xs\ ys = \texttt{choose}\ (\texttt{fst}\ xs\ ys)\ (\texttt{snd}\ xs\ ys) \\
\texttt{const}\ x\ y = x \qquad \texttt{fst}\ x = \texttt{const}\ x \\
\texttt{id}\ x = x \qquad\qquad \texttt{snd}\ x = \texttt{id}
\end{array}$$

The problem is that `fst` and `snd` have output type $\texttt{nat} \Rightarrow \texttt{nat}$ of order 1, which is not used in other clauses: in the `start` clause, both are called with a second argument. To avoid conflicts with the notion of data order, we disallow such programs. The program above *is* proper for data order 1, and may be adapted to respect data order 0 by increasing the arity of the two offending functions; i.e., replacing them by $\texttt{fst}\ x\ y = \texttt{const}\ x\ y$ and $\texttt{snd}\ x\ y = \texttt{id}\ y$ respectively.

2.3 Semantics

Like Jones, our language has a closure-based call-by-value semantics. We let data expressions, values and environments be defined by the grammar in Fig. 5.

$$d, b \in \mathtt{Data} ::= c\ d_1 \cdots d_m \mid (d, b)$$
$$v, w \in \mathtt{Value} ::= d \mid (v, w) \mid f\ v_1 \cdots v_n \quad (n < \mathtt{arity_p}(f))$$
$$\gamma, \delta \in \mathtt{Env} ::= \mathcal{V} \to \mathtt{Value}$$

Instantiation:

$$x\gamma := \gamma(x)$$
$$(\mathtt{c}\ \ell_1 \cdots \ell_n)\gamma := \mathtt{c}\ (\ell_1\gamma) \cdots (\ell_n\gamma)$$

Fig. 5. Data expressions, values and environments

We let $\mathtt{dom}(\gamma)$ denote the domain of an environment (partial function) γ. Note that values are ground expressions, and we only use well-typed values with fully applied data constructors. To every pattern ℓ and environment γ with $\mathtt{dom}(\gamma) \supseteq Var(\ell)$, we associate a value $\ell\gamma$ by instantiation the obvious way, see Figure 5.

Note that, for every value v and pattern ℓ, there is at most one environment γ with $\ell\gamma = v$. We say that an expression $\mathtt{f}\ s_1 \cdots s_n$ *instantiates* the left-hand side of a clause $\mathtt{f}\ \ell_1 \cdots \ell_k$ if $n = k$ and there is an environment γ with each $s_i = \ell_i\gamma$.

Both input and output to the program are data expressions. If $\mathtt{f}_1$ has type $\kappa_1 \Rightarrow \ldots \Rightarrow \kappa_M \Rightarrow \kappa$, we can think of the program as calculating a function $[\![\mathtt{p}]\!](d_1, \ldots, d_M)$ from M input data arguments to one output data expression.

Expression and program evaluation are given by the rules in Figure 6. Note that, in the [Call] rule, there is at most one suitable γ on domain $Var(\mathtt{f}\ \ell_1 \cdots \ell_k)$; therefore, the only source of non-determinism in the language is the `choose` operator. Programs without this operator are called *deterministic*.

Example 6. For the program from Example 1, $[\![\mathtt{p}]\!]((\mathtt{true}::\mathtt{false}::\mathtt{true}::[])) \mapsto \mathtt{false}::\mathtt{true}::\mathtt{true}::[]$, giving that $5 + 1 = 6$ as expected. In the (fixed) program from Example 5, we can both derive $[\![\mathtt{p}]\!](\mathtt{false}, \mathtt{s}\ \mathtt{false}) \mapsto \mathtt{false}$ and $[\![\mathtt{p}]\!](\mathtt{false}, \mathtt{s}\ \mathtt{false}) \mapsto \mathtt{s}\ \mathtt{false}$.

The language is easily seen to be Turing-complete unless further restrictions are imposed. In order to assuage any fears the reader may harbour about whether the complexity-theoretic characterisations we obtain are due to brittle design choices, we here give a few brief remarks on the language design.

Remark 2. We have omitted some constructs common to even some toy pure functional languages, but these are in general simple syntactic sugar that can be readily expressed by the existing constructs in the language, even in the presence of non-determinism. For instance, a let-binding $\mathtt{let}\ x = s_1\ \mathtt{in}\ s_2$ can be straightforwardly encoded by a function call in a pure call-by-value setting (replacing $\mathtt{let}\ x = s_1\ \mathtt{in}\ s_2$ by $\mathtt{helper}\ s_1$ and adding a clause $\mathtt{helper}\ x = s_2$).

Remark 3. We do not require the clauses of a function definition to exhaust all possible patterns. For instance, it is possible to have a clause $\mathtt{f}\ \mathtt{true} = \cdots$ without a clause for $\mathtt{f}\ \mathtt{false}$. Thus, a program has zero or more values.

3 Cons-free programs

Jones defines a cons-free program as one where the list constructor :: does not occur in any clause. In our setting (where functions like `hd` and `tl` are not predefined since they can be handled using pattern matching, and where more

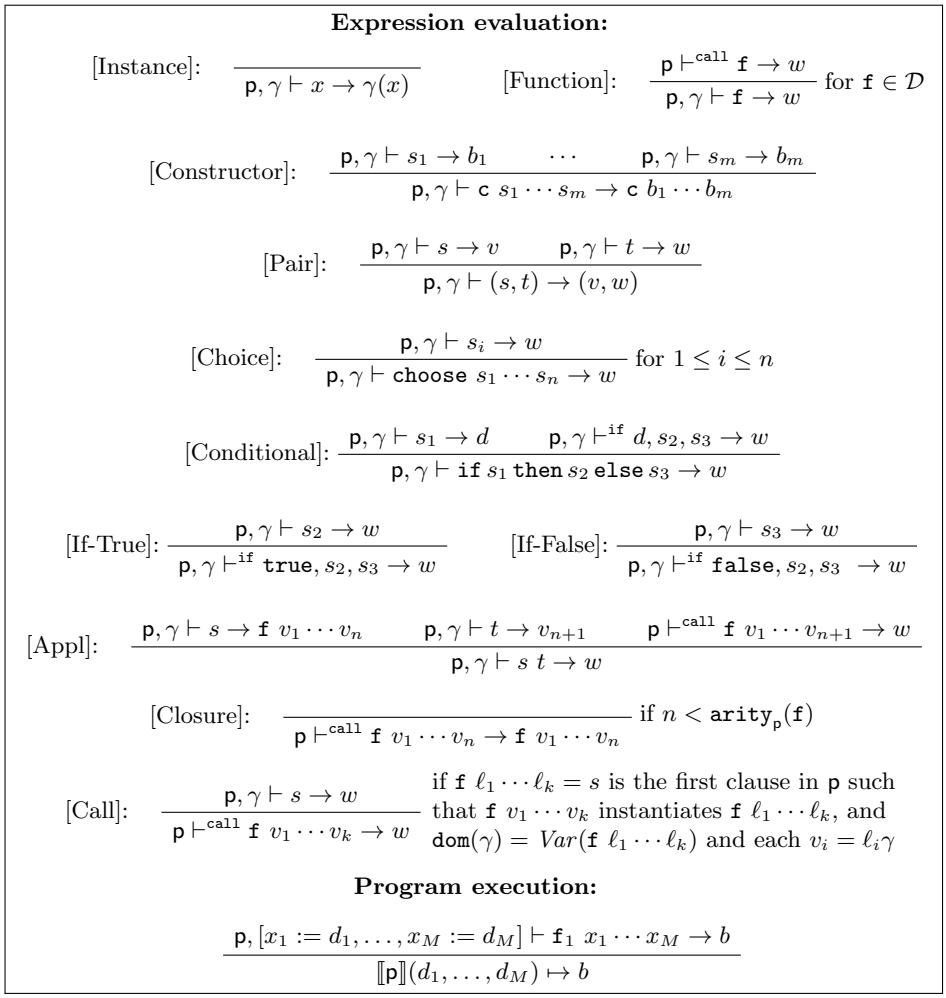

Fig. 6. Call-by-value semantics

constructors are in principle admitted), this translates to disallowing non-constant data constructors from occurring in the right-hand side of a clause. We define:

Definition 6. *A program* p *is cons-free, if all clauses in* p *are cons-free. A clause* $f_i\ \ell_1 \cdots \ell_n = s$ *is cons-free if for all* $s \trianglerighteq t$*: if* $t = \mathtt{c}\ s_1 \cdots s_m$ *with* $\mathtt{c} \in \mathcal{C}$*, then* t *is a data expression or* $\ell_i \trianglerighteq t$ *for some* i*.*

Example 7. Example 1 is not cons-free, due to the second and third clause. Examples 4 and 5 are both cons-free.

The key property of cons-free programming is that no *new* data structures can be created during program execution. Formally, this means that in a derivation tree in the operational semantics having root $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$, all data values (including b) are in the set $\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$, defined as follows:

Definition 7. *Let* $\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M} := \{d \mid \exists 1 \leq i \leq M[d_i \trianglerighteq d]\} \cup \{d \in \mathtt{Data} \mid \exists(\ell = s) \in \mathsf{p}[s \trianglerighteq d]\}$.

$\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$ is a set of data expressions, is closed under $\triangleright$ and, for p fixed, has a linear number of elements in the size of $d_1, \ldots, d_M$. The property that no new data structures can be created during execution is formally expressed by the following lemma.

Lemma 1. *Let* p *be a cons-free program, and suppose that* $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ *is obtained by a derivation tree* T*. Then for all statements* $\mathsf{p}, \gamma \vdash s \to w$ *or* $\mathsf{p}, \gamma \vdash^{\mathtt{if}} b', s_1, s_2 \to w$ *or* $\mathsf{p} \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$*, and all sub-expressions* d *such that (a)* $w \trianglerighteq d$*, (b)* $b' \trianglerighteq d$*, (c)* $\gamma(x) \trianglerighteq d$ *for some* x *or (d)* $v_i \trianglerighteq d$ *for some* i*: if* d *has the form* $\mathtt{c}\ b_1 \cdots b_m$ *with* $\mathtt{c} \in \mathcal{C}$*, then* $d \in \mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$.

That is, any data expression in the derivation tree of $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ (including occurrences as a sub-expression of other values) is also in $\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$.

Proof (Sketch). Induction on the form of T, using the assumption that for the root of T, (1) the requirements on γ and the v_i are satisfied, and (2) γ maps sub-expressions $t \trianglelefteq s, s_1, s_2$ to elements of $\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$ if t has the form $\mathtt{c}\ t_1 \cdots t_m$ with $\mathtt{c} \in \mathcal{C}$. (Full details are given in Appendix A.)

Note that Lemma 1 implies that the program result b is in $\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$. Recall also Remark 1: if we had admitted constructors with higher-order argument types, then Lemma 1 shows that they are never used, since any constructor appearing in a derivation for $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ must already occur in the (data!) input.

Example 8. By Lemma 1, functions in a cons-free programs cannot build inductive data structures. Therefore, it is often necessary to "code around" a problem, for instance by using sub-expressions of the input as a measure of size. Consider the task of determining whether a given string of booleans is a palindrome. We cannot use a clause such as $\mathtt{palindrome}\ cs = \mathtt{equal}\ cs\ (\mathtt{rev}\ cs)$, because, by Lemma 1, it is impossible to define $\mathtt{rev}\ cs$. Instead, in the solution below, $\mathtt{chk}\ b\ (c_0 :: \ldots :: c_{n-1})\ ys$ returns whether $c_{|ys|} = b$, where $|ys|$ is the length of ys.

$$
\begin{array}{ll}
\mathtt{palindrome}\ cs = \mathtt{pal}\ cs\ xs & \\
\mathtt{pal}\ cs\ (y :: ys) = \mathtt{if\ pal}\ cs\ ys\ \mathtt{then\ chk}\ y\ cs\ ys\ \mathtt{else\ false} & \\
\mathtt{pal}\ cs\ [] = \mathtt{true} & \mathtt{equal\ 0\ 0} = \mathtt{true} \\
\mathtt{chk}\ b\ (x :: xs)\ [] = \mathtt{equal}\ b\ x & \mathtt{equal\ 1\ 1} = \mathtt{true} \\
\mathtt{chk}\ b\ (x :: xs)\ (y :: ys) = \mathtt{chk}\ b\ xs\ ys & \mathtt{equal}\ x\ y = \mathtt{false}
\end{array}
$$

4 Turing Machines, decision problems and complexity

We assume familiarity with the standard notions of Turing Machines and complexity classes (see, e.g., [18,11,19]); in this section, we fix the notation we use.

4.1 (Deterministic) Turing Machines

Turing Machines (TMs) are triples (A, S, T) where A is a finite set of tape symbols such that $A \supseteq \{0, 1, \llcorner\}$, $S \supseteq \{\mathtt{start}, \mathtt{accept}, \mathtt{reject}\}$ is a finite set of states, and T is a finite set of transitions (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*). We sometimes denote this transition as $i \overset{r/w\ d}{\Longrightarrow} j$. A *deterministic* TM is a TM such that every pair (i, r) with $i \in S \setminus \{\mathtt{accept}, \mathtt{reject}\}$ and $r \in A$ is associated with exactly one transition (i, r, w, d, j). Every TM in this paper has a single, right-infinite tape.

A *valid tape* is an element t of $A^{\mathbb{N}}$ with $t(p) \neq \llcorner$ for only finitely many p. A *configuration* is a triple (t, p, s) with t a valid tape, $p \in \mathbb{N}$ and $s \in S$. The transitions T induce a relation $\Rightarrow$ between configurations in the obvious way.

4.2 Decision problems

A *decision problem* is a set $X \subseteq \{0, 1\}^+$.

A deterministic TM *decides* X if for any $x \in \{0, 1\}^+$ we have $x \in X$ iff $(\llcorner x_1 \dots x_n \llcorner\llcorner \dots, 0, \mathtt{start}) \Rightarrow^* (t, i, \mathtt{accept})$ for some t, i, and $(\llcorner x_1 \dots x_n \llcorner\llcorner \dots, 0, \mathtt{start}) \Rightarrow^* (t, i, \mathtt{reject})$ iff $x \notin X$. Thus, a deterministic machine which decides X halts on all inputs, ending in `accept` or `reject` depending on whether $x \in X$.

If $h : \mathbb{N} \longrightarrow \mathbb{N}$ is a function, a deterministic TM *runs in time* $\lambda n.h(n)$ if for all $n \in \mathbb{N} \setminus \{0\}$ and $x \in \{0, 1\}^n$: any evaluation starting in $(\llcorner x_1 \dots x_n \llcorner\llcorner \dots, 0, \mathtt{start})$ ends in the `accept` or `reject` state in at most $h(n)$ transitions.

4.3 Complexity and the EXPTIME hierarchy

We define classes of decision problem based on the *time* needed to accept them.

Definition 8. *Let $h : \mathbb{N} \to \mathbb{N}$ be a function. Then, TIME$(h(n))$ is the set of all $X \subseteq \{0, 1\}^+$ such that there exist $a > 0$ and a deterministic TM running in time $\lambda n.a \cdot h(n)$ that decides X.*

Observe that, by design, $\mathrm{TIME}(h(n)))$ is closed under $\mathcal{O}$, that is $\mathrm{TIME}(h(n)) = \mathrm{TIME}(\mathcal{O}(h(n)))$.

Definition 9. *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 $\mathsf{EXP}^K\mathsf{TIME} \triangleq \bigcup_{a,b \in \mathbb{N}} \mathit{TIME}\left(\mathit{exp}_2^K(an^b)\right)$.*

Observing that for every polynomial h, there are $a, b \in \mathbb{N}$ such that $h(n) \leq a \cdot n^b$ for all $n > 0$, we have that $\mathsf{EXP}^0\mathsf{TIME} = \mathsf{P}$ and $\mathsf{EXP}^1\mathsf{TIME} = \mathsf{EXP}$ (where EXP is the usual complexity class of this name, see e.g., [18, Ch. 20]). Note that in the literature, EXP is sometimes called $\mathsf{EXPTIME}$ or $\mathsf{DEXPTIME}$ (e.g., in the celebrated proof that ML typability is complete for $\mathsf{DEXPTIME}$ [13]).

Using the Time Hierarchy Theorem [19], it is easy to see that $\mathsf{P} = \mathsf{EXP}^0\mathsf{TIME} \subsetneq \mathsf{EXP}^1\mathsf{TIME} \subsetneq \mathsf{EXP}^2\mathsf{TIME} \subsetneq \cdots$.

Definition 10. *The set* $\mathsf{ELEMENTARY}$ *of elementary-time computable languages is* $\bigcup_{K \in \mathbb{N}} \mathsf{EXP}^K\mathsf{TIME}$.

4.4 Decision problems and programs

To solve decision problems by (cons-free) programs, we will consider programs with constructors $\mathtt{true}, \mathtt{false}$ of type $\mathtt{bool}$, $[]$ of type $\mathtt{list}$ and $::$ of type $\mathtt{bool} \Rightarrow \mathtt{list} \Rightarrow \mathtt{list}$, and whose main function $\mathtt{f}_1$ has type $\mathtt{list} \Rightarrow \mathtt{bool}$.

Definition 11. *We define:*

- *A program* $\mathtt{p}$ accepts $\mathtt{a}_1\mathtt{a}_2 \dots \mathtt{a}_n \in \{0,1\}^*$ *if* $[\![\mathtt{p}]\!](\overline{\mathtt{a}_1} :: \dots :: \overline{\mathtt{a}_n}) \mapsto \mathtt{true}$, *where* $\overline{\mathtt{a}_i} = \mathtt{true}$ *if* $\mathtt{a}_i = 1$ *and* $\overline{\mathtt{a}_i} = \mathtt{false}$ *otherwise.*
- *The* set accepted by *program* $\mathtt{p}$ *is* $\{\mathtt{a} \in \{0,1\}^* \mid \mathtt{p}$ *accepts* $\mathtt{a}\}$.

Although we focus on programs of this form, our proofs will allow for arbitrary input and output—with the limitation (as guaranteed by the rule for program execution) that both are data. This makes it possible to for instance consider decision problems on a larger input alphabet without needing encodings.

Example 9. Example 8 accepts the problem $\{x \in \{0,1\}^* \mid x \text{ is a palindrome}\}$.

5 Deterministic characterisations

As a basis, we transfer Jones' basic result on *time* classes to our more general language. That is, we obtain the first line of the first table in Figure 1.

	data order 0	**data order 1**	**data order 2**	**data order 3**	...
cons-free deterministic	$\mathsf{P} = \mathsf{EXP}^0\mathsf{TIME}$	$\mathsf{EXP} = \mathsf{EXP}^1\mathsf{TIME}$	$\mathsf{EXP}^2\mathsf{TIME}$	$\mathsf{EXP}^3\mathsf{TIME}$	...

To show that deterministic cons-free programs of data order K characterise $\mathsf{EXP}^K\mathsf{TIME}$ it is necessary to prove two things:

1. if $h(n) \leq \exp_2^K(a \cdot n^b)$ for all n, then for every deterministic Turing Machine M running in TIME $(h(n))$, there is a deterministic, cons-free program with data order at most K, which accepts some $x \in \{0,1\}^+$ if and only if M does;
2. for every deterministic cons-free program $\mathtt{p}$ with data order K, there is a deterministic algorithm operating in TIME $\left(\exp_2^K(a \cdot n^b)\right)$ for some a, b which, given input expressions $d_1, \dots, d_M$, determines b such that $[\![\mathtt{p}]\!](d_1, \dots, d_M) \mapsto b$ (if such b exists). As in Jones [12] we assume our algorithms to be implemented on a sufficiently expressive Turing-equivalent machine like the RAM. As any such machine can be simulated by, and can simulate, a Turing machine with polynomial-time overhead, the desired result follows.

We will show part (1) in Section 5.1, and part (2) in Section 5.2.

5.1 Simulating Turing Machines using deterministic cons-free programs

Let $M := (A, S, T)$ be a deterministic Turing Machine running in time $\lambda n.h(n)$. Like Jones, we start by assuming that we have a way to represent the numbers $0, \dots, h(n)$ as expressions, along with successor and predecessor operators and checks for equality. Our simulation uses the following data constructors:

- $\mathtt{true} : \mathtt{bool}, \mathtt{false} : \mathtt{bool}, [] : \mathtt{list}$ and $:: : \mathtt{bool} \Rightarrow \mathtt{list} \Rightarrow \mathtt{list}$ as discussed in Section 4.4;
- $\mathtt{a} : \mathtt{symbol}$ for $a \in A$ (writing $\mathtt{B}$ for the blank symbol), and $\mathtt{L}, \mathtt{R} : \mathtt{direc}$ and $\mathtt{s} : \mathtt{state}$ for $s \in S$;
- $\mathtt{action} : \mathtt{symbol} \Rightarrow \mathtt{direc} \Rightarrow \mathtt{state} \Rightarrow \mathtt{trans}$;
- $\mathtt{end} : \mathtt{state} \Rightarrow \mathtt{trans}$.

The rules to simulate the machine are given in Figure 7.

$$\mathtt{run}\ cs = \mathtt{test}\ (\mathtt{state}\ cs\ [h(|cs|)])$$

$$\begin{array}{ll} \mathtt{test\ accept} = \mathtt{true} & \mathtt{transition\ i\ r} = \mathtt{action\ w\ d\ j} \quad \text{for all } i \xrightarrow{r/w\ d} j \in T \\ \mathtt{test\ reject} = \mathtt{false} & \mathtt{transition\ i}\ x = \mathtt{end\ i} \quad \text{for } i \in \{\mathtt{accept}, \mathtt{reject}\} \end{array}$$

$$\begin{array}{l} \mathtt{state}\ cs\ [n] = \mathtt{if}\ [n = 0]\ \mathtt{then\ start\ else\ get3}\ (\mathtt{transat}\ cs\ [n-1]) \\ \mathtt{transat}\ cs\ [n] = \mathtt{transition}\ (\mathtt{state}\ cs\ [n])\ (\mathtt{tapesymb}\ cs\ [n]) \end{array}$$

$$\begin{array}{ll} \mathtt{get1}\ (\mathtt{action}\ x\ y\ z) = x & \mathtt{get1}\ (\mathtt{end}\ x) = \mathtt{B} \\ \mathtt{get2}\ (\mathtt{action}\ x\ y\ z) = y & \mathtt{get2}\ (\mathtt{end}\ x) = \mathtt{R} \\ \mathtt{get3}\ (\mathtt{action}\ x\ y\ z) = z & \mathtt{get3}\ (\mathtt{end}\ x) = x \end{array}$$

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

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

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

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

Fig. 7. Simulating a deterministic Turing Machine (A, S, T)

Types of defined symbols are easily derived. The intended meaning is that $\mathtt{state}\ cs\ [n]$, for cs the input list and $[n]$ a number in $\{0, \ldots, h(|cs|)\}$, returns the state of the machine at time $[n]$; $\mathtt{pos}\ cs\ [n]$ returns the position of the reader at time $[n]$, and $\mathtt{tape}\ cs[n]\ [p]$ the symbol at time $[n]$ and position $[p]$.

Clearly, the program is highly exponential, even when $h(|cs|)$ is polynomial, since the same expressions are repeatedly evaluated. This apparent contradiction is not problematic: we do not claim that all cons-free programs with data order 0 (say) have a derivation tree of at most polynomial size. Rather, as we will see in Section 5.2, we can find their *result* in polynomial time by essentially using a caching mechanism to avoid reevaluating the same expression.

What remains is to simulate numbers and counting. For a machine running in $\mathrm{TIME}\,(h(n))$, it suffices to find a value $[i]$ representing i for all $i \in \{0, \ldots, h(n)\}$ and cons-free clauses to calculate predecessor and successor functions and to perform zero and equality checks. This is given by a $(\lambda n.h(n) + 1)$-counting module:

Definition 12 (Adapted from [12]). *For $P : \mathbb{N} \to \mathbb{N} \setminus \{0\}$, a P-counting module is a tuple $C_\pi = (\alpha_\pi, \mathcal{D}_\pi, \mathcal{A}_\pi, \langle\cdot\rangle_\pi, \mathsf{p}_\pi)$ such that:*

- *α_π is a type (this will be the type of* numbers*);*
- *$\mathcal{D}_\pi$ is a set of defined symbols disjoint from $\mathcal{C}, \mathcal{D}, \mathcal{V}$, containing symbols $\mathtt{seed}_\pi$, $\mathtt{pred}_\pi$ and $\mathtt{zero}_\pi$, with types $\mathtt{seed}_\pi : \mathtt{list} \Rightarrow \alpha_\pi$, $\mathtt{pred}_\pi : \mathtt{list} \Rightarrow \alpha_\pi \Rightarrow \alpha_\pi$ and $\mathtt{zero}_\pi : \mathtt{list} \Rightarrow \alpha_\pi \Rightarrow \mathtt{bool}$;*
- *for $n \in \mathbb{N}$, $\mathcal{A}_\pi^n$ is a set of values of type α_π, all built over $\mathcal{C} \cup \mathcal{D}_\pi$ (this is the set of values used to represent numbers);*
- *for $n \in \mathbb{N}$, $\langle\cdot\rangle_\pi^n$ is a total function from $A_{\alpha_\pi}^n$ to $\mathbb{N}$;*
- *p_π is a list of cons-free clauses on the symbols in $\mathcal{D}_\pi$, such that, for all lists $cs : \mathtt{list} \in \mathtt{Data}$ with length n:*
  - *there is a unique value v such that $\mathsf{p}_\pi \vdash^{\mathtt{call}} \mathtt{seed}_\pi\ cs \to v$;*
  - *if $\mathsf{p}_\pi \vdash^{\mathtt{call}} \mathtt{seed}_\pi\ cs \to v$, then $v \in \mathcal{A}_\pi^n$ and $\langle v\rangle_\pi^n = P(n) - 1$;*
  - *if $v \in \mathcal{A}_\pi$ and $\langle v\rangle_\pi^n = i > 0$, then there is a unique value w such that $\mathsf{p}_\pi \vdash^{\mathtt{call}} \mathtt{pred}_\pi\ v \to w$; we have $w \in \mathcal{A}_\pi^n$ and $\langle w\rangle_\pi^n = i - 1$;*
  - *for $v \in \mathcal{A}_\pi^n$ with $\langle v\rangle_\pi^n = i$: $\mathsf{p}_\pi \vdash^{\mathtt{call}} \mathtt{zero}_\pi\ cs\ v \to \mathtt{true}$ if and only if $i = 0$, and $\mathsf{p}_\pi \vdash^{\mathtt{call}} \mathtt{zero}_\pi\ cs\ v \to \mathtt{false}$ if and only if $i > 0$.*

It is easy to see how a P-counting module can be plugged into the program of Figure 7. We only lack successor and equality functions, which are easily defined:

$$\begin{aligned}
&\mathtt{succ}_\pi\ cs\ i = \mathtt{sc}_\pi\ cs\ (\mathtt{seed}_\pi\ cs)\ i\\
&\mathtt{sc}_\pi\ cs\ j\ i = \mathtt{if}\ \mathtt{equal}_\pi\ cs\ (\mathtt{pred}_\pi\ cs\ j)\ i\ \mathtt{then}\ j\ \mathtt{else}\ \mathtt{sc}\ cs\ (\mathtt{pred}_\pi\ cs\ j)\ i\\
&\mathtt{equal}_\pi\ cs\ i\ j = \mathtt{if}\ \mathtt{zero}_\pi\ cs\ i\ \mathtt{then}\ \mathtt{zero}_\pi\ cs\ j\\
&\qquad\qquad \mathtt{else\ if}\ \mathtt{zero}_\pi\ cs\ j\ \mathtt{then\ false}\\
&\qquad\qquad \mathtt{else}\ \mathtt{equal}_\pi\ cs\ (\mathtt{pred}_\pi\ cs\ i)\ (\mathtt{pred}_\pi\ cs\ j)
\end{aligned}$$

Since the clauses in Figure 7 are cons-free and have data order 0, we obtain:

Lemma 2. *Let X be a decision problem which can be decided by a deterministic TM running in TIME$(h(n))$. If there is a cons-free $(\lambda n.h(n)+1)$-counting module C_π with data order K, then X is accepted by a cons-free program with data order K; the program is deterministic if the counting module is.*

Proof. By the argument given above.

The obvious difficulty is the restriction to cons-free clauses: we cannot simply construct a new number type, but will have to represent numbers using only sub-expressions of the input list cs, and constant data expressions.

Example 10. We consider a P-counting module $C_{\mathtt{x}}$ where $P(n) = 3 \cdot (n+1)^2$. Let $\alpha_{\mathtt{x}} = \mathtt{list} \times \mathtt{list} \times \mathtt{list}$ and for given n, let $\mathcal{A}_\pi^n = \{(d_0, d_1, d_2) \mid d_0$ is a list of length ≤ 2 and d_1, d_2 are lists of length $\leq n\}$. Writing $|\ x_1{::}\ldots{::}x_k{::}[]\ | = k$, we let $\langle(d_0, d_1, d_2)\rangle_{\mathtt{x}}^n := |d_0| \cdot (n+1)^2 + |d_1| \cdot (n+1) + |d_2|$. Essentially, we think of a number i as a 3-digit number $i_0 i_1 i_2$ in base $n+1$, with each i_j represented by a list. For $\mathsf{p}_{\mathtt{x}}$ we use:

$$\begin{aligned}
&\mathtt{seed}_{\mathtt{x}}\ cs = (\mathtt{false}{::}\mathtt{false}{::}[],\ cs,\ cs)\\
&\mathtt{pred}_{\mathtt{x}}\ cs\ (x_0, x_1, y{::}ys) = (x_0, x_1, ys) \qquad \mathtt{zero}_{\mathtt{x}}\ cs\ (x_0, x_1, y{::}ys) = \mathtt{false}\\
&\mathtt{pred}_{\mathtt{x}}\ cs\ (x_0, y{::}ys, []) = (x_0, ys, cs) \qquad \mathtt{zero}_{\mathtt{x}}\ cs\ (x_0, y{::}ys, []) = \mathtt{false}\\
&\mathtt{pred}_{\mathtt{x}}\ cs\ (y{::}ys, [], []) = (ys, cs, cs) \qquad \mathtt{zero}_{\mathtt{x}}\ cs\ (y{::}ys, [], []) = \mathtt{false}\\
&\mathtt{pred}_{\mathtt{x}}\ cs\ ([], [], []) = ([], [], []) \qquad \mathtt{zero}_{\mathtt{x}}\ cs\ ([], [], []) = \mathtt{true}
\end{aligned}$$

It is easy to see that the evaluation requirements are satisfied. For example, if $cs = \texttt{true}::\texttt{false}::\texttt{true}::[]$, one value in $\mathcal{A}^3_{\mathtt{x}}$ is $v = (\texttt{false}::[],\ \texttt{false}::\texttt{true}::[],\ [])$, which is mapped to the number $1 \cdot 4^2 + 2 \cdot 4 + 0 = 24$. Then $\mathtt{p_x} \vdash^{\mathtt{call}} \mathtt{pred_x}\ cs\ v \to w := (\texttt{false}::[],\ \texttt{true}::[],\ cs)$, which is mapped to $1 \cdot 4^2 + 1 \cdot 4 + 3 = 23$ as desired.

Example 10 suggests a systematic way to create polynomial counting modules.

Lemma 3. *For any $a, b \in \mathbb{N} \setminus \{0\}$, there is a $(\lambda n.a \cdot (n+1)^b)$-counting module $C_{\langle a,b\rangle}$ with data order 0.*

Proof (Sketch). A straightforward generalisation of Example 10.

By increasing type orders, we can obtain an exponential increase of magnitude.

Lemma 4. *If there is a P-counting module C_π of data order K, then there is a $(\lambda n.2^{P(n)})$-counting module $C_{\mathtt{e}[\pi]}$ of data order $K+1$.*

Proof (Sketch). Let $\alpha_{\mathtt{e}[\pi]} := \alpha_\pi \Rightarrow \texttt{bool}$; then $o(\alpha_{\mathtt{e}[\pi]}) \leq K+1$. A number i with bit representation $b_0 \dots b_{P(n)-1}$ (with b_0 the most significant digit) is represented by a value v such that, for w with $\langle w\rangle_\pi = i$: $\mathtt{p}_{\mathtt{e}[\pi]} \vdash^{\mathtt{call}} v\ w \to \texttt{true}$ iff $b_i = 1$, and $\mathtt{p}_{\mathtt{e}[\pi]} \vdash^{\mathtt{call}} v\ w \to \texttt{false}$ iff $b_i = 0$. We use the following clauses:

$$
\begin{aligned}
&\mathtt{seed}_{\mathtt{e}[\pi]}\ cs\ x = \texttt{true}\\
&\mathtt{zero}_{\mathtt{e}[\pi]}\ cs\ F = \mathtt{zhelp}_{\mathtt{e}[\pi]}\ cs\ F\ (\mathtt{seed}_\pi\ cs)\\
&\mathtt{zhelp}_{\mathtt{e}[\pi]}\ cs\ F\ k = \texttt{if}\ F\ k\ \texttt{then false}\\
&\qquad\qquad\qquad\qquad \texttt{else if}\ \mathtt{zero}_\pi\ cs\ k\ \texttt{then true}\\
&\qquad\qquad\qquad\qquad \texttt{else}\ \mathtt{zhelp}_{\mathtt{e}[\pi]}\ cs\ F\ (\mathtt{pred}_\pi\ cs\ k)\\
&\mathtt{pred}_{\mathtt{e}[\pi]}\ cs\ F = \mathtt{phelp}_{\mathtt{e}[\pi]}\ cs\ F\ (\mathtt{seed}_\pi\ cs)\\
&\mathtt{phelp}_{\mathtt{e}[\pi]}\ cs\ F\ k = \texttt{if}\ F\ k\ \texttt{then}\ \mathtt{flip}_{\mathtt{e}[\pi]}\ cs\ F\ k\\
&\qquad\qquad\qquad\qquad \texttt{else if}\ \mathtt{zero}_\pi\ cs\ k\ \texttt{then}\ \mathtt{seed}_{\mathtt{e}[\pi]}\ cs\\
&\qquad\qquad\qquad\qquad \texttt{else}\ \mathtt{phelp}_{\mathtt{e}[\pi]}\ cs\ (\mathtt{flip}_{\mathtt{e}[\pi]}\ cs\ F\ k)\ (\mathtt{pred}_\pi\ cs\ k)\\
&\mathtt{flip}_{\mathtt{e}[\pi]}\ cs\ F\ k\ i = \texttt{if}\ \mathtt{equal}_\pi\ cs\ k\ i\ \texttt{then not}\ (F\ i)\ \texttt{else}\ F\ i\\
&\texttt{not}\ b = \texttt{if}\ b\ \texttt{then false else true}
\end{aligned}
$$

We also include all clauses in $\mathtt{p}_\pi$. Here, note that a bitstring $b_0 \dots b_m$ represents 0 if each $b_i = 0$, and that the predecessor of $b_0 \dots b_i 10 \dots 0$ is $b_0 \dots b_i 01 \dots 1$.

Combining these results, we obtain:

Lemma 5. *Every decision problem in $\mathsf{EXP}^K\mathsf{TIME}$ is accepted by a deterministic cons-free program with data order K.*

Proof. A decision problem is in $\mathsf{EXP}^K\mathsf{TIME}$ if it is decided by a deterministic TM operating in time $\exp_2^K(a \cdot n^b))$ for some a, b. By Lemma 2, it therefore suffices if there is a Q-counting module for some $Q \geq \lambda n.\exp_2^K(a \cdot n^b) + 1$, with data order K. Certainly $Q(n) := \exp_2^K(a \cdot (n+1)^b)$ is large enough. By Lemma 3, there is a $(\lambda n.a \cdot (n+1)^b)$-counting module $C_{\langle a,b\rangle}$ with data order 0. Applying Lemma 4 K times, we obtain the required Q-counting module $C_{\mathtt{e}[\dots[\mathtt{e}[\langle a,b\rangle]]]}$.

Remark 4. Our definition of a counting module significantly differs from the one in [12]. The changes serve to allow for an easy formulation of the *non-deterministic* counting module in Section 6.

5.2 Simulating deterministic cons-free programs using an algorithm

We now turn to the second part of characterisation: that every decision problem solved by a deterministic cons-free program of data order K is in $\mathsf{EXP}^K\mathsf{TIME}$. We give an algorithm which determines the result of a fixed program (if any) on a given input in $\mathrm{TIME}\left(\exp_2^K(a \cdot n^b)\right)$ for some a, b. The algorithm is designed to extend easily to the non-deterministic characterisations in subsequent settings.

Key idea. The principle of our algorithm is easy to explain in the setting without higher types, so where all variables have data order 0. Using Lemma 1, all such variables must be instantiated by (tuples of) elements of $\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$, of which there are only polynomially many in the size of the input. Thus, we can make a comprehensive list of all expressions that might occur as the left-hand side of a [Call] in the derivation tree. Now we can go over the list repeatedly, filling in reductions to essentially trace a top-down derivation of the tree.

In the higher-order setting, there are infinitely many possible values. Therefore, instead of looking directly at values we consider an extensional replacement: partial functions with the same effect on the elements of $\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$.

Definition 13. *Let $\mathcal{B}$ be a set of data expressions closed under $\rhd$. For $\iota \in \mathcal{S}$, let $(\!|\iota|\!)_{\mathcal{B}} = \{d \in \mathcal{B} \mid \vdash d : \iota\}$. Inductively, define $(\!|\sigma \times \tau|\!)_{\mathcal{B}} = (\!|\sigma|\!)_{\mathcal{B}} \times (\!|\tau|\!)_{\mathcal{B}}$ and $(\!|\sigma \Rightarrow \tau|\!)_{\mathcal{B}} = \{A_{\sigma\Rightarrow\tau} \mid A \subseteq (\!|\sigma|\!)_{\mathcal{B}} \times (\!|\tau|\!)_{\mathcal{B}} \wedge \forall e \in (\!|\sigma|\!)_{\mathcal{B}}$ there is at most one u such that $(e,u) \in A_{\sigma\Rightarrow\tau}\}_{\sigma\Rightarrow\tau}$. We call the elements of any $(\!|\sigma|\!)_{\mathcal{B}}$* extensional values*.*

Note that extensional values are data expressions in $\mathcal{B}$ if σ is a sort, *pairs* if σ is a pair type, and sets of pairs (marked with a type) otherwise; these sets are exactly partial functions, and can be used as such:

Definition 14. *For $e \in (\!|\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau|\!)_{\mathcal{B}}$ and $u_1 \in (\!|\sigma_1|\!)_{\mathcal{B}}, \ldots, u_n \in (\!|\sigma_n|\!)_{\mathcal{B}}$, we inductively define $e(u_1,\ldots,u_n) \subseteq (\!|\tau|\!)_{\mathcal{B}}$:*

- *if $n = 0$, then $e(u_1,\ldots,u_n) = e() = \{e\}$;*
- *if $n \geq 1$, then $e(u_1,\ldots,u_n) = \bigcup_{A_{\sigma_n\Rightarrow\tau} \in e(u_1,\ldots,u_{n-1})} \{o \in (\!|\tau|\!)_{\mathcal{B}} \mid (u_n, o) \in A\}$.*

By induction on n, each $e(u_1,\ldots,u_n)$ has at most one element as would be expected of a partial function. We also consider a form of matching.

Definition 15. *Fix a set $\mathcal{B}$ of data expressions. An* extensional expression *has the form $\mathtt{f}\ e_1 \cdots e_n$ where $\mathtt{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau \in \mathcal{D}$ and each $e_i \in (\!|\sigma_i|\!)_{\mathcal{B}}$. Given a clause ρ: $\mathtt{f}\ \ell_1 \cdots \ell_k = r$ with $\mathtt{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_k \Rightarrow \tau \in \mathcal{F}$ and variable environment Γ, an* ext-environment *for ρ is a partial function η mapping each $x : \tau \in \Gamma$ to an element of $(\!|\tau|\!)_{\mathcal{B}}$, such that for $1 \leq j \leq n$: $\ell_j\eta \in (\!|\sigma_j|\!)_{\mathcal{B}}$. Here,*

- *$\ell\eta = \eta(\ell)$ if ℓ is a variable;*
- *$\ell\eta = (\ell^{(1)}\eta, \ell^{(2)}\eta)$ if $\ell = (\ell^{(1)}, \ell^{(2)})$;*
- *$\ell\eta = \ell[x := \eta(x) \mid x \in Var(\ell)]$ otherwise (in this case, ℓ is a pattern with data order 0, so all its variables have data order 0, so each $\eta(x) \in \mathtt{Data}$).*

Then $\ell\eta$ is an extensional value for ℓ a pattern. We say ρ matches *an extensional expression $\mathtt{f}\ e_1 \cdots e_k$ with each $e_i \in (\!|\sigma_i|\!)_{\mathcal{B}}$ if there is an ext-environment η for ρ such that for all $1 \leq i \leq k$: $\ell_i\eta = e_i$. We call η the* matching ext-environment*.*

Finally, for technical reasons we will need an ordering on extensional values:

Definition 16. *We define a relation $\sqsupseteq$ on extensional values of the same type:*

- *For $d, b \in \langle\!|\iota|\!\rangle_{\mathcal{B}}$ with $\iota \in \mathcal{S}$: $d \sqsupseteq b$ iff $d = b$.*
- *For $(e_1, e_2), (u_1, u_2) \in \langle\!|\sigma \times \tau|\!\rangle_{\mathcal{B}}$: $(e_1, e_2) \sqsupseteq (u_1, u_2)$ if each $e_i \sqsupseteq u_i$.*
- *For $A_\sigma, B_\sigma \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$ with σ functional: $A_\sigma \sqsupseteq B_\sigma$ if for all $(e, u) \in B$ there is $u' \sqsupseteq u$ such that $(e, u') \in A$.*

The algorithm. With these preparations, we are now ready to define our algorithm.

Algorithm 6 Let $\mathtt{p}$ be a fixed, deterministic cons-free program which is proper for some data order K, and suppose $\mathtt{f}_1$ has a type $\kappa_1 \Rightarrow \ldots \Rightarrow \kappa_M \Rightarrow \kappa \in \mathcal{F}$.

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

Output: The set of values b with $[\![\mathtt{p}]\!](d_1, \ldots, d_M) \mapsto b$.

1. Preparation.
   (a) Let $\mathtt{p}'$ by obtained from $\mathtt{p}$ as follows:
   - replace any sub-expression $(\mathtt{if}\, b\, \mathtt{then}\, s_1\, \mathtt{else}\, s_2)\ t_1 \cdots t_n$ in the right-hand side of a clause in $\mathtt{p}$ by $\mathtt{if}\, b\, \mathtt{then}\, (s_1\ t_1 \cdots t_n)\, \mathtt{else}\, (s_2\ t_1 \cdots t_n)$;
   - replace any sub-expression $(\mathtt{choose}\ s_1 \cdots s_m)\ t_1 \cdots t_n$ in the right-hand side of a clause in $\mathtt{p}$ by $\mathtt{choose}\ (s_1\ t_1 \cdots t_n) \cdots (s_m\ t_1 \cdots t_m)$;
   - add a symbol $\mathtt{start}$ to $\mathcal{D}$ and a clause $\mathtt{start}\ x_1 \cdots x_M = \mathtt{f}_1\ x_1 \cdots x_M$ at the head of $\mathtt{p}'$ (so that $[\![\mathtt{p}']\!](d_1, \ldots, d_M) \mapsto b$ iff $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \to b$).

   (b) Let $\mathcal{B} := \mathcal{B}^{\mathtt{p}}_{d_1,\ldots,d_M}$ and let $\mathcal{X}$ be the set of all "statements":
   - $\vdash \mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$ for (a) $\mathtt{f} \in \mathcal{D}$ with $\mathtt{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa' \in \mathcal{F}$, (b) $0 \le n \le \mathtt{arity}_{\mathtt{p}}(\mathtt{f})$ such that $o(\sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa') \le K$, (c) $e_i \in \langle\!|\sigma_i|\!\rangle_{\mathcal{B}}$ for $1 \le i \le n$ and (d) $o \in \langle\!|\sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa'|\!\rangle_{\mathcal{B}}$;
   - $\eta \vdash t \rightsquigarrow o$ for (a) ρ: $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ a clause in $\mathtt{p}'$, (b) $s \trianglerighteq t : \tau$, (c) $o \in \langle\!|\tau|\!\rangle_{\mathcal{B}}$ and (d) η an ext-environment for ρ.

   (c) Mark statements of the form $\eta \vdash t \rightsquigarrow o$ in $\mathcal{X}$ as confirmed if:
   i. $t \in \mathcal{V}$ and $\eta(t) \sqsupseteq o$, *or*
   ii. $t = \mathtt{c}\ t_1 \cdots t_m$ with $\mathtt{c} \in \mathcal{C}$ and $t\eta = o$.

   All statements not of either form are marked unconfirmed.
2. Iteration: repeat the following steps, until we complete an iteration where no changes are made.
   (a) For all unconfirmed statements $\vdash \mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$ in $\mathcal{X}$ with $n < \mathtt{arity}_{\mathtt{p}}(\mathtt{f})$: write $o = O_\sigma$ and mark the statement as confirmed if for all $(e_{n+1}, u) \in o$ there exists $u' \sqsupseteq u$ such that $\vdash \mathtt{f}\ e_1 \cdots e_{n+1} \rightsquigarrow u'$ is marked confirmed.
   (b) For all unconfirmed statements $\vdash \mathtt{f}\ e_1 \cdots e_k \rightsquigarrow o$ in $\mathcal{X}$ with $k = \mathtt{arity}_{\mathtt{p}}(\mathtt{f})$:
   i. find the first clause ρ: $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ in $\mathtt{p}'$ that matches $\mathtt{f}\ e_1 \cdots e_k$ and let η be the matching ext-environment (if no such clause exists, continue with the next statement);
   ii. determine whether $\eta \vdash s \rightsquigarrow o$ is confirmed and if so, mark the statement $\mathtt{f}\ e_1 \cdots e_k \rightsquigarrow o$ as confirmed.

(c) For all unconfirmed statements of the form $\eta \vdash \texttt{if}\ s_1\ \texttt{then}\ s_2\ \texttt{else}\ s_3 \rightsquigarrow o$ in $\mathcal{X}$, mark the statement as confirmed if:
   i. both $\eta \vdash s_1 \rightsquigarrow \texttt{true}$ and $\eta \vdash s_2 \rightsquigarrow o$ are marked confirmed, *or*
   ii. both $\eta \vdash s_1 \rightsquigarrow \texttt{false}$ and $\eta \vdash s_3 \rightsquigarrow o$ are marked confirmed.

(d) For all unconfirmed statements $\eta \vdash \texttt{choose}\ s_1 \cdots s_n \rightsquigarrow o$ in $\mathcal{X}$, mark the statement as confirmed if $\eta \vdash s_i \rightsquigarrow o$ for any $i \in \{1, \ldots, n\}$.

(e) For all unconfirmed statements $\eta \vdash (s_1, s_2) \rightsquigarrow (o_1, o_2)$ in $\mathcal{X}$, mark the statement confirmed if both $\eta \vdash s_1 \rightsquigarrow o_1$ and $\eta \vdash s_2 \rightsquigarrow o_2$ are confirmed.

(f) For all unconfirmed statements $\eta \vdash x\ s_1 \cdots s_n \rightsquigarrow o$ in $\mathcal{X}$ with $x \in \mathcal{V}$, mark the statement as confirmed if there are $e_1 \in \langle\!|\sigma_1|\!\rangle_{\mathcal{B}}, \ldots, e_n \in \langle\!|\sigma_n|\!\rangle_{\mathcal{B}}$ such that each $\eta \vdash s_i \rightsquigarrow e_i$ is marked confirmed, and there exists $o' \in \eta(a)(e_1, \ldots, e_n)$ such that $o' \sqsupseteq o$.

(g) For all unconfirmed statements $\eta \vdash \texttt{f}\ s_1 \cdots s_n \rightsquigarrow o$ in $\mathcal{X}$ with $\texttt{f} \in \mathcal{D}$, mark the statement as confirmed if there are $e_1 \in \langle\!|\sigma_1|\!\rangle_{\mathcal{B}}, \ldots, e_n \in \langle\!|\sigma_n|\!\rangle_{\mathcal{B}}$ such that each $\eta \vdash s_i \rightsquigarrow e_i$ is marked confirmed, and:
   i. $n \leq \texttt{arity}_{\texttt{p}}(a)$ and $\vdash \texttt{f}\ e_1 \cdots e_n \rightsquigarrow o$ is marked confirmed, *or*
   ii. $n > k := \texttt{arity}_{\texttt{p}}(a)$ and there are u, o' such that $\vdash \texttt{f}\ e_1 \cdots e_k \rightsquigarrow u$ is marked confirmed and $u(e_{k+1}, \ldots, e_n) \ni o' \sqsupseteq o$.

3. Completion: return $\{b \mid b \in \mathcal{B}\wedge \vdash \texttt{start}\ d_1 \cdots d_M \rightsquigarrow b \text{ is marked confirmed}\}$.

Note that, for programs of data order 0, this algorithm closely follows the sketch presented at the start of the subsection. Values of a higher type are abstracted to extensional values. The use of $\sqsupseteq$ is needed because a value of higher type is associated to many different extensional values: for example, to confirm a statement $\vdash \texttt{plus}\ \texttt{3} \rightsquigarrow \{(\texttt{1}, \texttt{4}), (\texttt{0}, \texttt{3})\}_{\texttt{nat}\Rightarrow\texttt{nat}}$ in some program, it may be necessary to first confirm $\vdash \texttt{plus}\ \texttt{3} \rightsquigarrow \{(\texttt{0}, \texttt{3})\}_{\texttt{nat}\Rightarrow\texttt{nat}}$.

Remark 5. Case 2d is not relevant in the current deterministic setting, since a deterministic algorithm is defined as one without `choose`. The case is included to enable reuse of the algorithm in the non-deterministic settings to follow.

The complexity of the algorithm relies on the following key observation:

Lemma 7. *Let* $\texttt{p}$ *be a cons-free program, proper for data order* K*. Let* Σ *be the set of all types* σ *with* $o(\sigma) \leq K$ *which occur as part of an argument type, or as an output type of some* $\texttt{f} \in \mathcal{D}$*.*

Suppose that, given input of total size n*,* $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$ *has cardinality at most* $F(n)$ *for all* $\sigma \in \Sigma$*, and testing whether* $e_1 \sqsupseteq e_2$ *for* $e_1, e_2 \in [\![\sigma]\!]_{\mathcal{B}}$ *takes at most* $F(n)$ *steps. Then Algorithm 6 runs in TIME* $\big(a \cdot F(n)^b\big)$ *for some* a, b*.*

Proof (Sketch). Due to the use of $\texttt{p}'$, all intensional values occurring in Algorithm 6 are in $\bigcup_{\sigma\in\Sigma}\langle\!|\sigma|\!\rangle_{\mathcal{B}}$. Writing $\texttt{a}$ for the greatest number of arguments any defined symbol $\texttt{f}$ or variable x in $\texttt{p}'$ may take and $\texttt{r}$ for the greatest number of subexpressions of any right-hand side in $\texttt{p}'$ (which is independent of the input!), $\mathcal{X}$ contains at most $\texttt{a} \cdot |\mathcal{D}| \cdot F(n)^{\texttt{a}+1} + |\texttt{p}'| \cdot \texttt{r} \cdot F(n)^{\texttt{a}+1}$ statements. Since in all but the last step of the iteration at least one statement is flipped from unconfirmed to confirmed, there are at most $|\mathcal{X}| + 1$ iterations, each considering $|\mathcal{X}|$ statements. It is easy to see that the individual steps in both the preparation and iteration are all polynomial in $|\mathcal{X}|$ and $F(n)$, resulting in a polynomial overall complexity.

The result follows as $\mathsf{Card}(\langle\!|\sigma|\!\rangle_{\mathcal{B}})$ is given by a tower of exponentials in $o(\sigma)$:

Lemma 8. *If* $1 \leq \mathsf{Card}(\mathcal{B}) < N$*, then for each* σ *on* L *sorts, with* $o(\sigma) \leq K$*:* $\mathsf{Card}(\langle\!|\sigma|\!\rangle_{\mathcal{B}}) < \exp_2^K(N^L)$*, where* $\mathsf{Card}(X)$ *denotes the cardinality of* X*. Testing* $e \sqsupseteq u$ *for* $e, u \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$ *takes at most* $\exp_2^K(N^{(L+1)^3})$ *comparisons between* $b, d \in \mathcal{B}$*.*

Proof (Sketch). An easy induction on the form of σ, using that $\exp_2^K(X) \cdot \exp_2^K(Y) \leq \exp_2^K(X \cdot Y)$ for $X \geq 2$, and that for $A_{\sigma_1 \Rightarrow \sigma_2}$, each key $e \in \langle\!|\sigma_1|\!\rangle_{\mathcal{B}}$ is assigned one of $\mathsf{Card}(\langle\!|\sigma_2|\!\rangle_{\mathcal{B}}) + 1$ choices: an element u of $\langle\!|\sigma_2|\!\rangle_{\mathcal{B}}$ such that $(e, u) \in A$, or non-membership; the second part uses the first.

We will postpone showing correctness of the algorithm until Section 6.3, where we can show the result together with the one for non-deterministic programs. Assuming correctness for now, we may conclude:

Lemma 9. *Every decision problem accepted by a deterministic cons-free program* p *with data order* K *is in* $\mathsf{EXP}^K\mathsf{TIME}$*.*

Proof. We will see in Lemma 19 in Section 6.3 that $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ if and only if Algorithm 6 returns a set containing b. For a program of data order K, Lemmas 8 and 7 together give that Algorithm 6 is in $\mathsf{EXP}^K\mathsf{TIME}$.

Theorem 1. *The class of deterministic cons-free programs with data order* K *characterises* $\mathsf{EXP}^K\mathsf{TIME}$ *for all* $K \in \mathbb{N}$*.*

Proof. A combination of Lemmas 5 and 9.

6 Non-deterministic characterisations

A natural question is what the result would be if we do not impose the limitation to deterministic programs. For data order 0, Bonfante [3] shows that adding the choice operator to Jones' language does not increase expressivity. We will recover this result for our generalised language in Section 7.

However, in the higher-order setting, non-deterministic choice *does* increase expressivity—dramatically so. Indeed, we obtain:

	data order 0	**data order 1**	**data order 2**	**data order 3**	...
cons-free	P	ELEMENTARY	ELEMENTARY	ELEMENTARY	...

As before, we will show the result—for data orders 1 and above—in two parts: in Section 6.1 we see that cons-free programs of data order 1 suffice to accept all problems in ELEMENTARY; in Section 6.2 we see that they cannot go beyond.

6.1 Simulating Turing Machines using (non-deterministic) cons-free programs

We start by showing how Turing Machines in ELEMENTARY can be simulated by non-deterministic cons-free programs. For this, we reuse the core simulation from Figure 7. The reason for the jump in expressivity lies in Lemma 2: by taking advantage of non-determinism, we can count up to arbitrarily high numbers.

Lemma 10. *If there is a P-counting module C_π with data order $K \leq 1$, there is a (non-deterministic) $(\lambda n.2^{P(n)-1})$-counting module $C_{\psi[\pi]}$ with data order 1.*

Proof. We let:

- $\alpha_{\psi[\pi]} := \mathtt{bool} \Rightarrow \alpha_\pi$, which has type order $\max(1, o(\alpha_\pi))$;
- $\mathcal{A}^n_{\psi[\pi]} :=$ the set of those values $v : \alpha_{\psi[\pi]}$ such that:
  - there is $w \in \mathcal{A}_\pi$ with $\langle w \rangle^n_\pi = 0$ such that $\mathtt{p}_{\psi[\pi]} \vdash^{\mathtt{call}} v\ \mathtt{true} \to w$;
  - there is $w \in \mathcal{A}_\pi$ with $\langle w \rangle^n_\pi = 0$ such that $\mathtt{p}_{\psi[\pi]} \vdash^{\mathtt{call}} v\ \mathtt{false} \to w$;

  and for all $1 \leq i < P(n)$ exactly one of the following holds:
  - there is $w \in \mathcal{A}^n_\pi$ with $\langle w \rangle^n_\pi = i$ such that $\mathtt{p}_{\psi[\pi]} \vdash^{\mathtt{call}} v\ \mathtt{true} \to w$;
  - there is $w \in \mathcal{A}^n_\pi$ with $\langle w \rangle^n_\pi = i$ such that $\mathtt{p}_{\psi[\pi]} \vdash^{\mathtt{call}} v\ \mathtt{false} \to w$;

  We will say that $v\ \mathtt{true} \mapsto i$ or $v\ \mathtt{false} \mapsto i$ respectively.
- $\langle v \rangle^n_{\psi[\pi]} := \sum_{i=1}^{P(n)-1} \{2^{P(n)-1-i} \mid v\ \mathtt{true} \mapsto i\}$;
- $\mathtt{p}_{\psi[\pi]}$ be given by the clauses in Figure 8 appended to $\mathtt{p}_\pi$, and $\mathcal{D}_{\psi[\pi]}$ be the defined symbols occurring in it.

So, we interpret a value v as the number given by the bitstring $b_1 \dots b_{P(n)-1}$ with the most significant digit first, where b_i is 1 if $v\ \mathtt{true}$ has a value representing i in C_π, and b_i is 0 otherwise—that is, exactly if $v\ \mathtt{false}$ has such a value.

– core elements; $\mathtt{st}i\ n\ F$ sets bit n in F to the value i

$\mathtt{base}_{\psi[\pi]}\ x\ b = x$

$\mathtt{st1}_{\psi[\pi]}\ n\ F\ \mathtt{true} = \mathtt{choose}\ n\ (F\ \mathtt{true})$ $\quad$ $\mathtt{st0}_{\psi[\pi]}\ n\ F\ \mathtt{true} = F\ \mathtt{true}$

$\mathtt{st1}_{\psi[\pi]}\ n\ F\ \mathtt{false} = F\ \mathtt{false}$ $\quad$ $\mathtt{st0}_{\psi[\pi]}\ n\ F\ \mathtt{false} = \mathtt{choose}\ n\ (F\ \mathtt{false})$

– testing bit values (using non-determinism and non-termination)

$\mathtt{bitset}_{\psi[\pi]}\ cs\ F\ i = \mathtt{if}\ \mathtt{equal}_\pi\ cs\ (F\ \mathtt{true})\ i\ \mathtt{then}\ \mathtt{true}$
$\quad \mathtt{else}\ \mathtt{if}\ \mathtt{equal}_\pi\ cs\ (F\ \mathtt{false})\ i\ \mathtt{then}\ \mathtt{false}$
$\quad \mathtt{else}\ \mathtt{bitset}_{\psi[\pi]}\ cs\ F\ i$

– the seed function

$\mathtt{nul}_\pi\ cs = \mathtt{nul}'_\pi\ cs\ (\mathtt{seed}_\pi\ cs)$

$\mathtt{nul}'_\pi\ cs\ n = \mathtt{if}\ \mathtt{zero}_\pi\ cs\ n\ \mathtt{then}\ n\ \mathtt{else}\ \mathtt{nul}'_\pi\ cs\ (\mathtt{pred}_\pi\ cs\ n)$

$\mathtt{seed}_{\psi[\pi]}\ cs = \mathtt{seed}'_{\psi[\pi]}\ cs\ (\mathtt{seed}_\pi\ cs)\ (\mathtt{base}_{\psi[\pi]}\ (\mathtt{nul}_\pi\ cs))$

$\mathtt{seed}'_{\psi[\pi]}\ cs\ i\ F = \mathtt{if}\ \mathtt{zero}_\pi\ cs\ i\ \mathtt{then}\ F\ \mathtt{else}\ \mathtt{seed}'_{\psi[\pi]}\ cs\ (\mathtt{pred}_\pi\ cs\ i)\ (\mathtt{st1}_{\psi[\pi]}\ i\ F)$

– the zero test

$\mathtt{zero}_{\psi[\pi]}\ cs\ F = \mathtt{zero}'_{\psi[\pi]}\ cs\ F\ (\mathtt{seed}_\pi\ cs)$

$\mathtt{zero}'_{\psi[\pi]}\ cs\ F\ i = \mathtt{if}\ \mathtt{zero}_\pi\ i\ \mathtt{then}\ \mathtt{true}$
$\quad \mathtt{else}\ \mathtt{if}\ \mathtt{bitset}_{\psi[\pi]}\ cs\ F\ i\ \mathtt{then}\ \mathtt{false}$
$\quad \mathtt{else}\ \mathtt{zero}'_{\psi[\pi]}\ cs\ F\ (\mathtt{pred}_\pi\ cs\ i)$

– the predecessor

$\mathtt{pred}_{\psi[\pi]}\ cs\ F = \mathtt{pr}_{\psi[\pi]}\ cs\ F\ (\mathtt{seed}_\pi\ cs)\ (\mathtt{base}_{\psi[\pi]}\ (\mathtt{nul}_\pi\ cs))$

$\mathtt{pr}_{\psi[\pi]}\ cs\ F\ i\ G = \mathtt{if}\ \mathtt{bitset}_{\psi[\pi]}\ cs\ F\ i\ \mathtt{then}\ \mathtt{cp}_{\psi[\pi]}\ cs\ F\ (\mathtt{pred}_\pi\ cs\ i)\ (\mathtt{st0}_{\psi[\pi]}\ i\ G)$
$\quad \mathtt{else}\ \mathtt{pr}_{\psi[\pi]}\ cs\ F\ (\mathtt{pred}_\pi\ cs\ i)\ (\mathtt{st1}_{\psi[\pi]}\ i\ G)$

$\mathtt{cp}\ cs\ F\ i\ G = \mathtt{if}\ \mathtt{zero}_\pi\ cs\ i\ \mathtt{then}\ G$
$\quad \mathtt{else}\ \mathtt{if}\ \mathtt{bitset}_{\psi[\pi]}\ cs\ F\ i\ \mathtt{then}\ \mathtt{cp}_{\psi[\pi]}\ cs\ F\ (\mathtt{pred}_\pi\ cs\ i)\ (\mathtt{st1}_{\psi[\pi]}\ i\ G)$
$\quad \mathtt{else}\ \mathtt{cp}_{\psi[\pi]}\ cs\ F\ (\mathtt{pred}_\pi\ cs\ i)\ (\mathtt{st0}_{\psi[\pi]}\ i\ G)$

Fig. 8. Clauses for the counting module $C_{\psi[\pi]}$.

To understand the counting program, consider the number 4, with bit representation 100. If the numbers $0, 1, 2, 3$ are represented in C_π by values O, w_1, w_2, w_3 respectively, then this bitstring corresponds for example to Q:

$$\texttt{st1}\ w_1\ (\texttt{st0}\ w_2\ (\texttt{st0}\ w_3\ (\texttt{base}_{\psi[\pi]}\ O)))$$

The null-value O functions as a default, and is a possible value of both $Q\ \texttt{true}$ and $Q\ \texttt{false}$ for any function Q representing a bitstring.

The non-determinism truly comes into play when determining whether $Q\ \texttt{true} \mapsto i$ or not: we can evaluate $F\ \texttt{true}$ to *some* value, but we cannot guarantee that it reduces to the value we need. Therefore, we find some value of both $F\ \texttt{true}$ and $F\ \texttt{false}$; if either gives a representation of i, then we have confirmed or rejected that $b_i = 1$. If both evaluations give a different value, we gives a non-terminating program, but there is always exactly one value b such that $\mathtt{p}_{\psi[\pi]} \vdash^{\texttt{call}} \texttt{bitset}_{\psi[\pi]}\ cs\ F\ i \to b$.

The $\texttt{seed}_{\psi[\pi]}$ function generates the bit string $1 \ldots 1$, so the function F with $F\ \texttt{true} \mapsto i$ for all $i \in \{0, \ldots, P(n) - 1\}$ and $F\ \texttt{false} \mapsto i$ for only $i = 0$. The $\texttt{zero}_{\psi[\pi]}$ function iterates through $b_{P(n)-1}, b_{P(n)-2}, \ldots, b_1$ and tests whether all bits are set to 0. The clauses for $\texttt{pred}_{\psi[\pi]}$ assume given a bitstring $b_1 \ldots b_{i-1}10 \ldots 0$, and recursively build $b_1 \ldots b_{i-1}01 \cdots 1$ in the parameter G.

Example 11. Consider an input string of length 3, say $\texttt{false}{::}\texttt{false}{::}\texttt{true}{::}[]$. There is a $(\lambda n.n + 1)$-counting module $C_{\langle 1,1\rangle}$ representing $i \in \{0, \ldots, 3\}$, and therefore a second-order $(\lambda n.2^n)$-counting module $C_{\psi[\langle 1,1\rangle]}$ representing $i \in \{0, \ldots, 7\}$. The number 6—with bitstring 110—is represented by the value w_6:

$$\texttt{set1}_{\psi[\langle 1,1\rangle]}\ (\texttt{true}{::}[])\ (\ \texttt{set1}_{\psi[\langle 1,1\rangle]}\ (\texttt{false}{::}\texttt{true}{::}[])\ (\ \texttt{set0}_{\psi[\langle 1,1\rangle]}\ (\texttt{false}{::}\texttt{false}{::}\texttt{true}{::}[])\ (\ \texttt{cons}_{\psi[\langle 1,1\rangle]}\ []\ )\ )\ ) : \texttt{bool} \Rightarrow \texttt{list}$$

But then there is also a $(\lambda n.2^{2^n-1})$-counting module $C_{\psi[\psi[\langle 1,1\rangle]]}$, representing $i \in \{0, \ldots, 2^7 - 1\}$. For example the number 97—with bit vector 1100001—is represented by the value S:

$$\begin{array}{l}\texttt{set1}_{\psi[\psi[\langle 1,1\rangle]]}\ w_1\ (\ \texttt{set1}_{\psi[\psi[\langle 1,1\rangle]]}\ w_2\ (\ \texttt{set0}_{\psi[\psi[\langle 1,1\rangle]]}\ w_3\ (\\ \texttt{set0}_{\psi[\psi[\langle 1,1\rangle]]}\ w_4\ (\ \texttt{set0}_{\psi[\psi[\langle 1,1\rangle]]}\ w_5\ (\ \texttt{set0}_{\psi[\psi[\langle 1,1\rangle]]}\ w_6\ (\\ \quad \texttt{set1}_{\psi[\psi[\langle 1,1\rangle]]}\ w_7\ (\ \texttt{cons}_{\psi[\psi[\langle 1,1\rangle]]}\ w_7\ )\ )\ )\ )\ )\ )\end{array}$$

Where $\texttt{set1}_{\psi[\psi[\langle 1,1\rangle]]}$ and $\texttt{set0}_{\psi[\psi[\langle 1,1\rangle]]}$ have the type $(\texttt{bool} \Rightarrow \texttt{list}) \Rightarrow (\texttt{bool} \Rightarrow \texttt{bool} \Rightarrow \texttt{list}) \Rightarrow \texttt{bool} \Rightarrow \texttt{bool} \Rightarrow \texttt{list}$ and each w_i is a value representing the number i in $C_{\psi[\langle 1,1\rangle]}$, as shown for w_6 above. Note that $S\ \texttt{true} \mapsto w_1, w_2, w_7$ and $S\ \texttt{false} \mapsto w_3, w_4, w_5, w_6$.

Since $2^{2^m-1} - 1 \geq 2^m$ for all $m \geq 2$, we can count up to arbitrarily high bounds using this module. Thus, already with data order 1, we can simulate Turing Machines in any class $\mathsf{EXP}^K\mathsf{TIME}$.

Lemma 11. *Every decision problem in* ELEMENTARY *is accepted by a cons-free program with data order* 1.

Proof. A decision problem is in ELEMENTARY if it is in some $\mathsf{EXP}^K\mathsf{TIME}$ which, by Lemma 2, is certainly the case if for any a, b there is a Q-counting module with $Q \geq \lambda n.\exp_2^K(a \cdot n^b)$. Such a module exists for data order 1 by Lemma 10.

6.2 Simulating cons-free programs using an algorithm

For the other part of obtaining a characterisation, we must see that the result of every cons-free program can be obtained by an algorithm in ELEMENTARY—so which runs in TIME $\left(\exp_2^K(a \cdot n^b)\right)$ for some a, b, K. For this, we can use Algorithm 6 *almost* unmodified: the difference is in the definition of $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$.

Definition 17. *Let $\mathcal{B}$ be a set of data expressions closed under $\rhd$. For $\iota \in \mathcal{S}$, let $[\![\iota]\!]_{\mathcal{B}} = \{d \in \mathcal{B} \mid\!\vdash d : \iota\}$. Inductively, define $[\![\sigma \times \tau]\!]_{\mathcal{B}} = [\![\sigma]\!]_{\mathcal{B}} \times [\![\tau]\!]_{\mathcal{B}}$ and $[\![\sigma \Rightarrow \tau]\!]_{\mathcal{B}} = \{A_{\sigma\Rightarrow\tau} \mid A \subseteq [\![\sigma]\!]_{\mathcal{B}} \times [\![\tau]\!]_{\mathcal{B}}\}$. We call the elements of any $[\![\sigma]\!]_{\mathcal{B}}$* non-deterministic extensional values.

Where the elements of $\langle\!|\sigma \Rightarrow \tau|\!\rangle_{\mathcal{B}}$ are partial functions, $[\![\sigma \Rightarrow \tau]\!]_{\mathcal{B}}$ contains arbitrary relations: a value v is associated to a set of pairs (e, u) such that $v\ e$ *might* evaluate to u. The notions of extensional expression, $e(u_1, \ldots, u_n)$ and $\sqsupseteq$ immediately extend to non-deterministic extensional values. Thus we can define:

Algorithm 12 *Let* p *be a fixed, deterministic cons-free program, with* $\mathtt{f}_1 : \kappa_1 \Rightarrow \ldots \Rightarrow \kappa_M \Rightarrow \kappa \in \mathcal{F}$.

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

Output: *The set of values* b *with* $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$.

Execute Algorithm 6, but using $[\![\sigma]\!]_{\mathcal{B}}$ *in place of* $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$.

In Section 6.3, we will see that indeed $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ if and only if Algorithm 12 returns a set containing b. But first we observe:

Lemma 13. *Defining the* arrow depth *of a type by: $\mathit{depth}(\iota) = 0$, $\mathit{depth}(\sigma \times \tau) = \max(\mathit{depth}(\sigma), \mathit{depth}(\tau))$ and $\mathit{depth}(\sigma \Rightarrow \tau) = 1 + \max(\mathit{depth}(\sigma), \mathit{depth}(\tau))$, then if $1 \leq \mathsf{Card}(\mathcal{B}) < N$, $\mathit{depth}(\sigma) \leq K$ and σ has L sorts: $\mathsf{Card}([\![\sigma]\!]_{\mathcal{B}}) < \exp_2^K(N^L)$. For $e, u \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$, testing $e \sqsupseteq u$ requires at most $\exp_2^K(N^{(L+1)^3})$ comparisons.*

Proof (Sketch). A straightforward induction on the form of σ, like Lemma 8.

Thus, once more assuming correctness for now, we may conclude:

Lemma 14. *Every decision problem accepted by a cons-free program* p *is in* ELEMENTARY.

Proof. We will see in Lemma 17 in Section 6.3 that $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ if and only if Algorithm 12 returns a set containing b. For a program where, for all σ in the set Σ (as defined in Lemma 7), $\mathit{depth}(\sigma) \leq K$, Lemmas 13 and 7 together give that Algorithm 12 is in $\mathsf{EXP}^K\mathsf{TIME} \subseteq \mathsf{ELEMENTARY}$.

Theorem 2. *The class of deterministic cons-free programs with data order K characterises* ELEMENTARY *for all $K \in \mathbb{N} \setminus \{0\}$.*

Proof. A combination of Lemmas 11 and 14.

6.3 Correctness proofs of Algorithms 6 and 12

Algorithms 6 and 12 are the same—merely parametrised with a different set of extensional values which may be used in Algorithm 1b. Due to this similarity, and because $\langle\!|\sigma|\!\rangle_{\mathcal{B}} \subseteq [\![\sigma]\!]_{\mathcal{B}}$, we can mostly prove their correctness together. The proofs are somewhat intricate, however; details are provided in the appendix.

We begin with *soundness*:

Lemma 15. *If Algorithm 6 or 12 returns a set $A \cup \{b\}$, then $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$.*

Proof (Sketch). We define for every value $v : \sigma$ and $e \in [\![\sigma]\!]_{\mathcal{B}}$: $v \Downarrow e$ iff:

- $\sigma \in \mathcal{S}$ and $v = e$;
- $\sigma = \sigma_1 \times \sigma_2$ and $v = (v_1, v_2)$ and $e = (e_1, e_2)$ with $v_1 \Downarrow e_1$ and $v_2 \Downarrow e_2$;
- $\sigma = \sigma_1 \Rightarrow \sigma_2$ and $e = A_\sigma$ with $A \subseteq \varphi(v) := \{(u_1, u_2) \mid u_1 \in [\![\sigma_1]\!]_{\mathcal{B}} \wedge u_2 \in [\![\sigma_2]\!]_{\mathcal{B}} \wedge$ for all values $w_1 : \sigma_1$ with $w_1 \Downarrow u_1$ there is some value $w_2 : \sigma_2$ with $w_2 \Downarrow u_2$ such that $\mathsf{p}' \vdash^{\mathtt{call}} v\ w_1 \to w_2\}$

We now prove the following two statements together by induction on the core algorithm, which is equipped with some unspecified subsets $[\sigma]$ of $[\![\sigma]\!]_{\mathcal{B}}$:

1. Let: (a) $\mathtt{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa \in \mathcal{F}$ be a defined symbol; (b) $v_1 : \sigma_1, \ldots, v_n : \sigma_n$ be values, for $1 \leq n \leq \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$; (c) $e_1 \in [\![\sigma_1]\!]_{\mathcal{B}}, \ldots, e_n \in [\![\sigma_n]\!]_{\mathcal{B}}$ be such that each $v_i \Downarrow e_i$; (d) $o \in [\![\sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa]\!]_{\mathcal{B}}$. If $\vdash \mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$ is eventually confirmed, then $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$ for some w with $w \Downarrow o$.
2. Let: (a) ρ: $\mathtt{f}\ \boldsymbol{\ell} = s$ be a clause in p'; (b) $t : \tau$ be a sub-expression of s; (c) η be an ext-environment for ρ; (d) γ be an environment such that $\gamma(x) \Downarrow \eta(x)$ for all $x \in \mathit{Var}(\mathtt{f}\ \boldsymbol{\ell})$; (e) $o \in [\![\tau]\!]_{\mathcal{B}}$. If the statement $\eta \vdash t \rightsquigarrow o$ is eventually confirmed, then $\mathsf{p}', \gamma \vdash t \to w$ for some w with $w \Downarrow o$.

Given the way p' is defined from p, the lemma follows from the first statement. The induction is not hard, but requires minor sub-steps such as transitivity of $\sqsupseteq$.

The harder part, where the algorithms diverge, is *completeness*:

Lemma 16. *If $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$, then Algorithm 12 returns a set $A \cup \{b\}$.*

Proof (Sketch). If $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$, then $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \to b$. We label the nodes in the derivation trees with strings of numbers (a node with label l has immediate subtrees of the form $l \cdot i$), and let $>$ denote lexicographic comparison of these strings, and $\succ$ lexicographic comparison without prefixes (e.g., $1 \cdot 2 > 1$ but not $1 \cdot 2 \succ 1$). We define the following relation:

- $\psi(v, l) = v$ if $v \in \mathcal{B}$, and $(\psi(v_1, l), \psi(v_2, l))$ if $v = (v_1, v_2)$;
- for $\mathtt{f}\ v_1 \cdots v_n : \tau = \sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa$ with $m > n$, let $\psi(\mathtt{f}\ v_1 \cdots v_n, l) = \{(e_{n+1}, u) \mid \exists q \succ p > l$ [the subtree with index p has a root $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_{n+1} \to w$ with $\psi(w, q) = u$ and $e_{n+1} \sqsupseteq' \psi(v_{n+1}, p)$]$\}_\tau$.

Here, $\sqsupseteq'$ is defined the same as $\sqsupseteq$, except that $A_\sigma \sqsupseteq' B_\sigma$ iff $A \supseteq B$. Note that clearly $A \sqsupseteq' B$ implies $A \sqsupseteq B$, and that $\sqsupseteq'$ is transitive by transitivity of $\supseteq$. Then, using induction on the labels of the tree in reverse lexicographical order (so going through the tree right-to-left, top-to-bottom), we can prove:

1. If the subtree labelled l has root $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$, then for all $e_1, \ldots, e_n$ such that each $e_i \sqsupseteq' \psi(v_i, l)$, and for all $p \succ l$ there exists $o \sqsupseteq' \psi(w, p)$ such that $\vdash \mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$ is eventually confirmed.

2. If the subtree labelled l has root $\mathsf{p}', \gamma \vdash t \to w$ and $\eta(x) \sqsupseteq' \psi(\gamma(x), l)$ for all $x \in \mathit{Var}(t)$, then for all $p \succ l$ there exists $o \sqsupseteq' \psi(w, p)$ such that $\eta \Vdash t \rightsquigarrow o$ is eventually confirmed.

The case for the main tree—which has label 0 so there exists $p \succ 0$—gives that $\vdash \mathtt{start}\ d_1 \cdots d_M \rightsquigarrow b$ is eventually confirmed, so b is indeed returned.

We conclude:

Lemma 17. *$[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ iff Algorithm 12 returns a set containing b.*

Proof. This is a combination of Lemmas 15 and 16.

The proof of the general case provides a basis for the deterministic case:

Lemma 18. *If $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ and p is deterministic, then Algorithm 12 returns a set $A \cup \{b\}$.*

Proof. We define a consistency measure $\wr$ on non-deterministic extensional values, with, e.g., $A_\sigma \wr B_\sigma$ iff for all $(e_1, u_1) \in A$ and $(e_2, u_2) \in B$: $e_1 \wr e_2$ implies $u_1 \wr u_2$.

In the proof of Lemma 16, we trace a derivation in the algorithm. In a deterministic program, we can see that if both $\vdash \mathtt{f}\ e_1 \cdots e_n \to o$ and $\vdash \mathtt{f}\ e'_1 \cdots e'_n \to o'$ are confirmed, and each $e_i \wr e'_n$, then $o \wr o'$—and similar for statements $\eta \vdash s \Rightarrow o$. We use this to remove statements which are not necessary, ultimately leaving only those which use extensional values. This gives a result using Algorithm 6.

Lemma 19. *$[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ iff Algorithm 6 returns a set containing b.*

Proof. This is a combination of Lemmas 15 and 18.

7 Recovering the EXPTIME hierarchy

While interesting, Lemma 11 exposes a problem: non-determinism is unexpectedly powerful in the higher-order setting. If we still want to use non-deterministic programs towards characterising non-deterministic complexity classes, we must surely start by considering restrictions which avoid this explosion of expressivity.

A possible direction is to consider *arrow depth* instead of data order: following Lemma 13 and the counting module of Figure 8, we can see that the class of cons-free programs with "data arrow depth K" characterises $\mathsf{EXP}^K\mathsf{TIME}$. However, this characterisation is not *practical*: it is hard for a human to take advantage of the expressivity in programs with a high arrow depth but low data order, as it relies so much on using non-deterministic choice.

Instead, we propose a very simple restriction: *unary variables*. This imposes no conditions in the setting with data order 0, which gives us the first column in the table from Section 6, and in general gives us the last line from Figure 1.

	data order 0	**data order 1**	**data order 2**	**data order 3**
cons-free unary variables	$\mathsf{P} = \mathsf{EXP}^0\mathsf{TIME}$	$\mathsf{EXP} = \mathsf{EXP}^1\mathsf{TIME}$	$\mathsf{EXP}^2\mathsf{TIME}$	$\mathsf{EXP}^3\mathsf{TIME}$

Definition 18. *A program* p *has* unary variables *if clauses can be typed with an assignment that maps each variable x to either a type κ, or $\sigma \Rightarrow \kappa$ with $o(\kappa) = 0$.*

Thus, in a program with unary variables, a variable of a type $(\texttt{list} \times \texttt{list} \times \texttt{list}) \Rightarrow \texttt{list}$ is admitted, but $\texttt{list} \Rightarrow \texttt{list} \Rightarrow \texttt{list} \Rightarrow \texttt{list}$ is not. The crucial difference is that the former must be applied to all its arguments at the same time, while the latter may be partially applied. Since the type order and arrow depth of a unary type are the same, the results in the table above follow easily:

Lemma 20. *Every decision problem in* $\mathsf{EXP}^K\mathsf{TIME}$ *is accepted by a cons-free program with data order K and unary variables.*

Proof. Both the base program in Figure 7 and the counting modules of Lemmas 3 and 4 have unary variables. Note that the program is deterministic.

Lemma 21. *Every decision problem accepted by a cons-free program with data order K and unary variables is in* $\mathsf{EXP}^K\mathsf{TIME}$.

Proof. For programs with unary variables, the arrow depth of a type is exactly its order. Thus, by Lemma 13, Algorithm 12 operates within $\mathsf{EXP}^K\mathsf{TIME}$, and by Lemma 17, it is correct.

Theorem 3. *The class of cons-free programs with unary variables of data order K characterises* $\mathsf{EXP}^K\mathsf{TIME}$.

Proof. Immediate by Lemmas 20 and 21.

8 Conclusion and future work

We have studied the effect of combining higher types and non-determinism for cons-free programs. This has resulted in the—highly surprising—conclusion that naively adding non-deterministic choice to a language that characterises the $\mathsf{EXP}^K\mathsf{TIME}$ hierarchy for increasing data orders immediately increases the expressivity of the language to ELEMENTARY. Recovering a more fine-grained complexity hierarchy can be done, but at the cost of further syntactical restrictions.

The primary goal of future work is to use non-deterministic cons-free programs to characterise hierarchies of *non*-deterministic complexity classes such as the classes $\mathsf{NEXP}^K\mathsf{TIME}$ for $K \in \mathbb{N}$. In addition, a full study must be made of the ramifications of imposing restrictions on recursion, such as tail-recursion or primitive recursion, in combination with non-determinism and higher types. We intend to study functional complexity using (deterministic and non-deterministic) cons-free programs, and characterisations of classes more restrictive than P, such as LOGTIME and LOGSPACE.

Finally, given the surprising nature of the results in the paper, we urge readers to investigate the effect of adding non-determinism to other programming languages used in implicit complexity that manipulate higher-order data. We conjecture that the effect on expressivity in these will essentially be the same as those we have observed.

References

1. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
2. Stephen Bellantoni. PhD thesis, University of Toronto, 1993.
3. G. Bonfante. Some programming languages for logspace and ptime. In *AMAST*, volume 4019 of *LNCS*, pages 66–80, 2006.
4. P. Clote. Computation models and function algebras. In *Handbook of Computability Theory*, pages 589–681. Elsevier, 1999.
5. S.A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *ACM*, 18(1):4–18, 1971.
6. Ugo Dal Lago. *A Short Introduction to Implicit Computational Complexity*, pages 89–109. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. URL: http://dx.doi.org/10.1007/978-3-642-31485-8_3, doi:10.1007/978-3-642-31485-8_3.
7. D. de Carvalho and J. Simonsen. An implicit characterization of the polynomial-time decidable sets by cons-free rewriting. In *RTA-TLCA*, volume 8560 of *LNCS*, pages 179–193, 2014.
8. Andreas Goerdt. Characterizing complexity classes by general recursive definitions in higher types. *Inf. Comput.*, 101(2):202–218, 1992. URL: http://dx.doi.org/10.1016/0890-5401(92)90062-K, doi:10.1016/0890-5401(92)90062-K.
9. Andreas Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theor. Comput. Sci.*, 100(1):45–66, 1992. URL: http://dx.doi.org/10.1016/0304-3975(92)90363-K, doi:10.1016/0304-3975(92)90363-K.
10. Neil Immerman. *Descriptive Complexity*. Springer-Verlag, 1999.
11. N. Jones. *Computability and Complexity from a Programming Perspective*. MIT Press, 1997.
12. N. Jones. The expressive power of higher-order types or, life without CONS. *JFP*, 11(1):55–94, 2001.
13. A. J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. An analysis of ML typability. *J. ACM*, 41(2):368–398, 1994. URL: http://doi.acm.org/10.1145/174652.174659, doi:10.1145/174652.174659.
14. Cynthia Kop and Jakob Grue Simonsen. Complexity hierarchies and higher-order cons-free rewriting. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, volume 52 of *LIPIcs*, pages 23:1–23:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. URL: http://dx.doi.org/10.4230/LIPIcs.FSCD.2016.23, doi:10.4230/LIPIcs.FSCD.2016.23.
15. L. Kristiansen and K.-H. Niggl. Implicit computational complexity on the computational complexity of imperative programming languages. *Theoretical Computer Science*, 318(1):139 – 161, 2004. URL: http://www.sciencedirect.com/science/article/pii/S0304397503005218, doi:http://dx.doi.org/10.1016/j.tcs.2003.10.016.
16. Lars Kristiansen and Paul J. Voda. Programming languages capturing complexity classes. *Nord. J. Comput.*, 12(2):89–115, 2005.
17. Isabel Oitavem. A recursion-theoretic approach to NP. *Ann. Pure Appl. Logic*, 162(8):661–666, 2011. URL: http://dx.doi.org/10.1016/j.apal.2011.01.010, doi:10.1016/j.apal.2011.01.010.
18. C. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
19. M. Sipser. *Introduction to the Theory of Computation*. Thomson Course Technology, 2006.

This appendix contains full proofs of the results presented in the text.

A Properties of cons-free programs (Section 3)

Before recalling Lemma 1, we present a reformulation with extra information that will be easier to use in later appendixes.

Lemma A1. *Let* p *be a cons-free program,* $d_1, \dots, d_M$ *be data expressions, and let* ***Value'*** *be given by the grammar:*

$$v, w \in \textbf{\textit{Value'}} ::= d \in \mathcal{B}^{\mathsf{p}}_{d_1,\dots,d_M} \mid (v, w) \mid f\ v_1 \cdots v_n \ \ (n < \mathtt{arity}_{\mathsf{p}}(f))$$

Let T *be a derivation tree for* $[\![\mathsf{p}]\!](d_1, \dots, d_M) \mapsto b$*. Then for all subtrees* T' *of* T*:*

- *if* T' *has root* $\mathsf{p}, \gamma \vdash s \to w$*, then both* w *and all* $\gamma(x)$ *are in* ***Value'****;*
- *if* T' *has root* $\mathsf{p}, \gamma \vdash^{\mathtt{if}} d, s_1, s_2 \to w$*, then* $d \in \mathcal{B}^{\mathsf{p}}_{d_1,\dots,d_M}$ *and both* w *and all* $\gamma(x)$ *are in* ***Value'****;*
- *if* T' *has a root* $\mathsf{p} \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$ *with* $\mathtt{f} \in \mathcal{D}$*, then both* w *and all* v_i *are in* ***Value'****;*
- *if* T' *has a root* $\mathsf{p}, \gamma \vdash \mathtt{c}\ s_1 \cdots s_m \to \mathtt{c}\ b_1 \cdots b_m$ *with* $\mathtt{c} \in \mathcal{C}$*, then each* $s_i\gamma = b_i \in$ ***Data*** *and* $\mathtt{c}\ b_1 \cdots b_m \in \mathcal{B}^{\mathsf{p}}_{d_1,\dots,d_M}$*.*

Proof. Write $\mathcal{B} := \mathcal{B}^{\mathsf{p}}_{d_1,\dots,d_M}$.

In order to obtain the last bullet, we first prove: (*) if T' is a tree with root $\mathsf{p}, \gamma \vdash s \to w$, and if s is a pattern, then $s\gamma = w$. We prove this by induction on the form of T'. Note that the roots of [Function], [Choice] and [Conditional] have the wrong shape. [Instance] immediately gives the required result, and the cases for [Constructor] and [Pair] follow by the induction hypothesis. Finally, we can prove by induction on n that [Appl] is not applicable: if $n = 1$, then [Appl] would require a subtree $\mathsf{p}, \gamma \vdash \mathtt{c} \to \mathtt{f}$ with $\mathtt{f} \in \mathcal{D}$, for which there are no inference rules. If $n > 1$, then [Appl] would require a subtree $\mathsf{p}, \gamma \vdash \mathtt{c}\ s_1 \cdots s_{n-1} \to \mathtt{f}\ v_1 \cdots v_i$ which, by the induction hypothesis, must be obtained by an inference rule other than [Appl]; again, there are no suitable inference rules.

Next we prove by induction on the depth of T' that: (**) the properties hold for both T' and all its strict subtrees if $\mathit{root}(T')$ has one of the following forms:

- $\mathsf{p}, \gamma \vdash s \to w$ with all $\gamma(x) \in$ `Value'`, and $t\gamma \in \mathcal{B}$ for all sub-expressions $t \trianglelefteq s$ such that $t = \mathtt{c}\ s_1 \cdots s_m$ for some $\mathtt{c} \in \mathcal{C}$;
- $\mathsf{p}, \gamma \vdash^{\mathtt{if}} d, s_1, s_2 \to w$ with $d \in \mathcal{B}$ and all $\gamma(x) \in$ `Value'`, and $t\gamma \in \mathcal{B}$ for all sub-expressions $t \trianglelefteq s_1$ or $t \trianglelefteq s_2$ such that $t = \mathtt{c}\ s_1 \cdots s_m$ for some $\mathtt{c} \in \mathcal{C}$;
- $\mathsf{p} \vdash^{\mathtt{call}} f\ v_1 \cdots v_n \to w$ with all $v_i \in$ `Value'`.

Note that proving this suffices: the immediate subtree T' of T has a root $\mathsf{p}, \gamma \vdash \mathtt{f}_1\ x_1 \cdots x_M \to b$, where each $\gamma(x_i) = d_i \in \mathcal{B}$, and $\mathtt{f}_1\ x_1 \cdots x_M$ has no sub-expressions with a data constructor at the head. Thus, the lemma holds for both T' and all its strict subtrees, which implies that it holds for T.

We prove (**). Assume that $\mathit{root}(T')$ has one of the given forms, and consider the rule used to obtain this root.

Instance Then T' has a root $\mathtt{p}, \gamma \vdash x \to \gamma(x)$; the requirement that all $\gamma(y) \in \mathtt{Value'}$ is satisfied by the assumption, and this also gives that the right-hand side $\gamma(x) \in \mathtt{Value'}$.

Function Then T' has a root $\mathtt{p}, \gamma \vdash \mathtt{f} \to v$ and a subtree $\mathtt{p} \vdash^{\mathtt{call}} \mathtt{f} \to v$; by the induction hypothesis, the properties hold for this subtree, which also implies that $v \in \mathtt{Value'}$ and therefore the properties hold for T' as well.

Constructor Then T' has a root $\mathtt{p}, \gamma \vdash \mathtt{c}\ s_1 \cdots s_m \to \mathtt{c}\ b_1 \cdots b_m$ with $\mathtt{c} \in \mathcal{C}$, and the immediate subtrees have the form $\mathtt{p}, \gamma \vdash s_i \to b_i$; by the induction hypothesis (and the assumption), the properties are satisfied for each such subtree. Also by the assumption, $(\mathtt{c}\ s_1 \cdots s_m)\gamma \in \mathcal{B}$, so necessarily each $s_i\gamma \in \mathcal{B} \subseteq \mathtt{Data}$. By (*), each $s_i\gamma = b_i$, and $\mathtt{c}\ b_1 \cdots b_m = (\mathtt{c}\ s_1 \cdots s_m)\gamma \in \mathcal{B}$.

Pair Then T' has a root $\mathtt{p}, \gamma \vdash (s_1, s_2) \to (w_1, w_2)$ and subtrees with roots $\mathtt{p}, \gamma \vdash s_1 \to w_1$ and $\mathtt{p}, \gamma \vdash s_2 \to w_2$. The assumption and induction hypothesis give that the properties are satisfied for both subtrees, and therefore both w_1 and w_2 are in $\mathtt{Value'}$, giving also $(w_1, w_2) \in \mathtt{Value'}$.

Choice Then T' has a root $\mathtt{p}, \gamma \vdash \mathtt{choose}\ s_1 \cdots s_n \to v$ and a subtree $\mathtt{p}, \gamma \vdash s_i \to v$ for some i. By the induction hypothesis, the properties hold for the subtree, and therefore $v \in \mathtt{Value'}$.

Conditional Then T' has a root $\mathtt{p}, \gamma \vdash \mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3 \to w$ and subtrees with roots $\mathtt{p}, \gamma \vdash s_1 \to d$ and $\mathtt{p}, \gamma \vdash^{\mathtt{if}} d, s_2, s_3 \to w$. The requirement that all $\gamma(x) \in \mathtt{Value'}$ is satisfied by the assumption, and by both the assumption and the induction hypothesis, the lemma is satisfied for the first subtrees. Thus, $d \in \mathtt{Value'}$; for typing reasons $d \in \mathcal{B}$. We may apply the induction hypothesis on the second subtree, which gives that the lemma is satisfied for it, and that $w \in \mathtt{Value'}$.

If-True or If-False Then $root(T')$ has the form $\mathtt{p}, \gamma \vdash^{\mathtt{if}} d, s_1, s_2 \to w$. The requirement that $d \in \mathcal{B}$ and all $\gamma(x) \in \mathtt{Value'}$ is satisfied by the assumption. T' has one immediate subtree T'', whose root is either $\mathtt{p}, \gamma \vdash s_2 \to w$ or $\mathtt{p}, \gamma \vdash s_3 \to w$. Since the assumptions are satisfied, T'' satisfies the lemma by the induction hypothesis, which also gives that $w \in \mathtt{Value'}$.

Appl Then T' has a root $\mathtt{p}, \gamma \vdash s\ t \to w$ and subtrees $\mathtt{p}, \gamma \vdash s \to \mathtt{f}\ v_1 \cdots v_n$ and $\mathtt{p}, \gamma \vdash t \to v_{n+1}$ and $\mathtt{p} \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_{n+1} \to w$. The assumption gives that all $\gamma(x) \in \mathtt{Value'}$, and the assumption and induction hypothesis together give that the lemma is satisfied for the first two subtrees. Since this implies that all $v_i \in \mathtt{Value'}$, we may also apply the induction hypothesis on the last subtree, which gives that $w \in \mathtt{Value'}$.

Closure Then $root(T')$ has the form $\mathtt{p}, \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$ with $\mathtt{f} \in \mathcal{D}$. The requirement that all v_i are in $\mathtt{Value'}$ is satisfied by the assumption, therefore $w = f\ v_1 \cdots v_n \in \mathtt{Value'}$ as well, and there are no strict subtrees.

Call Then $root(T')$ has the form $\mathtt{p}, \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_k \to w$ with $\mathtt{f} \in \mathcal{D}$, and there exist a clause $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ and an environment γ with domain $Var(\mathtt{f}\ \ell_1 \cdots \ell_k)$ such that each $v_i = \ell_i\gamma$, and T' has one immediate subtree T'' with root $\mathtt{p}, \gamma \vdash s \to w$. Then, for $1 \leq i \leq n$ we observe that $v_i \trianglerighteq \gamma(x)$ for all $x \in Var(\ell_i)$, since (by definition of a pattern) $\ell_i \trianglerighteq x$ for all such x. Since all sub-expressions of a value in $\mathtt{Value'}$ are themselves in $\mathtt{Value'}$, we thus have: each $\gamma(x) \in \mathtt{Value'}$.

Moreover, for $s \trianglerighteq t = \mathtt{c}\ s_1 \cdots s_m$ with $\mathtt{c} \in \mathcal{C}$, also $\ell_i \trianglerighteq t$ for some i by definition of a cons-free program. But then also $\ell_i\gamma = v_i \trianglerighteq t\gamma$.
Thus, we can apply the induction hypothesis, and obtain that the lemma is satisfied for T''. This implies that $w \in \mathtt{Value'}$, so the last requirement on the root of T' is also satisfied.

Now recall Lemma 1:

Lemma 1. *Let* p *be a cons-free program, and suppose that* $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ *is obtained by a derivation tree* T*. Then for all statements* $\mathsf{p}, \gamma \vdash s \to w$ *or* $\mathsf{p}, \gamma \vdash^{\mathtt{if}} b', s_1, s_2 \to w$ *or* $\mathsf{p} \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$*, and all sub-expressions* d *such that (a)* $w \trianglerighteq d$*, (b)* $b' \trianglerighteq d$*, (c)* $\gamma(x) \trianglerighteq d$ *for some* x *or (d)* $v_i \trianglerighteq d$ *for some* i*: if* d *has the form* $\mathtt{c}\ b_1 \cdots b_m$ *with* $\mathtt{c} \in \mathcal{C}$*, then* $d \in \mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$.

Proof. This is an immediate consequence of Lemma A1, since the only sub-expressions of an element of $\mathtt{Value'}$ whose head symbol is a data constructor, are in $\mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$.

B Counting modules (Section 5.1)

We discuss the counting modules from Section 5.1 in more detail.

Lemma 3. *For any* $a, b \in \mathbb{N} \setminus \{0\}$*, there is a* $(\lambda n.a \cdot (n+1)^b)$*-counting module* $C_{\langle a,b\rangle}$ *with data order* 0.

Proof. Using pairing in a right-associative way—that is, (x, y, z) should be read as $(x, (y, z))$, we let:

- $\alpha_{\langle a,b\rangle} := \mathtt{list}^{b+1}$; that is, $\mathtt{list} \times \cdots \times \mathtt{list}$ with $b+1$ occurrences of $\mathtt{list}$
- $\mathcal{A}^n_{\langle a,b\rangle} := \{(d_0, \ldots, d_b) \mid \text{all } d_i \text{ are boolean lists, with } |d_0| < a \text{ and } |d_i| \leq n \text{ for } 1 \leq i \leq b$; here, we say $|x_1{::}\ldots{::}x_k{::}[]| = k$
- $\langle (d_0, \ldots, d_b)\rangle^n_{\langle a,b\rangle} := \sum_{i=0}^{b} |d_i| \cdot (n+1)^{b-i}$
- $\mathcal{D}_{\langle a,b\rangle} = \{\mathtt{seed}_{\langle a,b\rangle}, \mathtt{pred}_{\langle a,b\rangle}, \mathtt{zero}_{\langle a,b\rangle}\}$
- let $\mathtt{alist}$ be a list of length $a-1$, e.g., $\mathtt{false}{::}\ldots{::}\mathtt{false}{::}[]$ and let $\mathsf{p}_{\langle a,b\rangle}$ consist of the following clauses:

$$\mathtt{seed}_{\langle a,b\rangle}\ cs = (\mathtt{alist}, cs, \ldots, cs)$$

$$\begin{aligned}
&\mathtt{pred}_{\langle a,b\rangle}\ cs\ (x_0, \ldots, x_{b-1}, y{::}ys) = (x_0, \ldots, x_{b-1}, xs)\\
&\mathtt{pred}_{\langle a,b\rangle}\ cs\ (x_0, \ldots, x_{b-2}, y{::}ys, []) = (x_0, \ldots, x_{b-2}, ys, cs)\\
&\ldots\\
&\mathtt{pred}_{\langle a,b\rangle}\ cs\ (y{::}ys, [], \ldots, []) = (ys, cs, \ldots, cs)\\
&\mathtt{pred}_{\langle a,b\rangle}\ cs\ ([], [], \ldots, []) = ([], [], \ldots, [])
\end{aligned}$$

$$\begin{aligned}
&\mathtt{zero}_{\langle a,b\rangle}\ cs\ (x_0, \ldots, x_{b-1}, y{::}ys) = \mathtt{false}\\
&\mathtt{zero}_{\langle a,b\rangle}\ cs\ (x_0, \ldots, x_{b-2}, y{::}ys, []) = \mathtt{false}\\
&\ldots\\
&\mathtt{zero}_{\langle a,b\rangle}\ cs\ (y{::}ys, [], \ldots, []) = \mathtt{false}\\
&\mathtt{zero}_{\langle a,b\rangle}\ cs\ ([], \ldots, []) = \mathtt{true}
\end{aligned}$$

It is easy to see that the requirements on evaluation are satisfied. For example, $\mathsf{p}_{\langle a,b\rangle} \vdash^{\mathtt{call}} \mathtt{seed}_{\langle a,b\rangle}\ cs \to (\mathtt{alist}, cs, \ldots, cs)$, which consists of $b+1$ boolean lists with the appropriate lengths, and $\langle(\mathtt{alist}, cs, \ldots, cs)\rangle^n_{\langle a,b\rangle} = (a-1)\cdot(n+1)^b + n\cdot(n+1)^{b-1} + \cdots + n\cdot(n+1)^{b-b} = (a\cdot(n+1)^b - (n+1)^b) + ((n+1)^b - (n+1)^{b-1}) + \cdots + ((n+1)^1 - (n+1)^0) = a\cdot(n+1)^b - 1$; since the program is deterministic, this is the only value. The evaluation requirements for $\mathtt{pred}_{\langle a,b\rangle}$ and $\mathtt{zero}_{\langle a,b\rangle}$ are similarly easy.

Lemma 4. *If there is a P-counting module C_π of data order K, then there is a $(\lambda n.2^{P(n)})$-counting module $C_{\mathsf{e}[\pi]}$ of data order $K+1$.*

Proof. We let:

- $\alpha_{\mathsf{e}[\pi]} := \alpha_\pi \Rightarrow \mathtt{bool}$; then $o(\alpha_{\mathsf{e}[\pi]}) \leq K+1$;
- $\mathcal{A}^n_{\mathsf{e}[\pi]} := \{$values F such that, (a) for all $v \in \mathcal{A}^n_\pi$: either $\mathsf{p}_{\mathsf{e}[\pi]} \vdash^{\mathtt{call}} F\ v \to \mathtt{true}$ or $\mathsf{p}_{\mathsf{e}[\pi]} \vdash^{\mathtt{call}} F\ v \to \mathtt{false}$ (but not both), and (b) for all $v, w \in \mathcal{A}^n_\pi$: if $\langle v\rangle^n_\pi = \langle w\rangle^n_\pi$ then $\mathsf{p}_{\mathsf{e}[\pi]} \vdash^{\mathtt{call}} F\ v \to b$ and $\mathsf{p}_{\mathsf{e}[\pi]} \vdash^{\mathtt{call}} F\ w \to d$ implies $b = d\}$; that is, $\mathcal{A}^n_{\mathsf{e}[\pi]}$ is the set of functions from α_π to $\mathtt{bool}$ such that $F\ [i]$ is uniquely defined for any representation $[i]$ of $i \in \{0, \ldots, P(n)-1\}$ in C_π;
- $\langle F\rangle^n_{\mathsf{e}[\pi]} = \sum_{i=0}^{P(n)-1}\{2^{P(n)-1-i} \mid \exists v \in \mathcal{A}^n_\pi[\langle v\rangle^n_\pi = i \wedge \mathsf{p}_{\mathsf{e}[\pi]} \vdash^{\mathtt{call}} F\ i \to \mathtt{true}]\}$; that is, F is mapped to the number i with a bitstring $b_0 \ldots b_{P(n)-1}$, where $b_i = 1$ if and only if $F\ [i]$ has value $\mathtt{true}$;
- $\mathcal{D}_{\mathsf{e}[\pi]} = \mathcal{D}_\pi \cup \{\mathtt{not}\} \cup \{\mathsf{f}_{\mathsf{e}[\pi]} \mid \mathsf{f}_{\mathsf{e}[\pi]}$ used in $\mathsf{p}_{\mathsf{e}[\pi]}$ below$\}$
- $\mathsf{p}_{\mathsf{e}[\pi]}$ consists of the following clauses, followed by the clauses in p_π:

$$\begin{aligned}
&\mathtt{seed}_{\mathsf{e}[\pi]}\ cs = \mathtt{alwaystrue}_{\mathsf{e}[\pi]}\\
&\mathtt{alwaystrue}_{\mathsf{e}[\pi]}\ x = \mathtt{true}\\
&\\
&\mathtt{zero}_{\mathsf{e}[\pi]}\ cs\ F = \mathtt{zhelp}_{\mathsf{e}[\pi]}\ cs\ F\ (\mathtt{seed}_\pi\ cs)\\
&\mathtt{zhelp}_{\mathsf{e}[\pi]}\ cs\ F\ k = \mathtt{if}\ F\ k\ \mathtt{then\ false}\\
&\qquad\qquad\qquad\quad \mathtt{else\ if}\ \mathtt{zero}_\pi\ cs\ k\ \mathtt{then\ true}\\
&\qquad\qquad\qquad\quad \mathtt{else}\ \mathtt{zhelp}_{\mathsf{e}[\pi]}\ cs\ F\ (\mathtt{pred}_\pi\ cs\ k)\\
&\\
&\mathtt{pred}_{\mathsf{e}[\pi]}\ cs\ F = \mathtt{phelp}_{\mathsf{e}[\pi]}\ cs\ F\ (\mathtt{seed}_\pi\ cs)\\
&\mathtt{phelp}_{\mathsf{e}[\pi]}\ cs\ F\ k = \mathtt{if}\ F\ k\ \mathtt{then}\ \mathtt{flip}_{\mathsf{e}[\pi]}\ cs\ F\ k\\
&\qquad\qquad\qquad\quad \mathtt{else\ if}\ \mathtt{zero}_\pi\ cs\ k\ \mathtt{then}\ \mathtt{seed}_{\mathsf{e}[\pi]}\ cs\\
&\qquad\qquad\qquad\quad \mathtt{else}\ \mathtt{phelp}_{\mathsf{e}[\pi]}\ cs\ (\mathtt{flip}_{\mathsf{e}[\pi]}\ cs\ F\ k)\ (\mathtt{pred}_\pi\ cs\ k)\\
&\mathtt{flip}_{\mathsf{e}[\pi]}\ cs\ F\ k\ i = \mathtt{if}\ \mathtt{equal}_\pi\ cs\ k\ i\ \mathtt{then\ not}\ (F\ i)\ \mathtt{else}\ F\ i\\
&\mathtt{not}\ b = \mathtt{if}\ b\ \mathtt{then\ false\ else\ true}
\end{aligned}$$

By standard bitvector arithmetic—in particular the observation that the predecessor of $b_0 \ldots b_i 10 \ldots 0$ is $b_0 \ldots b_i 01 \ldots 1$—we see that the evaluation requirements are satisfied.

C Algorithm complexity (Sections 5.2 and 6.2)

Recall the key lemma from Section 5.2:

Lemma 13. *Let* p *be a cons-free program, proper for data order* K*. Let* Σ *be the set of all types* σ *with* $o(\sigma) \leq K$ *which occur as part of an argument type, or as an output type of some* $\mathtt{f} \in \mathcal{D}$*.*

Suppose that, given input of total size n*,* $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$ *has cardinality at most* $F(n)$ *for all* $\sigma \in \Sigma$*, and testing whether* $e_1 \sqsupseteq e_2$ *for* $e_1, e_2 \in [\![\sigma]\!]_{\mathcal{B}}$ *takes at most* $F(n)$ *steps. Then Algorithm 6 runs in TIME* $\left(a \cdot F(n)^b\right)$ *for some* a, b*.*

Proof. We first observe that, for any $e \in [\![\sigma]\!]_{\mathcal{B}}$ occurring in the algorithm, $\sigma \in \Sigma$. This is due to the preparation step where p is replaced by p'.

Write $\mathtt{a}$ for the greatest number of arguments any defined symbol $\mathtt{f}$ or variable x occurring in p' may take, and write $\mathtt{r}$ for the greatest number of sub-expressions of any right-hand side in p' (which does not depend on the input!). We start by observing that $\mathcal{X}$ contains at most $\mathtt{a} \cdot |\mathcal{D}| \cdot F(n)^{\mathtt{a}+1}$ statements $\mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$, and at most $|\mathsf{p}'| \cdot \mathtt{r} \cdot F(n)^{\mathtt{a}+1}$ statements $t\eta \rightsquigarrow o$.

We observe that step 1a does not depend on the input, so takes a constant number of steps. Step 1b and 1c both take $|\mathcal{X}|$ steps. The exact time cost of each step depends on implementation concerns, but is certainly limited by some polynomial of $F(n)$, by the assumption on $\sqsupseteq$.. Thus, the preparation step is polynomial in $F(n)$; say its cost is $P_1(F(n))$.

In every step of the iteration, at least one statement is flipped from unconfirmed to confirmed, or the iteration ends. Thus, there are at most $|\mathcal{X}| + 1$ iterations. In each iteration, Step 2a has a cost limited by $\mathit{Card}(O) \cdot |\mathcal{X}| \cdot \langle$cost of checking $u' \sqsupseteq u\rangle \leq F(n)^3 \cdot |\mathcal{X}| \cdot \langle$ some implementation-dependent constant$\rangle$. Step 2b has a cost limited by $|\mathsf{p}'| \cdot \langle$cost of matching$\rangle \cdot |\mathcal{X}| \cdot \langle$some implementation-dependent constant$\rangle$. Both Steps 2c and 2e are limited by $2 \cdot \langle$some constant$\rangle \cdot |\mathcal{X}|$ as well (the cost for looking up confirmation status of two given statements), and Step 2d is certainly limited by $r \cdot \langle$some constant$\rangle \cdot |\mathcal{X}|$.

For each statement $s\eta \rightsquigarrow o$ in Steps 2f and 2g, we must check all suitable tuples $(e_1, \ldots, e_{n'})$—of which there are at most $F(n)^{\mathtt{a}}$—and test confirmation for each $s_i\eta \rightsquigarrow e_i$. In Step 2f, we must additionally do $\sqsupseteq$ tests for all $o' \in \eta(x)(e_1, \ldots, e_{n'})$ for all tuples; even if we ignore that $\eta(x)$ is a partial function, this takes at most $F(n)^{\mathtt{a}} \cdot F(n)^{\mathtt{a}} \cdot F(n) \cdot \langle$some constant$\rangle$ steps. In Step 2(g)i, a single lookup over $|\mathcal{X}|$ statements must be done; in Step 2(g)ii this is combined with a lookup. Both cases certainly stay below $F(n)^{2\cdot\mathtt{a}+2} \cdot \langle$some constant$\rangle$ steps.

In total, the cost of iterating is thus limited by $(|\mathcal{X}|+1) \cdot |\mathcal{X}| \cdot \langle$some constant$\rangle \cdot \max(F(n)^3 \cdot |\mathcal{X}|, |\mathsf{p}'| \cdot |\mathcal{X}|, 2 \cdot |\mathcal{X}|, r \cdot |\mathcal{X}|, F(n)^{2\cdot\mathtt{a}+2})$. Since $|\mathcal{X}|$ is a polynomial in $F(n)$, this is certainly bounded by $P_2(F(n))$ for some polynomial P_2.

Finally, completion requires at most $|\mathcal{X}|$ tests. Overall, all steps together gives a polynomial time complexity in $F(n)$.

To derive the sizes of $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$ and $[\![\sigma]\!]_{\mathcal{B}}$, we first obtain a simple helper lemma:

Lemma C2. *If* $X, Y \geq 2$*, then* $\exp_2^K(X) \cdot \exp_2^K(Y)$ *for all* $K \in \mathbb{N}$*.*

Proof. For $X, Y \geq 2$ always (**) $X \cdot Y \geq X + Y$:

– $2 \cdot 2 = 4 = 2 + 2$;

- if $X \cdot Y \geq X + Y$, then $X \cdot (Y+1) = X \cdot Y + X \geq (X+Y) + X \geq X + (Y+1)$;
- if $X \cdot Y \geq X + Y$, then $(X+1) \cdot Y \geq Y + X + 1$ in the same way.

By induction on K we also see: (***) if $X \geq 2$ then $\exp_2^K(X) \geq 2$ for all K. Now the lemma follows by another induction on K:

- for $K = 0$: $\exp_2^K(X) \cdot \exp_2^K(Y) = X \cdot Y = \exp_2^K(X \cdot Y)$;
- for $K \geq 0$: $\exp_2^{K+1}(X) \cdot \exp_2^{K+1}(Y) = 2^{\exp_2^K(X)} \cdot 2^{\exp_2^K(Y)} = 2^{\exp_2^K(X) + \exp_2^K(Y)} \leq 2^{\exp_2^K(X) \cdot \exp_2^K(Y)}$ by (**) and (***), $\leq 2^{\exp_2^K(X) \cdot \exp_2^K(Y)} = \exp_2^{K+1}(X \cdot Y)$ by the induction hypothesis.

We can prove the result on $\sqsupseteq$ for Lemmas 8 and 13 together:

Lemma C3. *Let $[\sigma]$ be one of $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$ or $[\![\sigma]\!]_{\mathcal{B}}$, and suppose that we know that for all subtypes of σ containing L sorts: $\mathsf{Card}([\sigma]) < \exp_2^K(N^L)$ for some fixed K, and $N \geq 2$. Then for any $e, u \in [\sigma]$: testing $e \sqsupseteq u$ requires $< \exp_2^K(N^{(L+1)^3})$ comparisons between elements of $\mathcal{B}$.*

Proof. We let C_σ be the maximum cost of either $\sqsupseteq$ tests or equality tests for elements of $[\sigma]$. We first observe:

1. $(X+Y+1)^3 = X^3 + Y^3 + 3X^2Y + 3XY^2 + 3X^2 + 3Y^2 + 6XY + 3X + 3Y + 1$;
2. $(X+1)^3 = X^3 + 3X^2 + 3X + 1$;
3. $(X+Y+1)^3 - (X+1)^3 - (Y+1)^3 = 3X^2Y + 3XY^2 + 6XY - 1$.

Now, $C_\iota = 1 < N^8 = \exp_2^K(N^{2^3})$ for $\iota \in \mathcal{S}$. Writing L_1 for the number of sorts in σ_1 and L_2 for the number of sorts in σ_2, we have:

$$\begin{aligned}
C_{\sigma_1 \times \sigma_2} &= C_{\sigma_1} + C_{\sigma_2} \\
&< \exp_2^K(N^{(L_1+1)^3}) + \exp_2^K(N^{(L_2+1)^3}) \text{ by the induction hypothesis} \\
&\leq \exp_2^K(N^{(L_1+1)^3} \cdot N^{(L_2+1)^3}) \text{ because both sides are at least 2} \\
&\leq \exp_2^K(N^{(L_1+1)^3 + (L_2+1)^3}) \text{ by Lemma C2} \\
&\leq \exp_2^K(N^{(L_1+L_2+1)^3}) \text{ by observation 3 above} \\
&= \exp_2^K(N^{(L+1)^3})
\end{aligned}$$

To compare $A_{\sigma_1 \Rightarrow \sigma_2}$ and $B_{\sigma_1 \Rightarrow \tau_1}$, we may for instance do the following:

- for all $(u_1, u_2) \in B$:
  - for all $(e_1, e_2) \in A$, test $e_1 = u_1$ and either $e_2 = e_2$ or $e_2 \sqsupseteq u_2$;
  - conclude failure if we didn't find a match
- in the case of $\sqsupseteq$, conclude success if we haven't concluded failure yet; in the case of $=$, also do the test in the other direction

This gives, roughly:

$$\begin{aligned}
C_{\sigma \Rightarrow \tau} &\leq 2 \cdot \mathsf{Card}([\sigma_1 \times \sigma_2]) \cdot \mathsf{Card}([\sigma_1 \times \sigma_2]) \cdot (C_{\sigma_1} + C_{\sigma_2}) \\
&\leq 2 \cdot \exp_2^K(N^L) \cdot \exp_2^K(N^L) \cdot (C_{\sigma_1} + C_{\sigma_2}) \\
&< 2 \cdot \exp_2^K(N^L) \cdot \exp_2^K(N^L) \cdot \exp_2^K(N^{(L_1+1)^3 + (L_2+1)^3}) \text{ as above} \\
&\leq 2 \cdot \exp_2^K(N^{2 \cdot L + (L_1+1)^3 + (L_2+1)^3}) \text{ by Lemma C2} \\
&\leq \exp_2^K(N^{2 \cdot L + (L_1+1)^3 + (L_2+1)^3 + 1}) \text{ because } N \geq 2 \\
&\leq \exp_2^K(N^{(L_1+L_2+1)^3}) \text{ by observation 3 above} \\
&\quad \text{because } (X + 6L_1L_2 - 1) - (2L_1 + 2L_2 + 1) \geq 0 \text{ when } L_1, L_2 \geq 1
\end{aligned}$$

Now we can derive the cardinalities of $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$ and $[\![\sigma]\!]_{\mathcal{B}}$ as in the text:

Lemma 8. *If* $1 \leq \mathsf{Card}(\mathcal{B}) < N$*, then for each* σ *with* $o(\sigma) \leq K$ *and containing* L *sorts:* $\mathsf{Card}(\langle\!|\sigma|\!\rangle_{\mathcal{B}}) < \exp_2^K(N^L)$*, where* $\mathsf{Card}(X)$ *denotes the cardinality of* X*. For* $e, u \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$*, testing* $e \sqsupseteq u$ *requires at most* $\exp_2^K(N^{(L+1)^3})$ *comparisons between elements of* $\mathcal{B}$*.*

Proof. We prove the first part by induction on the form of σ.

For $\sigma \in \mathcal{S}$, $\langle\!|\sigma|\!\rangle_{\mathcal{B}} \subseteq \mathcal{B}$ so $\mathsf{Card}(\langle\!|\sigma|\!\rangle_{\mathcal{B}}) \leq \mathsf{Card}(\mathcal{B}) < N$.

For $\sigma = \sigma_1 \times \sigma_2$ with σ_1 having L_1 sorts and σ_2 having L_2, we have

$$\begin{aligned}
\mathsf{Card}(\langle\!|\sigma_1 \times \sigma_2|\!\rangle_{\mathcal{B}}) &= \mathsf{Card}(\langle\!|\sigma_1|\!\rangle_{\mathcal{B}}) \cdot \mathsf{Card}(\langle\!|\sigma_2|\!\rangle_{\mathcal{B}}) \\
&< \exp_2^K(N^{L_1}) \cdot \exp_2^K(N^{L_2}) \\
&\leq \exp_2^K(N^{L_1} \cdot N^{L_2}) \text{ by Lemma C2} \\
&= \exp_2^K(N^{L_1+L_2}) = \exp_2^K(L)
\end{aligned}$$

For $\sigma = \sigma_1 \Rightarrow \sigma_2$ with σ_1 having L_1 sorts and σ_2 having L_2, each element of $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$ can be seen as a *total* function from $\langle\!|\sigma_1|\!\rangle_{\mathcal{B}}$ to $\langle\!|\sigma_2|\!\rangle_{\mathcal{B}} \cup \{\bot\}$. Therefore,

$$\begin{aligned}
\mathsf{Card}(\langle\!|\sigma_1 \Rightarrow \sigma_2|\!\rangle_{\mathcal{B}}) &= (\mathsf{Card}(\langle\!|\sigma_2|\!\rangle_{\mathcal{B}}) + 1)^{\mathsf{Card}(\langle\!|\sigma_1|\!\rangle_{\mathcal{B}})} \\
&\leq \exp_2^K(N^{L_2})^{\mathsf{Card}(\langle\!|\sigma_1|\!\rangle_{\mathcal{B}})} \\
&< \exp_2^K(N^{L_2})\hat{}(\exp_2^{K-1}(N^{L_1})) \\
&= 2\hat{}(\ \exp_2^{K-1}(N^{L_2}) \cdot \exp_2^{K-1}(N^{L_1})\ ) \\
&\leq 2\hat{}(\ \exp_2^{K-1}(N^{L})\ ) \text{ by Lemma C2} \\
&= \exp_2^K(N^L)
\end{aligned}$$

The second part is given by Lemma C3.

Lemma 13. *Defining the* arrow depth *of a type by:* $depth(\iota) = 0$*,* $depth(\sigma \times \tau) = \max(depth(\sigma), depth(\tau))$ *and* $depth(\sigma \Rightarrow \tau) = 1 + \max(depth(\sigma), depth(\tau))$*, then if* $1 \leq \mathsf{Card}(\mathcal{B}) < N$*,* $depth(\sigma) \leq K$ *and* σ *has* L *sorts:* $\mathsf{Card}([\![\sigma]\!]_{\mathcal{B}}) < \exp_2^K(N^L)$*. For* $e, u \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$*, testing* $e \sqsupseteq u$ *requires at most* $\exp_2^K(N^{(L+1)^3})$ *comparisons between elements of* $\mathcal{B}$*.*

Proof. We prove the first part by induction on the form of σ.

For $\sigma \in \mathcal{S}$, $[\![\sigma]\!]_{\mathcal{B}} \subseteq \mathcal{B}$ so $\mathsf{Card}([\![\sigma]\!]_{\mathcal{B}}) \leq \mathsf{Card}(\mathcal{B}) < N$.

For $\sigma = \sigma_1 \times \sigma_2$, we obtain $\mathsf{Card}([\![\sigma]\!]_{\mathcal{B}}) \leq \exp_2^K(N^L)$ in exactly the same way as in Lemma 8.

For $\sigma = \sigma_1 \Rightarrow \sigma_2$ with σ_1 having L_1 sorts and σ_2 having L_2, each element of $[\![\sigma]\!]_{\mathcal{B}}$ is a subset of $[\![\sigma_1]\!]_{\mathcal{B}} \times [\![\sigma_2]\!]_{\mathcal{B}}$; therefore,

$$\begin{aligned}
\mathsf{Card}(\langle\!|\sigma_1 \Rightarrow \sigma_2|\!\rangle_{\mathcal{B}}) &= 2\hat{}(\ \mathsf{Card}(\langle\!|\sigma_1|\!\rangle_{\mathcal{B}} \times \langle\!|\sigma_2|\!\rangle_{\mathcal{B}})\ ) \\
&\leq 2\hat{}(\ \exp_2^{K-1}(N^{L_1}) \cdot \exp_2^{K-1}(N^{L_2})\ ) \\
&\leq 2\hat{}(\ \exp_2^{K-1}(N^{L})\ ) \text{ by Lemma C2} \\
&= \exp_2^K(N^L)
\end{aligned}$$

The second part is given by Lemma C3.

D Algorithm correctness (Section 6.3)

We prove that for both Algorithm 6 and Algorithm 12: $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ if and only if b is in the set returned by the algorithm. We do this in four steps:

- we obtain some properties on (non-deterministic) extensional values and p';
- we prove that for both algorithms: if b is returned by the algorithm, then $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$;
- we prove that for Algorithm 12: if $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$, then b is returned by the algorithm;
- we adapt this proof to the deterministic setting.

This is done in the sections D.1–D.4.

D.1 Properties of extensional values and p'

We begin by giving obtaining some properties relevant to both the soundness and completeness proofs.

Lemma D4. *Fix a set $\mathcal{B}$ of data expressions, closed under taking sub-expressions. Let $\Downarrow$ be a relation, relating values v of type σ to (non-deterministic) extensional values $e \in [\![\sigma]\!]_{\mathcal{B}}$, notation $v \Downarrow e$, such that:*

- *$v \Downarrow e$ for v, e data if and only if $v = e$, and*
- *$(v, w) \Downarrow (e, u)$ if and only if both $v \Downarrow e$ and $w \Downarrow u$.*

Let $v_1 : \sigma_1, \ldots, v_k : \sigma_k$ and $e_1 \in [\![\sigma_1]\!]_{\mathcal{B}}, \ldots, e_k \in [\![\sigma_k]\!]_{\mathcal{B}}$ be such that $s_i \Downarrow e_i$ for each i, and let ρ: $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ be a clause. Then there is an environment γ such that each $v_i = \ell_i\gamma$ if and only if there is an ext-environment η such that each $e_i = \ell_i\eta$, and if both are satisfied then $\gamma(x) \Downarrow \eta(x)$ for all $x \in Var(f\ \ell_1 \cdots \ell_k)$.

Essentially, this lemma says that no matter how we associate values of a higher type to extensional values, if data and pairing are handled as expected, then matching functions in the natural way.

Proof. For ℓ a pattern of type σ, $v : \sigma$ a value and $e \in [\![\sigma]\!]_{\mathcal{B}}$ such that $v \Downarrow e$, the lemma follows easily once we prove the following by induction on ℓ:

- If $v = \ell\gamma$ for some γ, then there exists η on domain $Var(\ell)$ such that $e = \ell\eta$ and $\gamma(x) \Downarrow \eta(x)$ for all x in the domain:
  - If ℓ is a variable, then $\gamma(\ell) = v$, so choose $\eta := [\ell := e]$.
  - If ℓ is a pair (ℓ_1, ℓ_2), then $v = (v_1, v_2)$ and therefore $e = (e_1, e_2)$ with both $v_1 \Downarrow e_1$ and $v_2 \Downarrow e_2$; by the induction hypothesis, we find η_1 and η_2 on domains $Var(\ell_1)$ and $Var(\ell_2)$ respectively; we are done with $\eta := \eta_1 \cup \eta_2$.
  - If $\ell = \mathtt{c}\ \ell_1 \cdots \ell_m$ with $\mathtt{c} \in \mathcal{C}$, then v and e are both data expressions, so $v = e$; since the argument types of constructors have order 0, all $x \in Var(\ell)$ have type order 0, so we can choose $\eta(x) := \gamma(x)$ for such x.

– If $e = \ell\eta$ for some η, then there exists γ on domain $Var(\ell)$ such that $s = \ell\gamma$ and $\gamma(x) \Downarrow \eta(x)$ for x in $Var(\ell)$; this reasoning is parallel to the case above.

Lemma D5. $\sqsupseteq$ *is transitive.*

Proof. Let $e \sqsupseteq u \sqsupseteq o$ with $e, u, o \in [\![\sigma]\!]_{\mathcal{B}}$; we prove that $e \sqsupseteq o$ by induction on the form of σ.

– if $\sigma \in \mathcal{S}$, then $e = u = o$;
– if $\sigma = \sigma_1 \times \sigma_2$, then $e = (e_1, e_2)$, $v = (u_1, u_2)$ and $o = (o_1, o_2)$ with both $e_1 \sqsupseteq u_1 \sqsupseteq o_1$ and $e_2 \sqsupseteq u_2 \sqsupseteq o_2$; by the induction hypothesis indeed $e_1 \sqsupseteq o_1$ and $e_2 \sqsupseteq o_2$;
– if $\sigma = \sigma_1 \Rightarrow \sigma_2$, then we can write $e = A_\sigma$, $u = B_\sigma$ and $o = C_\sigma$ and:
  • for all $(o_1, o_2) \in C$ there exists $u_2 \sqsupseteq o_2$ such that $(o_1, u_2) \in B$;
  • for all $(o_1, u_2) \in B$ there exists $e_2 \sqsupseteq u_2$ such that $(o_1, e_2) \in A$.

  As the induction hypothesis gives $e_2 \sqsupseteq o_2$, also $e \sqsupseteq o$.

Lemma D6. *Let* $\mathtt{p}'$ *be obtained from* $\mathtt{p}$ *following step 1a in Algorithm 6. Then* $[\![\mathtt{p}]\!](d_1, \ldots, d_M) \mapsto b$ *if and only if* $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \to b$.

Proof. First observe that $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \to b$ if and only if $\mathtt{p}', [x_1 := d_1, \ldots, x_M := d_M] \vdash \mathtt{f}_1\ x_1 \cdots x_M \to b$, and that by definition $[\![\mathtt{p}]\!](d_1, \ldots, d_M) \mapsto b$ if and only if $\mathtt{p}, [x_1 := d_1, \ldots, x_M := d_M] \vdash \mathtt{f}_1\ x_1 \cdots x_M \to b$.

Second, let $fix(s)$ be the result of replacing all sub-expressions of the form $(\mathtt{if}\, b\, \mathtt{then}\, s_1\, \mathtt{else}\, s_2)\ t_1 \cdots t_n$ in s by $\mathtt{if}\, b\, \mathtt{then}\, (s_1\ t_1 \cdots t_n)\, \mathtt{else}\, (s_2\ t_1 \cdots t_n)$, and expressions $(\mathtt{choose}\ s_1 \cdots s_m)\ t_1 \cdots t_n$ by $\mathtt{choose}\ (s_1\ t_1 \cdots t_n) \cdots (s_m\ t_1 \cdots t_n)$. Then we see that:

– $\mathtt{p} \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$ iff $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$, and
– $\mathtt{p}, \gamma \vdash s \to w$ iff $\mathtt{p}, \gamma \vdash fix(s) \to w$.

By induction on the size of the derivation tree; the case where s has one of the fixable forms mostly requires the swapping of some subtrees.

D.2 Soundness of Algorithms 6 and 12

We turn to soundness. We will see that for every b in the output set of Algorithms 6 and 12 indeed $[\![\mathtt{p}]\!](d_1, \ldots, d_M)$; since each $\langle\!|\sigma|\!\rangle_{\mathcal{B}} \subseteq [\![\sigma]\!]_{\mathcal{B}}$, it suffices to prove this for the non-deterministic algorithm, as the deterministic case follows directly.

To achieve this end, we first give a definition to relate values and (non-deterministic) extensional values, and obtain two further helper results:

Definition 19. *For a value* $v : \sigma$ *and a (non-deterministic) extensional value* $e \in [\![\sigma]\!]_{\mathcal{B}}$*, we recursively define* $v \Downarrow e$ *if one of the following holds:*

– $\sigma \in \mathcal{S}$ *and* $v = e$*;*
– $\sigma = \sigma_1 \times \sigma_2$ *and* $v = (v_1, v_2)$ *and* $e = (e_1, e_2)$ *with* $v_1 \Downarrow e_1$ *and* $v_2 \Downarrow e_2$*;*

– $\sigma = \sigma_1 \Rightarrow \sigma_2$ *and* $e = A_\sigma$ *with* $A \subseteq \varphi(v) := \{(u_1, u_2) \mid u_1 \in [\![\sigma_1]\!]_{\mathcal{B}} \wedge u_2 \in [\![\sigma_2]\!]_{\mathcal{B}} \wedge$ *for all values* $w_1 : \sigma_1$ *with* $w \Downarrow u_1$ *there is some value* $w_2 : \sigma_2$ *with* $w_2 \Downarrow u_2$ *such that* $\mathsf{p}' \vdash^{\mathtt{call}} v\ w_1 \to w_2\}$.

It is easy to see that $\Downarrow$ satisfies the requirements of Lemma D4.

Lemma D7. *Assume given an environment* γ. *Let* $s : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau$, *and* $e \in [\![\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau]\!]_{\mathcal{B}}$ *be such that* $v \Downarrow e$ *for some value* v *with* $\mathsf{p}', \gamma \vdash s \to v$. *For* $1 \leq i \leq n$, *let* $t_i, v_i : \sigma_i$ *and* $u_i \in [\![\sigma_i]\!]_{\mathcal{B}}$ *be such that* $\mathsf{p}', \gamma \vdash t_i \to v_i \Downarrow u_i$. *Then for any* $o \in e(u_1, \ldots, u_n)$ *there exists* $w : \tau$ *such that* $w \Downarrow o$ *and* $\mathsf{p}', \gamma \vdash s\ t_1 \cdots t_n \to w$.

Proof. By induction on $n \geq 0$.

If $n = 0$, then $o = e$ and $\mathsf{p}', \gamma \vdash^{\mathtt{call}} s \to v$ is given; we choose $w := v$.

If $n \geq 1$, then there is some $o' := A_{\sigma_n \Rightarrow \tau} \in e(u_1, \ldots, u_{n-1})$ such that $(u_n, o) \in A$. By the induction hypothesis, there exists a value w' such that $\mathsf{p}', \gamma \vdash s\ t_1 \cdots t_{n-1} \to w' \Downarrow o'$. Since also $v_n \Downarrow u_n$, the definition of $\Downarrow$ provides a value w such that $\mathsf{p}' \vdash^{\mathtt{call}} w'\ v_n \to w \Downarrow o$. As w' is a value of higher type, it must have a form $\mathtt{f}\ w_1 \cdots w_i$, so we can apply [Appl] to obtain $\mathsf{p}', \gamma \vdash (s\ t_1 \cdots t_{n-1})\ t_n \to w$.

The following property is closely related to transitivity of $\sqsupseteq$:

Lemma D8. *For any value* $v : \sigma$ *and (non-deterministic) extensional values* $e, u \in [\![\sigma]\!]_{\mathcal{B}}$: *if* $v \Downarrow e \sqsupseteq u$ *then* $v \Downarrow u$.

Proof. By induction on the form of σ:

– if v is data, then $v = e = u$;
– if $v = (v_1, v_2)$, then $v \Downarrow e \sqsupseteq u$ implies $e = (e_1, e_2)$ and $u = (u_1, u_2)$ with $v_i \Downarrow e_i \sqsupseteq u_i$ for $i \in \{1, 2\}$, so $v_i \Downarrow u_i$ by the induction hypothesis;
– if v is a functional value, then $e = A_\sigma$ and $u = B_\sigma$, and for all $(o_1, o_2) \in B$ there exists $o_2' \sqsupseteq o_2$ such that $(o_1, o_2') \in A$; thus, for all values $w_1 \Downarrow o_1$, the property that $v \Downarrow e$ gives some w_2 such that $\mathsf{p}' \vdash^{\mathtt{call}} v\ w_1 \to w_2 \Downarrow o_2' \sqsupseteq o_2$, which by the induction hypothesis implies $w_2 \Downarrow o_2$ as well. Thus, indeed $v \Downarrow u$.

With these preparations, we are ready to tackle the soundness proof:

Lemma 15. *If Algorithm 6 or 12 returns a set* $A \cup \{b\}$, *then* $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$.

Proof. We prove the lemma by obtaining the following results:

1. Let:
   – $\mathtt{f} : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa \in \mathcal{F}$ be a defined symbol;
   – $v_1 : \sigma_1, \ldots, v_n : \sigma_n$ be values, for $1 \leq n \leq \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$;
   – $e_1 \in [\![\sigma_1]\!]_{\mathcal{B}}, \ldots, e_n \in [\![\sigma_n]\!]_{\mathcal{B}}$ be such that each $v_i \Downarrow e_i$;
   – $o \in [\![\sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa]\!]_{\mathcal{B}}$.

   If the statement $\vdash \mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$ is eventually confirmed, then we can derive $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$ for some w with $w \Downarrow o$.
2. Let:

– ρ: $\mathtt{f}\ \boldsymbol{\ell} = s$ be a clause in p';
– $t : \tau$ be a sub-expression of s;
– η be an ext-environment for ρ;
– γ be an environment such that $\gamma(x) \Downarrow \eta(x)$ for all $x \in \mathit{Var}(\mathtt{f}\ \boldsymbol{\ell})$;
– $o \in [\![\tau]\!]_{\mathcal{B}}$.

If the statement $\eta \vdash t \rightsquigarrow o$ is eventually confirmed, then we can derive $\mathsf{p}', \gamma \vdash t \to w$ for some w with $w \Downarrow o$.

This proves the lemma: if the algorithm returns b, then $\mathtt{start}\ d_1 \cdots d_M \rightsquigarrow b$ is confirmed, so $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \mapsto b$. By Lemma D6, $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$.

We prove both statements together by induction on the algorithm.

1. $\mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$ can only be confirmed in two ways:
   (2a) $n < \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$, $o = O_{\sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa}$ and for all $(e_{n+1}, u) \in O$ there is some $u' \sqsupseteq u$ such that also $\mathtt{f}\ e_1 \cdots e_{n+1} \rightsquigarrow u'$ is confirmed. By the induction hypothesis, this implies that for all such e_{n+1} and u', and for all $v_{n+1} : \sigma_{n+1}$ with $v_{n+1} \Downarrow e_{n+1}$, there exists w' with $w' \Downarrow u'$ such that $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_{n+1} \to w'$. By Lemma D8, also $w' \Downarrow u$. Thus, $O \subseteq \varphi(\mathtt{f}\ v_1 \cdots v_n)$, and $(\mathtt{f}\ v_1 \cdots v_n) \Downarrow o$. We are done choosing $w := \mathtt{f}\ v_1 \cdots v_n$, since $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to \mathtt{f}\ v_1 \cdots v_n$ by [Closure].
   (2b) $n = \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$ and, for ρ: $\mathtt{f}\ \ell_1 \cdots \ell_k = s$ the first matching clause in p' and η the matching ext-environment, $\eta \vdash s \rightsquigarrow o$ is confirmed. Following Lemma D4, there exists an environment γ on domain $\mathit{Var}(\mathtt{f}\ \ell_1 \cdots \ell_k)$ with each $\ell_j \gamma = v_j$ and $\gamma(x) \Downarrow \eta(x)$ for each x in the mutual domain. By the induction hypothesis, we can derive $\mathsf{p}', \gamma \vdash s \to w$ for some w with $w \Downarrow o$; by [Call] therefore $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$ (necessarily $n = k$).
2. $\eta \vdash t \rightsquigarrow o$ can be confirmed in seven ways:
   (1(c)i) $t \in \mathcal{V}$ and $\eta(t) \sqsupseteq o$; choosing $w = \gamma(t)$, we have $\mathsf{p}', \gamma \vdash t \to w$ by [Instance], and $w \Downarrow o$ by Lemma D8.
   (1(c)ii) $t = \mathtt{c}\ t_1 \cdots t_m$ with $\mathtt{c} \in \mathcal{C}$ and $t\eta = o$; choosing $w = t\gamma = o$, we clearly have $w \Downarrow o$ and $\mathsf{p}', \gamma \vdash t \to w$ by [Constructor].
   (2c) $t = \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3$ and either
      (2(c)i) $\eta \vdash t_1 \rightsquigarrow \mathtt{true}$ and $\eta \vdash t_2 \rightsquigarrow o$ are both confirmed; by the induction hypothesis, $\mathsf{p}', \gamma \vdash t_1 \to \mathtt{true}$ and $\mathsf{p}', \gamma \vdash t_2 \to w$ for some w with $w \Downarrow o$;
      (2(c)ii) $\eta \vdash t_1 \rightsquigarrow \mathtt{false}$ and $\eta \vdash t_3 \rightsquigarrow o$ are both confirmed; by the induction hypothesis, $\mathsf{p}', \gamma \vdash t_1 \to \mathtt{true}$ and $\mathsf{p}', \gamma \vdash t_3 \to w$ for some w with $w \Downarrow o$.

      In either case we complete with [Conditional], using [Cond-True] in the former and [Cond-False] in the latter case.
   (2d) $t = \mathtt{choose}\ t_1 \cdots t_n$ and $\eta \vdash t_i \to o$ is confirmed for some i; by the induction hypothesis, $\mathsf{p}', \gamma \vdash t_i \to w$ for a suitable w, so $\mathsf{p}', \gamma \vdash t \to w$ by [Choice].
   (2f) $t = x\ t_1 \cdots t_n$ with $x \in \mathcal{V}$ and $n > 0$, and there are $e_1, \ldots, e_n$ such that $\eta \vdash t_i \rightsquigarrow e_i$ is confirmed for all i, and $\eta(x)(e_1, \ldots, e_n) \ni o' \sqsupseteq o$ for some o'; by the induction hypothesis, there are $v_1, \ldots, v_n$ such that $\mathsf{p}', \gamma \vdash t_i \to v_i$ for all i. Since also $\mathsf{p}', \gamma \vdash x \to \gamma(x) \Downarrow \eta(x)$ by [Instance], Lemma D7

provides w such that $\mathsf{p}', \gamma \vdash x\ t_1 \cdots t_n \to w \Downarrow o'$; by Lemma D8, also $w \Downarrow o$.

(2(g)i) $t = \mathtt{f}\ t_1 \cdots t_n$ with $\mathtt{f} \in \mathcal{D}$ and $0 \leq n \leq \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$, and there are $e_1, \ldots, e_n$ such that $\eta \vdash t_i \rightsquigarrow e_i$ is confirmed for all i, and $\vdash \mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$ is marked confirmed. By the second induction hypothesis, there are $v_1, \ldots, v_n$ such that $\mathsf{p}', \gamma \vdash t_i \to v_i \Downarrow e_i$ for all i, and therefore by the first induction hypothesis, there is w such that $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w \Downarrow o$. Combining this with [Function] and n [Appl]s, we have $\mathsf{p}', \gamma \vdash \mathtt{f}\ t_1 \cdots t_n \to w$ as well.

(2(g)ii) $t = \mathtt{f}\ t_1 \cdots t_n$ with $\mathtt{f} \in \mathcal{D}$ and $n > k := \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$, and there are $e_1, \ldots, e_n$ such that, just as in the previous two cases, $\mathsf{p}', \gamma \vdash t_i \to v_i \Downarrow e_i$ for each i. Moreover, $u(e_{k+1}, \ldots, e_n) \ni o' \sqsupseteq o$ for some u with $\mathtt{f}\ e_1 \cdots e_k \rightsquigarrow u$ confirmed. As in the previous case, there exists v such that $\mathsf{p}', \gamma \vdash \mathtt{f}\ t_1 \cdots t_k \to v \Downarrow u$. Lemma D7 provides w with $\mathsf{p}', \gamma \vdash \mathtt{f}\ t_1 \cdots t_n \to w \Downarrow o'$; since $o' \sqsupseteq o$ also $w \Downarrow o$ by Lemma D8.

D.3 Completeness of Algorithm 12

Now for completeness of the algorithms, we will use induction on *positions* in a derivation. To conveniently speak of this, we we will need a form of labelling.

Definition 20. *For a given derivation tree T, we label the nodes by strings of numbers as follows: the root is labelled 0, and for a tree*

$$\frac{T_1 \qquad \ldots \qquad T_n}{\pi}$$

if node π is labelled with l, then we label each T_i with $l \cdot i$.

We say that $l > p$ if l is larger than p in the lexicographic ordering (with $l \cdot i > l$), and $l \succ p$ if $l > p$ but p is not a prefix of l.

Thus, for nodes labelled l and p, we have $l \succ p$ if l occurs to the right of p, and $l > p$ if l occurs to the right or above of p. We have $1 \succ l$ for all l in the tree.

In order to have a basis for the completeness proof that we can reuse for Algorithm 6, we transform a given derivation tree into one which uses non-deterministic extensional values. To this end, we define:

Definition 21. *Let T be a derivation tree and L the set of its labels, which must all have the form $0 \cdot l$. For any $v \in$ `Value'` (see Lemma A1) and $l \in L \cup \{1\}$, let:*

- $\psi(v, l) = v$ *if* $v \in \mathcal{B}$
- $\psi(v, l) = (\psi(v_1, l), \psi(v_2, l))$ *if* $v = (v_1, v_2)$
- *for* $\mathtt{f}\ v_1 \cdots v_n : \tau = \sigma_{n+1} \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \kappa$ *with* $m > n$, *let* $\psi(\mathtt{f}\ v_1 \cdots v_n, l) = \{(e_{n+1}, u) \mid \exists q \succ p > l$ *[the subtree with index p has a root* $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_{n+1} \to w$ *with* $\psi(w, q) = u$ *and* $e_{n+1} \sqsupseteq' \psi(v_{n+1}, p)]\}_\tau$. *In this, q is allowed to be 1 (but p is not).*

Here, $\sqsupseteq'$ is defined the same as $\sqsupseteq$, except that $A_\sigma \sqsupseteq' B_\sigma$ iff $A \supseteq B$. Note that clearly $e \sqsupseteq' u$ implies $e \sqsupseteq u$, and that $\sqsupseteq'$ is transitive by transitivity of $\supseteq$.

Thus, $\psi(v,l) \in [\![\sigma]\!]_{\mathcal{B}}$ for $v : \sigma$, but *not* $\psi(v,l) \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$. Note that $\psi(v,l) \sqsupseteq' \psi(v,p)$ if $p > l$ by transitivity of $>$. Note also that, in the derivation tree for $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \to b$, all values are in `Value'` as all d_i are in $\mathcal{B}$.

The special label 1 is used because we will make statements of the form "for all $p \succ l$ there exists $o \sqsupseteq' \psi(w,p)$ with property P": if we did not include 1 in this quantification, it would give no information about, e.g., the root of the tree.

Before proving completeness, we will use ψ to build an alternative derivation tree using the rules in Figure 9. Derivations using these rules are closely connected to Algorithm 6, and will make the completeness result straightforward to prove.

$$[\text{Constructor}]\quad \frac{}{\mathsf{p}', \eta \Vdash \mathtt{c}\ s_1 \cdots s_m \Rightarrow \mathtt{c}\ (s_1\eta) \cdots (s_m\eta)}$$

$$[\text{Pair}]\quad \frac{\mathsf{p}', \eta \Vdash s \Rightarrow o_1 \qquad \mathsf{p}', \eta \Vdash t \Rightarrow o_2}{\mathsf{p}', \eta \Vdash (s,t) \Rightarrow (o_1, o_2)}$$

$$[\text{Choice}]\quad \frac{\mathsf{p}', \eta \Vdash s_i \Rightarrow o}{\mathsf{p}', \eta \Vdash \mathtt{choose}\ s_1 \cdots s_n \Rightarrow o} \text{ for } 1 \le i \le n$$

$$[\text{Cond-True}]\quad \frac{\mathsf{p}', \eta \vdash s_1 \Rightarrow \mathtt{true} \qquad \mathsf{p}', \eta \Vdash s_2 \Rightarrow o}{\mathsf{p}', \eta \Vdash \mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3 \Rightarrow o}$$

$$[\text{Cond-False}]\quad \frac{\mathsf{p}', \eta \vdash s_1 \Rightarrow \mathtt{false} \qquad \mathsf{p}', \eta \Vdash s_3 \Rightarrow o}{\mathsf{p}', \eta \Vdash \mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3 \Rightarrow o}$$

$$[\text{Variable}]\quad \frac{\mathsf{p}', \eta \Vdash s_i \Rightarrow e_i \text{ for } 1 \le i \le n}{\mathsf{p}', \eta \Vdash x\ s_1 \cdots s_n \Rightarrow o}\ \exists o' \in \eta(x)(e_1, \ldots, e_n)[o' \sqsupseteq o]$$

$$[\text{Func}]\quad \frac{\mathsf{p}', \eta \Vdash s_i \Rightarrow e_i \text{ for } 1 \le i \le n \qquad \mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n \Rightarrow o}{\mathsf{p}', \eta \Vdash \mathtt{f}\ s_1 \cdots s_n \Rightarrow o} \text{ for } \mathtt{f} \in \mathcal{D},\ n \le \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$$

$$[\text{Applied}]\quad \frac{\mathsf{p}', \eta \Vdash s_i \Rightarrow e_1 \text{ for } 1 \le i \le n \qquad \mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_k \Rightarrow u}{\mathsf{p}', \eta \Vdash \mathtt{f}\ s_1 \cdots s_n \Rightarrow o} \begin{array}{l}\text{for } \mathtt{f} \in \mathcal{D},\\ n > \mathtt{arity}_{\mathsf{p}}(\mathtt{f}),\\ o' \in u(e_{k+1}, \ldots, e_n),\\ o' \sqsupseteq o\end{array}$$

$$[\text{Value}]\quad \frac{\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_{n+1} \Rightarrow u' \sqsupseteq u \text{ for all } (e_{n+1}, u) \in O}{\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n \Rightarrow O_\sigma} \text{ if } n < \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$$

$$[\text{Call}]\quad \frac{\mathsf{p}', \eta \Vdash s \Rightarrow o}{\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_k \Rightarrow o} \begin{array}{l}\text{if } \mathtt{f}\ \ell_1 \cdots \ell_k = s \text{ is the first clause in } \mathsf{p}' \text{ which}\\ \text{matches } \mathtt{f}\ e_1 \cdots e_k, \text{ and } \eta \text{ is the matching}\\ \text{ext-environment}\end{array}$$

Fig. 9. Alternative semantics using (non-deterministic) extensional values

We prove:

Lemma D9. *If* $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$, *then* $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \Rightarrow b$.

Proof. Given $[\![\mathtt{p}]\!](d_1, \ldots, d_M) \mapsto b$, Lemma D6 allows us to assume that $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \to b$. Let T be the derivation tree with this root (with root label 0) and L the set of its labels. We prove, by induction on l with greater labels handled first (which is well-founded because T has only finitely many subtrees):

1. If the subtree with label l has root $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$, then for all $e_1, \ldots, e_n$ such that each $e_i \sqsupseteq' \psi(v_i, l)$, and for all $p \succ l$ there exists $o \sqsupseteq' \psi(w, p)$ such that $\mathtt{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n \Rightarrow o$.
2. If the subtree with label l has root $\mathtt{p}', \gamma \vdash t \to w$ and $\eta(x) \sqsupseteq' \psi(\gamma(x), l)$ for all $x \in Var(t)$, then for all $p \succ l$ there exists $o \sqsupseteq' \psi(w, p)$ such that $\mathtt{p}', \eta \Vdash t \Rightarrow o$.

Here, for $p \succ l$ we allow $p \in L \cup \{1\}$. Therefore, in both cases, there must exist a suitable $o \sqsupseteq' \psi(w, 1)$ if w is a data expression; this o can only be w itself. The first item gives the desired result for $l = 0$, as $o \sqsupseteq' \psi(b, 1)$ implies $o = b$.

We prove both items together by induction on l, with greater labels handled first. Consider the first item. There are two cases:

- If $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$ by [Closure], then $n < \mathtt{arity}_{\mathtt{p}}(\mathtt{f})$ and $w = \mathtt{f}\ v_1 \cdots v_n$. Given $p \succ l$, let $o := \psi(w, l)$; then clearly $o \sqsupseteq' \psi(w, p)$. We must see that $\mathtt{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n \Rightarrow o$; by [Value], this is the case if for all (e_{n+1}, u) in the set underlying o we can derive $\mathtt{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_{n+1} \Rightarrow u'$ for some $u' \sqsupseteq' u$. So let (e_{n+1}, u) be in this underlying set.
  By definition of ψ, we can find $q \succ p' > l$ and v_{n+1}, w' such that the subtree with label p' has a root $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_{n+1} \to w'$ and $e_{n+1} \sqsupseteq' \psi(v_{n+1}, p')$ and $u = \psi(w', q)$. Since $p' > l$, also $e_i \sqsupseteq' \psi(v_i, p')$ for $1 \leq i \leq n$; thus, the induction hypothesis provides $u' \sqsupseteq' \psi(w', q) = u$ with $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_{n+1} \Rightarrow u'$ as required.
- If $\mathtt{p}' \vdash^{\mathtt{call}} \mathtt{f}\ v_1 \cdots v_n \to w$ by [Call], then $n = \mathtt{arity}_{\mathtt{p}}(\mathtt{f})$ and we can find a clause, say $\rho\colon \mathtt{f}\ \ell_1 \cdots \ell_n = s$ and an environment γ such that
  1. ρ is the first clause in $\mathtt{p}'$ whose right-hand side is instantiated by $\mathtt{f}\ v_1 \cdots v_n$;
  2. each $v_i = \ell_i\gamma$;
  3. $\mathtt{p}', \gamma \vdash s \to w$.

  By Lemma D4, using $v \Downarrow V$ iff $V \sqsupseteq' \psi(v, l)$, also ρ is the first clause which matches $\mathtt{f}\ e_1 \cdots e_n$, and for the matching ext-environment η, each $\eta(x) \sqsupseteq' \psi(\gamma(x), l) \sqsupseteq' \psi(\gamma(x), l \cdot 1)$. Thus using the induction hypothesis for observation 3, we find $o \sqsupseteq' \psi(w, p)$ for all $p \succ l \cdot 1$. As this includes every label p with $p \succ l$, we are done.

Now for the second claim, assume that $\mathtt{p}', \gamma \vdash t \to w$ (with label l) and that $\eta(x) \sqsupseteq' \psi(\gamma(x), l)$ for all $x \in Var(t)$; let $p \succ l$ which (**) implies $p \succ l \cdot i$ for any string i as well. Consider the form of t (taking into account that, following the transformation of $\mathtt{p}$ to $\mathtt{p}'$, we do not need to consider applications whose head is an `if then else` or `choose` statement).

- $t = (t_1, t_2)$; then we can write $w = (w_1, w_2)$ and the trees with labels $l \cdot 1$ and $l \cdot 2$ have roots $\mathtt{p}', \gamma \vdash t_1 \to w_1$ and $\mathtt{p}', \gamma \vdash t_2 \to w_2$ respectively. Using observation (**), the induction hypothesis provides o_1, o_2 such that each $\mathtt{p}', \eta \Vdash s_i \Rightarrow o_i \sqsupseteq' \psi(w_i, p)$; we are done choosing $o := (o_1, o_2)$.

- $t = \mathtt{c}\ t_1 \cdots t_m$ with $\mathtt{c} \in \mathcal{C}$; then by Lemma A1, each $s_i\gamma = b_i \in \mathcal{B}$; this implies that all $\gamma(x) \in \mathcal{B}$, so $\eta(x) = \gamma(x)$, and $\mathtt{p}', \eta \vdash t \Rightarrow o := t\eta$ by [Constructor].
- $t = \mathtt{choose}\ t_1 \cdots t_n$; then the immediate subtree is $\mathtt{p}', \gamma \vdash t_i \to w$ for some i. By observation (**), the induction hypothesis provides a suitable o, which suffices by rule [Choice] from $\Vdash$.
- $t = \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3$; then, as the immediate subtree can only be obtained by [If-True] or [If-False], we have either $\mathtt{p}', \gamma \vdash s_1 \to \mathtt{true}$ and $\mathtt{p}', \gamma \vdash s_2 \to w$, or $\mathtt{p}', \gamma \vdash s_1 \to \mathtt{false}$ and $\mathtt{p}', \gamma \vdash s_3 \to w$. Using the induction hypothesis for $p = 1$, we have $\mathtt{p}', \eta \Rightarrow \mathtt{true}$ in the first case and $\mathtt{p}', \eta \Rightarrow \mathtt{false}$ in the second. Using (**) and the induction hypothesis as before, we obtain a suitable o using the inference rule [Cond-True] or [Cond-False] of $\Vdash$.
- $t \in \mathcal{V}$, so the tree is obtained by [Instance]; choosing $o := \eta(t) \sqsupseteq' \psi(\gamma(t), l) \sqsupseteq' \psi(\gamma(t), p)$ by (**), we have $o \sqsupseteq' \psi(w, p)$ by transitivity of $\sqsupseteq'$, and $\mathtt{p}', \eta \Vdash t \Rightarrow o$ by [Variable] (as $o \in \{o\} = o()$).
- $t = x\ t_1 \cdots t_n$ with $n > 0$; then there are $w_0, \ldots, w_n$ such that the root is obtained using:
  - $\mathtt{p}', \gamma \vdash x \to \gamma(x) =: w_0$ by [Instance] with label $l \cdot 1^n$;
  - n subtrees of the form $\mathtt{p}', \gamma \vdash t_i \to v_i$ with label $l \cdot 1^{n-i} \cdot 2$ for $1 \le i \le n$;
  - n subtrees of the form $\mathtt{p}' \vdash^{\mathtt{call}} w_{i-1}\ v_i \to w_i$ with label $l \cdot 1^{n-i} \cdot 3$ for $1 \le i \le n$;
  - n uses of [Appl], each with conclusion w_i and label $l \cdot 1^{n-i}$ for $1 \le i \le n$.

  Note that here $w_n = w$. For $1 \le i \le n$ we define e_i and o_{i-1} as follows:
  - observing that $l \cdot 1^{n-i} \cdot 3 \succ l \cdot 1^{n-i} \cdot 2$, the induction hypothesis provides e_i such that $\mathtt{p}', \eta \vdash t_i \Rightarrow e_i \sqsupseteq' \psi(v_i, l \cdot 1^{n-i} \cdot 3)$
  - $o_0 := \psi(\gamma(x), l) \sqsupseteq' \psi(w_i, l \cdot 1^{n-1} \cdot 2)$;
  - for $1 < i \le n$, let $o_{i-1} := \psi(w_{i-1}, l \cdot 1^{n-i} \cdot 2)$.

  We also define $o_n := \psi(w_n, p)$. Then by definition of ψ, because $l \cdot 1^{n-i} \cdot 3 > l \cdot 1^{n-i} \cdot 2$ and the former is the label of $\mathtt{p}' \vdash^{\mathtt{call}} w_{i-1}\ v_i \to w_i$, there is an element $(e_i, \psi(w_i, q))$ in the set underlying o_{i-1} for any $q \succ l \cdot 1^{n-i} \cdot 3$. In particular, this means (e_i, o_i) is in this set, whether $i < n$ or $i = n$. Thus, by a quick induction on i we have $o_i \in \eta(x)(e_1, \ldots, e_i)$, so $\mathtt{p}', \eta \Vdash s \Rightarrow o_n = \psi(w, p)$ by [Variable].
- $t = \mathtt{f}\ t_1 \cdots t_n$ with $n \le \mathtt{arity}_{\mathtt{p}}(\mathtt{f})$; then there are subtrees $\mathtt{p}', \gamma \vdash t_i \to v_i$ labelled $l \cdot 1^{n-i} \cdot 2$ and $\mathtt{p}' \vdash \mathtt{f}\ v_1 \cdots v_n \to w$ labelled $l \cdot 3$. By the induction hypothesis, there are $e_1, \ldots, e_n$ such that $\mathtt{p}', \eta \Vdash t_i \Rightarrow e_i \sqsupseteq' \psi(v_i, l \cdot 3)$ for $1 \le i \le n$. Therefore, by the $\vdash^{\mathtt{call}}$ part of the induction hypothesis and (**), there is $o \sqsupseteq' \psi(w, p)$ such that $\mathtt{p}', \eta \Vdash \mathtt{f}\ e_1 \cdots e_n \Rightarrow o$. But then $\mathtt{p}', \eta \Vdash t \Rightarrow o$ by [Func].
- $t = (\mathtt{f}\ s_1 \cdots s_k)\ t_1 \cdots t_0$ with $k = \mathtt{arity}_{\mathtt{p}}(\mathtt{f})$ and $n > 0$; then there are subtrees:
  - $\mathtt{p}', \gamma \vdash \mathtt{f}\ s_1 \cdots s_k \to w_0$ by [Function] or [Appl], with label $l \cdot 1^n$;
  - $\mathtt{p}', \gamma \vdash t_i \to v_i$ with label $l \cdot 1^{n-i} \cdot 2$ for $1 \le i \le n$;
  - $\mathtt{p}' \vdash^{\mathtt{call}} w_{i-1}\ v_i \to w_i$ with label $l \cdot 1^{n-i} \cdot 3$ for $1 \le i \le n$.

  For some $w_0, \ldots, w_n$ with $w_n = w$. In the same way as the previous case, there exists $o_0 \sqsupseteq' \psi(w_0, l \cdot 1^{n-1} \cdot 2)$ such that $\mathtt{p}', \eta \Vdash \mathtt{f}\ s_1 \cdots s_k \Rightarrow o_0$ (as $l \cdot 1^{n-1} \cdot 2$). The remainder of this case follows the case with $t = x\ t_1 \cdots t_n$.

Now we may forget ψ and $\sqsupseteq'$, and show how derivations using $\Vdash$ are connected with the algorithm.

Lemma D10. *If* $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \Rightarrow b$*, then Algorithm 12 returns a set containing* b*. If* $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \Rightarrow b$ *has a derivation tree which only uses deterministic extensional values, then so does Algorithm 6.*

Proof. Starting with $\mathcal{B} = \mathcal{B}^{\mathsf{p}}_{d_1,\ldots,d_M}$, we show that:

1. If $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n \Rightarrow o$, then $\vdash \mathtt{f}\ e_1 \cdots e_n \rightsquigarrow o$ is eventually confirmed.
2. If $\mathsf{p}', \eta \Vdash s \Rightarrow o$, then $\eta \vdash s \rightsquigarrow o$ is eventually confirmed.

Both statements hold regardless of which algorithm is used, provided that all extensional values in the derivation tree are among those considered by the algorithm. We prove the statements together by induction on the derivation tree. For the first, there are two inference rules that might have been used:

Value $o = O_\sigma$ and for all $(e_{n+1}, u) \in O$ there exists $u' \sqsupseteq u$ such that $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_{n+1} \Rightarrow u'$ is an immediate subtree. By the induction hypothesis, each such statement $\mathtt{f}\ e_1 \cdots e_{n+1} \rightsquigarrow u'$ is confirmed, so the current statement is confirmed by step 2a.
Call Immediate by the induction hypothesis and step 2b.

For the second, suppose $\mathsf{p}', \eta \Vdash s \Rightarrow o$, and consider the inference rule used to derive this.

Constructor Immediate by step 1(c)ii.
Pair Immediate by the induction hypothesis and step 2e.
Choice Immediate by the induction hypothesis and step 2d.
Cond-True Immediate by the induction hypothesis and step 2(c)i.
Variable If $n = 0$, then $\eta(x) \sqsupseteq o$, so the statement is confirmed in step 1(c)i. Otherwise, by the induction hypothesis $\eta \vdash s_i \rightsquigarrow e_i$ is confirmed for each i and $o' \sqsupseteq o$ for some $o' \in \eta(x)(e_1, \ldots, e_n)$; the statement is confirmed in step 2f.
Func Immediate by the induction hypothesis and step 2(g)i.
Applied Immediate by the induction hypothesis and step 2(g)ii.

At this point, we have all the components for Lemma 16.

Lemma 16. *If* $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$*, then Algorithm 12 returns a set* $A \cup \{b\}$*.*

Proof. Immediate by a combination of Lemmas D9 and D10.

D.4 Completeness of Algorithm 6

Now we turn to the deterministic case. By Lemma D10, it suffices if we can find a derivation of $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \Rightarrow b$ which uses only deterministic extensional values (elements of some $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$). While the tree that we built in Lemma D9 does not have this property, we will use it to build a tree which does.

To start, we will see that the conclusions in any derivation tree are consistent, where *consistency* of two (non-deterministic) extensional values is defined as follows:

- $d \wr b$ iff $d_1 = d_2$ for $\mathcal{B} \ni d_1, d_2 : \iota$;
- $(e_1, u_1) \wr (e_2, u_2)$ iff both $e_1 \wr e_2$ and $u_1 \wr u_2$;
- $A_\sigma \wr B_\sigma$ iff for all $(e_1, u_1) \in A$ and $(e_2, u_2) \in B$: if $e_1 \wr e_2$ then $u_1 \wr u_2$.

Consistency is preserved under taking "smaller" extensional values:

Lemma D11. *If $e_1' \sqsupseteq e_1$, $e_2' \sqsupseteq e_2$ and $e_1' \wr e_2'$, then also $e_1 \wr e_2$.*

Proof. By induction on the form of e_1. If $e_1 \in \mathcal{B}$, then $e_1' = e_1 = e_2 = e_2'$. If e_1 is a pair, then so is e_1' and we use the induction hypothesis. Finally, suppose $e_1 = B_\sigma^1, e_2 = B_\sigma^2, e_1' = A_\sigma^1$ and $e_2' = A_\sigma^2$. Then for all $(u_1, o_1) \in B^1$ and $(u_2, o_2) \in B^2$, there are $o_1' \sqsupseteq o_1$ and $o_2' \sqsupseteq o_2$ such that $(u_1, o_1') \in A^1$ and $(u_2, o_2') \in A^2$. Now suppose $u_1 \wr u_2$. By consistency of e_1 and e_2, we then have $o_1' \wr o_2'$, so by the induction hypothesis, also $o_1 \wr o_2$. This gives $B_\sigma^1 \wr B_\sigma^2$, so $e_1 \wr e_2$.

Thus we see that conclusions between derivation trees are consistent:

Lemma D12. *Let T_1, T_2 be derivation trees for $\Vdash$, and let $root(T_1), root(T_2)$ denote their roots. Suppose given o, o' such that one of the following holds:*

1. *There are $\mathtt{f}, e_1, \ldots, e_n, e_1', \ldots, e_n'$ such that:*
   - *$root(T_1) = \mathtt{f}\ e_1 \cdots e_n \Rightarrow o$;*
   - *$root(T_2) = \mathtt{f}\ e_1' \cdots e_n' \Rightarrow o'$;*
   - *$e_1 \wr e_1', \ldots, e_n \wr e_n'$.*
2. *There are η, η' on the same domain and s such that:*
   - *$root(T_1) = \mathtt{p}', \eta \Vdash s \Rightarrow o$;*
   - *$root(T_2) = \mathtt{p}', \eta' \Vdash s \Rightarrow o'$;*
   - *$\eta(x) \wr \eta'(x)$ for all x occurring in s.*

   Moreover, s has no sub-expressions of the form $(\mathtt{if}\, b\, \mathtt{then}\, s_1\, \mathtt{else}\, s_2)\ t_1 \cdots t_n$ with $n > 0$.

If `choose` *does not occur in s or any clause of $\mathtt{p}'$, then $o \wr o'$.*

Proof. Both statements are proved together by induction on the form of T_1. For the first, consider n. Since $root(T_1)$ could be derived, necessarily $n \leq \mathtt{arity}_\mathtt{p}(\mathtt{f})$. There are two cases:

- $n < \mathtt{arity}_\mathtt{p}(\mathtt{f})$; both trees were derived by [Value]. Thus, we can write $o = A_\sigma$ and $o' = A_\sigma'$ and have:
  - for all $(e_{n+1}, u_1) \in A$ there is some $u_2 \sqsupseteq u_1$ such that T_1 has an immediate subtree $\mathtt{p}' \Vdash \mathtt{f}\ e_1 \cdots e_n\ e_{n+1} \Rightarrow u_2$;
  - for all $(e_{n+1}', u_1') \in A'$ there is some $u_2' \sqsupseteq u_1'$ such that T_2 has an immediate subtree $\mathtt{p}' \Vdash \mathtt{f}\ e_1' \cdots e_n'\ e_{n+1}' \Rightarrow u_2'$.

  Now let $(e_{n+1}, u_1) \in A$ and $(e_{n+1}, u_1') \in B$ be such that $e_{n+1} \wr e_{n+1}'$. Considering the two relevant subtrees, the induction hypothesis gives that $u_2 \wr u_2'$. By Lemma D11 we then obtain the required property that $u_1 \wr u_1'$.

– $n = \mathtt{arity_p(f)}$; both trees were derived by [Call]. Given that extensional values of the form A_σ can only instantiate variables (not pairs or patterns with a constructor at the head), a reasoning much like the one in Lemma D4 gives us that both conclusions are obtained by the same clause $\mathtt{f}\ \ell_1 \cdots \ell_k = s$, the first with ext-environment η and the second with η' such that each $\eta(x) \wr \eta'(x)$. Then the immediate subtrees have roots $\mathtt{p}', \eta \Vdash s \Rightarrow o$ for T_1 and $\mathtt{p}', \eta' \Vdash s \Rightarrow o'$ for T_2, and we are done by the induction hypothesis.

For the second statement, let T_1 have a root $\eta \Vdash s \Rightarrow o$ and T_2 a root $\eta' \Vdash\!\Rightarrow o'$, and assume that s does not contain any `choose` operators or if-statements at the head of an application. In addition, let $\eta(x) \wr \eta'(x)$ for all (relevant) x. Then s may have one of six forms:

– $s = \mathtt{c}\ s_1 \cdots s_m$: then $o = s\eta$ and $o' = s\eta'$; as, in this case, necessarily all variables have a type of order 0, $o = o'$ which guarantees consistency.
– $s = (s_1, s_2)$: then $o = (o_1, o_2)$ and $o' = (o'_1, o'_2)$, and by the induction hypothesis both $o_1 \wr o'_1$ and $o_2 \wr o'_2$; thus indeed $o \wr o'$.
– $s = \mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3$: since not $\mathtt{true} \wr \mathtt{false}$, either both conclusions are derived by [Cond-True] or by [Cond-False]; consistency of o and o' follows immediately by the induction hypothesis on the second subtree.
– $s = x\ s_1 \cdots s_n$; the induction hypothesis provides $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_n$ such that each $e_i \wr e'_i$ and there are $u \sqsupseteq o, u' \sqsupseteq o'$ such that $u \in \eta(x)(e_1, \ldots, e_n)$ and $u' \in \eta'(x)(e'_1, \ldots, e'_n)$. By Lemma D11, it suffices if u and u' are consistent. We prove this by induction on n:
  - if $n = 0$ then $u = \eta(x)$ and $u' = \eta'(x)$ and consistency is assumed;
  - if $n > 0$ then there are $A_\sigma \in \eta(x)(e_1, \ldots, e_{n-1})$ and $B_\sigma \in \eta'(x)(e'_1, \ldots, e'_{n-1})$ such that $(e_n, u) \in A$ and $(e'_n, u') \in B$. By the induction hypothesis, $A_\sigma \wr B_\sigma$. Since also $e_n \wr e'_n$, this implies $u \wr u'$.
– $s = \mathtt{f}\ s_1 \cdots s_n$ with $n \leq \mathtt{arity_p(f)}$; then both conclusions follow by [Func]. The immediate subtrees provide $e_1, \ldots, e_n$ and $e'_1, \ldots, e'_n$ such that, by the induction hypothesis, each $e_i \wr e'_i$, as well as a conclusion $\mathtt{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n \Rightarrow o$ in T_1 and $\mathtt{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e'_1 \cdots e'_n \Rightarrow o'$ in T_2; we can use the first part of the induction hypothesis to conclude $o \wr o'$.
– $s = \mathtt{f}\ s_1 \cdots s_n$ with $n > \mathtt{arity_p(f)}$; then both conclusions follow by [Applied]. There are $e_1, \ldots, e_n, e'_1, \ldots, e'_n$ such that, by the induction hypothesis, each $e_i \wr e'_i$. Moreover, there are u, u' such that T_1 has a subtree with root $\mathtt{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_k \Rightarrow u$ and T_2 has a subtree with root $\mathtt{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e'_1 \cdots e'_k \Rightarrow u'$, where $k = \mathtt{arity_p(f)}$; by the induction hypothesis, clearly $u \wr u'$, and since there are o_2, o'_2 such that $u(e_{k+1}, \ldots, e_n) \ni o_2 \sqsupseteq o$ and $u'(e'_{k+1}, \ldots, e'_n) \ni o'_2 \sqsupseteq o'$, the induction argument in the variable case provides $o_2 \wr o'_2$, so $o \wr o'$ by Lemma D11.

This result implies that all (non-deterministic) extensional values in the derivation tree are *internally consistent*: $o \wr o$. We can derive a functional counterpart—an element of some $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$—for any internally consistent extensional value. For an easier inductive definition, we consider multiple consistent values together.

Definition 22. *Given a* non-empty, consistent *set* X*—so* $\emptyset \neq X \subseteq [\![\sigma]\!]_{\mathcal{B}}$ *with* $e \wr u$ *for all* $e, u \in X$*—let* $\sqcup X \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$ *be defined as follows:*

- *if* $\sigma \in \mathcal{S}$*, then by consistency* X *can only have one element; we let* $\sqcup\{d\} = d$*;*
- *if* $\sigma = \sigma_1 \times \sigma_2$*, then* $\sqcup X = (\sqcup\{e \mid (e,u) \in X\}, \sqcup\{u \mid (e,u) \in X\})$
  (this is well-defined because $(e_1,u_1) \wr (e_2,u_2)$ implies both $e_1 \wr e_2$ and $u_1 \wr u_2$, so indeed the two sub-sets are consistent)
- *if* $\sigma = \sigma_1 \Rightarrow \tau$*, then* $\sqcup X = \{(e, \sqcup Y_e) \mid e \in \langle\!|\sigma|\!\rangle_{\mathcal{B}} \wedge Y_e = \bigcup_{A_\sigma \in X}\{o \mid (u,o) \in A \wedge e \sqsupseteq \sqcup\{u\}\} \wedge Y_e \neq \emptyset\}_{\sigma_1 \Rightarrow \sigma_2}$
  (this is well-defined because for every e there is only one Y_e, and Y_e is indeed consistent: if $o_1, o_2 \in Y$, then there are $A^{(1)}_\sigma, A^{(2)}_\sigma \in X$ and there exist u_1, u_2 such that $(u_1, p_1) \in A^{(1)}, (u_2, o_2) \in A^{(2)}$ and both $e \sqsupseteq u_1$ and $e \sqsupseteq u_2$; by Lemma D11—using that $e \wr e$ because $e \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$—the latter implies that $u_1 \wr u_2$, so by consistency of $A^{(1)}_\sigma$ and $A^{(2)}_\sigma$ indeed $o_1 \wr o_2$)

We make a number of observations regarding $\sqcup$.

Lemma D13. *Let* $e \in \langle\!|\sigma|\!\rangle_{\mathcal{B}}$ *and* $X = X^{(1)} \cup \cdots \cup X^{(n)} \subseteq [\![\tau]\!]_{\mathcal{B}}$ *be such that* X *is consistent,* $n > 0$ *and for all* $1 \leq i \leq n$*:* $X^{(i)}$ *is non-empty and* $e \sqsupseteq \sqcup X^{(i)}$*. Then* $e \sqsupseteq \sqcup X$*.*

Proof. If $\sigma \in \mathcal{S}$, then each $\sqcup X^{(i)} = e$; thus, $X^{(1)} = \cdots = X^{(n)} = X = \{e\}$ and $\sqcup X = e$ as well.

If $\sigma = \sigma_1 \times \sigma_2$, then $e = (e_1, e_2)$ and $\sqcup X = (\sqcup Y_1, \sqcup Y_2)$, where $Y_j = \{u_j \mid (u_1,u_2) \in X\}$ for $j \in \{1,2\}$. Let $Y^{(i)}_j = \{u_j \mid (u_1,u_2) \in X^{(i)}\}$. Then clearly each $Y_j = Y^{(1)}_j \cup \cdots \cup Y^{(n)}_j$, and $e \sqsupseteq \sqcup X^{(i)}$ implies that each $e_j \sqsupseteq \sqcup Y^{(i)}_j$. The induction hypothesis gives $e_j \sqsupseteq Y_j$ for both j.

If $\sigma = \sigma_1 \Rightarrow \sigma_2$, then write $e = A_\sigma$. Now,

- for $u \in \langle\!|\sigma_1|\!\rangle_{\mathcal{B}}$, denote $Y^{(i)}_u = \bigcup_{B_\sigma \in X^{(i)}}\{o \mid (u',o) \in B \wedge u \sqsupseteq \sqcup\{u'\}\}$;
- for $(u, \sqcup Y_u) \in \sqcup X$, we can write $Y = Y^{(1)}_u \cup \cdots \cup Y^{(N)}_u$;
- for $(u, \sqcup Y_u) \in \sqcup X$, some $Y^{(i)}_u$ must be non-empty;
- as $(u, \sqcup Y^{(i)}_u) \in \sqcup X^{(i)}$, there exists $(u,o') \in A$ with $o' \sqsupseteq \sqcup Y^{(i)}_u$;
- as there is only one o' with $(u,o') \in A$, we obtain $o' \sqsupseteq \sqcup Y^{(j)}_u$ for all non-empty $Y^{(j)}_u$;
- by the induction hypothesis, $o' \sqsupseteq \sqcup(Y^{(1)}_u \cup \cdots \cup Y^{(N)}_u) = \sqcup Y_u$.

Thus, $A_\sigma \sqsupseteq \sqcup X$ as required.

Lemma D14. *Let* $X, Y \subseteq [\![\sigma]\!]_{\mathcal{B}}$ *be non-empty consistent sets, and suppose that for every* $e \in Y$ *there is some* $e' \in X$ *such that* $e' \sqsupseteq e$*. Then* $\sqcup X \sqsupseteq \sqcup Y$*.*

Proof. By induction on the form of σ.

If $\sigma \in \mathcal{S}$ there is little to prove: X and Y contain the same single element.

If $\sigma = \sigma_1 \times \sigma_2$, then $\sqcup X = (\sqcup\{u \mid (u,o) \in X\}, \sqcup\{o \mid (u,o) \in X\})$ and $\sqcup Y = (\sqcup\{u \mid (u,o) \in Y\}, \sqcup\{o \mid (u,o) \in Y\})$. Since, for every u in $\{u \mid (u,o) \in Y\}$

there is some $(u', o') \in X$ with $u' \sqsupseteq u$ (by definition of $\sqsupseteq$ for pairs), the containment property also holds for the first sub-set; it is as easily obtained for the second. Thus we complete by the induction hypothesis and the definition of $\sqsupseteq$.

Otherwise $\sigma = \sigma_1 \Rightarrow \sigma_2$; denote $\sqcup X = A_\sigma$ and $\sqcup Y = B_\sigma$. Now, all elements of B can be written as $(u, \sqcup Y_u)$ where $Y_u = \bigcup_{D_\sigma \in Y} \{o \mid (u', o) \in D \wedge u \sqsupseteq \sqcup\{u'\}\}$, and all elements of A as $(u, \sqcup X_u)$, where $X_u = \bigcup_{C_\sigma \in X} \{o \mid (u', o) \in C \wedge u \sqsupseteq \sqcup\{u'\}\}$. Let $(u, \sqcup Y_u) \in B$; we claim that (1) X_u is non-empty, (2) $(u, \sqcup X_u) \in A$ and (3) $\sqcup X_u \sqsupseteq \sqcup Y_u$, which suffices to conclude $\sqcup X \sqsupseteq \sqcup Y$.

1. $(u, \sqcup Y_u) \in B$ gives that Y_u is non-empty, so it has at least one element o with $(u', o) \in D$ for some $D_\sigma \in Y$; by assumption, there is $C_\sigma \in X$ with $C_\sigma \sqsupseteq D_\sigma$, which implies that $(u', o') \in C_\sigma$ for some $o' \sqsupseteq o$; as $u \sqsupseteq u'$ we have $o' \in X_u$;
2. follows from (1);
3. for all $o \in Y_u$, there are u' with $u \sqsupseteq u'$ and $D_\sigma \in Y$ such that $(u', o) \in D$, and by assumption $C_\sigma \in X$ and $(u', o') \in C$ with $o' \sqsupseteq o$; as $u \sqsupseteq u'$, we have $o' \in X_u$. The induction hypothesis therefore gives $\sqcup X_u \sqsupseteq \sqcup Y_u$.

We will regularly use the following simpler variation of Lemma D14:

Lemma D15. *Let $X, Y \subseteq [\![\sigma]\!]_{\mathcal{B}}$ be non-empty sets of consistent extensional values. If $X \supseteq Y$ then $\sqcup X \sqsupseteq \sqcup Y$.*

Proof. Since $\sqsupseteq$ is reflexive, this follows immediately from Lemma D14.

Lemma D16. *Let $n \geq 0$ and suppose that:*

- $\langle\!|\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau|\!\rangle_{\mathcal{B}} \ni e \sqsupseteq \sqcup\{e^{(1)}, \ldots, e^{(m)}\}$;
- $\langle\!|\sigma_i|\!\rangle_{\mathcal{B}} \ni u_i = \sqcup\{u_i^{(1)}, \ldots, u_i^{(m)}\}$ *for* $1 \leq i \leq n$;
- $e^{(j)}(u_1^{(j)}, \ldots, u_n^{(j)}) \ni c^{(j)} \sqsupseteq o^{(j)}$ *for* $1 \leq j \leq m$;
- $o = \sqcup\{o^{(1)}, \ldots, o^{(m)}\}$.

Then there exists $c \in \langle\!|\tau|\!\rangle_{\mathcal{B}}$ such that $e(u_1, \ldots, u_n) \ni c \sqsupseteq o$.

Proof. By induction on n. First suppose that $n = 0$, so each $e^{(j)} = c^{(j)} \sqsupseteq o^{(j)}$. Then $e \sqsupseteq \sqcup\{e^{(1)}, \ldots, e^{(m)}\} \sqsupseteq \sqcup\{o^{(1)}, \ldots, o^{(m)}\} = o$ by Lemma D14, so $e() \ni e \sqsupseteq o$ by transitivity of $\sqsupseteq$.

Now let $n > 0$. For $1 \leq j \leq m$ the third observation gives $A^{(j)}$ such that $e^{(j)}(u_1^{(j)}, \ldots, u_{n-1}^{(j)}) \ni A^{(j)}_{\sigma_n \Rightarrow \tau}$ and $(u_n^{(j)}, c^{(j)}) \in A^{(j)}$. Then by the induction hypothesis, there exists $A_{\sigma \Rightarrow \tau} \in e(u_1, \ldots, u_{n-1})$ such that $A_{\sigma_n \Rightarrow \tau} \sqsupseteq \sqcup\{A^{(1)}, \ldots, A^{(m)}\}$. That is, omitting the subscript n:

- $\langle\!|\sigma \Rightarrow \tau|\!\rangle_{\mathcal{B}} \ni A_{\sigma \Rightarrow \tau} \sqsupseteq \sqcup\{A^{(1)}_{\sigma \Rightarrow \tau}, \ldots, A^{(m)}_{\sigma \Rightarrow \tau}\}$;
- $\langle\!|\sigma|\!\rangle_{\mathcal{B}} \ni u = \sqcup\{u^{(1)}, \ldots, u^{(m)}\}$;
- for $1 \leq j \leq m$: $(u^{(j)}, c^{(j)}) \in A^{(j)}$ for some $c^{(j)} \sqsupseteq o^{(j)}$;
- $o = \sqcup\{o^{(1)}, \ldots, o^{(m)}\}$.

Moreover, for every c such that $(u, c) \in A$ also $c \in e(u_1, \ldots, u_n)$; thus, we are done if we can identify such $c \sqsupseteq o$.

Let $B_u^{(j)} := \{o' \mid (u', o') \in A^{(j)} \wedge u \sqsupseteq \sqcup\{u'\}\}$ and let $B_u := B_u^{(1)} \cup \cdots \cup B_u^{(m)}$. Then we have:

- $c^{(j)} \in B_u$ for $1 \leq j \leq m$: since $(u^{(j)}, c^{(j)}) \in A^{(j)}$, and $u = \sqcup\{u^{(1)}, \ldots, u^{(m)}\} \sqsupseteq \sqcup\{u^{(j)}\}$ by Lemma D15, we have $c^{(j)} \in B_u^{(j)} \subseteq B_u$;
- since therefore $B_u \neq \emptyset$, the pair $(u, \sqcup B_u)$ occurs in the set underlying $\sqcup\{A^{(1)}, \ldots, A^{(m)}\}$;
- since $A_{\sigma\Rightarrow\tau} \sqsupseteq \sqcup\{A^{(1)}_{\sigma\Rightarrow\tau}, \ldots, A^{(m)}_{\sigma\Rightarrow\tau}\}$, there exists $c \sqsupseteq \sqcup B_u$ such that $(u, c) \in A$;
- since $A_{\sigma\Rightarrow\tau} \in \langle\!|\sigma_n \Rightarrow \tau|\!\rangle_{\mathcal{B}}$, there is only one choice for c;
- $c \sqsupseteq \sqcup B_u \sqsupseteq \sqcup\{c^{(j)}\} \sqsupseteq \sqcup\{o^{(j)}\}$ for all $1 \leq j \leq m$ by Lemmas D15 and D14;
- therefore $c \sqsupseteq o$ by Lemma D13.

All preparations done, we now turn to the proof that in a deterministic setting, it suffices to consider deterministic extensional values.

Lemma D17. *If* p *is deterministic and* $\mathsf{p}' \vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \Rightarrow b$*, then this can be derived using a* functional tree*: a derivation tree where all extensional values are in some* $\langle\!|\sigma|\!\rangle_{\mathcal{B}}$*.*

Proof. Let p' be deterministic (so also p' is). We prove the following statements:

1. Suppose $T_1, \ldots, T_N$ are derivation trees, and there are fixed $\mathtt{f}, n$ such that each tree T_j has a root $\mathsf{p}' \Vdash \mathtt{f}\ e_1^{(j)} \cdots e_n^{(j)} \Rightarrow o^{(j)}$, where $e_i^{(j)} \wr e_i^{(k)}$ for all $1 \leq j, k \leq N$ and $1 \leq i \leq n$. Let $e_1, \ldots, e_n$ be deterministic extensional values such that $e_i \sqsupseteq \sqcup\{e_i^{(j)} \mid 1 \leq j \leq N\}$ for $1 \leq i \leq n$. We can derive $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n \Rightarrow o := \sqcup\{o^{(j)} \mid 1 \leq j \leq N\}$ by a functional tree.
2. Suppose $T_1, \ldots, T_N$ are derivation trees, and there is some fixed s such that each tree T_j has a root $\mathsf{p}', \eta^{(j)} \Vdash s \Rightarrow o^{(j)}$, where $\eta^{(j)}(x) \wr \eta^{(k)}(x)$ for all $1 \leq j, k \leq N$ and variables x in the shared domain. Let η be an ext-environment on the same domain mapping to functional extensional values such that $\eta(x) \sqsupseteq \sqcup\{\eta^{(j)}(x) \mid 1 \leq j \leq N\}$ for all x. Writing $o := \sqcup\{o^{(j)} \mid 1 \leq j \leq N\}$, we can derive $\mathsf{p}', \eta \Vdash s \Rightarrow o$ by a functional tree. (We assume that no sub-expression of s has an if-then-else at the head of an application.)

The first of these claims proves the lemma for $N = 1$: clearly data expressions are self-consistent, and the only $o \sqsupseteq b = \sqcup\{b\}$ is b itself, so the claim says that the root can be derived using a functional tree.

We prove the claims together by a shared induction on the maximum depth of any T_j. We start with the first claim. There are two cases:

- $n = \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$: then for each T_j there is a clause $\rho_j : \mathtt{f}\ \ell_1 \cdots \ell_n = s$ which imposes $\eta^{(j)}$ such that the immediate subtree of T_j is $\mathsf{p}', \eta^{(j)} \Vdash o^{(j)}$.
  Now, let $\ell : \sigma$ be a linear pattern, η an ext-environment and $e, u \in [\![\sigma]\!]_{\mathcal{B}}$ be such that $e \wr u$ and $\ell\eta = e$. By a simple induction on the form of ℓ we find an ext-environment η' on domain $Var(\ell)$ such that $\ell\eta' = u$ and $\eta(x) \wr \eta'(x)$.
  Thus, the first matching clause ρ_j is necessarily the same for all T_j, and we have $\eta^{(j)}(x) \wr \eta^{(k)}(x)$ for all j, k, x. For all $1 \leq i \leq n$ and $1 \leq j \leq N$, we have $e_i^{(j)} = \ell_i\eta^{(j)}$. Another simple induction on ℓ_i proves that we can find η with each $\eta(x) \sqsupseteq \sqcup\{\eta^{(j)}(x) \mid 1 \leq j \leq N\}$ such that $e_i = \ell_i\eta$.
  The induction hypothesis gives $\mathsf{p}', \eta \Vdash\!\Rightarrow o$, so $\mathtt{f}\ e_1 \cdots e_n \Rightarrow o$ by [Call].

– $n < \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$: each of the trees T_j is derived by [Value]. Write $o = O_\sigma$ and $o^{(j)} = O_\sigma^{(j)}$ for $1 \leq j \leq N$. We are done by [Value] if $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n\ e_{n+1} \Rightarrow o'$ for all $(e_{n+1}, o') \in O$.
Since $o = \sqcup\{o^{(1)}, \ldots, o^{(N)}\}$, we can write $o' = \sqcup Y_{e_{n+1}}$ and identify a non-empty set $Pairs_{e_{n+1}} = \{(e,u) \in O^{(1)} \cup \cdots \cup O^{(N)} \mid e_{n+1} \sqsupseteq \sqcup\{e\}\}$ such that $Y_{e_{n+1}} = \{u \mid (e,u) \in Pairs_{e_{n+1}}\}$.
For each element (e,u) of $Pairs_C$, some T_j has a subtree with root $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1^{(j)} \cdots e_n^{(j)}\ e \Rightarrow u$. Let $Trees_{e_{n+1}}$ be the corresponding set of trees, and note that all trees in $Trees_{e_{n+1}}$ have a strictly smaller depth than the T_j they originate from, so certainly smaller than the maximum depth.
Now, for $1 \leq i \leq n+1$, let $Args_i := \{$ argument i of the root of $T \mid T \in Trees_{e_{n+1}}\}$. We observe that:
  - for $1 \leq i \leq n$: $e_i \sqsupseteq \sqcup Args_i$: we have $Args_i \subseteq \{e_i^{(1)}, \ldots, e_i^{(N)}\}$, so by Lemma D15, $e_i \sqsupseteq \sqcup\{e_i^{(j)} \mid 1 \leq j \leq N\} \sqsupseteq \sqcup Args_i$, which suffices by transitivity (Lemma D5);
  - $e_{n+1} \sqsupseteq \sqcup Args_{j+1}$: $e_{n+1} \sqsupseteq \{e\}$ for all $e \in Args_{j+1}$, so this is given by Lemma D13.
  - $o' = \sqcup Y_{e_{n+1}} = \sqcup\{$right-hand sides of the roots of $Trees_{e_{n+1}}\}$.

  Therefore $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n\ e_{n+1} \Rightarrow o'$ by the induction hypothesis as required.

For the second case, consider the form of s.

– $s = \mathtt{c}\ s_1 \cdots s_m$ with $\mathtt{c} \in \mathcal{C}$: then each $o^{(i)} = s\eta^{(i)} \in \mathcal{B}$, so $o = o^{(1)} = \cdots = o^{(N)}$ and—since the variables in s all have order 0—we have $\eta(x) = \eta^{(1)}(x) = \cdots = \eta^{(N)}(x)$ for all relevant x. Thus also $o = s\eta$ and we complete by [Constructor].
– $s = (s_1, s_2)$; each tree T_j has two immediate subtrees: one with root $\mathsf{p}', \eta^{(j)} \Vdash s_1 \Rightarrow o_1^{(j)}$ and one with root $\mathsf{p}', \eta^{(j)} \Vdash s_2 \Rightarrow o_2^{(j)}$, where $o^{(j)} = (o_1^{(j)}, o_2^{(j)})$.
We can write $o = (o_1, o_2)$ where $o_1 = \sqcup\{o_1^{(j)} \mid 1 \leq j \leq N\}$ and $o_2 = \sqcup\{o_2^{(j)} \mid 1 \leq j \leq N\}$, and as the induction hypothesis for both subtrees gives $\mathsf{p}', \eta \Vdash s_1 \Rightarrow o_1$ and $\mathsf{p}', \eta \Vdash s_2 \Rightarrow o_2$ respectively, we conclude $\mathsf{p}', \eta \Vdash s \Rightarrow o$ by [Pair].
– $s = \mathtt{if}\ s_1\ \mathtt{then}\ s_2\ \mathtt{else}\ s_3$: for each tree T_j, the first subtree has the form $\mathsf{p}', \eta^{(j)} \Vdash s_1 \Rightarrow \mathtt{true}$ or $\mathsf{p}', \eta^{(j)} \Vdash s_1 \Rightarrow \mathtt{false}$; by consistency of derivation trees (Lemma D12), either $\mathtt{true}$ or $\mathtt{false}$ is chosen for *all* these subtrees. We assume the former; the latter case is symmetric.
By the induction hypothesis for this first subtree, $\mathsf{p}', \eta \Vdash s_1 \Rightarrow \mathtt{true} = \sqcup\{\mathtt{true}, \ldots, \mathtt{true}\}$ as well.
The second immediate subtree of all trees T_j has a root of the form $\mathsf{p}', \eta^{(j)} \Vdash s_2 \Rightarrow o^{(j)}$. By the induction hypothesis for this second subtree, $\mathsf{p}', \eta \Vdash s_2 \Rightarrow o$. Thus we conclude $\mathsf{p}', \eta \Vdash s \Rightarrow o$ by [Cond-True].
– $s = x\ s_1 \cdots s_n$ with $x \in \mathcal{V}$: each of the trees T_j has n subtrees of the form $\mathsf{p}', \eta^{(j)} \Vdash s_1 \Rightarrow e_i^{(j)}$ (for $1 \leq i \leq n$); by the induction hypothesis, we have $\mathsf{p}', \eta \Vdash s_i \Rightarrow e_i$, where $e_i = \sqcup\{e_i^{(j)} \mid 1 \leq j \leq N\}$. But then:
  - $\langle\!|\sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau|\!\rangle_{\mathcal{B}} \ni \eta(x) \sqsupseteq \sqcup\{\eta^{(1)}(x), \ldots, \eta^{(N)}(x)\}$;

- $\langle\!|\sigma_i|\!\rangle_{\mathcal{B}} \ni e_i = \sqcup\{e_i^{(1)}, \ldots, e_i^{(N)}\}$ for $1 \leq i \leq n$;
- there are $u^{(j)}$ such that $\eta^{(j)}(e_1^{(j)}, \ldots, e_n^{(j)}) \ni u^{(j)} \sqsupseteq o^{(j)}$ for $1 \leq j \leq N$;
- $o = \sqcup\{o^{(1)}, \ldots, o^{(N)}\}$.

By Lemma D16, there exists $u \in o(e_1, \ldots, e_n)$ such that $u \sqsupseteq o$. We conclude $\mathsf{p}', \eta \Vdash s \Rightarrow o$ by [Variable].

– $s = \mathtt{f}\ s_1 \cdots s_n$ with $n \leq \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$: then necessarily each $\mathsf{p}', \eta^{(j)} \Vdash s \Rightarrow o^{(j)}$ follows by [Func]. Thus, for $1 \leq j \leq N$ there are $e_1^{(j)}, \ldots, e_n^{(j)}$ such that:
  - $\mathsf{p}', \eta^{(j)} \Vdash s_i \Rightarrow e_i^{(j)}$ for $1 \leq i \leq n$ and
  - $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1^{(j)} \cdots e_n^{(j)} \Rightarrow o^{(j)}$.

  Now, clearly each set $\{e_i^{(j)} \mid 1 \leq j \leq N\}$ is consistent by the simple fact that there are derivation trees for them: this is the result of Lemma D12. Defining $e_i := \sqcup\{e_i^{(1)}, \ldots, e_i^{(N)}\}$ for $1 \leq j \leq N$, the induction hypothesis gives that $\mathsf{p}', \eta \Vdash s_i \Rightarrow e_i$, and that $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_n \Rightarrow o$, all by functional trees. We complete with [Func].

– $s = \mathtt{f}\ s_1 \cdots s_n$ with $n > k := \mathtt{arity}_{\mathsf{p}}(\mathtt{f})$: then there are $e_1^{(j)}, \ldots, e_n^{(j)}, u^{(j)}, c^{(j)}$ such that for all $1 \leq j \leq N$:
  - tree T_j has subtrees $\mathsf{p}', \eta^{(j)} \Vdash s_i \Rightarrow e_i^{(j)}$ for $1 \leq i \leq n$;
  - tree T_j has a subtree $\Vdash^{\mathtt{call}} \mathtt{f}\ e_1^{(j)} \cdots e_k^{(j)} \Rightarrow u^{(j)}$;
  - $u^{(j)}(e_{k+1}^{(j)}, \ldots, e_n^{(j)}) \ni c^{(j)} \sqsupseteq o^{(i)}$.

  Therefore, by the induction hypothesis and Lemma D16, we can identify $e_1, \ldots, e_n, u, c$ such that:
  - $e_i = \sqcup\{e_i^{(1)}, \ldots, e_i^{(N)}\}$ and $\mathsf{p}', \eta \Vdash s_i \Rightarrow e_i$ for $1 \leq i \leq n$;
  - $\mathsf{p}' \Vdash^{\mathtt{call}} \mathtt{f}\ e_1 \cdots e_k \Rightarrow u = \sqcup\{u^{(1)}, \ldots, u^{(N)}\}$;
  - $u(e_{k+1}, \ldots, e_n) \ni c \sqsupseteq o$.

  Therefore $\mathsf{p}', \eta \Vdash s \Rightarrow o$ by [Apply].

With this, the one remaining lemma—completeness of Algorithm 6—is trivial.

Lemma D18. *If* $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ *and* p *is deterministic, then Algorithm 12 returns a set* $A \cup \{b\}$.

Proof. Suppose $[\![\mathsf{p}]\!](d_1, \ldots, d_M) \mapsto b$ for a deterministic program p. By Lemma D9, we can derive $\Vdash^{\mathtt{call}} \mathtt{start}\ d_1 \cdots d_M \Rightarrow b$. By Lemma D17, there is one which only uses deterministic extensional values. By Lemma D10, Algorithm 6 therefore returns a set containing b.