Cyclic Implicit Complexity

Gianluca Curzi
University of Birmingham
Birmingham, UK
g.curzi@bham.ac.uk

Anupam Das
University of Birmingham
Birmingham, UK
A.Das@bham.ac.uk

ABSTRACT

Circular (or *cyclic*) proofs have received increasing attention in recent years, and have been proposed as an alternative setting for studying (co)inductive reasoning. In particular, now several type systems based on circular reasoning have been proposed. However, little is known about the complexity theoretic aspects of circular proofs, which exhibit sophisticated loop structures atypical of more common 'recursion schemes'.

This paper attempts to bridge the gap between circular proofs and *implicit computational complexity* (ICC). Namely we introduce a circular proof system based on Bellantoni and Cook's famous safe-normal function algebra, and we identify proof theoretical constraints, inspired by ICC, to characterise the polynomial-time and elementary computable functions. Along the way we introduce new recursion theoretic implicit characterisations of these classes that may be of interest in their own right.

CCS CONCEPTS

• **Theory of computation** → **Complexity theory and logic**; **Proof theory**.

KEYWORDS

Cyclic proofs, implicit complexity, function algebras, safe recursion, higher-order complexity

ACM Reference Format:
Gianluca Curzi and Anupam Das. 2022. Cyclic Implicit Complexity. In *37th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (LICS '22), August 2–5, 2022, Haifa, Israel.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3531130.3533340

1 INTRODUCTION

Formal proofs are traditionally seen as finite objects modelling logical or mathematical reasoning. *Non-wellfounded* proofs are a generalisation of this notion to an infinitary (but finitely branching) setting, in which consistency is maintained by a standard global condition: the 'progressing' criterion. Special attention is devoted to *regular* (or *circular* or *cyclic*) proofs, i.e. those non-wellfounded proofs having only finitely many distinct sub-proofs, and which may thus be represented by finite (possibly cyclic) directed graphs. For such proofs the progressing criterion may be effectively decided by reduction to the universality problem for Büchi automata.

Non-wellfounded proofs have been employed to reason about the modal μ-calculus and fixed-point logics [16, 27], first-order inductive definitions [8], Kleene algebra [13, 14], linear logic [2, 17], arithmetic [7, 11, 30], system T [10, 12, 22], and continuous cut-elimination [1, 18, 26]. In particular, [22] and [10, 12] investigate the computational expressivity of circular proofs, with respect to the proofs-as-programs paradigm, in the setting of higher-order primitive recursion.

However little is known about the complexity-theoretic aspects of circular proofs. Usual termination arguments for circularly typed programs are nonconstructive, proceeding by contradiction and using a non-recursive 'totality' oracle (cf. [10, 12, 22]). As a result, these arguments are not appropriate for delivering feasible complexity bounds (cf. [11]).

The present paper aims to bridge this gap by proposing a circular foundation for Implicit Computational Complexity (ICC), a branch of computational complexity studying machine-free characterisations of complexity classes. Our starting point is Bellantoni and Cook's famous function algebra B characterising the polynomial time computable functions (**FPTIME**) using *safe recursion* [5]. The prevailing idea behind safe recursion (and its predecessor, 'ramified' recursion [23]) is to organise data into strata in a way that prevents recursive calls being substituted into recursive parameters of previously defined functions. This approach has been successfully employed to give resource-bound-free characterisations of polynomial-time [5], levels of the polynomial-time hierarchy [4], and levels of the Grzegorczyk hierarchy [31], and has been extended to higher-order settings too [19, 24].

Circular systems for implicit complexity

Construing B as a type system, we consider non-wellfounded proofs, or *coderivations*, generated by its recursion-free subsystem B^-. The circular proof system CNB is then obtained by considering the regular and progressing coderivations of B^- which satisfy a further criterion, *safety*, motivated by the eponymous notion from Bellantoni and Cook's work (cf. also 'ramification' in Leivant's work [23]). On the one hand, regularity and progressiveness ensure that coderivations of CNB define total computable functions; on the other hand, the latter criterion ensures that the corresponding equational programs are 'safe': the recursive call of a function is never substituted into the recursive parameter of a step function.

Despite CNB having only ground types, it is able to define equational programs that nest recursive calls, a property that typically arises only in higher-order recursion (cf., e.g., [19, 24]). In fact, we show that this system defines precisely the elementary computable functions (**FELEMENTARY**). Let us point out that the capacity of circular proofs to simulate some higher-order behaviour reflects an

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
LICS '22, August 2–5, 2022, Haifa, Israel
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9351-5/22/08…$15.00
https://doi.org/10.1145/3531130.3533340

emerging pattern in the literature. For instance in [10, 12] is shown that the number-theoretic functions definable by type level n proofs of a circular version of system T are exactly those definable by type level $n+1$ proofs of T.

In the setting of ICC, Hofmann [19] and Leivant [24] already observed that higher-order safe recursion mechanisms can be used to characterise **FELEMENTARY**. In particular, in [19], Hofmann presents the type system SLR (Safe Linear Recursion) as a higher-order version of B imposing a 'linearity' restriction on the higher-order safe recursion operator. He shows that this system defines just the polynomial-time computable functions on natural numbers (**FPTIME**).

Inspired by [19], we too introduce a linearity requirement for CNB that is able to control the interplay between cycles and the cut rule, called the *left-leaning criterion*. The resulting circular proof system is called CB, which we show defines precisely **FPTIME**.

Function algebras for safe nested recursion and safe recursion along well-founded relations

As well as introducing the circular systems CB and CNB just mentioned, we also develop novel function algebras for **FPTIME** and **FELEMENTARY** that allow us to prove the aforementioned complexity characterisations via a 'sandwich' technique, cf. Figure 1. This constitutes a novel (and more direct) approach to reducing circularity to recursion, that crucially takes advantage of safety.

We give a relativised formulation of B, i.e. with *oracles*, that allows us to define a form of safe *nested* recursion. The resulting function algebra is called NB and is comparable to the type 2 fragment of Leivant's extension of ramified recursion to finite types [24]. The algebras B and NB will serve as lower bounds for CB and CNB respectively.

The relativised formulation of function algebras also admits a robust notion of (safe) recursion along a well-founded relation. We identify a particular well-founded preorder $\subset$ ('permutation of prefixes') whose corresponding safe recursor induces algebras $\mathsf{B}^{\subset}$ and $\mathsf{NB}^{\subset}$ that will serve as upper bounds for CB and CNB respectively.

Outline

This paper is structured as follows. Section 2 presents B as a proof system. In Section 3 we define non-wellfounded proofs and their semantics and present the circular proof systems CNB and CB. In Section 4 we present the function algebras NB, $\mathsf{B}^{\subset}$ and $\mathsf{NB}^{\subset}$. Section 5 shows that $\mathsf{B}^{\subset}$ captures **FPTIME** (Corollary 40) and that both NB and $\mathsf{NB}^{\subset}$ capture **FELEMENTARY** (Corollary 42). These results require a delicate Bounding Lemma (Lemma 38) and an encoding of the elementary functions into NB (Theorem 41). In Section 6 we show that any function definable in B is also definable in CB (Theorem 43), and that any function definable in NB is also definable in CNB (Theorem 46). Finally, in Section 6.2, we present a translation of CNB into $\mathsf{NB}^{\subset}$ that maps CB coderivations into $\mathsf{B}^{\subset}$ functions (Lemma 47), by reducing circularity to a form of simultaneous recursion on $\subset$.

The main results of this paper are summarised in Figure 1. Full proofs of our results, as well as further discussion and examples, can be found in an extended version of this paper available [9].

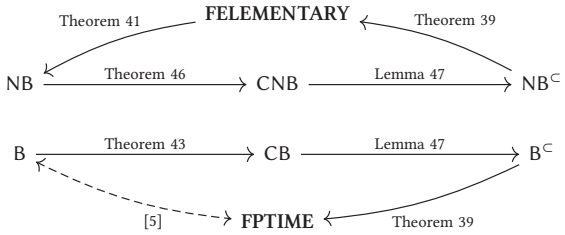


Figure 1: Summary of the main results of the paper, where $\rightarrow$ indicates an inclusion ($\subseteq$) of function classes.

2 PRELIMINARIES

Bellantoni and Cook introduced in [5] an algebra of functions based on a simple two-sorted structure. This idea was itself inspired by Leivant's characterisations, one of the founding works in Implicit Computational Complexity (ICC) [23]. The resulting 'tiering' of the underlying sorts has been a recurring theme in the ICC literature since, and so it is this structure that shall form the basis of the systems we consider in this work.

We consider functions on the natural numbers with inputs distinguished into two sorts: 'safe' and 'normal'. We shall write functions explicitly indicating inputs, namely writing $f(x_1, \ldots, x_m; y_1, \ldots, y_n)$ when f takes m normal inputs $\vec{x}$ and n safe inputs $\vec{y}$. Both sorts vary over the natural numbers, but their roles will be distinguished by the closure operations of the algebras and rules of the systems we consider.

Throughout this work, we write $|x|$ for the length of x (in binary notation), and if $\vec{x} = x_1, \ldots, x_m$ we write $|\vec{x}|$ for the list $|x_1|, \ldots, |x_m|$.

2.1 Bellantoni-Cook characterisation of FPTIME

We first recall Bellantoni-Cook in its original guise.

Definition 1 (Bellantoni-Cook algebra)**.** B is defined as the smallest class of (two-sorted) functions containing,

- $0(;) := 0 \in \mathbb{N}$.
- $\pi^{m;n}_{j;}(x_1, \ldots, x_m; y_1, \ldots, y_n) := x_j$, for $1 \leq j \leq m$.
- $\pi^{m;n}_{;j}(x_1, \ldots, x_m; y_1, \ldots, y_n) := y_j$, for $1 \leq j \leq n$.
- $\mathsf{s}_i(;x) := 2x + i$, for $i \in \{0, 1\}$.
- $\mathsf{p}(;x) := \lfloor x/2 \rfloor$.
- $\mathsf{cond}(;w,x,y,z) := \begin{cases} x & w = 0 \\ y & w = 0 \mod 2, w \neq 0 \\ z & w = 1 \mod 2 \end{cases}$

and closed under the following:

- (Safe composition)
  - If $f(\vec{x}, x; \vec{y})$, $g(\vec{x};) \in \mathsf{B}$ then $f(\vec{x}, g(\vec{x};); \vec{y}) \in \mathsf{B}$.
  - If $f(\vec{x}; \vec{y}, y)$, $g(\vec{x}; \vec{y}) \in \mathsf{B}$ then $f(\vec{x}; \vec{y}, g(\vec{x}; \vec{y})) \in \mathsf{B}$.
- (Safe recursion on notation) If $g(\vec{x}; \vec{y})$, $h_i(x, \vec{x}; \vec{y}, y) \in \mathsf{B}$ for $i = 0, 1$ then so is $f(x, \vec{x}; \vec{y})$ given by:

$$\begin{aligned} f(0, \vec{x}; \vec{y}) &:= g(\vec{x}; \vec{y}) \\ f(\mathsf{s}_0 x, \vec{x}; \vec{y}) &:= h_0(x, \vec{x}; \vec{y}, f(x, \vec{x}; \vec{y})) \text{ if } x \neq 0 \\ f(\mathsf{s}_1 x, \vec{x}; \vec{y}) &:= h_1(x, \vec{x}; \vec{y}, f(x, \vec{x}; \vec{y})) \end{aligned}$$

Intuitively, in a function $f(\vec{x};\vec{y}) \in \mathsf{B}$ only the normal arguments $\vec{x}$ can be used as recursive parameters. The idea behind safe recursion is that recursive calls can only appear in safe position, and hence they can never be used as recursive parameters of other previously defined functions. Safe composition preserves the distinction between normal and safe arguments by requiring that, when composing along a normal parameter, the pre-composing function has no safe parameter at all. As a result, we can effectively substitute a normal parameter into a safe position but not vice-versa.

Writing **FPTIME** for the class of functions computable in polynomial-time, the main result of Bellantoni and Cook is:

Theorem 2 ([5]). *$f(\vec{x};) \in \mathsf{B}$ if and only if $f(\vec{x}) \in$ **FPTIME**.*

2.2 Proof theoretic presentation of Bellantoni-Cook

We shall work with a formulation of Bellantoni and Cook's algebra as a type system with modalities to distinguish the two sorts (similarly to [19]). In order to facilitate the definition of the circular system that we present later, we here work with sequent-style typing derivations.

We only consider *types* (or *formulas*) N ('safe') and $\Box N$ ('normal') which intuitively vary over the natural numbers. We write A, B, etc. to vary over types.

A *sequent* is an expression $\Gamma \Rightarrow A$, where Γ is a list of types (called the *context* or *antecedent*) and A is a type (called the *succedent*). For a list of types $\Gamma = N, \overset{k}{\ldots}, N$, we write $\Box\Gamma$ for $\Box N, \overset{k}{\ldots}, \Box N$.

In what follows, we shall essentially identify B with the $S4$-style type system in Figure 2. The colouring of type occurrences may be ignored for now, they will become relevant in the next section. Derivations in this system are simply called B-*derivations*, and will be denoted $\mathcal{D}, \mathcal{E}, \ldots$. We write $\mathcal{D} : \Gamma \Rightarrow A$ if the derivation $\mathcal{D}$ has end-sequent $\Gamma \Rightarrow A$. We may write $\mathcal{D} = \mathsf{r}(\mathcal{D}_1, \ldots, \mathcal{D}_n)$ (for $n \leq 3$) if r is the last inference step of $\mathcal{D}$ whose immediate subderivations are, respectively, $\mathcal{D}_1, \ldots, \mathcal{D}_n$. Unless otherwise indicated, we assume that the r-instance is as typeset in Figure 2.

Convention 3 (Left normal, right safe)**.** In what follows, we assume that sequents have shape $\Box N, \ldots, \Box N, N, \ldots, N \Rightarrow A$, i.e. in the LHS all $\Box N$ occurrences are placed before all N occurrences. Note that this invariant is maintained by the typing rules of Figure 2, as long as we insist that $A = B$ in the exchange rule e. This effectively means that exchange steps have one of the following two forms:

$$\mathsf{e}_N \frac{\Gamma, N, N, \vec{N}' \Rightarrow A}{\Gamma, N, N, \vec{N}' \Rightarrow A} \qquad \mathsf{e}_\Box \frac{\Box\vec{N}, \Box N, \Box N, \Gamma' \Rightarrow A}{\Box\vec{N}, \Box N, \Box N, \Gamma' \Rightarrow A}$$

Let us point out that this convention does not change the class of definable functions with only normal inputs, under the semantics we are about to give.

We construe the system of B-derivations as a class of safe-normal functions by identifying each rule instance as an operation on safe-normal functions. Formally:

Definition 4 (Semantics)**.** Given a B-derivation $\mathcal{D}$ with conclusion $\Box N, \overset{m}{\ldots}, \Box N, N, \overset{n}{\ldots}, N \Rightarrow A$ we define a two-sorted function $f_\mathcal{D}(x_1, \ldots, x_m; y_1, \ldots, y_n)$ by induction on the structure of $\mathcal{D}$ as follows (all rules as typeset in Figure 2):

- If $\mathcal{D} = \mathsf{id}$ then $f_\mathcal{D}(; y) := y$.
- If $\mathcal{D} = \mathsf{w}_N(\mathcal{D}_0)$ then $f_\mathcal{D}(\vec{x};\vec{y}, y) := f_{\mathcal{D}_0}(\vec{x};\vec{y})$.
- If $\mathcal{D} = \mathsf{w}_\Box(\mathcal{D}_0)$ then $f_\mathcal{D}(x, \vec{x};\vec{y}) := f_{\mathcal{D}_0}(\vec{x};\vec{y})$.
- If $\mathcal{D} = \mathsf{e}_N(\mathcal{D}_0)$ then $f_\mathcal{D}(\vec{x};\vec{y}, y, y', \vec{y}') := f_{\mathcal{D}_0}(\vec{x};\vec{y}, y', y, \vec{y}')$.
- If $\mathcal{D} = \mathsf{e}_\Box(\mathcal{D}_0)$ then $f_\mathcal{D}(\vec{x}, x, x', \vec{x}';\vec{y}) := f_{\mathcal{D}_0}(\vec{x}, x', x, \vec{x}';\vec{y})$.
- If $\mathcal{D} = \Box_l(\mathcal{D}_0)$ then $f_\mathcal{D}(x, \vec{x};\vec{y}) := f_{\mathcal{D}_0}(\vec{x};\vec{y}, x)$.
- If $\mathcal{D} = \Box_r(\mathcal{D}_0)$ then $f_\mathcal{D}(\vec{x};) := f_{\mathcal{D}_0}(\vec{x};)$.
- If $\mathcal{D} = 0$ then $f_\mathcal{D}(;) := 0$.
- If $\mathcal{D} = \mathsf{s}_0(\mathcal{D}_0)$ then $f_\mathcal{D}(\vec{x};\vec{y}) := \mathsf{s}_0(; f_{\mathcal{D}_0}(\vec{x};\vec{y}))$.
- If $\mathcal{D} = \mathsf{s}_1(\mathcal{D}_0)$ then $f_\mathcal{D}(\vec{x};\vec{y}) := \mathsf{s}_1(; f_{\mathcal{D}_0}(\vec{x};\vec{y}))$.
- If $\mathcal{D} = \mathsf{cut}_N(\mathcal{D}_0, \mathcal{D}_1)$ then $f_\mathcal{D}(\vec{x};\vec{y}) := f_{\mathcal{D}_1}(\vec{x};\vec{y}, f_{\mathcal{D}_0}(\vec{x};\vec{y}))$.
- If $\mathcal{D} = \mathsf{cut}_\Box(\mathcal{D}_0, \mathcal{D}_1)$ then $f_\mathcal{D}(\vec{x};\vec{y}) := f_{\mathcal{D}_1}(f_{\mathcal{D}_0}(\vec{x};\vec{y}), \vec{x};\vec{y})$.
- If $\mathcal{D} = \mathsf{cond}_N(\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2)$ then:

$$\begin{aligned} f_\mathcal{D}(\vec{x};\vec{y}, 0) &:= f_{\mathcal{D}_0}(\vec{x};\vec{y}) \\ f_\mathcal{D}(\vec{x};\vec{y}, \mathsf{s}_0 y) &:= f_{\mathcal{D}_1}(\vec{x};\vec{y}, y) \text{ if } y \neq 0 \\ f_\mathcal{D}(\vec{x};\vec{y}, \mathsf{s}_1 y) &:= f_{\mathcal{D}_2}(\vec{x};\vec{y}, y) \end{aligned}$$

- If $\mathcal{D} = \mathsf{cond}_\Box(\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2)$ then:

$$\begin{aligned} f_\mathcal{D}(0, \vec{x};\vec{y}) &:= f_{\mathcal{D}_0}(\vec{x};\vec{y}) \\ f_\mathcal{D}(\mathsf{s}_0 x, \vec{x};\vec{y}) &:= f_{\mathcal{D}_1}(x, \vec{x};\vec{y}) \text{ if } x \neq 0 \\ f_\mathcal{D}(\mathsf{s}_1 x, \vec{x};\vec{y}) &:= f_{\mathcal{D}_2}(x, \vec{x};\vec{y}) \end{aligned}$$

- If $\mathcal{D} = \mathsf{srec}(\mathcal{D}_0, \mathcal{D}_1, \mathcal{D}_2)$ then:

$$\begin{aligned} f_\mathcal{D}(0, \vec{x};\vec{y}) &:= f_{\mathcal{D}_0}(\vec{x};\vec{y}) \\ f_\mathcal{D}(\mathsf{s}_0 x, \vec{x};\vec{y}) &:= f_{\mathcal{D}_1}(x, \vec{x};\vec{y}, f_\mathcal{D}(x, \vec{x};\vec{y})) \text{ if } x \neq 0 \\ f_\mathcal{D}(\mathsf{s}_1 x, \vec{x};\vec{y}) &:= f_{\mathcal{D}_2}(x, \vec{x};\vec{y}, f_\mathcal{D}(x, \vec{x};\vec{y})) \end{aligned}$$

This formal semantics exposes how B-derivations and B functions relate. The rule srec in Figure 2 corresponds to safe recursion, and safe composition along safe parameters is expressed by means of the rules cut_N. Note, however, that the function $f_\mathcal{D}$ is not quite defined according to function algebra B, due to the interpretation of the $\mathsf{cut}_\Box$ rule apparently not satisfying the required constraint on safe composition along a normal parameter. However, this admission turns out to be harmless, as exposited in the following proposition:

Proposition 5. *Given a* B*-derivation $\mathcal{D} : \Box\Gamma, \vec{N} \Rightarrow \Box N$, there is a smaller* B*-derivation $\mathcal{D}^* : \Box\Gamma \Rightarrow \Box N$ such that:*

$$f_\mathcal{D}(\vec{x};\vec{y}) = f_{\mathcal{D}^*}(\vec{x};).$$

Our overloading of the notation B, for both a function algebra and for a type system, is now justified by:

Proposition 6. *$f(\vec{x};\vec{y}) \in \mathsf{B}$ iff there is a* B*-derivation $\mathcal{D}$ for which $f_\mathcal{D}(\vec{x};\vec{y}) = f(\vec{x};\vec{y})$.*

Proof sketch. The left-right implication follows by a routine induction on the definition of $f(\vec{x};\vec{y})$. For the right-left implication we proceed by induction on the structure of $\mathcal{D}$. The only non-trivial case is when the last rule is $\mathsf{cut}_\Box$, for which we appeal to Proposition 5 to recover a correct instance of safe composition along a normal parameter. □

Convention 7. Given Proposition 6 above, we shall henceforth freely write $f(\vec{x};\vec{y}) \in \mathsf{B}$ if there is a derivation $\mathcal{D} : \Box\Gamma, \vec{N} \Rightarrow N$ with $f_\mathcal{D}(\vec{x};\vec{y}) = f(\vec{x};\vec{y})$.

$$\mathrm{id}\,\frac{}{N \Rightarrow N} \quad \mathrm{cut}_N\,\frac{\Gamma \Rightarrow N \quad \Gamma, N \Rightarrow B}{\Gamma \Rightarrow B} \quad \mathrm{cut}_\Box\,\frac{\Gamma \Rightarrow \Box N \quad \Box N, \Gamma \Rightarrow B}{\Gamma \Rightarrow B} \quad \mathrm{w}_N\,\frac{\Gamma \Rightarrow B}{\Gamma, N \Rightarrow B} \quad \mathrm{w}_\Box\,\frac{\Gamma \Rightarrow B}{\Box N, \Gamma \Rightarrow B} \quad \mathrm{e}\,\frac{\Gamma, A, B, \Gamma' \Rightarrow C}{\Gamma, B, A, \Gamma' \Rightarrow C}$$

$$\Box_l\,\frac{\Gamma, N \Rightarrow A}{\Box N, \Gamma \Rightarrow A} \quad \Box_r\,\frac{\Box\Gamma \Rightarrow N}{\Box\Gamma \Rightarrow \Box N} \quad 0\,\frac{}{\Rightarrow N} \quad \mathrm{s}_0\,\frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A} \quad \mathrm{s}_1\,\frac{\Gamma \Rightarrow A}{\Gamma \Rightarrow A} \quad \mathrm{srec}\,\frac{\Gamma \Rightarrow N \quad \Box N, \Gamma, N \Rightarrow N \quad \Box N, \Gamma, N \Rightarrow N}{\Box N, \Gamma \Rightarrow N}$$

$$\mathrm{cond}_N\,\frac{\Gamma \Rightarrow N \quad \Gamma, N \Rightarrow N \quad \Gamma, N \Rightarrow N}{\Gamma, N \Rightarrow N} \qquad \mathrm{cond}_\Box\,\frac{\Gamma \Rightarrow N \quad \Box N, \Gamma \Rightarrow N \quad \Box N, \Gamma \Rightarrow N}{\Box N, \Gamma \Rightarrow N}$$

Figure 2: System B, as a sequent-style type system.

3 TWO-SORTED CIRCULAR SYSTEMS ON NOTATION

In this section we introduce a 'coinductive' version of B, and we study global criteria that tame its computational strength. This proof-theoretic investigation will lead us to two relevant circular systems: CNB, which morally permits 'nested' versions of safe recursion, and CB, which will turn out to be closer to usual safe recursion.

Throughout this section we shall work with the set of typing rules $\mathsf{B}^- := \mathsf{B} - \{\mathrm{srec}\}$.

3.1 Non-wellfounded typing derivations

To begin with, we define the notion of 'coderivation', which is the fundamental formal object of this section.

Definition 8 (Coderivations)**.** A (B^--)*coderivation* $\mathcal{D}$ is a possibly infinite rooted tree (of height $\leq \omega$) generated by the rules of B^-. Formally, we identify $\mathcal{D}$ with a prefix-closed subset of $\{0,1,2\}^*$ (i.e. a ternary tree) where each node is labelled by an inference step from B^- such that, whenever $\nu \in \mathcal{D}$ is labelled by a step $\frac{S_1 \quad \cdots \quad S_n}{S}$, for $n \leq 3$, ν has n children in $\mathcal{D}$ labelled by steps with conclusions $S_1, \ldots, S_n$ respectively. Sub-coderivations of a coderivation $\mathcal{D}$ rooted at position $\nu \in \{0,1,2\}^*$ are typically denoted $\mathcal{D}_\nu$, so that $\mathcal{D}_\epsilon = \mathcal{D}$. We write $\nu \sqsubseteq \mu$ (or $\nu \sqsubset \mu$) if ν is a prefix (respectively, a strict prefix) of μ, and in this case we say that μ is *above* (respectively, *strictly above*) ν or that ν is *below* (respectively, *strictly below*) μ. We extend this order from nodes to sequents in the obvious way.

We say that a coderivation is *regular* (or *circular*) if it has only finitely many distinct sub-coderivations.

Note that, while usual derivations may be naturally written as finite trees or dags, regular coderivations may be naturally written as finite directed (possibly cyclic) graphs. Some examples of regular coderivations can be found in Figure 3, employing the following writing conventions:

Convention 9 (Representing coderivations)**.** Henceforth, we may mark steps by • (or similar) in a regular coderivation to indicate roots of identical sub-coderivations. Moreover, to avoid ambiguities and to ease parsing of (co)derivations, we shall often underline principal formulas of a rule instance in a given coderivation and omit instances of $\mathrm{w}_\Box$ and w_N as well as certain structural steps, e.g. during a cut step.

Finally, when the sub-coderivations $\mathcal{D}_0$ and $\mathcal{D}_1$ above the second and the third premise of the conditional rule (from left) are similar (or identical), we may compress them into a single parametrised sub-coderivation $\mathcal{D}_i$ (for $i = 0, 1$).

As discussed in [10, 12, 22], coderivations can be identified with Kleene-Herbrand-Gödel style equational programs, in general computing partial recursive functionals (see, e.g., [20, §63] for further details). We shall specialise this idea to our two-sorted setting.

Definition 10 (Semantics of coderivations)**.** To each B^--coderivation $\mathcal{D}$ we associate a two-sorted Kleene-Herbrand-Gödel partial function $f_\mathcal{D}$ obtained by construing the semantics of Definition 4 as a (possibly infinite) equational program. Given a two-sorted function $f(\vec{x};\vec{y})$, we say that f is *defined* by a B^--coderivation $\mathcal{D}$ if $f_\mathcal{D}(\vec{x};\vec{y}) = f(\vec{x};\vec{y})$.

Remark 11. Note, in particular, that from a regular coderivation $\mathcal{D}$ we obtain a *finite* equational program determining $f_\mathcal{D}$. Of course, our overloading of the notation $f_\mathcal{D}$ is suggestive since it is consistent with that of Definition 4.

Example 12 (Regular coderivations, revisited)**.** Let us consider the semantics of coderivations $\mathcal{I}$, $\mathcal{R}$ and $\mathcal{C}$ from Figure 3.

- The partial functions $f_{\mathcal{I}_\nu}$ are given by the following equational program:

$$\begin{aligned} f_{\mathcal{I}_\epsilon}(x;) &= f_{\mathcal{I}_1}(f_{\mathcal{I}_0}(x;);) \\ f_{\mathcal{I}_0}(x;) &= f_{\mathcal{I}_{00}}(x;) \\ f_{\mathcal{I}_{00}}(x;) &= f_{\mathcal{I}_{000}}(;x) \\ f_{\mathcal{I}_{000}}(;x) &= \mathsf{s}_1(;x) \\ f_{\mathcal{I}_1}(x;) &= f_{\mathcal{I}_\epsilon}(x;) \end{aligned}$$

  By purely equational reasoning, we can simplify this program to obtain $f_{\mathcal{I}_\epsilon}(x;) = f_{\mathcal{I}_\epsilon}(\mathsf{s}_1 x;)$ Since the above equational program keeps increasing the input, the function $f_\mathcal{I} = f_{\mathcal{I}_\epsilon}$ is always undefined.
- Let $f_\mathcal{G}(\vec{x};\vec{y})$ and $f_{\mathcal{H}_i}(x,\vec{x},z;\vec{y})$ ($i = 0, 1$) be the functions defined by the regular B^--coderivations $\mathcal{G}$ and $\mathcal{H}_i$, respectively. Then the equational program for $\mathcal{R}$ can be rewritten as follows:

$$\begin{aligned} f_\mathcal{R}(0,\vec{x};) &= f_\mathcal{G}(\vec{x};) \\ f_\mathcal{R}(\mathsf{s}_i x,\vec{x};) &= f_{\mathcal{H}_i}(x,\vec{x},f_\mathcal{R}(x,\vec{x};);) \end{aligned} \tag{1}$$

  which is an instance of a *non-safe* recursion scheme (on notation).

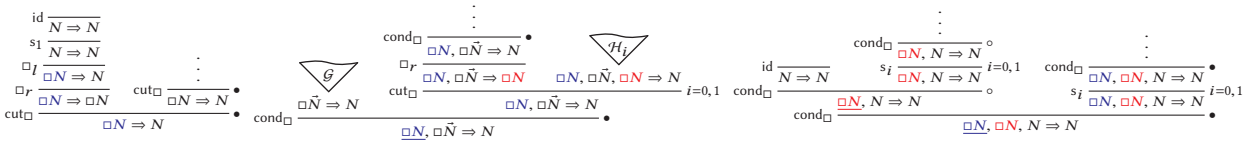


Figure 3: Examples of regular coderivations $\mathcal{I}$, $\mathcal{R}$ and C, from left (assuming $\mathcal{G}$, $\mathcal{H}_0$ and $\mathcal{H}_1$ regular).

- The equational program of C can be written as:

$$\begin{aligned} f_{C_\epsilon}(0,0;z) &= z \\ f_{C_\epsilon}(0,\mathsf{s}_i y;z) &= \mathsf{s}_i f_{C_\epsilon}(x,y;z) \neq 0 \\ f_{C_\epsilon}(\mathsf{s}_i x,y;z) &= \mathsf{s}_i f_{C_\epsilon}(x,y;z) \neq 0 \end{aligned}$$

  which computes concatenation of the binary representation of three natural numbers.

The above examples illustrate two undesirable features of regular B^--coderivations, from the point of view of implicit complexity:

I. on the one hand, despite being finitely presentable, they can define *partial functions*;
II. on the other hand, despite the presence of modalities implementing the normal/safe distinction of function arguments, they can define *non-safe recursion schemes*.

3.2 The progressing criterion

To address Problem I we shall adapt to our setting a well-known 'termination criterion' from non-wellfounded proof theory. First, let us recall some standard proof theoretic concepts about (co)derivations, similar to those in [10, 12, 22].

Definition 13 (Ancestry)**.** Fix a coderivation $\mathcal{D}$. We say that a type occurrence A is an *immediate ancestor* of a type occurrence B in $\mathcal{D}$ if they are types in a premiss and conclusion (respectively) of an inference step and, as typeset in Figure 2, have the same colour. If A and B are in some Γ or Γ', then furthermore they must be in the same position in the list.

Being a binary relation, immediate ancestry forms a directed graph upon which our correctness criterion is built.

Definition 14 (Progressing coderivations)**.** A *thread* is a maximal path in the graph of immediate ancestry. We say that a (infinite) thread is *progressing* if it is eventually constant $\Box N$ and infinitely often principal for a $\mathsf{cond}_{\Box N}$ rule.

A coderivation is *progressing* if each of its infinite branches has a progressing thread.

Example 15 (Regular coderivations, re-visited)**.** In Figure 3, $\mathcal{I}$ has precisely one infinite branch (that loops on $\bullet$) which contains no instances of $\mathsf{cond}_\Box$ at all. Therefore, $\mathcal{I}$ is not progressing. On the other hand, C has two simple loops, one on $\bullet$ and the other one on $\circ$. For any infinite branch B we have two cases:

- if B crosses the bottommost conditional infinitely many times, it contains a progressing blue thread;
- otherwise, B crosses the topmost conditional infinitely many times, so that it contains a progressing red thread.

Therefore, C is progressing. By the same reasoning, we can conclude that $\mathcal{R}$ is progressing whenever $\mathcal{G}$ and $\mathcal{H}_i$ are.

Like in [10, 12, 22], the progressing criterion is sufficient to guarantee that the partial function computed is, in fact, a well-defined total function:

Proposition 16. *If $\mathcal{D}$ is progressing then $f_\mathcal{D}$ is total.*

Proof sketch. We proceed by contradiction. If $f_\mathcal{D}$ is non-total then, since each rule preserves totality top-down, we must have that $f_{\mathcal{D}'}$ is non-total for one of $\mathcal{D}$'s immediate sub-coderivations $\mathcal{D}'$. Continuing this reasoning we can build an infinite leftmost 'non-total' branch $B = (\mathcal{D}^i)_{i<\omega}$. Let $(\Box N^i)_{i\geq k}$ be a progressing thread along B, and assign to each $\Box N^i$ the least natural number $n_i \in \mathbb{N}$ such that $f_{\mathcal{D}^i}$ is non-total when n_i is assigned to the type occurrence $\Box N^i$.

Now, notice that:

- $(n_i)_{i\geq k}$ is monotone non-increasing, by inspection of the rules and their interpretations from Definition 4.
- $(n_i)_{i\geq k}$ does not converge, since $(\Box N^i)_{i\geq k}$ is progressing and so is infinitely often principal for $\mathsf{cond}_\Box$, where the value of n_i must strictly decrease (cf., again, Definition 4).

This contradicts the well-ordering property of the natural numbers. □

One of the most appealing features of the progressing criterion is that, while being rather expressive and admitting many natural programs, e.g. as we will see in the next subsections, it remains effective (for regular coderivations) thanks to well known arguments in automaton theory:

Fact 17 (Folklore). *It is decidable whether a regular coderivation is progressing.*

This well-known result (see, e.g., [15] for an exposition for a similar circular system) follows from the fact that the progressing criterion is equivalent to the universality of a Büchi automaton of size determined by the (finite) representation of the input coderivation. This problem is decidable in polynomial space, though the correctness of this algorithm requires nontrivial infinitary combinatorics, as formally demonstrated in [21].

Let us finally observe that the progressing condition turns out to be sufficient to restate Proposition 5 in the setting of non-wellfounded coderivations:

Proposition 18. *Given a progressing B^--coderivation $\mathcal{D}$: $\Box\Gamma, \vec{N} \Rightarrow \Box N$, there is a progressing B^--coderivation $\mathcal{D}^*$: $\Box\Gamma \Rightarrow \Box N$ such that:*

$$f_\mathcal{D}(\vec{x};\vec{y}) = f_{\mathcal{D}^*}(\vec{x};).$$

Proof sketch. By progressiveness, any infinite branch contains a $\mathsf{cond}_\Box$-step, which has non-modal succedent. Thus there is a set of $\mathsf{cond}_\Box$-occurrences that forms a bar across $\mathcal{D}$. By König Lemma, the set of all nodes of $\mathcal{D}$ below this bar, say $X_\mathcal{D}$, is finite. The proof

now follows by induction on the cardinality of $X_{\mathcal{D}}$ and is analogous to Proposition 5. □

Note that the above proof uniquely depends on progressiveness, and so it holds for non-regular progressing B^--coderivations as well.

3.3 Computational expressivity of coderivations

Problem II indicates that the modal/non-modal distinction for (progressing) B^--coderivations, by itself, does not suffice to control complexity. Indeed it is not hard to see that, as it stands, this distinction is somewhat redundant for definable functions with only normal inputs. By inspection of Figure 2, we may safely replace each N by $\Box N$ in any such coderivation, preserving progressiveness:

Proposition 19. *Let $\mathcal{D} : \Box N, .^n., \Box N, N, .^m., N \Rightarrow N$ be a B^--coderivation. Then, there exists a B^--coderivation $\mathcal{D}^{\Box} : \Box N, .^{n+m}., \Box N \Rightarrow N$ s.t.:*

- *$f_{\mathcal{D}}(\vec{x};\vec{y}) = f_{\mathcal{D}^{\Box}}(\vec{x},\vec{y};)$;*
- *$\mathcal{D}^{\Box}$ does not contain instances of $\mathsf{w}_N, \mathsf{e}_N, \mathsf{cut}_N, \mathsf{cond}_N$.*

Moreover, $\mathcal{D}^{\Box}$ is regular (resp. progressing) if $\mathcal{D}$ is.

Proof sketch. We construct $\mathcal{D}^{\Box}$ coinductively. The only interesting cases are when $\mathcal{D}$ is id or it is obtained from $\mathcal{D}_0 : \Gamma \Rightarrow N$ and $\mathcal{D}_1 : \Gamma, N \Rightarrow A$ by applying a cut_N-step. Then, $\mathcal{D}^{\Box}$ is constructed, respectively, as follows:

$$\dfrac{}{N \Rightarrow N}\,\mathsf{id} \qquad \dfrac{\dfrac{\dfrac{\mathcal{D}_0^{\Box}}{\Box\Gamma \Rightarrow N}}{\Box\Gamma \Rightarrow \Box N}\,\Box_r \qquad \dfrac{\mathcal{D}_1^{\Box}}{\Box\Gamma, N \Rightarrow N}}{\Box\Gamma, \Box N \Rightarrow N}\,\Box_l}{\Box\Gamma \Rightarrow N}\,\mathsf{cut}_{\Box N}$$

□

Consequently, we may view (regular, progressing) B^--coderivations as the type 0 (regular, progressing) fragment of the system CT from [10, 12, 22]. As a result we inherit the following characterisations:

Proposition 20. *We have the following:*

(1) Any function $f(\vec{x};)$ is defined by a progressing B^--coderivation.
(2) The class of regular B^--coderivations is Turing-complete, i.e. they define every partial recursive function.
(3) $f(\vec{x};)$ is defined by a regular progressing B^--coderivation if and only if $f(\vec{x})$ is type-1-primitive-recursive, i.e. it is in the level 1 fragment T_1 of Gödel's T.

Given the computationally equivalent system CT_0 with contraction from [12], we can view the above result as a sort of 'contraction admissibility' for regular progressing B^--coderivations. Call $\mathsf{B}^- + \{\mathsf{c}_N, \mathsf{c}_{\Box N}\}$ the extension of B^- with the rules c_N and $\mathsf{c}_{\Box N}$ below:

$$\mathsf{c}_N\,\dfrac{\Gamma, N, N \Rightarrow B}{\Gamma, N \Rightarrow B} \qquad \mathsf{c}_{\Box}\,\dfrac{\Gamma, \Box N, \Box N \Rightarrow B}{\Gamma, \Box N \Rightarrow B} \tag{2}$$

where the semantics for the new system extends the one for B^- in the obvious way, and the notion of (progressing) thread is induced by the given colouring.[1] We have:

Corollary 21. *$f(\vec{x};)$ is definable by a regular progressing $\mathsf{B}^- + \{\mathsf{c}_N, \mathsf{c}_{\Box N}\}$-coderivation if and only if it is definable by a regular progressing B^--coderivation.*

3.4 Proof-level conditions motivated by implicit complexity

B-derivations locally introduce safe recursion by means of the rule srec, and Proposition 5 ensures that the composition schemes defined by the cut rules are safe. As suggested by Problem II, a different situation arises when we move to B^--coderivations, where the lack of further constraints means that we can define 'non-safe' equational programs. We may recover safety by a natural proof-level condition:

Definition 22 (Safety)**.** A B^--coderivation is *safe* if each infinite branch crosses only finitely many $\mathsf{cut}_{\Box}$-steps.

The corresponding equational programs of safe coderivations indeed only have safe inputs in hereditarily safe positions, as we shall soon see. Let us illustrate this by means of examples.

Example 23. The coderivation $\mathcal{I}$ in Figure 3 is not safe, as there is an instance of $\mathsf{cut}_{\Box}$ in the loop on •, which means that there is an infinite branch crossing infinitely many $\mathsf{cut}_{\Box}$-steps. By contrast, using the same reasoning, we can infer that the coderivation $\mathcal{C}$ is safe. Finally, by inspecting the coderivation $\mathcal{R}$ of Figure 3, we notice that the infinite branch that loops on • contains infinitely many $\mathsf{cut}_{\Box}$ steps, so it too is not safe.

Perhaps surprisingly, however, the safety condition is not enough to restrict the set of B^--definable functions to **FPTIME**, as the following example shows.

Example 24 (Safe exponentiation)**.** Consider the following coderivation $\mathcal{E}$,

$$\dfrac{\dfrac{\dfrac{}{N \Rightarrow N}\,\mathsf{id}}{N \Rightarrow N}\,\mathsf{s}_0 \qquad \dfrac{\vdots}{\Box N, N \Rightarrow N}\,\mathsf{cond}_{\Box}\bullet \qquad \dfrac{\vdots}{\Box N, N \Rightarrow N}\,\mathsf{cond}_{\Box}\bullet}{\Box N, N \Rightarrow N}\,\mathsf{cut}_N\ i{=}0,1}{\Box N, N \Rightarrow N}\,\mathsf{cond}_{\Box}\bullet$$

where we identify the sub-coderivations above the second and the third premises of the conditional. The coderivation is clearly progressing. Moreover it is safe, as $\mathcal{E}$ has no instances of $\mathsf{cut}_{\Box}$. Its associated equational program can be written as follows:

$$\begin{aligned} f_{\mathcal{E}}(0;y) &= \mathsf{s}_0(;y) \\ f_{\mathcal{E}}(\mathsf{s}_0 x;y) &= f_{\mathcal{E}}(x; f_{\mathcal{E}}(x;y)) \quad x \neq 0 \\ f_{\mathcal{E}}(\mathsf{s}_1 x;y) &= f_{\mathcal{E}}(x; f_{\mathcal{E}}(x;y)) \end{aligned} \tag{3}$$

The above equational program has already appeared in [19, 24]. It is not hard to show, by induction on x, that $f_{\mathcal{E}}(x;y) = 2^{2^{|x|}} \cdot y$. Thus $f_{\mathcal{E}}$ has exponential growth rate (as long as $y \neq 0$), despite being defined by a 'safe' recursion scheme.

[1] Note that the totality argument of Proposition 16 still applies in the presence of these rules, cf. also [12].

The above coderivation exemplifies a safe recursion scheme that is able to *nest* one recursive call inside another in order to obtain exponential growth rate. This is in fact a peculiar feature of circular proofs, and it is worth discussing.

Remark 25 (On nesting and higher-order recursion). As we have already seen, namely in Proposition 20.3, (progressing) B^--coderivations are able to simulate some sort of higher-order recursion, namely at type 1 (cf. also [12]). In this way it is arguably not so surprising that the sort of 'nested recursion' in Equation (3) is definable since type 1 recursion, in particular, allows such nesting of the recursive calls. To make this point more apparent, consider the following higher-order 'safe' recursion operator:

$$\mathrm{rec}_A : \Box N \to (\Box N \to A \to A) \to A \to A$$

with $A = N \to N$, and $f(x) = \mathrm{rec}_A(x, h, g)$ is defined as $f(0) = g$ and $f(\mathsf{s}_i x) = h(x, f(x))$ for $x > 0$. By setting $g := \lambda y : N.\mathsf{s}_0 y$ and $h := \lambda x : \Box N.\lambda u : N \to N.(\lambda y : N.u(u\, y))$ we can easily check that $f_{\mathcal{E}}(x; y) = \mathrm{rec}_A(x, h, g)(y)$, where $\mathcal{E}$ is as in Example 24. Hence, the function $f_{\mathcal{E}}(x; y)$ can be defined by means of a higher-order version of safe recursion.

As noticed by Hofmann [19], and formally proved by Leivant [24], higher-order safe recursion can be used to characterise **FELEMENTARY**, thanks to this capacity to nest recursive calls. Moreover, Hofmann showed in [19] that by introducing a 'linearity' restriction on the operator rec_A, which prevents duplication of recursive calls, it is possible to recover the class **FPTIME**. The resulting type system, called SLR ('Safe Linear Recursion'), can thus be regarded as a higher-order formulation of B.

Following [19], we shall impose a linearity criterion to rule out those coderivations that nest recursive calls. This is achieved by observing that the duplication of the recursive calls of $f_{\mathcal{E}}$ in Example 24 is due to the presence in $\mathcal{E}$ of loops on $\bullet$ crossing both premises of a cut_N step. Hence, our circular-proof-theoretic counterpart of Hofmann's linearity restriction can be obtained by demanding that at most one premise of each cut_N step is crossed by such loops. Again, we rather give a more natural proof-level criterion which does not depend on our intuitive notion of loop.

Definition 26 (Left-leaning)**.** A B^--coderivation is said to be *left-leaning* if each infinite branch goes right at a cut_N-step only finitely often.

Example 27. In Figure 3, $\mathcal{I}$ is trivially left-leaning, as it contains no instances of cut_N at all. The coderivations $\mathcal{C}$ and $\mathcal{R}$ are also left-leaning, since no infinite branch can go right at the cut_N steps. By contrast, the coderivation $\mathcal{E}$ in Example 24 is not left-leaning, as there is an infinite branch looping at $\bullet$ and crossing infinitely many times the rightmost premise of the cut_N-step.

We are now ready to present our circular systems:

Definition 28 (Circular Implicit Systems)**.** CNB is the class of safe regular progressing B^--coderivations. CB is the restriction of CNB to only left-leaning coderivations. A two-sorted function $f(\vec{x}; \vec{y})$ is CNB-*definable* (or CB-*definable*) if there is a coderivation $\mathcal{D} \in$ CNB (resp., $\mathcal{D} \in$ CB) such that $f_{\mathcal{D}}(\vec{x}; \vec{y}) = f(\vec{x}; \vec{y})$.

Let us point out that Proposition 18 can be strengthened to preserve safety and left-leaningness:

Proposition 29. *Let $\mathcal{D} : \Box\Gamma, \vec{N} \Rightarrow \Box N$ be a coderivation in* CNB *(or* CB*). There exists a* CNB*-coderivation (resp.,* CB*-coderivation) $\mathcal{D}^* : \Box\Gamma \Rightarrow \Box N$ such that $f_{\mathcal{D}}(\vec{x}; \vec{y}) = f_{\mathcal{D}^*}(\vec{x};)$.*

3.5 On the complexity of proof-checking

Note that both the safety and the left-leaning conditions above are defined at the level of arbitrary coderivations, not just regular and/or progressing ones. Moreover, since these conditions are defined at the proof-level rather than the thread-level, they are easy to check on regular coderivations:

Proposition 30. *The safety and the left-leaning condition are* **NL***-decidable for regular coderivations.*

The idea here is that, for regular coderivations, checking that no branch has infinitely many occurrences of a particular rule can be reduced to checking acyclicity of a certain subgraph, which is well-known to be in **coNL** = **NL**.

Recall that progressiveness of regular coderivations is decidable by reduction to universality of Büchi automata, a **PSPACE**-complete problem. Indeed progressiveness itself is **PSPACE**-complete in many settings, cf. [28]. It is perhaps surprising, therefore, that the safety of a regular coderivation also allows us to decide progressiveness efficiently too, thanks to the following reduction:

Proposition 31. *A safe B^--coderivation is progressing iff every infinite branch has infinitely many $\mathsf{cond}_\Box$-steps.*

Proof. The left-right implication is trivial. For the right-left implication, let us consider an infinite branch B of a safe B^--coderivation $\mathcal{D}$. By safety, there exists a node v of B such that any sequent above v in B is not the conclusion of a $\mathsf{cut}_\Box$-step. Now, by inspecting the rules of $\mathsf{B}^- - \{\mathsf{cut}_\Box\}$ we observe that:

- every modal formula occurrence in B has a unique thread along B;
- infinite threads along B cannot start strictly above v.

Hence, setting k to be the number of $\Box N$ occurrences in the antecedent of v, B has (at most) k infinite threads. Moreover, since B contains infinitely many $\mathsf{cond}_\Box$-steps, by the Infinite Pigeonhole Principle we conclude that one of these threads is infinitely often principal for the $\mathsf{cond}_\Box$ rule. □

Thus, using similar reasoning to that of Proposition 30 we may conclude from Proposition 31 the following:

Corollary 32. *Given a regular B^--coderivation $\mathcal{D}$, the problem of deciding if $\mathcal{D}$ is in* CNB *(resp.* CB*) is in* **NL***.*

Let us point out that the reduction above is similar to (and indeed generalises) an analogous one for cut-free extensions of Kleene algebra, cf. [14, Proposition 8].

4 SOME VARIANTS OF SAFE RECURSION

In this section we shall introduce various extensions of B to ultimately classify the expressivity of the circular systems CB and CNB. First, starting from the analysis of Example 24 and the subsequent system CNB, we shall define a version of B with safe *nested* recursion, called NB. Second, motivated by the more liberal way of defining functions in both CB and CNB, we shall endow the

safe recursion	*on notation*	*on* $\subset$
unnested	B	$\mathsf{B}^{\subset}$
unnested, with compositions	SB	$\mathsf{SB}^{\subset}$
nested	NB	$\mathsf{NB}^{\subset}$

Figure 4: The function algebras considered in Section 4. Any algebra is included in one below it and to the right of it.

function algebras B and NB with forms of safe recursion over a well-founded relation $\subset$ on lists of normal parameters. Figure 4 summarises the function algebras considered and their relations.

4.1 Relativised algebras and nested recursion

One of the key features of the Bellantoni-Cook algebra B is that 'nesting' of recursive calls is not permitted. For instance, let us recall the equational program from Example 24:

$$\begin{aligned} \mathsf{ex}(0;y) &= \mathsf{s}_0 y \\ \mathsf{ex}(\mathsf{s}_i x;y) &= \mathsf{ex}(x;\mathsf{ex}(x;y)) \end{aligned} \tag{4}$$

Recall that $\mathsf{ex}(x;y) = f_{\mathcal{E}}(x;y) = 2^{2^{|x|}} \cdot y$. The 'recursion step' on the second line is compatible with safe composition, in that safe inputs only occur in hereditarily safe positions, but one of the recursive calls takes another recursive call among its safe inputs. In Example 24 we showed how the above function $\mathsf{ex}(x;y)$ can be CNB-defined. We thus seek a suitable extension of B able to formalise such nested recursion to serve as a function algebraic counterpart to CNB.

It will be convenient for us to work with generalisations of B including *oracles*. Formally speaking, these can be seen as variables ranging over two-sorted functions, though we shall often treat them as explicit functions too.

Definition 33. For all sets of oracles $\vec{a} = a_1, \ldots, a_k$, we define the algebra of B^- functions *over* $\vec{a}$ to include all the initial functions of B^- and,

- (oracles). $a_i(\vec{x};\vec{y})$ is a function over $\vec{a}$, for $1 \leq i \leq k$, (where $\vec{x}$,$\vec{y}$ have appropriate length for a_i).

and closed under:

- (Safe Composition).
  (1) from $g(\vec{x};\vec{y}), h(\vec{x};\vec{y},y)$ over $\vec{a}$ define $f(\vec{x};\vec{y})$ over $\vec{a}$ by $f(\vec{x};\vec{y}) = h(\vec{x};\vec{y},g(\vec{x};\vec{y}))$.
  (2) from $g(\vec{x};)$ over $\emptyset$ and $h(\vec{x},x;\vec{y})$ over $\emptyset$ define $f(\vec{x};\vec{y})$ over $\vec{a}$ by $f(\vec{x};\vec{y}) = h(\vec{x},g(\vec{x};);\vec{y})$.

We write $\mathsf{B}^-(\vec{a})$ for the class of functions over $\vec{a}$ generated in this way.

Note that Safe Composition along normal parameters (Item 2 above) comes with the condition that $g(\vec{x};)$ is oracle-free. This restriction prevents oracles (and hence the recursive calls) appearing in normal position. The same condition on $h(\vec{x},x;\vec{y})$ being oracle-free is not strictly necessary for the complexity bounds we are after, as we shall see in the next section when we define more expressive algebras, but is convenient in order to facilitate the 'grand tour' strategy of this paper (cf. Figure 1).

We shall write, say, $\lambda\vec{v}.f(\vec{x};\vec{v})$ for the function taking only safe arguments $\vec{v}$ with $(\lambda\vec{v}.f(\vec{x};\vec{v}))(;\vec{y}) = f(\vec{x};\vec{y})$ (here $\vec{x}$ may be seen as parameters). Nested recursion can be formalised in the setting of algebras-with-oracles as follows:

Definition 34 (Safe Nested Recursion)**.** We write snrec for the scheme:

- from $g(\vec{x};\vec{y})$ over $\emptyset$ and $h_i(a)(x,\vec{x};\vec{y})$ over $a,\vec{a}$, define $f(x,\vec{x};\vec{y})$ over $\vec{a}$ by:

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(\mathsf{s}_0 x,\vec{x};\vec{y}) &= h_0(\lambda\vec{v}.f(x,\vec{x};\vec{v}))(x,\vec{x};\vec{y}) \qquad x \neq 0 \\ f(\mathsf{s}_1 x,\vec{x};\vec{y}) &= h_1(\lambda\vec{v}.f(x,\vec{x};\vec{v}))(x,\vec{x};\vec{y}) \end{aligned}$$

We write $\mathsf{NB}(\vec{a})$ for the class of functions over $\vec{a}$ generated from B^- under snrec and Safe Composition (from Definition 33), and write simply NB for $\mathsf{NB}(\emptyset)$.

Note that Safe Nested Recursion also admits definitions that are not morally 'nested' but rather use a form of 'composition during recursion':

- from $g(\vec{x};\vec{y})$, $\left(\vec{g}_j(x,\vec{x};\vec{y})\right)_{1\leq j\leq k}$ and $h_i\left(x,\vec{x};\vec{y},(z_j)_{1\leq j\leq k}\right)$ define:

$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(\mathsf{s}_i x,\vec{x};\vec{y}) &= h_i\left(x,\vec{x};\vec{y},\left(f(x,\vec{x};\vec{g}_j(x,\vec{x};\vec{y}))\right)_{1\leq j\leq k}\right) \end{aligned} \tag{5}$$

In this case, note that we have allowed the safe inputs of f to take arbitrary values given by previously defined functions, but at the same time f never calls itself in a safe position, as in (4). This safe recursion scheme does not require the use of oracles in its statement, but is equivalent to the following one using a notion of 'nesting' for Safe Composition:

Definition 35 (Safe Recursion with Composition During Recursion)**.** An instance of Safe Composition along Safe Parameters (Item 1 from Definition 33) is called *unnested* if (at least) one of $g(\vec{x};\vec{y})$ and $h(\vec{x};\vec{y},y)$ is over $\emptyset$ (i.e. is oracle-free). We define $\mathsf{SB}(\vec{a})$ to be the restriction of $\mathsf{NB}(\vec{a})$ using only unnested Safe Composition along Safe Parameters, and write simply SB for $\mathsf{SB}(\emptyset)$.

It is not hard to see that, indeed, SB defines the same class of functions of the extension of B^- by the scheme given in Equation (5). The 'S' in SB thus indicates that the recursion scheme is 'shallow', as opposed to nested. It should be said that we will not actually use SB in this work, but rather an extension of it defined in the next subsection, but we have included it for completeness of the exposition.

4.2 Safe recursion on well-founded relations

Relativised function algebras may be readily extended by recursion on arbitrary well-founded relations. For instance, given a well-founded preorder $\trianglelefteq$, and writing $\triangleleft$ for its strict variant,[2] 'safe recursion on $\triangleleft$' is given by the scheme:

- from $h(a)(x,\vec{x};\vec{y})$ over $a,\vec{a}$, define $f(x,\vec{x};\vec{y})$ over $\vec{a}$ by:

$$f(x,\vec{x};\vec{y}) = h(\lambda v \triangleleft x.f(v,\vec{x};\vec{y}))(x,\vec{x};\vec{y})$$

[2]To be precise, for a preorder $\trianglelefteq$ we write $x \triangleleft y$ if $x \trianglelefteq y$ and $y \not\trianglelefteq x$. As abuse of terminology, we say that $\trianglelefteq$ is well-founded just when $\triangleleft$ is.

Note here that we employ the notation $\lambda v \lhd x$ for a 'guarded abstraction'. Formally:

$$(\lambda v \lhd x.f(v,\vec{x};\vec{y}))(z) := \begin{cases} f(z,\vec{x};\vec{y}) & z \lhd x \\ 0 & \text{otherwise} \end{cases}$$

It is now not hard to see that total functions (with oracles) are closed under the recursion scheme above, by reduction to induction on the well-founded relation $\lhd$.

Note that such schemes can be naturally extended to preorders on *tuples* of numbers too, by abstracting several inputs. We shall specialise this idea to a particular well-founded preorder that will be helpful later to bound the complexity of definable functions in our systems CB and CNB.

Recall that we say that x is a *prefix* of y if y has the form xz in binary notation, i.e. y can be written $x2^n + z$ for some $n \geq 0$ and some $z < 2^n$. We say that x is a *strict prefix* of y if x is a prefix of y but $x \neq y$.

Definition 36 (Permutations of prefixes)**.** Let $[n]$ denote $\{0,\ldots,n-1\}$. We write $(x_0,\ldots,x_{n-1}) \subseteq (y_0,\ldots,y_{n-1})$ if, for some permutation $\pi : [n] \to [n]$, we have that x_i is a prefix of $y_{\pi i}$, for all $i < n$. We write $\vec{x} \subset \vec{y}$ if $\vec{x} \subseteq \vec{y}$ but $\vec{y} \not\subseteq \vec{x}$, i.e. there is a permutation $\pi : [n] \to [n]$ with x_i a prefix of y_i for each $i < n$ and, for some $i < n$, x_i is a strict prefix of y_i.

It is not hard to see that $\subseteq$ is a well-founded preorder, by reduction to the fact that the prefix relation is a well-founded partial order. As a result, we may duly devise a version of safe (nested) recursion on $\subset$:

Definition 37 (Safe (nested) recursion on permutations of prefixes)**.** We write $\mathsf{NB}^{\subset}(\vec{a})$ for the class of functions over $\vec{a}$ generated from B^- under Safe Composition, the scheme $\mathrm{snrec}_{\subset}$,

- from $h(a)(\vec{x};\vec{y})$ over $a,\vec{a}$ define $f(\vec{x};\vec{y})$ over $\vec{a}$ by:

$$f(\vec{x};\vec{y}) = h(\lambda\vec{u} \subset \vec{x}, \lambda\vec{v}.f(\vec{u};\vec{v}))(\vec{x};\vec{y})$$

and the following generalisation of Safe Composition along a Normal Parameter:

(2)′ from $g(\vec{x};)$ over $\emptyset$ and $h(\vec{x},x;\vec{y})$ over $\vec{a}$ define $f(\vec{x};\vec{y})$ over $\vec{a}$ by $f(\vec{x};\vec{y}) = h(\vec{x},g(\vec{x};);\vec{y})$.

Recalling Definition 35, we write $\mathsf{SB}^{\subset}(\vec{a})$ to be the restriction of $\mathsf{NB}^{\subset}(\vec{a})$ using only unnested Safe Composition along Safe Parameters. Finally we define $\mathsf{B}^{\subset}(\vec{a})$ to be the restriction of $\mathsf{SB}^{\subset}(\vec{a})$ where every instance of $\mathrm{snrec}_{\subset}$ has the form:

- from $h(a)(\vec{x};\vec{y})$ over $a,\vec{a}$, define $f(\vec{x};\vec{y})$ over $\vec{a}$:

$$f(\vec{x};\vec{y}) = h(\lambda\vec{u} \subset \vec{x}, \lambda\vec{v} \subseteq \vec{y}.f(\vec{u};\vec{v}))(\vec{x};\vec{y})$$

We call this latter recursion scheme $\mathrm{srec}_{\subset}$, e.g. if we need to distinguish it from $\mathrm{snrec}_{\subset}$.

Note that the version of safe composition along a normal parameter above differs from the previous one, Item 2 from Definition 33, since the function h is allowed to use oracles. Again, this difference is inessential in terms of computational complexity, as we shall see. However, as we have mentioned, the greater expressivity of $\mathsf{B}^{\subset}$ and $\mathsf{NB}^{\subset}$ will facilitate our overall strategy for characterising CB and CNB, cf. Figure 1.

Let us take a moment to point out that $\mathsf{NB}^{\subset}(\vec{a}) \supseteq \mathsf{SB}^{\subset}(\vec{a}) \supseteq \mathsf{B}^{\subset}(\vec{a})$ indeed contain only well-defined total functions over the oracles $\vec{a}$, by reduction to induction on $\subset$.

5 CHARACTERISATIONS FOR FUNCTION ALGEBRAS

In this section we characterise the complexities of the function algebras we introduced in the previous section. Namely, despite apparently extending B, $\mathsf{B}^{\subset}$ still contains just the polynomial-time functions, whereas both NB and $\mathsf{NB}^{\subset}$ are shown to contain just the elementary functions. All such results rely on a 'bounding lemma' inspired by [5].

5.1 A relativised Bounding Lemma

Bellantoni and Cook showed in [5] that any function $f(\vec{x};\vec{y}) \in \mathsf{B}$ satisfies the 'poly-max bounding lemma': there is a polynomial $p_f(\vec{n})$ such that:[3]

$$|f(\vec{x};\vec{y})| \leq p_f(|\vec{x}|) + \max|\vec{y}| \tag{6}$$

This provided a suitable invariant for eventually proving that all B-functions were polynomial-time computable.

In this work, inspired by that result, we generalise the bounding lemma to a form suitable to the relativised algebras from the previous section. To this end we establish in the next result a sort of 'elementary-max' bounding lemma that accounts for the usual poly-max bounding as a special case, by appealing to the notion of (un)nested safe composition. Both the statement and the proof are quite delicate due to our algebras' formulation using oracles; we must assume an appropriate bound for the oracles themselves, and the various (mutual) dependencies in the statement are subtle.

To state and prove the Bounding Lemma, let us employ the notation $\|\vec{x}\| := \sum|\vec{x}|$.

Lemma 38 (Bounding Lemma). *Let $f(\vec{a})(\vec{x};\vec{y}) \in \mathsf{NB}^{\subset}(\vec{a})$, with $\vec{a} = a_1,\ldots,a_k$. There is a function $m^{\vec{c}}_f(\vec{x},\vec{y})$ with,*

$$m^{\vec{c}}_f(\vec{x},\vec{y}) = e_f(\|\vec{x}\|) + d_f\sum\vec{c} + \max|\vec{y}|$$

for an elementary function $e_f(n)$ and a constant $d_f \geq 1$, such that whenever there are constants $\vec{c} = c_1,\ldots,c_k$ such that,

$$|a_i(\vec{x}_i;\vec{y}_i)| \leq c_i + d_f\sum_{j\neq i} c_j + \max|\vec{y}_i| \tag{7}$$

for $1 \leq i \leq k$, we have:[4]

$$|f(\vec{a})(\vec{x};\vec{y})| \leq m^{\vec{c}}_f(\vec{x},\vec{y}) \tag{8}$$

$$f(\vec{a})(\vec{x};\vec{y}) \;=\; f\left(\lambda\vec{x}_i.\lambda|\vec{y}_i| \leq m^{\vec{c}}_f(\vec{x},\vec{y}).a_i(\vec{x}_i;\vec{y}_i)\right)_i(\vec{x};\vec{y}) \tag{9}$$

Moreover, if in fact $f(\vec{x};\vec{y}) \in \mathsf{SB}^{\subset}(\vec{a})$, then $d_f = 1$ and $e_f(n)$ is a polynomial.

Proof idea. Both Equations (8) and (9) are proved simultaneously by induction on the definition of $f(\vec{x};\vec{y})$, always assuming that we have $\vec{c}$ satisfying Equation (7). □

[3] Recall that, for $\vec{x} = x_1,\ldots,x_n$, we write $|\vec{x}|$ for $|x_1|,\ldots,|x_n|$.

[4] To be clear, here we write $|\vec{y}_i| \leq m^{\vec{c}}_f(\vec{x},\vec{y})$ here as an abbreviation for $\{|y_{ij}| \leq m^{\vec{c}}_f(\vec{x},\vec{y})\}_j$.

Unwinding the statement above, note that e_f and d_f depend *only* on the function f itself, not on the constants $\vec{c}$ given for the (mutual) oracle bounds in Equation (7). This is crucial for the proof, namely in the case when f is defined by recursion, substituting different values for $\vec{c}$ during an inductive argument.

While the role of the elementary bounding function e_f is a natural counterpart of p_f in Bellantoni and Cook's bounding lemma, cf. Equation (6), the role of d_f is perhaps slightly less clear. Morally, d_f represents the amount of 'nesting' in the definition of f, increasing whenever oracle calls are substituted into arguments for other oracles, cf. Definition 35. Hence, if f uses only unnested Safe Composition, then $d_f = 1$ as required. In fact, it is only important to distinguish whether $d_f = 1$ or not, since d_f forms the base of an exponent for defining e_f when f is defined by safe recursion.

Finally let us note that Equations (8) and (9) are somewhat dual: while the former bounds the *output* of a function (serving as a modulus of *growth*), the latter bounds the *inputs* (serving as a modulus of *continuity*).

5.2 Soundness results

In this subsection we show that the function algebras $\mathsf{B}^{\subset}$ and $\mathsf{NB}^{\subset}$ (as well as NB) capture precisely the classes **FPTIME** and **FELEMENTARY**, respectively. We start with $\mathsf{B}^{\subset}$:

Theorem 39. *Suppose $\vec{a}$ satisfies Equation (7) for some constants $\vec{c}$. We have the following:*

*(1) If $f(\vec{x};\vec{y}) \in \mathsf{B}^{\subset}(\vec{a})$ then $f(\vec{x},\vec{y}) \in$ **FPTIME**$(\vec{a})$.*
*(2) If $f(\vec{x};\vec{y}) \in \mathsf{NB}^{\subset}(\vec{a})$ then $f(\vec{x},\vec{y}) \in$ **FELEMENTARY**$(\vec{a})$.*

Note in particular that, for $f(\vec{x};\vec{y})$ in $\mathsf{B}^{\subset}$ or $\mathsf{NB}^{\subset}$, i.e. not using any oracles, we immediately obtain membership in **FPTIME** or **FELEMENTARY**, respectively. However, the reliance on intermediate oracles during a function definition causes some difficulties that we must take into account. At a high level, the idea is to use the Bounding Lemma (namely Equation (9)) to replace certain oracle calls with explicit appropriately bounded functions computing their graphs. From here we compute $f(\vec{x};\vec{y})$ by a sort of 'course-of-values' recursion on $\subset$, storing previous values in a lookup table. In the case of $\mathsf{B}^{\subset}$, it is important that this table has polynomial-size, since there are only $m! \prod |\vec{x}|$ permutations of prefixes of a list $\vec{x} = x_1, \ldots, x_m$ (which is a polynomial of degree m).

5.3 Completeness and characterisations

We are now ready to give our main function algebraic characterisation results for polynomial-time:

Corollary 40. *The following are equivalent:*

(1) $f(\vec{x};) \in \mathsf{B}$.
(2) $f(\vec{x};) \in \mathsf{B}^{\subset}$.
*(3) $f(\vec{x}) \in$ **FPTIME**.*

Proof. (1) $\Longrightarrow$ (2) is trivial, and (2) $\Longrightarrow$ (3) is given by Theorem 39.(1). Finally, (3) $\Longrightarrow$ (1) is from [5], stated in Theorem 2 earlier. □

The remainder of this subsection is devoted to establishing a similar characterisation for NB, $\mathsf{NB}^{\subset}$ and **FELEMENTARY**. To this end, we naturally require the following result:

Theorem 41. *If $f(\vec{x}) \in$ **FELEMENTARY** then $f(\vec{x};) \in \mathsf{NB}$.*

Proof sketch. Using a formulation of **FELEMENTARY** in binary notation, and we prove by induction on the definition of $f(\vec{x}) \in$ **FELEMENTARY** that there exists $f^*(x;\vec{x}) \in \mathsf{NB}$ and a monotone function $t_f \in$ **FELEMENTARY** such that for all integers $\vec{x}$ and all $w \geq t_f(\vec{x})$ we have $f^*(w;\vec{x}) = f(\vec{x})$. Then, we use the function $\mathrm{ex}(x;y) \in \mathsf{NB}$ in (4) to achieve arbitrary elementary growth rate, which implies $f(\vec{x}) \in \mathsf{NB}$. The proof technique based on the construction of the pair $(f^*(x;\vec{x}), t_f)$ is well-known since [5], and has been adapted to the case of the elementary functions in [31]. □

Now, by the same argument as for Corollary 40, only using Theorem 41 above instead of appealing to [5], we can give our main characterisation result for algebras for elementary computation:

Corollary 42. *The following are equivalent:*

(1) $f(\vec{x};) \in \mathsf{NB}$.
(2) $f(\vec{x};) \in \mathsf{NB}^{\subset}$.
*(3) $f(\vec{x}) \in$ **FELEMENTARY**.*

6 CHARACTERISATIONS FOR CIRCULAR SYSTEMS

We now return our attention to the circular systems CB and CNB that we introduced in Section 3. We will address the complexity of their definable functions by 'sandwiching' them between function algebras of Section 4, given their characterisations that we have just established.

6.1 Completeness

To show that CB contains all polynomial-time functions, we may simply simulate Bellantoni and Cook's algebra:

Theorem 43. *If $f(\vec{x};\vec{y}) \in \mathsf{B}$ then $f(\vec{x};\vec{y}) \in \mathsf{CB}$.*

Proof sketch. The proof is by induction on the definition of f. The only interesting case is when $f(x,\vec{x};\vec{y})$ is defined by safe recursion on notation from the functions $g(\vec{x};\vec{y})$ and $h_i(x,\vec{x};\vec{y},y)$. Given $\mathcal{D}_g$ and $\mathcal{D}_{h_i}$ defining g and h_i by induction hypothesis, we may define $f(x,\vec{x};\vec{y})$ by the coderivation,

$$\dfrac{\mathcal{D}_g \quad \Gamma \Rightarrow N}{\dfrac{\dfrac{\vdots}{\Box N, \Gamma \Rightarrow N}\,\mathrm{cond}_\Box \quad \dfrac{\mathcal{D}_{h_i}}{\Box N, \Gamma, N \Rightarrow N}}{\Box N, \Gamma \Rightarrow N}\,\mathrm{cut}_N\ i=0,1}\,\mathrm{cond}_\Box \bullet$$
$$\underline{\Box N}, \Gamma \Rightarrow N$$

where $\Gamma = \Box\vec{N}, \vec{N}$. The only infinite branch (outside $\mathcal{D}_g$ and $\mathcal{D}_{h_i}$) loops on $\bullet$ and has a progressing thread in blue. □

We can also show that CNB is complete for elementary functions by simulating our nested algebra NB. First, we need to introduce the notion of *oracle* for coderivations.

Definition 44 (Oracles for coderivations)**.** Let $\vec{a} = a_1, \ldots, a_n$ be a set of safe-normal functions. A $\mathsf{B}^-(\vec{a})$-coderivations is just a usual B^--coderivation that may use initial sequents of the form $\overset{a_i}{\Box N^{n_i}, N^{m_i} \Rightarrow N}$, when a_i takes n_i normal and m_i safe inputs. We write:

$$\dfrac{\dfrac{}{\Box N^{n_i}, N^{m_i} \Rightarrow N}\, a_i \quad \mathcal{D}(\vec{a})}{\Gamma \Rightarrow A}\, i$$

for a coderivation $\mathcal{D}$ whose initial sequents are among the initial sequents $\dfrac{}{\Box N^{n_i}, N^{m_i} \Rightarrow N}\, a_i$, with $i = 1, \ldots, n$. We write $\mathsf{CNB}(\vec{a})$ for the set of CNB-coderivations with initial functions $\vec{a}$. We may sometimes omit indicating some oracles $\vec{a}$ if it is clear from context.

The semantics of such coderivations and the notion of $\mathsf{CNB}(\vec{a})$-definability are as expected, with coderivations representing functions over the oracles $\vec{a}$, and $f_{\mathcal{D}(\vec{a})} \in \mathsf{CNB}(\vec{a})$ denoting the induced interpretation of $\mathcal{D}(\vec{a})$.

Before giving our main completeness result for CNB, we need the following lemma allowing us to 'pass' parameters to oracle calls. It is similar to the notion $\mathcal{D}^{\vec{\rho}}$ from [12, Lemma 42], only we must give a more refined argument due to the unavailability of contraction in our system.

Lemma 45. *Let $\mathcal{D}(a)$ be a regular coderivation over initial sequents $\vec{a}, a$ of form:*

$$\dfrac{\dfrac{}{\Delta \Rightarrow N}\, a \qquad \dfrac{}{\Delta_i \Rightarrow N}\, a_i \quad \mathcal{D}(a)}{\Box N, \overset{k}{\ldots}, \Box N, \Gamma \Rightarrow N}\, i$$

where Γ and Δ are lists of non-modal formulas, and the path from the conclusion to each initial sequent a does not contain $\mathsf{cut}_\Box$-steps, $\Box_l$-steps and the leftmost premise of a $\mathsf{cond}_\Box$-step. Then, there exists an a^* and a regular coderivation $\mathcal{D}^*(a^*)$ with shape:

$$\dfrac{\dfrac{}{\Box N, \overset{k}{\ldots}, \Box N, \Delta \Rightarrow N}\, a^* \qquad \dfrac{}{\Delta_i \Rightarrow N}\, a_i \quad \mathcal{D}^*(a^*)}{\Box N, \overset{k}{\ldots}, \Box N, \Gamma \Rightarrow N}\, i$$

such that:

- $f_{\mathcal{D}^*(a^*)}(\vec{x}; \vec{y}) = f_{\mathcal{D}(a(\vec{x}))}(\vec{x}; \vec{y})$;
- *there exists a $\Box N$-thread from the j^{th} $\Box N$ in the LHS of the end-sequent to the j^{th} $\Box N$ in the context of any occurrence of the initial sequent a^* in $\mathcal{D}^*(a^*)$, for $1 \leq j \leq k$.*

Moreover, if $\mathcal{D}(a)$ is progressing, safe or left-leaning, then $\mathcal{D}^*(a^*)$ is also progressing, safe or left-leaning, respectively.

Theorem 46. *If $f(\vec{x}; \vec{y}) \in \mathsf{NB}$ then $f(\vec{x}; \vec{y}) \in \mathsf{CNB}$.*

6.2 The Translation Lemma

We now state a translation of CNB-coderivations into functions of $\mathsf{NB}^\subset$ which, in particular, maps CB-derivations into functions of $\mathsf{B}^\subset$, thus concluding our characterisation of CB and CNB in terms of computational complexity.

Lemma 47 (Translation Lemma). *Let $\mathcal{D}$ be a CNB-coderivation. Then, there exists a set of n functions $(f_i)_{1 \leq i \leq n}$ such that $f_1 = f_{\mathcal{D}}$ and, for all i:*

$$f_i(\vec{x}; \vec{y}) = h_i\left(\lambda \vec{u} \subset \vec{x}, \lambda \vec{v}.f_j(\vec{u}; \vec{v})\right)_{1 \leq j \leq n}(\vec{x}; \vec{y}) \tag{10}$$

where $h_i \in \mathsf{NB}^\subset(a_i)_{1 \leq i \leq n}$. Moreover, if $\mathcal{D}$ is a CB-coderivation then, for all i:

$$f_i(\vec{x}; \vec{y}) = h_i\left(\lambda \vec{u} \subset \vec{x}, \lambda \vec{v} \subseteq \vec{y}.f_j(\vec{u}; \vec{v})\right)_{1 \leq j \leq n}(\vec{x}; \vec{y}) \tag{11}$$

and $h_i \in \mathsf{B}^\subset(a_i)_{1 \leq i \leq n}$.

Let us give the idea of how this lemma is proved. By Definition 10 there exists a system of equations $S_{\mathcal{D}}$ containing, for each node v of $\mathcal{D}$, an equation that defines the function $f_{\mathcal{D}_v}$ in terms of the functions $f_{\mathcal{D}_u}$ with u immediately above v. Moreover, since $\mathcal{D}$ is regular, by Remark 11 we may assume $S_{\mathcal{D}}$ is a *finite* system of equations defining $f_{\mathcal{D}}$ (i.e. $f_{\mathcal{D}_\epsilon}$). It is then suggestive to represent $\mathcal{D}$ as a 'definition tree', by replacing each node v with the corresponding function $f_{\mathcal{D}_v}$ as in Figure 5a, where each function in the tree is defined by one of the equations of $S_{\mathcal{D}}$. This representation of $\mathcal{D}$ has the advantage of highlighting the inter-dependencies of the functions defined in $S_{\mathcal{D}}$.

We now discuss how the proof-theoretical properties of $\mathcal{D}$ impose conditions on the equations of $S_{\mathcal{D}}$:

(1) By safety of $\mathcal{D}$, the tree in Figure 5acannot contain the instance of $\mathsf{cut}_\Box$ in Figure 5b. Hence, whenever $S_{\mathcal{D}}$ has functions $f_{\mathcal{D}_u}$ and $f_{\mathcal{D}_v}$ that are inter-dependent, no equation of $S_{\mathcal{D}}$ can define one of these functions by means of a composition along its normal parameters. By inspecting Definition 10 this implies that, whenever an equation in $S_{\mathcal{D}}$ defines a function $f_{\mathcal{D}_u}$ by affecting its normal parameters, u must be the conclusion of an instance of $\mathsf{cond}_\Box$, i.e. this equation must be of the form $f_{\mathcal{D}_u}(\mathsf{s}_i x, \vec{z}; \vec{y}) = f_{\mathcal{D}_v}(x, \vec{z}; \vec{y})$, for some v.
(2) By progressiveness of $\mathcal{D}$, there always exists an occurrence of $\mathsf{cond}_\Box$ in-between two 'backpointers' $\bullet_i$ of Figure 5a.
(3) If $\mathcal{D}$ is left-leaning, then the tree in Figure 5acannot contain the instance of cut_N in Figure 5c.Hence, whenever the system of equations for $\mathcal{D}$ contains two functions $f_{\mathcal{D}_u}$ and $f_{\mathcal{D}_v}$ that are inter-dependent, no equation can define one of these functions by means of a composition along its safe parameters. By inspecting Definition 10 this implies that, whenever an equation in $S_{\mathcal{D}}$ defines a function $f_{\mathcal{D}_u}$ by affecting its safe parameters, u must be the conclusion of an instance of cond_N, i.e. the equation must be of the form $f_{\mathcal{D}_u}(\vec{x}; \vec{y}, \mathsf{s}_i z) = f_{\mathcal{D}_v}(\vec{z}; \vec{y}, z)$, for some v.

We can now simplify the equations in $S_{\mathcal{D}}$ to obtain a 'minimal' system of equations $S^*_{\mathcal{D}}$, where each equation defines a function in $(f_1)_{1 \leq i \leq n}$, with $f_1 = f_{\mathcal{D}}$ and $f_i = f_{\mathcal{D}_{v_i}}$ (see Figure 5a). More precisely, for all $1 \leq i \leq n$ there exists $h_i \in \mathsf{NB}^\subset(a_i)_{1 \leq i \leq n}$, for some oracles $(a_i)_{1 \leq i \leq n}$, such that the following equation is in $S^*_{\mathcal{D}}$:

$$f_i(\vec{x}; \vec{y}) = h_i\left(\lambda \vec{u}.\lambda \vec{v}.f_j(\vec{u}; \vec{v})\right)_{1 \leq j \leq n}(\vec{x}; \vec{y})$$

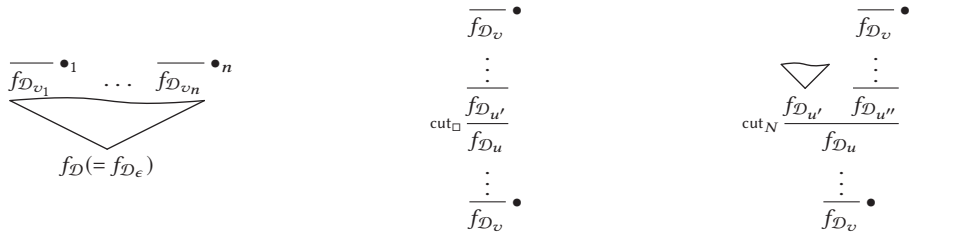
By point 1, each such equation can be rewritten as:

$$f_i(\vec{x}; \vec{y}) = h_i\left(\lambda \vec{u} \subseteq \vec{x}.\lambda \vec{v}.f_j(\vec{u}; \vec{v})\right)_{1 \leq j \leq n}(\vec{x}; \vec{y}) \tag{12}$$

Moreover, by point 2 the relation $\subseteq$ is strict (i.e. $\vec{u} \subset \vec{x}$) when $j = i$. In particular, by point 3, if $\mathcal{D}$ is left-leaning then $h_i \in \mathsf{B}^\subset(a_i)_{1 \leq i \leq n}$ and (12) above can be rewritten as:

$$f_i(\vec{x}; \vec{y}) = h_i\left(\lambda \vec{u} \subseteq \vec{x}.\lambda \vec{v} \subseteq \vec{y}.f_j(\vec{u}; \vec{v})\right)_{1 \leq j \leq n}(\vec{x}; \vec{y})$$

It remains to show that the relation $\subseteq$ is strict (i.e. $\vec{u} \subset \vec{x}$) when $j \neq i$ in the above equations. This can be established by repeatedly



(a) $\mathcal{D}$ as a definition tree. **(b) A $\mathsf{cut}_\square$ violating safety.** **(c) A cut_N violating left-leaning.**

Figure 5: Structure of the definition tree of $\mathcal{D}$.

applying the following operation for each equation in $S^*_{\mathcal{D}}$ with shape (12): for all j, if $\subseteq$ is not strict in $\lambda\vec{u} \subseteq \vec{x}.\lambda\vec{v} \subseteq \vec{y}.f_j(\vec{u};\vec{v})$ (i.e. $\vec{u} \subset \vec{x}$), replace f_j with its definition given by the corresponding equation of $S^*_{\mathcal{D}}$. Such a procedure keeps 'unfolding' equations, and terminates as a consequence of point 2.

Finally, we can establish the main result of this paper:

Corollary 48. *We have the following:*

- $f(\vec{x};) \in \mathsf{CB}$ *if and only if* $f(\vec{x}) \in$ **FPTIME**;
- $f(\vec{x};) \in \mathsf{CNB}$ *if and only if* $f(\vec{x}) \in$ **FELEMENTARY**.

Proof sketch. Soundness ($\Rightarrow$) follows from Lemma 47, by showing that $\mathsf{B}^\subset$ and $\mathsf{NB}^\subset$ are closed under *simultaneous* versions of their recursion schemes. Completeness ($\Leftarrow$) follows from Theorem 43 and Theorem 46. □

7 CONCLUSIONS AND FURTHER REMARKS

In this work we presented two-tiered circular type systems CB and CNB and showed that they capture polynomial-time and elementary computation, respectively. This is the first time that methods of circular proof theory have been applied in implicit computational complexity (ICC). Along the way we gave novel relativised algebras for these classes based on safe (nested) recursion on well-founded relations.

7.1 Unary notation and linear space

It is well-known that **FLINSPACE**, i.e. the class of functions computable in linear space, can be captured by reformulating B in *unary* notation (see [3]). A similar result can be obtained for CB by just defining a unary version of the conditional in $\mathsf{B}^\subset$ (similarly to the ones in [12, 22]) and by adapting the proofs of Lemma 38, Theorem 43 and Lemma 47. On the other hand, CNB is (unsurprisingly) not sensitive to such choice of notation.

7.2 On unnested recursion with compositions

Notice that Lemma 38 implies a polynomial bound on the growth rate of functions in $\mathsf{SB}^\subset$, and hence for functions in SB. We conjecture that both function algebras capture precisely the class **FPSPACE** (but proving this formally is beyond the scope of this work). Indeed, as already observed, the unnested version of the recursion scheme snrec can be replaced by the scheme in (5), which allows both multiple recursive calls and composition during recursion. Several function algebras for **FPSPACE** have been proposed in the literature, and all of them involve variants of (5) (see [25, 29]).

These recursion schemes reflect the parallel nature of polynomial space functions, which in fact can be defined in terms of alternating polynomial time computation. We suspect that a circular proof theoretic characterisation of this class can thus be achieved by extending CB with a 'parallel' version of the cut rule and by adapting the left-leaning criterion appropriately. Parallel cuts might also play a fundamental role for potential circular proof theoretic characterisations of circuit complexity classes, like **ALOGTIME** or **NC**.

7.3 Towards higher-order cyclic implicit complexity

It would be pertinent to pursue higher-order versions of both CNB and CB, in light of precursory works in circular proof theory [12, 22] as well as ICC [6, 19, 24]. In the case of polynomial-time, for instance, a soundness result for some higher-order version of CB might follow by translation to (a sequent-style formulation of) Hofmann's SLR [19]. Analogous translations might be defined for a higher-order version of CNB once the linearity restrictions on the recursion operator of SLR are dropped. Finally, as SLR is essentially a subsystem of Gödel's system T, such translations could refine the results on the abstraction complexity (i.e. type level) of the circular version of system T in [10, 12].

ACKNOWLEDGMENTS

The authors would like to thank Patrick Baillot, Alexis Saurin, Denis Kuperberg, and the anonymous reviewers for useful comments and discussions. This work was supported by a UKRI Future Leaders Fellowship, 'Structure vs Invariants in Proofs', project reference MR/S035540/1.

REFERENCES

[1] David Baelde, Amina Doumane, Denis Kuperberg, and Alexis Saurin. 2020. Bouncing threads for infinitary and circular proofs. *CoRR* abs/2005.08257 (2020). arXiv:2005.08257 https://arxiv.org/abs/2005.08257
[2] David Baelde, Amina Doumane, and Alexis Saurin. 2016. Infinitary Proof Theory: the Multiplicative Additive Case. 62 (2016), 42:1–42:17.
[3] Stephen Bellantoni. 1992. *Predicative recursion and computational complexity.* PhD thesis.
[4] Stephen Bellantoni. 1995. Predicative Recursion and The Polytime Hierarchy. In *Feasible Mathematics II*, Peter Clote and Jeffrey B. Remmel (Eds.). Birkhäuser Boston, Boston, MA, 15–29.
[5] Stephen Bellantoni and Stephen Cook. 1992. A New Recursion-Theoretic Characterization of the Polytime Functions (Extended Abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing* (Victoria, British Columbia, Canada) *(STOC '92)*. Association for Computing Machinery, New York, NY, USA, 283–293.

[6] Stephen J. Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. 2000. Higher type recursion, ramification and polynomial time. *Ann. Pure Appl. Log.* 104, 1-3 (2000), 17–30.

[7] Stefano Berardi and Makoto Tatsuta. 2017. Equivalence of inductive definitions and cyclic proofs under arithmetic. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12.

[8] James Brotherston and Alex Simpson. 2011. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation* 21, 6 (2011), 1177–1216.

[9] Gianluca Curzi and Anupam Das. 2021. Cyclic Implicit Complexity. *CoRR* abs/2110.01114 (2021). arXiv:2110.01114 https://arxiv.org/abs/2110.01114

[10] Anupam Das. 2020. A circular version of Gödel's T and its abstraction complexity. *CoRR* abs/2012.14421 (2020). arXiv:2012.14421 https://arxiv.org/abs/2012.14421

[11] Anupam Das. 2020. On the logical complexity of cyclic arithmetic. *Log. Methods Comput. Sci.* 16, 1 (2020).

[12] Anupam Das. 2021. On the Logical Strength of Confluence and Normalisation for Cyclic Proofs. In *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference) (LIPIcs, Vol. 195)*, Naoki Kobayashi (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 29:1–29:23.

[13] Anupam Das and Damien Pous. 2017. A cut-free cyclic proof system for Kleene algebra. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 261–277.

[14] Anupam Das and Damien Pous. 2018. Non-Wellfounded Proof Theory For (Kleene+Action)(Algebras+Lattices). In *27th EACSL Annual Conference on Computer Science Logic (CSL 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 119)*, Dan Ghica and Achim Jung (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:18.

[15] Christian Dax, Martin Hofmann, and Martin Lange. 2006. A Proof System for the Linear Time μ-Calculus. In *FSTTCS 2006: Foundations of Software Technology and Theoretical Computer Science, 26th International Conference, Kolkata, India, December 13-15, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4337)*, S. Arun-Kumar and Naveen Garg (Eds.). Springer, 273–284.

[16] Christian Dax, Martin Hofmann, and Martin Lange. 2006. A proof system for the linear time μ-calculus. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer, 273–284.

[17] Abhishek De and Alexis Saurin. 2019. Infinets: The Parallel Syntax for Non-wellfounded Proof-Theory. In *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11714)*, Serenella Cerrito and Andrei Popescu (Eds.). Springer, 297–316.

[18] Jérôme Fortier and Luigi Santocanale. 2013. Cuts for circular proofs: semantics and cut-elimination. In *Computer Science Logic 2013 (CSL 2013)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[19] Martin Hofmann. 1997. A Mixed Modal/Linear Lambda Calculus with Applications to Bellantoni-Cook Safe Recursion. In *Computer Science Logic, 11th International Workshop, CSL '97, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers (Lecture Notes in Computer Science, Vol. 1414)*, Mogens Nielsen and Wolfgang Thomas (Eds.). Springer, 275–294.

[20] Stephen Cole Kleene. 1971. *Introduction to Metamathematics* (7 ed.). Wolters-Noordhoff Publishing.

[21] Leszek Aleksander Kolodziejczyk, Henryk Michalewski, Pierre Pradic, and Michal Skrzypczak. 2019. The logical strength of Büchi's decidability theorem. *Log. Methods Comput. Sci.* 15, 2 (2019).

[22] Denis Kuperberg, Laureline Pinault, and Damien Pous. 2021. Cyclic proofs, system t, and the power of contraction. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–28.

[23] Daniel Leivant. 1991. A Foundational Delineation of Computational Feasiblity. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*. IEEE Computer Society, 2–11.

[24] Daniel Leivant. 1999. Ramified Recurrence and Computational Complexity III: Higher Type Recurrence and Elementary Complexity. *Ann. Pure Appl. Log.* 96, 1-3 (1999), 209–229.

[25] Daniel Leivant and Jean-Yves Marion. 1994. Ramified Recurrence and Computational Complexity II: Substitution and Poly-Space. In *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers (Lecture Notes in Computer Science, Vol. 933)*, Leszek Pacholski and Jerzy Tiuryn (Eds.). Springer, 486–500.

[26] Grigori E Mints. 1978. Finite investigations of transfinite derivations. *Journal of Soviet Mathematics* 10, 4 (1978), 548–596.

[27] Damian Niwiński and Igor Walukiewicz. 1996. Games for the μ-calculus. *Theoretical Computer Science* 163, 1-2 (1996), 99–116.

[28] Rémi Nollet, Alexis Saurin, and Christine Tasson. 2019. PSPACE-Completeness of a Thread Criterion for Circular Proofs in Linear Logic with Least and Greatest Fixed Points. In *Automated Reasoning with Analytic Tableaux and Related Methods - 28th International Conference, TABLEAUX 2019, London, UK, September 3-5, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11714)*, Serenella Cerrito and Andrei Popescu (Eds.). Springer, 317–334.

[29] Isabel Oitavem. 2008. Characterizing PSPACE with pointers. *Math. Log. Q.* 54, 3 (2008), 323–329.

[30] Alex Simpson. 2017. Cyclic Arithmetic Is Equivalent to Peano Arithmetic. In *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10203)*, Javier Esparza and Andrzej S. Murawski (Eds.). 283–300.

[31] Marc Wirz. 1999. Characterizing the Grzegorczyk hierarchy by safe recursion.