



HAL
open science

Cyclic proofs, system T, and the power of contraction

Denis Kuperberg, Laureline Pinault, Damien Pous

► **To cite this version:**

Denis Kuperberg, Laureline Pinault, Damien Pous. Cyclic proofs, system T, and the power of contraction. 2020. hal-02487175v1

HAL Id: hal-02487175

<https://hal.science/hal-02487175v1>

Preprint submitted on 21 Feb 2020 (v1), last revised 24 Nov 2020 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cyclic proofs, system T, and the power of contraction*

Denis Kuperberg

Plume team, LIP, CNRS, ENS de Lyon
Lyon, France

Laureline Pinault

Plume team, LIP, CNRS, ENS de Lyon
Lyon, France

Damien Pous

Plume team, LIP, CNRS, ENS de Lyon
Lyon, France

Abstract

We study a cyclic proof system C over regular expression types, inspired by linear logic and non-wellfounded proof theory. Proofs in C denote total computable functions; we analyse the relative strength of C and Gödel’s system T , showing that contraction plays a crucial role. In the general case, we show that the two systems capture the same functions on natural numbers. In the affine case, we manage to give a direct and uniform encoding of C into T , translating cycles into explicit recursions. We also show that for functions on natural numbers, removing contraction reduces the expressivity precisely to primitive recursive functions—providing an alternative and more general proof of a result by Dal Lago.

The two upper bounds on the expressivity of C w.r.t. functions on natural numbers are obtained by formalising weak normalisation of a small step reduction semantics in subsystems of second-order arithmetic: ACA_0 and RCA_0 .

Whether a direct and uniform translation from C to T can be given in the presence of contraction remains open.

1 Introduction

In recent years there has been a surge of interest in the theory of non-wellfounded proofs. This is an approach to infinitary proof theory where proofs remain finitely branching but are permitted to be infinitely deep. A correctness criterion is usually required to guarantee consistency, typically some ω -regular condition on the infinite branches. Proofs whose dependency graphs are regular trees are known as *cyclic* proofs; being finite objects, they can be exchanged and checked, thus playing the role of traditional *inductive* proofs. A natural question is whether specific cyclic and inductive proof systems have the same logical strength. Inductive proofs can usually be translated easily into cyclic ones (see, e.g., [9]), while the converse problem is often harder [7, 29], or impossible [6, 12]. Cyclic proofs systems have been recently used in the context of the mu-calculus [2, 16] and Kleene algebra [13–15], in order to obtain completeness results, and in the context of linear logics [17, 18]

Here we propose a cyclic proof system which we study from the other side of the Curry-Howard correspondence. We look at cyclic proofs as computational devices, and we characterise their computational strength in terms of more

traditional devices: primitive recursive functions and Gödel’s system T (i.e., simply typed lambda-calculus with natural numbers and recursion).

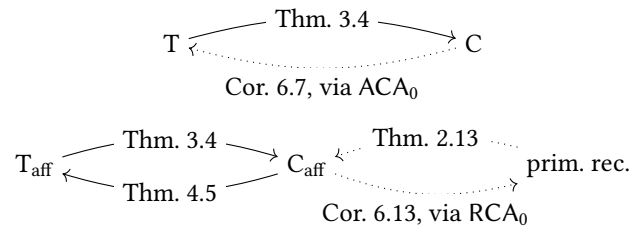
We consider the formulas of intuitionistic multiplicative additive linear logic (IMALL) with a least fixpoint operator for lists. We can thus manipulate datatypes consisting of natural numbers and functions, but also pairs, lists, or sums, without the need for encodings. Our cyclic proof system, which we call *system C*, is basically the sequent system LAL for action lattices from [15], to which we add the three usual structural rules: exchange, weakening and contraction. Proceeding this way makes it possible to consider the *affine* fragment C_{aff} of C , where the contraction rule is forbidden.

Accordingly, we use a variant of Gödel’s system T with the same formulas/types as C in order to ease comparisons. We define this type system in a slightly non-standard way: like for C , we use explicit structural rules in order to be able to talk about the affine fragment T_{aff} of T .

Contraction indeed plays an important role in those systems: we show that

1. affine C and affine T are equally expressive (at all types), and their functions on the natural numbers (\mathbb{N}) are the primitive recursive functions;
2. C and T capture the same functions on \mathbb{N} .

We obtain those results via the translations summarised below, where dotted arrows denote encodings restricted to functions on natural numbers.



As expected, we can easily translate terms of T into cyclic proofs of C (Thm. 3.4); this translation is uniform and maps affine terms to affine proofs. We also observe that we do not need contraction to encode primitive recursive functions into C (Thm. 2.13).

Encoding cyclic proofs into T is much harder: we have to delineate possibly complex cycle structures in order to use the very basic recursion capacities of T . We provide a direct and uniform encoding in the affine case (Thm. 4.5), which we do not know how to extend in presence of contraction.

In order to get our upper bounds on the expressivity of C and affine C for functions on \mathbb{N} (Cor. 6.7 and Cor. 6.13), we

*This work has been supported by the European Research Council (ERC) under the European Union’s Horizon 2020 programme (CoVeCe, grant agreement No 678157), and by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon, within the program "Investissements d’Avenir" (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

define a small steps reduction semantics for C. This semantics matches the higher-level and higher-order semantics we use elsewhere in the paper, and we prove that it is weakly normalising. We obtain Cor. 6.7 by observing that this weak normalisation proof can be performed inside the subsystem ACA_0 of second order arithmetic [30], whose provably recursive functions are precisely those from system T.

For the affine case (Cor. 6.13), Dal Lago’s system $\mathcal{H}(\emptyset)$ [27] is a variant of Gödel’s system T which characterises primitive recursive functions and which is really close to our affine version of T. Unfortunately, we need additive pairs in order to translate affine C into affine T. Those are not available in $\mathcal{H}(\emptyset)$, and it is not clear how to extend Dal Lago’s proof to deal with such operations: his proof is complex and relies on a semantics based on geometry of interaction, whose extension to additives is notoriously difficult [1, 5, 19, 23].

We actually prove Cor. 6.13 by using another proof of weak normalisation, which works only on the image of our translation from affine T to C. This argument can be formalised into another subsystem of second order arithmetic, RCA_0 , which is known to define only the primitive recursive functions [3]. This eventually gives us an alternative and more general proof of Dal Lago’s result.

Related work System T was originally introduced by Gödel in [21] as an equational theory built up over a fragment of the term calculus that we identify as T here. That work introduced the celebrated ‘Dialectica’ *functional interpretation*, that allows T to interpret Peano Arithmetic.¹ Our work can be seen as a natural counterpart in T to recent work on cyclic versions of arithmetic [12, 29].

Other infinitary versions of system T are well-known, in particular [31]. These also induce a ‘term model’ of T where recursors are replaced by infinitely long yet well-founded terms. This difference resembles the difference between logical systems with ω -branching versus their non-wellfounded counterparts, e.g. as in arithmetic [12, 29].

The role of contraction w.r.t expressivity we exhibit in the present work is reminiscent of a recent result [26]: in a specific cut-free fragment of C, affine proofs capture precisely the regular languages while proofs with contraction capture the DLOGSPACE ones.

Notation. Given two sets X, Y , we write $X \times Y$ for their Cartesian product, $X + Y$ for their disjoint union, Y^X for the set of functions from X to Y , and X^* for the set of finite sequences (lists) over X . Given such a sequence l , we write $|l|$ for its length and l_i for its i th element. We write 1 for the singleton set $\{\langle \rangle\}$ and $\langle x, y, z \rangle$ for tuples. We use commas to denote concatenation of both sequences and tuples, and ϵ or just a blank to denote the empty sequence.

¹Gödel only treated Heyting Arithmetic, the intuitionistic counterpart of Peano Arithmetic. An interpretation of the latter is duly obtained by composition with an appropriate double-negation translation.

2 System C and its semantics

2.1 Regular expressions as types

We let the letters a, b range over the elements of a fixed set A of *type variables*. We define *types* with the following syntax.

$$e, f := a \mid e \cdot f \mid e + f \mid e^* \mid 1 \mid e \rightarrow f \mid e \cap f$$

The five first entries correspond to regular expressions; the arrow adds function spaces. The role of the intersection operator will be explained later. We call types *formulas* when this is more natural.

We assume a family $(D_a)_{a \in A}$ of sets indexed by A . To every type e , we associate a set $[e]$ of *values*, by induction on e :

$$[e \cdot f] \triangleq [e] \times [f] \quad [e + f] \triangleq [e] + [f] \quad [e \rightarrow f] \triangleq [f]^{[e]}$$

$$[e^*] \triangleq [e]^* \quad [a] \triangleq D_a \quad [1] \triangleq 1 \quad [e \cap f] \triangleq [e] \times [f]$$

We let E, F range over finite sequences of types. Given such a sequence $E = e_1, \dots, e_n$, we write $[E]$ for $[e_1] \times \dots \times [e_n]$. We define a sequent proof system, where sequents have the shape $E \vdash e$, and where proofs of such sequents denote functions from $[E]$ to $[e]$.

2.2 Non-wellfounded proofs

The rules are given in Fig. 1; in addition to the *structural rules* (exchange, weakening, contraction, axiom, and cut), we have introduction rules on the left and on the right for each type connective (*logical rules*). Those rules are standard, they are those of intuitionistic multiplicative additive linear logic, when interpreting \cdot as multiplicative conjunction (\otimes), $+$ as additive disjunction (\oplus), \cap as additive conjunction ($\&$), and \rightarrow as linear arrow (\multimap). The rules for type e^* correspond to unfolding rules, looking at e^* as the least fixpoint expression $\mu x. 1 + e \cdot x$ (e.g., from the μ -calculus).

Those rules are also essentially the same as those used for action lattices in [15]. The only differences are that they can be slightly simplified here since we have the exchange rule, and that we have only one arrow operation, being in a commutative setting (again, due to the exchange rule).

A (binary, possibly infinite) *tree* is a non-empty and prefix-closed subset of $\{0, 1\}^*$, which we view with the root, ϵ , at the bottom; elements of $\{0, 1\}^*$ are called *addresses*.

Definition 2.1. A *preproof* is a labelling π of a tree by sequents such that, for every node v with children v_1, \dots, v_n

($n = 0, 1, 2$), the expression $\frac{\pi(v_1) \cdots \pi(v_n)}{\pi(v)}$ is an instance

of a rule from Fig. 1. Given an address v in a preproof π , we write π_v for the sub-preproof rooted at v , defined by $\pi_v(w) = \pi(vw)$. A preproof is *regular* if it has finitely many distinct subtrees. A preproof is *cut-free* (resp. *affine*, *linear*) if it does not use the *cut* rule (resp. *c* rule, *c* and *w* rules).

We write \sqsubseteq (resp. \sqsubset) for the prefix relation (resp. strict prefix) on $\{0, 1\}^*$. The formula e in an instance of the *cut* rule is called the *cut formula*; the formulas appearing in lists

$$\begin{array}{c}
 \frac{E, f, e, F \vdash g}{E, e, f, F \vdash g} \quad \frac{E \vdash g}{e, E \vdash g} \quad \frac{e, e, E \vdash g}{e, E \vdash g} \\
 \\
 \frac{id}{e \vdash e} \quad \frac{cut}{\frac{E \vdash e \quad e, F \vdash g}{E, F \vdash g}} \\
 \\
 \frac{-l}{\frac{e, f, E \vdash g}{e \cdot f, E \vdash g}} \quad \frac{-r}{\frac{E \vdash e \quad F \vdash f}{E, F \vdash e \cdot f}} \\
 \\
 \frac{+l}{\frac{e, E \vdash g \quad f, E \vdash g}{e + f, E \vdash g}} \quad \frac{+r_i}{\frac{E \vdash e_i}{E \vdash e_0 + e_1}} \quad i \in \{0, 1\} \\
 \\
 \frac{*l}{\frac{E \vdash g \quad e, e^*, E \vdash g}{e^*, E \vdash g}} \quad \frac{*r_e}{\frac{}{\vdash e^*}} \quad \frac{*r_f}{\frac{E \vdash e \quad F \vdash e^*}{E, F \vdash e^*}} \\
 \\
 \frac{1-l}{\frac{E \vdash g}{1, E \vdash g}} \quad \frac{1-r}{\frac{}{\vdash 1}} \\
 \\
 \frac{->l}{\frac{E \vdash e \quad f, F \vdash g}{e \rightarrow f, E, F \vdash g}} \quad \frac{->r}{\frac{e, E \vdash f}{E \vdash e \rightarrow f}} \\
 \\
 \frac{\cap-l_i}{\frac{e_i, E \vdash g}{e_0 \cap e_1, E \vdash g}} \quad i \in \{0, 1\} \quad \frac{\cap-r}{\frac{E \vdash e \quad E \vdash f}{E \vdash e \cap f}}
 \end{array}$$

Figure 1. The rules of C.

E, F of any rule instance are called *auxiliary formulas*, and the non auxiliary formula appearing in the antecedent of the conclusion of the logical rules is called the *principal formula*.

Three examples of regular preproofs are depicted in Fig. 2. The backpointers are used to denote circularity: the actual preproofs are obtained by unfolding. Only the topmost pre-proof satisfies the validity criterion which we define below. Before that, we need to define a notion of thread, which are the branches of the shaded trees depicted on the preproofs.

All rules but the cut rule have the subformula property: every formula appearing in the premisses is a subformula of one of the formulas appearing in the conclusion, usually called its immediate descendant in the literature. We use a slightly stricter notion of ancestry in the present paper.

Definition 2.2. A *position* in a preproof π is a pair $\langle v, i \rangle$ consisting of an address v and an index i such that $\pi(v) = E \vdash f$ and E_i is a star formula. A **-l address* is an address pointing at the conclusion of a *-l step; $\langle v, i \rangle$ is a **-l position* when v is a *-l address and $i = 0$.

A position $\langle v, i \rangle$ is a *parent* of a position $\langle w, j \rangle$ if $|v| = |w| + 1$ and, looking at the rule applied at address w the two positions point at the same place in the lists E, F of auxiliary formulas, or at the formula e (resp. e or f) when this is the contraction rule (resp. exchange rule), or at the principal formula e^* when this is the *-l rule and $v = w1$. We write $\langle v, i \rangle \triangleleft \langle w, j \rangle$ in the former cases, and $\langle v, i \rangle \triangleleft \langle w, j \rangle$ in the latter case (in which case $i = 1$ and $j = 0$). $\langle v, i \rangle$ is an *ancestor*

of $\langle w, j \rangle$ when those positions are related by the transitive closure of the parentship relation.

The graph of the parentship relation is depicted in Fig. 2 using shaded thick lines and an additional bullet to indicate when we pass principal steps (\triangleleft). Note that in rule *-l, the occurrence of e in the second premiss is not a parent of e^* in the conclusion. Due to this restriction, positions linked by the ancestry relation all point to the same star formula.

Remark 2.3. Notice that if $u \sqsubseteq v$ are addresses in a preproof π , then a position at v has at most one ancestor at u . Moreover, it is only in the presence of contraction that a position at u may have two ancestors at v .

Definition 2.4. A *thread* is a branch of the ancestry graph; it is *principal* when it visits a *-l position, *spectator* if it is never principal, *valid* if it is principal infinitely many often.

In the first preproof of Fig. 2, the infinite red thread $\langle \epsilon, 0 \rangle \triangleright \langle 1, 1 \rangle \triangleright \langle 10, 0 \rangle \triangleright \langle 101, 1 \rangle \triangleright \langle 1010, 0 \rangle \dots$ is valid while the infinite green thread $\langle \epsilon, 1 \rangle \triangleright \langle 1, 2 \rangle \triangleright \langle 10, 1 \rangle \triangleright \langle 101, 2 \rangle \triangleright \langle 1010, 1 \rangle \dots$ is spectator. In the second preproof, all threads are finite: the instances of the *cut* rule disconnect the various copies of the thread $\langle \epsilon, 0 \rangle \triangleright \langle 1, 1 \rangle$ occurring in the only infinite branch of the preproof. In the third preproof, all infinite threads are spectator: principal steps force the thread to terminate.

Definition 2.5. A preproof is *valid* if every infinite branch contains a valid thread. A *proof* is a valid preproof. We write $\pi : E \vdash e$ when π is a proof whose root is labelled by $E \vdash e$.

In Fig. 2, only the first preproof is valid, thanks to the infinite red thread. The second preproof is invalid: every thread is finite. The third preproof is invalid: infinite threads along the (infinitely many) infinite branches are all spectator.

This validity criterion is essentially the same as in LKA [15], which in turn is an instance of the one for μ MALL [17]: we just had to extend the notion of ancestry to cover the weakening and contraction rules. This induces some subtleties:

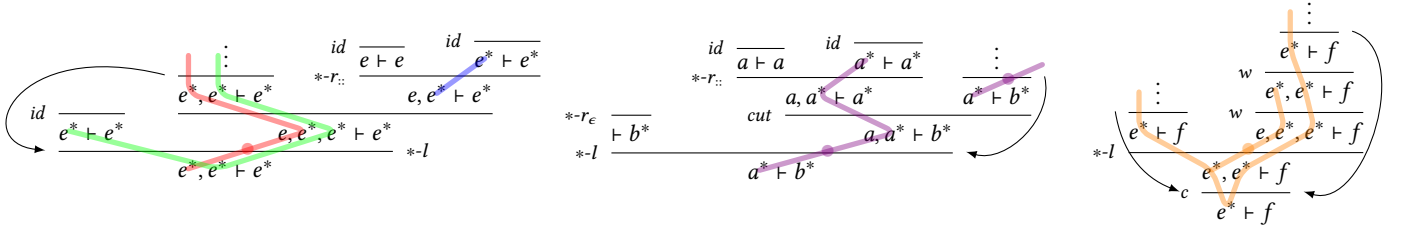
Remark 2.6. In a fixed branch of an affine preproof, every maximal thread is determined by its first element (a position). This is not true with contraction since we can choose which parent position to follow at each contraction step.

2.3 Computational interpretation of system C

We now show how to interpret a proof $\pi : E \vdash e$ as a function $[\pi] : [E] \rightarrow [e]$. Since proofs are not well-founded, we cannot reason directly by induction on proofs. We use instead the following relation, which we prove to be well-founded.

Definition 2.7. A *computation* in a fixed proof π is a pair $\langle v, s \rangle$ consisting of an address v of π with $\pi(v) = E \vdash e$, and a value $s \in [E]$. Given two computations, we write $\langle v, s \rangle < \langle w, t \rangle$ when $|v| = |w| + 1$ and

1. for all i, j s.t. $\langle v, i \rangle \triangleleft \langle w, j \rangle$, we have $s_i = t_j$, and
2. for all i, j s.t. $\langle v, i \rangle \triangleleft \langle w, j \rangle$, we have $|s_i| < |t_j|$.


Figure 2. Three regular preproofs.

The two conditions state that the values assigned to star formulas should remain the same along auxiliary steps and decrease in length along principal steps.

Lemma 2.8. *The relation $<$ on computations is well-founded.*

Proof. An infinite descending sequence would correspond to an infinite branch of π . This branch would contain a valid thread, which is forbidden by 1/ and 2/: we would obtain an infinite sequence of lists of decreasing length. \square

Definition 2.9. The *return value* $[v](s)$ of a computation $\langle v, s \rangle$ with $\pi(v) = E \vdash e$ is a value in $[e]$ defined by well-founded induction on $<$ and case analysis on the rule used at address v . We list only the most interesting cases below; see App. A.1 for the complete enumeration.

$$\begin{aligned}
 id &: [v](s) \triangleq s \\
 cut &: [v](s, t) \triangleq [v1]([v0](s), t) \\
 c &: [v](x, s) \triangleq [v0](x, x, s) \\
 \cdot r &: [v](s, t) \triangleq \langle [v0](s), [v1](t) \rangle \\
 \rightarrow l &: [v](h, s, t) \triangleq [v1](h([v0](s)), t) \\
 \rightarrow r &: [v](h) \triangleq (x \mapsto [v0](x, h)) \\
 *l &: [v](l, s) \text{ is defined by case analysis on the list } l: \\
 &\quad \bullet [v](\epsilon, s) \triangleq [v0](s) \\
 &\quad \bullet [v](x :: q, s) \triangleq [v1](x, q, s)
 \end{aligned}$$

In each case, the recursive calls are made on strictly smaller computations: they occur on direct subproofs, the values associated to auxiliary formulas are left unchanged, and in the second subcase of the $*l$ case, the length of the list associated to the principal formula decreases by one.

Note that in the *cut* and $\rightarrow l$ cases, the values $[v0](s)$ and $h([v0](s))$ might be arbitrary large. This is not problematic: the corresponding positions have no children, so that those values are left unconstrained by the relation $<$.

Definition 2.10. The semantics of a proof $\pi : E \vdash e$ is the function $[\pi] : [E] \rightarrow [e]$ defined by $[\pi](s) \triangleq [\epsilon](s)$.

Let us compute the semantics of the first (and only) proof in Fig. 2. We have

$$\begin{aligned}
 [\epsilon](\epsilon, l) &= [0](l) = l \\
 [\epsilon](x :: q, l) &= [1](x, q, l) = [11](x, [10](q, l)) \\
 &= [110](x) :: [111]([10](q, l)) \\
 &= x :: [10](q, l) = x :: [\epsilon](q, l)
 \end{aligned}$$

$$\begin{array}{c}
 \text{rem} \frac{\frac{}{e \vdash 1} \quad \frac{}{1, E \vdash f}}{e, E \vdash f} \quad \frac{}{E \vdash f} \\
 \text{cut} \\
 \text{dup} \frac{\frac{}{e \vdash e \cdot e} \quad \frac{}{e, e, E \vdash f}}{e, E \vdash f} \quad \frac{}{e, e, E \vdash f} \\
 \text{cut}
 \end{array}$$

Figure 3. Deriving weakening and contraction.

In the last equality we used the fact that $\pi_{01} = \pi_\epsilon$, so that $[01] = [\epsilon]$. We recognise for $[\epsilon]$ the standard definition of list concatenation, which is recursive on its first argument. Trying to perform such computations on the two invalid preproofs from Fig. 2 would give rise to non-terminating behaviours, e.g., $[\epsilon](x :: q) \rightsquigarrow [11](x :: q) = [\epsilon](x :: q)$ in the second preproof.

2.4 Weakening and contraction

A type is *closed* when it does not contain variables; it is *positive* when it does not contain negative connectives (\rightarrow, \cap).

Lemma 2.11. *For every closed type e , there is a linear regular proof $\text{rem}_e : e \vdash 1$.*

Proof. By induction on e , see App. A.2. \square

As a consequence, weakening is admissible for closed types, by replacing it with the gadget on the left in Fig. 3.

The linear system also allows for some form of duplication: while arrow types cannot be duplicated, basic types such as natural numbers (1^*) or lists of natural numbers (1^{**}) can.

Lemma 2.12. *For every positive closed type e , there is a linear regular proof $\text{dup}_e : e \vdash e \cdot e$ such that for all $x \in [e]$, $[\text{dup}_e](x) = \langle x, x \rangle$.*

Proof. Again by induction on e , see App. A.2. \square

Like above, it follows that positive closed instances of the contraction rule are derivable in the linear system using the gadget on the right in Fig. 3. However, they are not admissible in general: the gadget does cut the potential threads on the contracted formula, so that it cannot be freely used in arbitrary proofs. For instance, anticipating on Sect.2.5 below, if we use it to replace the contraction on a star formula in the proof from Fig. 5, the affine preproof we obtain is not valid: the green thread is cut at each iteration. Actually,

if contraction on closed types was derivable in a thread-preserving way, and thus admissible, we would obtain a counter-example to Cor. 6.13 below.

2.5 Functions on natural numbers

Natural numbers can be represented through the type 1^* of lists over the singleton set. The logical rules for this specific instance of the star type can be optimised as follows:

$$1^* \text{-}l \frac{E \vdash g \quad 1^*, E \vdash g}{1^*, E \vdash g} \quad 1^* \text{-}r_0 \frac{}{\vdash 1^*} \quad 1^* \text{-}r_s \frac{E \vdash 1^*}{E \vdash 1^*}$$

Those rules are immediate consequences of the logical rules for 1 and star. Using these rules, we deduce that for all $n \in \mathbb{N}$, we can build a finite proof $\underline{n} : \vdash 1^*$ such that $[\underline{n}]() = n$.

Similarly, for every function (even an uncomputable one) $f : \mathbb{N} \rightarrow \mathbb{N}$, we can obtain a proof $\underline{f} : 1^* \vdash 1^*$ such that $[\underline{f}] = f$: repeatedly apply the $1^* \text{-}l$ rule to obtain a comb-shape infinite tree, and fill the remaining leaves with finite proofs for the successive values of the function. This proof, which is essentially the graph of the function f , is linear and cut-free, but not regular in general.

Our first expressivity result for regular proofs is:

Theorem 2.13. *For every primitive recursive function $f : \mathbb{N} \times \dots \times \mathbb{N} \rightarrow \mathbb{N}$, there exists a linear and regular proof $\pi : 1^* \times \dots \times 1^* \vdash 1^*$ such that $[\pi] = f$.*

Proof. By induction on definition scheme for primitive recursive functions. The constant 0-ary function and the successor 1-ary functions give rise to simple finite proofs. The projection functions just require weakening for 1^* (Lem. 2.11). Function composition is implemented using the *cut* rule, as expected, but it also requires duplicating the arguments to provide them to the composed functions. For instance, to compose a 2-ary function h with two 1-ary functions f, g , we use the following scheme:

$$\text{cut} \frac{\text{cut} \frac{\pi_f}{1^* \vdash s} \quad \text{cut} \frac{\pi_g}{1^* \vdash t} \quad \pi_h}{s, t \vdash r}}{1^*, 1^* \vdash r} \quad c' \frac{}{1^* \vdash r}$$

We used the abbreviations $r = s = t = 1^*$ to distinguish between the respective return types of h, f and g , and we marked with c' our usage of the derivable contraction rule (Lem. 2.12). That this step cuts the threads is not problematic here: cycles cannot visit this contraction step.

It remains to deal with primitive recursion. Suppose f is defined by primitive recursion:

$$\begin{cases} f \ 0 \ \vec{y} & = g \ \vec{y} \\ f \ (Sx) \ \vec{y} & = h \ x \ (f \ x \ \vec{y}) \ \vec{y} \end{cases}$$

where g and h are primitive recursive functions of respective arity n and $n+2$. By induction hypothesis we have π_g and π_h ,

proofs that encode g and h . In the recursive definition above, one can observe that both x and the \vec{y} are used twice. The latter can easily be handled using the derivable contraction rule since they are not involved in the termination argument. On the contrary, the duplication of x is problematic since the corresponding thread should validate the recursion. To circumvent this difficulty, we perform a recursion that returns a copy of the recursive argument together with the expected return value. We write E for the sequence of 1^* s of length n (i.e., the types for \vec{y}). We use $r = 1^*$ to denote the return type of the primitive recursion scheme, and $e^* = 1^*$ to denote the type of the recursive argument. We set $r' = e^* \cdot r$ and we construct the proof in Fig. 4. \square

Note that when displaying proofs, we omit usages of the exchange rule, which typically make it possible to apply left introduction rules on arbitrary formulas rather than just on the first one. Moreover, we sometimes abbreviate sequences of steps or standalone proofs using double bars.

The above argument works in the fragment of C without arrows, sums, and intersections, and where star and unit are replaced with a base type for natural numbers together with the dedicated rules for 1^* . Pairs are necessary to avoid using the contraction rule and remain in the affine fragment.

As announced in the introduction, the contraction rule makes it possible to go beyond primitive recursion:

Example 2.14. We give a regular proof whose semantics is Ackermann-Péter's function in Fig. 5. The subproof labelled with S is a proof for the successor function. The subproof labelled with 1 is a proof for the constant value 1.

The preproof is valid: every infinite branch either goes infinitely often through loops (a) or (a') , in which case it is validated by the green thread where we go right on contraction steps whenever the next visited backpointer is a (b) ; or it eventually goes only through loop (b) , in which case it is validated by the red thread.

Its semantics satisfies the same recursive equations as those defining Ackermann's function $A(n, k)$: we have

$$\begin{aligned} [\epsilon](n, k) &= [0](n, n, k) = A(n, k) \\ [01](n, Sn, k) &= A(Sn, k) \\ [00](_, k) &= [000](k) = A(0, k) = Sk \\ [010](n, _) &= [0100](n) = A(Sn, 0) = A(n, 1) \\ [011](n, Sn, k) &= A(Sn, Sk) = A(n, A(Sn, k)) \quad \square \end{aligned}$$

We prove in the next section that we can actually represent all system T functions with regular proofs, the class of which we call *system C*. We can go beyond total functions by forgetting the validity criterion: we can encode the minimisation operator using a regular but invalid preproof, so that every computable partial function can be represented by a regular preproof (see App. A.3).

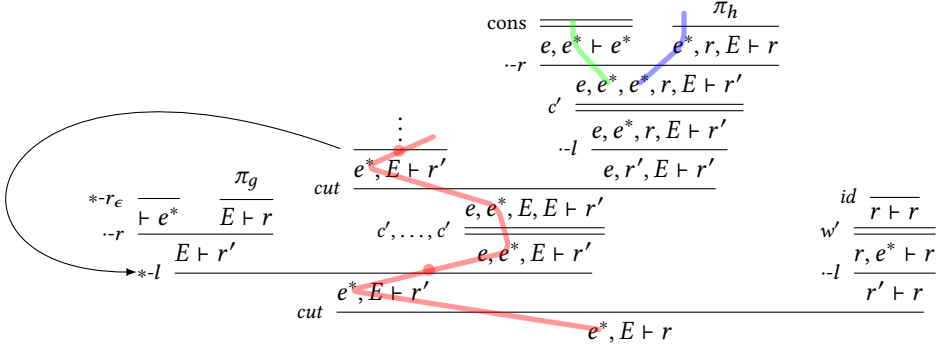


Figure 4. Regular linear proof for primitive recursion; $e \triangleq 1$, $r \triangleq 1^*$; $r' \triangleq e^* \cdot r$.

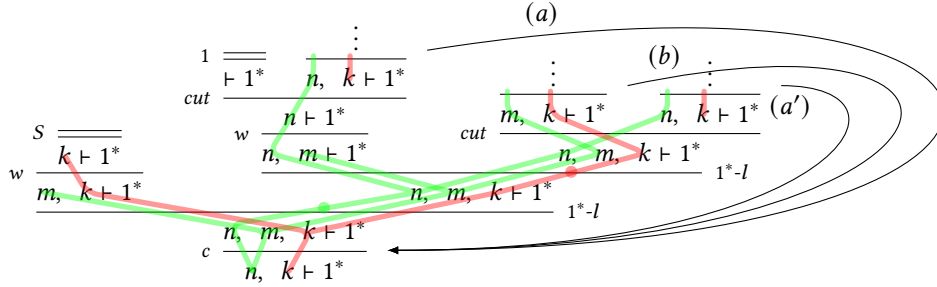


Figure 5. A regular proof for Ackermann-Péter's function; $n \triangleq m \triangleq k \triangleq 1^*$.

3 Extended, resource-tracking system T

We define in this section the variant of system T we will work with. We use the following syntax for terms, where x ranges over a set of *variables* and i ranges over $0, 1$.

$M, N, O ::= x$	$\lambda x.M$	MN
	$\langle M, N \rangle$	$\text{let } \langle x, y \rangle := M \text{ in } N$
	$\langle \rangle$	$\text{let } \langle \rangle := M \text{ in } N$
	$i_i M$	$D(M; x.N; x.O)$
	$\llbracket \rrbracket M :: N$	$R(M; N; x.y.O)$
	$\langle\langle M, N \rangle\rangle$	$p_i M$

It consists of a lambda-calculus extended with pairs, singletons, sums, lists, and additive pairs. We let Γ, Δ range *typing environments*, i.e., lists of pairs of a variable and a type. The type system is given in App. B (Fig. 9). Unlike for C, typing derivations are just finite trees built from the rules, as usual. This type system however departs from the standard presentations in that it keeps track of the usage of resources: the rules for the various connectives are those of a linearly typed lambda-calculus. We include contraction and weakening rules (c, w), so that the standard typing rules for system T are all admissible.

The structural and introduction rules are term-decorated versions of the corresponding rules of C (Fig. 1). In contrast, the elimination rules differ: they follow the ‘natural deduction’ scheme and each of them intuitively contains a *cut* on the corresponding formula.

Let us focus on the recursion operator on lists (R). This operator expects a list as first argument, and then two arguments for the case of the empty list and for the case of a non-empty list. Intuitively, we have

$$\begin{aligned} \mathbf{R}(\llbracket \rrbracket; M; x.y.N) &= M \\ \mathbf{R}(X::Q; M; x.y.N) &= N\{x \leftarrow X; y \leftarrow \mathbf{R}(Q; M; x.y.N)\} \end{aligned}$$

Note that this is an *iterator* rather than a *recursor*: the tail of the list (Q) is not given to N . This is not a restriction since recursors can be encoded from iterators and pairs. Its (elimination) typing rule is the following one:

$$**e \frac{\Gamma \vdash L : e^* \quad \Delta \vdash M : g \quad x : e, y : g \vdash N : g}{\Gamma, \Delta \vdash \mathbf{R}(L; M; x.y.N) : g}$$

Like in Dal Lago’s system $\mathcal{H}(\emptyset)$ [27], the important point is that the third argument (the one being iterated) is typed in the empty environment—except for its two variables x for the head of the list and y for the value of the recursive call on the tail of the list. This is crucial in the affine system to get a linear recursion operator; this is not a restriction in the full system, thanks to arrows and contraction (see App. B.1).

Terms should always be considered equipped with their typing derivation. A typed term is *affine* (resp. *linear*) when its typing derivation does not use c (resp. c and w).

Given a typing environment $\Gamma = x_1 : e_1, \dots, x_n : e_n$, we write $\underline{\Gamma}$ for the list of types e_1, \dots, e_n .

Definition 3.1. The semantics of a typed term $\Gamma \vdash M : e$ is the function $[M] : [\Gamma] \rightarrow [e]$ defined as follows by induction on the typing derivation:

$$\begin{aligned} id &: [x](s) \triangleq s \\ \rightarrow e &: [MN](s, t) \triangleq [M](s)([N](t)) \\ c &: [M](v, s) \triangleq [M](v, v, s) \\ \cdot i &: [\langle M, N \rangle](s, t) \triangleq \langle [M](s), [N](t) \rangle \\ *e &: [\mathbf{R}(L; M; x.y.N)](s, t) \triangleq h(x_1, h(x_2, \dots h(x_n, a) \dots)), \\ &\text{where the induction provided a list } [L](s) = x_1, \dots, x_n, \\ &\text{an element } a \triangleq [M](t), \text{ and a function } h \triangleq [N]. \end{aligned}$$

The other cases are given in App. B.2.

Note that in the contraction case (c), the two occurrences of M are shortands for two distinct stages of the typing derivation: the recursive call is made on a smaller typing derivation, even though the typed term remains unchanged.

Example 3.2. We can define list concatenation as follows:

$$\lambda h.\lambda k.\mathbf{R}(h; k; x.qk.x::qk)$$

This term has type $e^* \rightarrow e^* \rightarrow e^*$ for every type e . Note that this term is strictly linear: it is typed without exchange, contraction and weakening.

Example 3.3. Remember that we code natural numbers as lists over the singleton set. Writing 1 for the constant $\langle \cdot \rangle :: []$ and S for the successor function $\lambda n.\langle \cdot \rangle :: n$, we can define Ackermann's function as follows:

$$\lambda n.\mathbf{R}(n; S; _ . f.\lambda k.\mathbf{R}(k; f1; _ . r.fr))$$

This term can be typed with type $1^* \rightarrow 1^* \rightarrow 1^*$ in the empty environment. The outer recursion produces a function of type $1^* \rightarrow 1^*$. This term is not affine: we need the contraction rule since f is used twice in the outer recursion.

As announced before, system C contains system T:

Theorem 3.4. *For every typing derivation $\Gamma \vdash M : e$, there exists a regular proof $\underline{M} : \underline{\Gamma} \vdash e$ such that $[\underline{M}] = [M]$. If M is affine/linear, so is \underline{M} .*

The proof is given in App. B.3; all constructions of system T map directly to their counterpart in C, without forging any new formula (unlike in Fig. 4 for the encoding of the primitive recursion scheme).

Encoding the term given in Ex. 3.2 for list concatenation yields the first proof in Fig. 2. In contrast, encoding the term we provided for Ackermann's function (Ex. 3.3) does not yield the proof given in Fig. 5: the outer recursion in this term constructs functional values, which give rise through the encoding to cycles over sequents with arrow types on the right. More importantly, the proof in Fig. 5 has a non-trivial cycle structure, while in the proofs in the image of the encoding every infinite branch eventually loops on a single cycle of the finite presentation of the proof.

4 From affine C to affine T (using \cap)

The converse direction, encoding cyclic proofs into system T terms, is much harder since we have to delineate the possibly complex cycle structure of the starting proof in order to recover simple structural recursion schemes.

We provide a direct translation for the affine case in this section, where we proceed in two steps: first we show that affine regular proofs can be presented in such a way that cycles are associated to star formulas and occur in a hierarchic way (this is the notion of *ranked proof* in Sect. 4.3), this makes it possible to proceed bottom up in a second step, translating cycles associated to a given star formula into blocks of functions defined by mutual recursion (Sect. 4.4).

The second step is inspired by the one sketched in [15, Thm. 33] to translate regular proofs in LAL into equational proofs in action lattices. However, the authors of [15] did not realise that the first step we describe here is necessary, so that their argument is incorrect. The technique we present here makes it possible to repair it, fortunately.

4.1 Proofs with backpointers

We first formalise precisely how regular proofs are represented by finite proof graphs with backpointers, as pictured earlier in the paper. We start by defining auxiliary notions.

Definition 4.1. A *proof with backpointers* (*bp-proof* for short) is a pair $\pi_{bp} = \langle \pi, Pts \rangle$ where π is a proof, and Pts is a set of *backpointers*, where each backpointer pt has a *source* address $src(pt)$ and a *target* address $tgt(pt)$, such that

- For all $pt \in Pts$, $tgt(pt) \sqsubset src(pt)$ and the subtrees of π rooted in $src(pt)$ and $tgt(pt)$ are isomorphic.
- For every infinite branch B of π , there exists a unique $pt \in Pts$ with $src(pt) \in B$.

An address of a bp-proof is a *source* if it is the source of a backpointer, it is *canonical* if it is a prefix of a source address.

This definition is similar to that of ‘cycle normal form’ from [8]. Notice that the definition implies that in every bp-proof $\langle \pi, Pts \rangle$, the set Pts is finite. Moreover, to define such a bp-proof it suffices to describe the (finite) restriction of π to canonical addresses, as it was done earlier in the figures of this paper. Every regular proof can be represented as a bp-proof. We show below that backpointers can be assumed to satisfy additional properties related to threads.

4.2 Idempotent normal form

Let π be a regular proof and let s be the maximal length of sequent antecedents in π . Let \mathcal{F} be the set of partial functions $[0; s] \rightarrow [0; s]$. This set equipped with composition \circ is a finite monoid. An element $f \in \mathcal{F}$ is *idempotent* if $f \circ f = f$.

If $u \sqsubset v$ are addresses in π , we define $f_{u,v} \in \mathcal{F}$ by

$$f_{u,v}(j) \triangleq \begin{cases} i & \text{if } \langle v, j \rangle \text{ is an ancestor of } \langle u, i \rangle \\ \text{undefined} & \text{if no such } i \text{ exists} \end{cases}$$

5 Subsystems of second-order arithmetic

We define in this section the second-order logics ACA_0 and RCA_0 , as well as the properties we need about them. A comprehensive introduction to these theories and the ‘reverse mathematics’ program can be found in [22, 30]. Also, an excellent introduction to the functional interpretations of proofs, including for the theories covered here, is [4].

5.1 Some ‘second-order’ theories of arithmetic

We consider a two-sorted first-order language, henceforth called ‘second-order logic’ as is traditional, consisting of individual variables x, y, z etc., terms s, t, u etc., and set variables X, Y, Z etc. We have quantifiers for both the individual sort and the set sort. There is a single binary relation symbol \in connecting the two sorts, allowing us to write formulas of the form $t \in X$. (We may sometimes write $X(t)$ instead.) We have an equality relation for the individual sort; set equality is expressed by extensionality: $X = Y \triangleq \forall x(X(x) \equiv Y(x))$.

The *language of arithmetic* consists of the non-logical symbols $0, S, +, \times, <$, with their usual intended interpretations. A *theory* is just a set of closed formulas, and we say that a theory T *proves* a formula φ , if φ is a logical consequence of T . The base theory Q2 extends second-order logic by basic axioms governing the behaviour of the non-logical symbols, namely stating that $(0, S0, +, \times, <)$ is a commutative semiring discretely ordered by S . *Bounded* quantifiers are of the shape $\exists x(x < t \wedge \varphi)$ and $\forall x(x < t \Rightarrow \varphi)$.

Definition 5.1 (Arithmetical hierarchy). A possibly open formula is in $\Sigma_0^0 = \Pi_0^0 = \Delta_0^0$ if it has only bounded quantifiers. From here we define the *arithmetical hierarchy* as follows:

- Σ_{k+1}^0 formulas are those of the form $\exists \vec{x}.\varphi$ with $\varphi \in \Pi_k^0$.
- Π_{k+1}^0 formulas are those of the form $\forall \vec{x}.\varphi$ with $\varphi \in \Sigma_k^0$.

A formula is Δ_k^0 (provably in a theory T) if it is equivalent to both a Σ_k^0 formula and a Π_k^0 formula (resp., provably in T).

We define the following axiom schemata for *induction* and *comprehension*, where free variables may occur in φ :

- (φ -IND): $(\varphi(0) \wedge \forall x(\varphi(x) \Rightarrow \varphi(Sx))) \Rightarrow \forall x\varphi(x)$.
- (φ -CA): $\exists X\forall x(X(x) \equiv \varphi)$

Definition 5.2 ($\text{ACA}_0, \text{RCA}_0$). • ACA_0 extends Q2 by all instances of induction and comprehension.
• RCA_0 extends Q2 by axioms φ -IND where $\varphi \in \Sigma_1^0$ and φ -CA where φ is provably Δ_1^0 .

We often write formulas in natural language to stand for their obvious formalisation in arithmetic. We do not concern ourselves with such low-level encodings in the sequel. Statements written in natural language are typically robust under the choice of encoding.

5.2 Provably total computable functions

The utility of the second-order theories we have introduced, for this work, lies in the fact that they may reason about

programs and potentially infinite computations, by way of quantification over set variables. What is more, the functions they may well-define, or programs that they may prove terminating, are well-understood, in terms of their computational strength: we may freely use such functions in logical formulas without affecting logical complexity.

Proposition 5.3 (Witnessing for ACA_0). *Suppose ACA_0 proves $\forall \vec{x}\exists y\varphi(\vec{x}, y)$, where φ is Σ_1^0 and contains no set symbols. Then there is a term M of T with a typing derivation $x_1 : 1^*, \dots, x_n : 1^* \vdash M : 1^*$ such that $\mathbb{N} \models \forall \vec{x}.\varphi(\vec{x}, [M])$.*

This result follows immediately from the conservativity of ACA_0 over Peano Arithmetic and thence, under the Gödel-Gentzen double-negation translation, Gödel’s Dialectica functional interpretation of Heyting Arithmetic into T (see, e.g., [4] for more details).

A similar characterisation of RCA_0 is also known. This theory is conservative over $\text{I}\Sigma_1$, the restriction of Peano Arithmetic to Σ_1 -induction, which is known to well-define only primitive recursive natural number functions. This result was originally established by Parsons in his *predicative* functional interpretation [28], though there are also direct proofs available, e.g. by cut-elimination (see [10]).

Proposition 5.4 (Witnessing for RCA_0). *Suppose RCA_0 proves $\forall \vec{x}\exists y\varphi(\vec{x}, y)$, where φ is Σ_1^0 and contains no set symbols. Then there is a primitive recursive function $f(\vec{x})$ such that $\mathbb{N} \models \forall \vec{x}.\varphi(\vec{x}, f(\vec{x}))$.*

5.3 Reverse mathematics of cyclic proof checking

While the notion of preproof can easily be formalised already in RCA_0 , dealing with the validity criterion is non-trivial: we must be able to verify it within our theories too. In fact, the correctness of a generic cyclic proof checker is not available in RCA_0 due to Gödelian arguments applied to nontrivial relationships between cyclic and inductive fragments of arithmetic, cf. [12]. However, it is known that for any fixed preproof, RCA_0 can check whether it is valid or not:

Proposition 5.5 ([12], also implicit in [25]). *Let π be a regular proof. Then RCA_0 proves that π (written as a finite graph) is a proof, i.e., that each infinite branch contains a valid thread.*

This is a nontrivial result that is obtained by formalising the reduction of proof validity to the universality problem for nondeterministic Büchi automata and proving the correctness of a universality algorithm (see App. D.4).

6 Small steps reduction semantics for C

We fix a regular proof π in this section. We define a simplified version of the rewriting system used in [15] to prove cut-elimination in the system LAL. *Programs* are defined via the following syntax, where v ranges over addresses.

$$P, Q ::= \langle \rangle \mid [] \mid P :: P \mid v(P_1, \dots, P_n)$$

The first three entries correspond to constructors for singletons and lists. The fourth one corresponds to calling the node v of π with the given list of arguments. This syntax is much simpler than that used in [15]: we put constructors only for singletons and lists, which are the only types we want to observe in the present work. In particular, we do not need lambda abstractions to represent functional values. Also note that in contrast to [15], programs are always ‘closed’.

We use a simple type system to rule out ill-formed programs. Typing judgements have the form $\vdash P : e$; intuitively meaning that the program P produces values of type e .

$$\frac{}{\vdash \langle \rangle : 1} \quad \frac{}{\vdash [] : e^*} \quad \frac{\vdash P : e \quad \vdash Q : e^*}{\vdash P :: Q : e^*}$$

$$\frac{\vdash P_1 : e_1 \quad \dots \quad \vdash P_n : e_n}{\vdash v(P_1, \dots, P_n) : f} \quad \pi(v) = e_1, \dots, e_n \vdash f$$

Every program has at most one typing derivation, which can be computed in linear time. This argument is easily formalisable in RCA_0 .

We associate to every program P of type e a semantic value $[P] \in [e]$, by induction:

$$\langle \rangle \triangleq \langle \rangle \quad [[]] \triangleq \epsilon \quad [P::Q] \triangleq [P] :: [Q]$$

$$[v(P_1, \dots, P_n)] \triangleq [v]([P_1], \dots, [P_n])$$

Note that $[v]$ is the semantics of the node v in the proof π (Def. 2.9). This semantics cannot be defined in our second-order theories: values may be objects of arbitrary type.

Definition 6.1 (Reduction). *Reduction*, written \rightsquigarrow , is the smallest relation on programs which is closed under all contexts and satisfies the following rules, defined by case analysis on the rules used at addresses mentioned in the program. We omit some rules, see App. E.1 for an exhaustive list. We use v (resp. w) to range over addresses of left (resp. right) introduction rules, and u to range over other addresses.

$$\text{id} : u(P) \rightsquigarrow P \quad \text{cut} : u(\vec{P}, \vec{Q}) \rightsquigarrow u1(u0(\vec{P}), \vec{Q})$$

$$*l : v([], \vec{R}) \rightsquigarrow v0(\vec{R}) \quad *r_\epsilon : w() \rightsquigarrow []$$

$$*l : v(P::Q, \vec{R}) \rightsquigarrow v1(P, Q, \vec{R}) \quad *r_{::} : w(\vec{P}, \vec{Q}) \rightsquigarrow w0(\vec{P})::w1(\vec{Q})$$

$$\rightarrow l / \rightarrow r : v(w(\vec{P}), \vec{Q}, \vec{R}) \rightsquigarrow v1(w0(v0(\vec{Q}), \vec{P}), \vec{R})$$

As expected, subject reduction holds, so that we only work with well-typed programs in the sequel.

Notice that \rightsquigarrow is computable in RCA_0 , and so is provably Δ_1^0 . We also have the following characterisation of irreducible programs, still in RCA_0

Lemma 6.2. *If P is irreducible, then P is of the form*

- $\langle \rangle$, $[]$, or $P_1 :: P_2$ for some programs P_1, P_2 ; or,
- $v(\vec{P})$ for some v s.t. π_v ends with $+r_i, \cdot r, \cap r$ or $\rightarrow r$.

It follows that every irreducible program of type e^* is a list of irreducible programs of type e .

We also have that reductions preserve the semantics. We use this property only at the meta-level: it cannot even be stated in ACA_0 since it involves higher-order objects:

Proposition 6.3 (Semantic preservation). *For all programs P, P' , if $P \rightsquigarrow P'$ then $[P] = [P']$.*

Given a natural number n , let us write \underline{n} for its encoding as a closed program of type 1^* , such that $[\underline{n}] = n$. By Lem. 6.2, the irreducible programs of type 1^* are all of this shape. This simple encoding makes it possible to reason about proofs from natural numbers to natural numbers: if $\pi : 1^* \vdash 1^*$, then for all n , $[\pi](n)$ can be obtained by reducing the program $\pi(\underline{n})$. (Writing $\pi(\vec{P})$ for $\epsilon(\vec{P})$.)

6.1 Weak normalisation in ACA_0

We write $P \downarrow_\pi P'$ when P reduces to an irreducible P' via the left-most innermost strategy. We want to show:

Theorem 6.4 (Weak normalisation). *For all proofs π , ACA_0 proves that for all P , there exists P' with $P \downarrow_\pi P'$.*

To prove it, we use the following sets R_e of *reducible programs*, defined by induction on e . Those are inspired by reducibility candidates [20, 32].

$$\text{R}_{e^*} \triangleq \{P \mid P \downarrow_\pi Q_1 :: \dots :: Q_n, Q_1, \dots, Q_n \in \text{R}_e\}$$

$$\text{R}_{e \rightarrow f} \triangleq \{P \mid P \downarrow_\pi v(\vec{Q}), v a \rightarrow r, \forall Q \in \text{R}_e, v0(Q, \vec{Q}) \in \text{R}_f\}$$

The remaining cases are given in App. E.2. If $\vec{P} = P_1, \dots, P_n$ and $E = E_1, \dots, E_n$, we write $\vec{P} \in \text{R}_E$ when $P_i \in \text{R}_{E_i}$ for all i . Note that these sets are defined non-uniformly in ACA_0 : we use separate instances of comprehension at each stage. This is not a problem: we will need only finitely many of them since the starting proof is regular.

Every program in R_e is weakly normalisable by definition, so that it suffices to show that all programs of type e belong to R_e . We proceed by induction on the syntax of programs. The constructor cases are straightforward; for the remaining case we use the following proposition:

Proposition 6.5. *For every address v with $v : E \vdash e$, and for all programs $\vec{P} \in \text{R}_E$, we have $v(\vec{P}) \in \text{R}_e$.*

This property on addresses is locally preserved by the rules of C. This observation is not sufficient to conclude since we work with non-wellfounded proofs. We actually prove a strengthening of local preservation, by contrapositive:

Lemma 6.6. *For every address $w : E \vdash e$, for all $\vec{P} \in \text{R}_E$ such that $w(\vec{P}) \notin \text{R}_e$, there are v, F, f, \vec{Q} such that $|v| = |w| + 1$, $v : F \vdash f$, $v(\vec{Q}) \notin \text{R}_f$, and:*

1. for all i, j s.t. $\langle v, i \rangle \triangleleft \langle w, j \rangle$, we have $|Q_i| = |P_j|$, and
2. for all i, j s.t. $\langle v, i \rangle \triangleleft \langle w, j \rangle$, we have $|Q_i| < |P_j|$.

(Where given $P \in \text{R}_{e^*}$, we write $|P|$ for the length of the list given by the definition of R_{e^*} .)

Proof of Prop. 6.5. Suppose by contradiction that for some address $v : E \vdash e$ we have $\vec{P} \in R_E$ such that $v(\vec{P}) \notin R_e$. By using Lem. 6.6 repeatedly, we can construct an infinite branch of π starting at v . We conclude like in Lem. 2.8. \square

This concludes the ACA_0 proof of Thm. 6.4 and we deduce:

Corollary 6.7. *If $\pi : 1^* \dots 1^* \vdash 1^*$ is a regular proof, then there exists a term M from system T such that $[\pi] = [M]$.*

Proof (case of a unary function). By Prop. 5.5 and Thm. 6.4 we obtain a proof in ACA_0 of “ $\forall n, \exists m, \pi(\underline{n}) \downarrow_\pi \underline{m}$ ”. By Prop. 5.3, we can thus extract a system T term M such that for all n , $\pi(\underline{n}) \rightsquigarrow^* \underline{[M](n)}$. By Prop. 6.3, we deduce that for all n , $[\pi](n) = [\pi(\underline{n})] = [[M](n)] = [M](n)$. \square

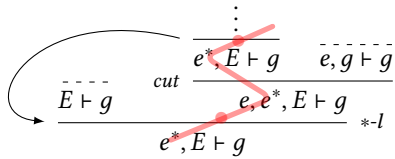
6.2 Weak normalisation in RCA_0

Given Prop. 5.4, it could be tempting to revisit the proof from the previous section, trying to see if we could use RCA_0 instead of ACA_0 in the absence of contraction. This fails, however, because the R_e sets already require set comprehensions outside Δ_1^0 (due to the quantifier alternation in the definition of $R_{e \rightarrow f}$). We need only finitely many such sets for a given regular proof, so that we could hope to use only their defining formulas, but then our main induction on the syntax of programs, to prove that all programs of type e belong to R_e , is not a Σ_1^0 -induction.

A different termination proof, inspired from [15], can be given in the affine case, using weak König’s lemma. RCA_0 extended with this axiom (WKL_0) is known to be conservative over RCA_0 for arithmetical formulas (see App. D.2), so that working in WKL_0 still makes it possible to extract primitive recursive functions. Unfortunately, this second proof does not seem to be formalisable in WKL_0 (see App. E.3).

We use a third termination argument instead, relying on the translation from Sect. 4.

Definition 6.8. A *simple proof* is an ibp-proof such that for every backpointer pt , $\text{src}(pt) = \text{tgt}(pt)10$ and the node at $\text{tgt}(pt)1$ is a *cut*.



In other words, a simple proof is equivalent to a well-founded proof using the following derivable rule:

$$\frac{E \vdash g \quad e, g \vdash g}{e^*, E \vdash g} \text{*-l'}$$

Our translation from T to C (Thm. 3.4) actually produces simple proofs, so that by Thm. 4.5, every affine proof can be translated into a simple affine proof with the same semantics.

Accordingly, we assume in the rest of this section that the fixed proof π is affine and simple.

We update the notion of reduction accordingly: we write \rightsquigarrow for the relation defined like in Def. 6.1, except that when v is the target of a backpointer, we use the following rule instead of the two $*-l$ reduction rules:

$$v(P_1 :: \dots :: P_n :: [], \vec{R}) \rightsquigarrow v11(P_1, \dots, v11(P_n, v0(\vec{R})))$$

This rule has to be compared with the $2n + 1$ reductions we can obtain with \rightsquigarrow :

$$\begin{aligned} v(P_1 :: \dots :: P_n :: [], \vec{R}) &\rightsquigarrow v1(P_1, P_2 :: \dots :: P_n :: [], \vec{R}) \\ &\rightsquigarrow v11(P_1, v10(P_2 :: \dots :: P_n :: [], \vec{R})) \\ \dots &\rightsquigarrow v11(P_1, \dots, v(10)^n 11(P_n, v(10)^n([], \vec{R}))) \\ &\rightsquigarrow v11(P_1, \dots, v(10)^n 11(P_n, v(10)^n 0(\vec{R}))) \end{aligned}$$

The main advantage of \rightsquigarrow is that when $P \rightsquigarrow P'$, if P contains only canonical addresses, then so does P' .

Lemma 6.9. *If there is an infinite leftmost innermost reduction sequence along \rightsquigarrow , then there is an infinite reduction sequence along \rightsquigarrow where programs only contain canonical addresses.*

Proof. By mapping addresses into their canonical addresses and compressing finite sequences of reductions as above. \square

We assume all programs only mention canonical addresses in the sequel. Let $m(P)$ be the finite multiset of (canonical) addresses mentioned in a program P . These multisets can be represented and computed in RCA_0 via appropriate encodings; we write $m(u)$ for the number of occurrences of an address u in a multiset m .

We write \geq for the multiset ordering, where addresses are ordered by reverse prefix ordering (i.e., longer addresses are considered as smaller):

$$m \geq m' \triangleq \forall v, m(v) \geq m'(v) \vee \exists u, u \sqsubseteq v, m(u) > m'(u)$$

Lemma 6.10. *If $P \rightsquigarrow P'$ then $m(P) > m(P')$.*

Proof. By straightforward analysis of the reduction rules. (Note that the reduction rule for contraction fails this property because it duplicates arbitrary addresses.) \square

This suffices to deduce at the meta-level that every leftmost innermost reduction sequence along \rightsquigarrow terminates. Indeed, since we have finitely many canonical addresses in π , the reverse prefix ordering on canonical addresses is well-founded, as well as the above multiset ordering.

This latter result cannot be proved uniformly in RCA_0 , however (see Cor. 6.14 below). Instead, we prove that the multiset order on a fixed and finite order is provably well-founded in RCA_0 :

Proposition 6.11. *For all $n \in \mathbb{N}$, RCA_0 proves that the multiset order on $[0; n]$ is well-founded.*

Proof. Write $\max m$ for the maximal number occurring in a finite multiset m of natural numbers (-1 if m is empty). We prove the following property by (meta-level) induction on n :

$$\text{RCA}_0 \text{ proves } \forall (m_i)_{i \in \mathbb{N}}, (\forall i, m_i > m_{i+1}) \Rightarrow \max m_0 \geq n$$

(This property entails well-foundedness over multisets on $[0; n - 1]$.)

- the case $n = 0$ is trivial since m_0 cannot be empty.
- for the inductive case, suppose by contradiction that there exists a decreasing sequence $(m_i)_{i \in \mathbb{N}}$ such that $\max m_0 < n + 1$, i.e. $\max m_0 \leq n$.
 - By a Δ_0^0 induction, we get $\forall i, m_i \leq n$.
 - By a second Δ_0^0 induction, we get $\forall i, m_{i+1}(n) \leq m_i(n)$. The function $i \mapsto m_i(n)$ is thus decreasing, so that it must stationate: there exists j such that for all k , $m_{j+k}(n) = m_j(n)$. (This can be proved by absurd and Π_1^0 -induction, which is available in RCA_0 [11].)

Now consider the sequence $m'_i \triangleq m_{j+i} \setminus n$, where $m \setminus n$ denotes the multiset m where all occurrences of n have been removed. This sequence is decreasing by Δ_0^0 -induction, and satisfies $\max m'_0 < n$, thus contradicting the induction hypothesis. \square

(That we restrict to multiset order on a finite total order in the above statement is not a restriction since every finite partial order—like our reverse prefix ordering on canonical addresses—embeds in a finite total order.)

Theorem 6.12 (Weak normalisation). *For all affine simple proof π , RCA_0 proves that for all P , there exists P' with $P \downarrow_\pi P'$.*

Proof. Write P_n for the n -th reduct of P via the leftmost innermost strategy (if any). It suffices to show that there exists n such that P_n is irreducible. Suppose by contradiction that for all n , P_n can be reduced, i.e., $P_n \rightsquigarrow P_{n+1}$ since we fixed a strategy. By Lem. 6.9 and Lem. 6.10, we find an infinite decreasing sequence of multisets over $[0; n]$ where n is the maximal length of canonical addresses in π , contradicting Prop. 6.11. \square

Corollary 6.13. *If $\pi : 1^* \dots 1^* \vdash 1^*$ is an affine regular proof, then $[\pi]$ is primitive recursive.*

Proof. We first translate π into an affine term and then back into a simple affine proof using Thms. 4.5 and 3.4. Then we proceed Like for Cor. 6.7, using Thm. 6.12 and Prop. 5.4 instead of Thm. 6.4 and Prop. 5.3. \square

Corollary 6.14. *RCA_0 cannot prove that the multiset order on \mathbb{N} is well-founded.*

Proof. If this was a theorem of RCA_0 , then we would get a uniform proof of Thm. 6.12, from which we could extract a ‘universal primitive recursive function’ whose complexity would bound the complexity of all primitive recursive functions (via Thm. 2.13). \square

7 Conclusions and future work

We proposed the cyclic sequent proof system C and we studied its expressive power as computational device, by comparing it with an appropriate version of Gödel’s system T. Encoding cyclic proofs into recursive ones is nontrivial, but we managed to give a direct encoding from C to T in the affine case. To measure the complexity of functions of C and its affine variant we then appealed to proofs of totality in systems of second-order arithmetic, thus obtaining simulations in T and primitive recursive arithmetic, respectively.

We used the connectives of IMALL plus a least fixpoint operator for lists to illustrate the genericity of our approach. Small fragments of C are already complete w.r.t. the considered classes of functions (e.g., 1^* and \cdot do suffice to capture primitive recursive functions). Conversely, other least fixpoint operators could easily be handled (e.g., $\mu x.e + x \cdot x$ for binary trees with leaves in e). Cyclic systems with both least and greatest fixpoints have been studied in the literature [17, 18]; whether they correspond to appropriate extensions of T is left for future work.

Our current translation of C into T (with contraction) works for natural number functions, but it is not immediate that it scales to higher types. Indeed, while usual reducibility and hereditary recursivity arguments may indeed be carried out in constructive arithmetic, our proof of totality by contradiction and infinite descent comprises nonconstructive reasoning. While the Dialectica functional interpretation ensures that our translation from C to T for natural number functions is constructive, it would be interesting to attain a ‘direct’ translation, e.g. in the style of Sect. 4, that could work at higher types too.

The type levels of recursors in T programs are closely related to the logical complexity of induction in Peano Arithmetic (in the sense of Def. 5.1). At this level of granularity, it was observed recently in [12] that there is indeed a difference between cyclic and inductive proofs: cyclic proofs using Σ_n formulas is equivalent to inductive proofs using Σ_{n+1} formulas (over Π_{n+1} theorems). It would be natural to expect, therefore, that C restricted to level n types is equivalent to T restricted to level $n + 1$ recursors (over level $n + 1$ functions), but that remains a topic for future work.

Acknowledgements. We are grateful to Anupam Das for early discussions on this work and later for convincing us to use second order theories of arithmetic like ACA_0 and RCA_0 and pointing out a bug in a preliminary version. We would also like to thank Olivier Laurent, Pierre Clairambault, Colin Riba, and Ludovic Patey for many helpful discussions.

References

- [1] S. Abramsky, E. Haghverdi, and P. Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002.

- [2] B. Afshari and G. E. Leigh. Cut-free completeness for modal mu-calculus. In *Proc. LiCS*, pages 1–12, 2017.
- [3] J. Avigad. Formalizing forcing arguments in subsystems of second-order arithmetic. *Annals of Pure and Applied Logic*, 82(2):165 – 191, 1996.
- [4] J. Avigad and S. Feferman. Gödel’s functional interpretation. In S. R. Buss, editor, *Handbook of Proof Theory*. Elsevier, 1998.
- [5] P. Baillot and M. Pedicini. Elementary complexity and geometry of interaction. *Fundamenta Informaticae*, 45(1-2):1–31, 2001.
- [6] S. Berardi and M. Tatsuta. Classical system of martin-löf’s inductive definitions is not equivalent to cyclic proof system. In *Proc. FoSSaCS*, pages 301–317, 2017.
- [7] S. Berardi and M. Tatsuta. Equivalence of inductive definitions and cyclic proofs under arithmetic. In *Proc. LiCS*, pages 1–12, 2017.
- [8] J. Brotherston. Cyclic proofs for first-order logic with inductive definitions. In *Proc. Proceedings of TABLEAUX-14*, volume 3702 of *LNAI*, pages 78–92. Springer-Verlag, 2005.
- [9] J. Brotherston and A. Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, 2011.
- [10] S. R. Buss. The witness function method and provably recursive functions of Peano arithmetic. In *Studies in Logic and the Foundations of Mathematics*, volume 134, pages 29–68. Elsevier, 1995.
- [11] S. R. Buss. *Handbook of proof theory*, volume 137. Elsevier, 1998.
- [12] A. Das. On the logical complexity of cyclic arithmetic. *Logical Methods in Computer Science*, Volume 16, Issue 1, Jan. 2020.
- [13] A. Das, A. Doumane, and D. Pous. Left-handed completeness for Kleene algebra, via cyclic proofs. In *Proc. LPAR*, volume 57 of *EPiC Series in Computing*, pages 271–289. EasyChair, 2018.
- [14] A. Das and D. Pous. A cut-free cyclic proof system for Kleene algebra. In *Proc. TABLEAUX*, volume 10501 of *Lecture Notes in Computer Science*, pages 261–277. Springer, 2017.
- [15] A. Das and D. Pous. Non-wellfounded proof theory for (Kleene+action)(algebras+lattices). In *Proc. CSL*, volume 119 of *LIPICs*, pages 18:1–18:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.
- [16] A. Doumane. Constructive completeness for the linear-time μ -calculus. In *Proc. LiCS*, pages 1–12, 2017.
- [17] A. Doumane, D. Baelde, and A. Saurin. Infinitary proof theory: the multiplicative additive case. In *Proc. CSL*, volume 62 of *LIPICs*, pages 42:1–42:17, Sept. 2016.
- [18] J. Fortier and L. Santocanale. Cuts for circular proofs: semantics and cut-elimination. In *Proc. CSL*, volume 23 of *LIPICs*, pages 248–262, 2013.
- [19] J.-Y. Girard. Geometry of interaction iii: accommodating the additives. In *Proc. Proceedings of the workshop on Advances in linear logic*, pages 329–389. Cambridge University Press, 1995.
- [20] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and Types*. Cambridge University Press, USA, 1989.
- [21] V. K. Gödel. Über eine bisher noch nicht benützte erweiterung des finiten standpunktes. *Dialectica*, 12(3-4):280–287, 1958.
- [22] D. R. Hirschfeldt. *Slicing the truth: On the computable and reverse mathematics of combinatorial principles*. World Scientific, 2014.
- [23] N. Hoshino, K. Muroya, and I. Hasuo. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In *Proc. Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, page 52. ACM, 2014.
- [24] U. Kohlenbach. Effective bounds from ineffective proofs in analysis: An application of functional interpretation and majorization. *The Journal of Symbolic Logic*, 57(4):1239–1273, 1992.
- [25] L. Kolodziejczyk, H. Michalewski, P. Pradic, and M. Skrzypczak. The logical strength of Büchi’s decidability theorem. *Logical Methods in Computer Science*, Volume 15, Issue 2, May 2019.
- [26] D. Kuperberg, L. Pinault, and D. Pous. Cyclic Proofs and Jumping Automata. In *Proc. FSTTCS*, Bombay, India, Dec. 2019.
- [27] U. D. Lago. The geometry of linear higher-order recursion. *ACM Trans. Comput. Log.*, 10(2):8:1–8:38, 2009.
- [28] C. Parsons. On n-quantifier induction. *The Journal of Symbolic Logic*, 37(3):466–482, 1972.
- [29] A. Simpson. Cyclic arithmetic is equivalent to peano arithmetic. In *Proc. FoSSaCS*, pages 283–300, 2017.
- [30] S. G. Simpson. *Subsystems of second order arithmetic*, volume 1. Cambridge University Press, 2009.
- [31] W. Tait. Infinitely long terms of transfinite type. In J. Crossley and M. Dummett, editors, *Formal Systems and Recursive Functions*, volume 40 of *Studies in Logic and the Foundations of Mathematics*, pages 176 – 185. Elsevier, 1965.
- [32] W. W. Tait. Intensional interpretations of functionals of finite type i. *The Journal of Symbolic Logic*, 32(2):198–212, 1967.

A Additional details for Sect. 2

A.1 Return value of a computation

We give the complete version of Def. 2.9.

The *return value* $[v](s)$ of a computation $\langle v, s \rangle$ with $\pi(v) = E \vdash e$ is a value in $[e]$ defined by well-founded induction on \prec and case analysis on the rule used at address v . We list all cases below.

$$\begin{aligned}
 id &: [v](s) \triangleq s \\
 cut &: [v](s, t) \triangleq [v1]([v0](s), t) \\
 c &: [v](x, s) \triangleq [v0](x, x, s) \\
 x &: [v](s, x, y, t) \triangleq [v0](s, y, x, t) \\
 w &: [v](x, s) \triangleq [v0](s) \\
 \cdot\cdot l &: [v](\langle x, y \rangle, s) \triangleq [v0](x, y, s) \\
 \cdot\cdot r &: [v](s, t) \triangleq \langle [v0](s), [v1](t) \rangle \\
 \rightarrow\cdot l &: [v](h, s, t) \triangleq [v1](h([v0](s)), t) \\
 \rightarrow\cdot r &: [v](h) \triangleq (x \mapsto [v0](x, h)) \\
 *l &: [v](l, s) \text{ is defined by case analysis on the list } l: \\
 &\quad \bullet [v]([], s) \triangleq [v0](s) \\
 &\quad \bullet [v](x :: q, s) \triangleq [v1](x, q, s) \\
 *r_e &: [v]() \triangleq \epsilon \\
 *r_r &: [v](s, t) \triangleq [v0](s) :: [v1](t) \\
 +l &: [v](x, s) \text{ is defined by case analysis on } x: \\
 &\quad \bullet \text{ if } x \in [e], [v](x, s) \triangleq [v0](x, s) \\
 &\quad \bullet \text{ if } x \in [f], [v](x, s) \triangleq [v1](x, s) \\
 +r_i &: [v](s) \triangleq [v0](s) \\
 1l &: [v](\langle \rangle, s) \triangleq [v0](s) \\
 1r &: [v]() \triangleq \langle \rangle \\
 \cap\cdot l_i &: [v](\langle x_0, x_1 \rangle, s) \triangleq [v0](x_i, s) \\
 \cap\cdot r &: [v](s) \triangleq \langle [v0](s), [v1](s) \rangle
 \end{aligned}$$

In each case, the recursive calls are made on strictly smaller computations: they occur on direct subproofs, the values associated to auxiliary formulas are left unchanged, and in the second subcase of the $*l$ case, the length of the list associated to the principal formula decreases by one.

A.2 Weakening and contraction

Proof of Lem. 2.11. We proceed by induction on e . The first interesting case is the weakening of a star formula e^* which

is depicted on the left of Fig. 6. The rule marked *IH* is the weakening rule derived for e by induction hypothesis and the widget on the left in Fig. 3. The second interesting case is the weakening of an arrow formula $e \rightarrow f$ depicted on the right of Fig. 6. The proof (inh_e) is a witness that every closed type e is inhabited, which is easily shown by induction on e . The rule *IH* is the weakening rule derived for f by induction hypothesis. \square

Proof of Lem. 2.12. We proceed by induction on e ; the interesting case is the duplication of a star formula e^* , which is depicted in Fig. 7. The subproofs labelled with 'cons' consist of an application of the $*-r_*$ rule followed by two identity axioms. The rule marked *IH* at address 110 is the contraction rule derived for e by induction hypothesis and the widget on the right in Fig. 3. \square

A.3 Minimisation operator

We show in this section that by dropping the validity condition, we can encode the minimisation operator μ , yielding Turing-completeness of the proof system.

We define μ with one integer parameter x , as any tuple of parameters can be encoded in one. Thus μ is defined as follows: if $f : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, then $\mu(f)(x)$ is the smallest $y \in \mathbb{N}$ such that $f(y, x) = 0$, and is undefined if no such y exists.

Therefore, the μ operator has type $(\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}$. The preproof π_μ is represented in Fig. 8. In this figure, x, y stand for 1^* , f stands for $1^* \cdot 1^* \rightarrow 1^*$, and k stands for 1^* : it stores the result $f(y, x)$. We note k^a the predecessor of k and y' the successor of y . Principal formulas may be emphasised by a red font.

The principle behind this preproof is simply to compute $k = f(y, x)$ for $y = 0, 1, 2, \dots$, and returns y as soon as $k = 0$. The preproof is not valid, as the infinite branch contains no validating thread. The only infinite thread in this branch is the one following x , which is never principal.

In order to give a semantic to such an invalid preproof (as a partial function on natural numbers), one can use the small-step semantic from Sect. 6.1: feed the proof with natural numbers and try to compute a result value with leftmost innermost reduction strategy. If this terminates, we can read back a natural number by Lem. 6.2, otherwise the function is undefined at the considered point.

B Additional details for Sect. 3

B.1 Encoding of classical system T

Let us show how our version of system T can encode the more classical recursion operator, thereby proving the two systems are equivalent.

Let us call \mathbf{R}_T the classical recursion operator from system T. We recall below the behaviour of \mathbf{R}_T , and the corresponding typing rule:

$$\mathbf{R}_T([], M, x.q.y.N) = M$$

$$\mathbf{R}_T(X::Q, M, x.q.y.N) = N\{x \leftarrow X; q \leftarrow Q; y \leftarrow \mathbf{R}_T(q, M, x.y.q.N)\}$$

$$\mathbf{R}_T \frac{\Gamma \vdash L : e^* \quad \Gamma \vdash M : g \quad \Gamma, x : e, q : e^*, y : g \vdash N : g}{\Gamma \vdash \mathbf{R}_T(L, M, x.q.y.N) : g}$$

There are several differences with our typing rule for \mathbf{R} : the tail q is fed to the function N , the context Γ can be duplicated, and a non-empty context can be used by the function N .

We show that \mathbf{R}_T can be encoded by a term of T (together with its typing derivation).

The idea is to duplicate the necessary information using contractions, and to use our restricted recursor with an enriched return type g' .

Let $\Gamma = \vec{u} : E$ be an arbitrary context, and e be a type. We define the type $g' \triangleq \vec{E} \rightarrow (e^* \cdot g)$. We use our affine recursor scheme \mathbf{R} with arguments L (unchanged), $M' \triangleq \lambda \vec{u}. \langle [], M \rangle$ and

$$x.y'.N' \triangleq \lambda \vec{u}. (\text{let } \langle q, y \rangle := y'(\vec{u}) \text{ in } x.q.y.N).$$

Notice that provided $M:g$ and $N:g$, we have $M':g'$ and $x.y'.N':g'$ as desired for the use of \mathbf{R} . Typing derivations showing this are omitted.

Finally, the term $\mathbf{R}_T(L, M, x.q.y.N)$ is now encoded as $\text{let } \langle l, r \rangle := \mathbf{R}(L, M', x'.y'.N')(\vec{u})$ in r .

The following typing derivation (admitting the recursion-free typing for M', N') in our system can then serve as a macro of the \mathbf{R}_T typing rule above. For clarity, we use $\dashv\text{p}_1$ as a shortcut for the projection on the second component of a product.

$$\begin{array}{c} *e \frac{\Gamma \vdash L : e^* \quad M' : g' \quad x : e, y' : g' \vdash N' : g'}{\Gamma \vdash \mathbf{R}(L, M', x'.y'.N') : g'} \quad id \frac{}{\Gamma \vdash \vec{u} : E} \\ \rightarrow e \frac{}{\Gamma, \Gamma \vdash \mathbf{R}(L, M', x'.y'.N')(\vec{u}) : e^* \cdot g} \\ c \frac{}{\Gamma \vdash \mathbf{R}(L, M', x'.y'.N')(\vec{u}) : e^* \cdot g} \\ \dashv\text{p}_1 \frac{}{\Gamma \vdash \mathbf{R}_T(L, M, x.q.y.N) : g} \end{array}$$

B.2 Complete list for Def. 3.1

We provide here the full list defining the semantic of terms from T, completing Def. 3.1.

$$\begin{array}{l} id : [x](s) \triangleq s \\ \rightarrow e : [MN](s, t) \triangleq [M](s)([N](t)) \\ c : [M](v, s) \triangleq [M](v, v, s) \\ \cdot i : [\langle M, N \rangle](s, t) \triangleq \langle [M](s), [N](t) \rangle \\ *e : [\mathbf{R}(L; M; x.y.N)](s, t) \triangleq h(x_1, h(x_2, \dots h(x_n, a) \dots)), \\ \quad \text{where the induction provided a list } [L](s) = x_1, \dots, x_n, \\ \quad \text{an element } a \triangleq [M](t), \text{ and a function } h \triangleq [N]. \\ x : [M](s, u, v, t) \triangleq [M](s, v, u, t) \\ w : [M](v, s) \triangleq [M](s) \\ \cdot e : [\text{let } \langle x, y \rangle := M \text{ in } N](s, t) \triangleq [N](u, v) \text{ where the in-} \\ \quad \text{duction provided } [M](s) = \langle u, v \rangle. \end{array}$$

$$\begin{array}{c}
 \vdots \\
 \text{IH} \frac{e^* \vdash 1}{e, e^* \vdash 1} \\
 \text{1-r} \frac{}{\vdash 1} \quad \text{* -l} \frac{}{e^* \vdash 1} \\
 \hline
 e^* \vdash 1
 \end{array}
 \quad
 \begin{array}{c}
 \text{inh}_e \frac{}{\vdash e} \quad \text{IH} \frac{}{f \vdash 1} \\
 \text{->-l} \frac{}{e \rightarrow f \vdash 1}
 \end{array}$$

Figure 6. Weakening of star and arrow formulas.

$$\begin{array}{c}
 \text{cons} \frac{}{e, e^* \vdash e^*} \quad \text{cons} \frac{}{e, e^* \vdash e^*} \\
 \text{->-r} \frac{}{e, e, e^*, e^* \vdash e^* \cdot e^*} \\
 \text{IH} \frac{}{e, e^*, e^* \vdash e^* \cdot e^*} \\
 \text{->-l} \frac{}{e, e^* \cdot e^* \vdash e^* \cdot e^*} \\
 \text{cut} \frac{}{e^* \vdash e^* \cdot e^*} \\
 \text{* -r}_\epsilon \frac{}{\vdash e^*} \quad \text{* -r}_\epsilon \frac{}{\vdash e^*} \\
 \text{->-r} \frac{}{y, x \vdash y \cdot x} \\
 \text{1* -l} \frac{}{y, f, x \vdash 1^*} \\
 \text{c, c, c} \frac{}{y, y, f, f, x, x \vdash 1^*} \\
 \text{cut} \frac{}{\vdash e^* \cdot e^*} \\
 \text{1* -r}_0 \frac{}{\vdash 1^*} \\
 \text{cut} \frac{}{f, x \vdash 1^*}
 \end{array}$$

Figure 7. Duplicating a star formula.

$$\begin{array}{c}
 \vdots \\
 \text{S} \frac{}{y \vdash y'} \quad \text{cut} \frac{}{y', f, x \vdash 1^*} \\
 \text{id} \frac{}{y \vdash 1^*} \\
 \text{w} \frac{}{y, f, x \vdash 1^*} \\
 \text{1* -l} \frac{}{y, f, x \vdash 1^*} \\
 \text{->-r} \frac{}{y, x \vdash y \cdot x} \\
 \text{->-l} \frac{}{k, y, f, x \vdash 1^*} \\
 \text{c, c, c} \frac{}{y, y, f, f, x, x \vdash 1^*} \\
 \text{cut} \frac{}{\vdash 1^*} \\
 \text{1* -r}_0 \frac{}{\vdash 1^*} \\
 \text{cut} \frac{}{f, x \vdash 1^*}
 \end{array}$$

Figure 8. The preproof π_μ for minimisation.

$+e$: $[D(S; x.M; y.N)](s, t) \triangleq [M]([S](s), t)$ if $[S](s) \in [e]$ and $[N]([S](s), t)$ otherwise.

$1-e$: $[\text{let } \langle \rangle := M \text{ in } N](s, t) \triangleq [N](t)$

$\cap -e_i$: $[\mathbf{p}_i M](s) \triangleq [M](s)$

$+i_j$: $[\mathbf{i}_j M](s) \triangleq [M](s)$

$*-i_\epsilon$: $[\mathbf{[]}]() \triangleq \epsilon$.

$*-i_{::}$: $[M :: N](s, t) \triangleq [M](s) :: [N](t)$

$1-i$: $[\langle \rangle]() \triangleq 1$

$\rightarrow -i$: $[\lambda x.M](s) \triangleq u \mapsto [M](u, s)$

$\cap -i$: $[\langle\langle M, N \rangle\rangle](s) \triangleq \langle [M](s), [N](s) \rangle$

B.3 From system T to system C

We give here the encoding from system T to system C.

Theorem B.1 (Thm. 3.4 in the main text). *For every typing derivation $\Gamma \vdash M : e$, there exists a regular proof $\underline{M} : \underline{\Gamma} \vdash e$ such that $[\underline{M}] = [M]$. If M is affine/linear, so is \underline{M} .*

Proof. We proceed by induction on the typing derivation. The structural rules (exchange, weakening, contraction and

identity) as well as the introduction rules of system T translate immediately to their counterparts in system C. It remains to deal with the elimination rules of system T. Leaving the $*-e$ rule aside, they all translate into a cut on the eliminated formula, followed by an application of the corresponding left introduction rule (and an identity rule for the negative connectives \cap and \rightarrow). For instance, for the $-e$ case (i.e., term $\text{let } \langle x, y \rangle := M \text{ in } N$), we obtain two regular proofs $\underline{M} : \underline{\Gamma} \vdash e \cdot f$ and $\underline{N} : e, f, \underline{\Delta} \vdash g$ by induction, and we construct the following preproof:

$$\text{cut} \frac{\frac{\underline{M}}{\underline{\Gamma} \vdash e \cdot f} \quad \frac{\underline{N}}{e, f, \underline{\Delta} \vdash g}}{\underline{\Gamma}, \underline{\Delta} \vdash g} \text{-l}$$

This preproof regular and valid: every infinite branch eventually belongs either to \underline{M} or \underline{N} .

The $*-e$ case (i.e., term $R(L; M; x.y.N)$) is the only one where we introduce circularities: we obtain by induction

$$\begin{array}{c}
 \frac{\Gamma, y : f, x : e, \Delta \vdash M : g}{\Gamma, x : e, y : f, \Delta \vdash M : g} \quad x \\
 \frac{\Gamma \vdash M : f}{x : e, \Gamma \vdash M : f} \quad w \\
 \frac{x : e, x : e, \Gamma \vdash M : f}{x : e, \Gamma \vdash M : f} \quad c \\
 \frac{id}{x : e \vdash x : e} \quad id \\
 \frac{\Gamma \vdash M : e \cdot f \quad x : e, y : f, \Delta \vdash N : g}{\Gamma, \Delta \vdash \text{let } \langle x, y \rangle := M \text{ in } N : g} \quad \cdot\text{-}e \\
 \frac{\Gamma \vdash S : e + f \quad x : e, \Delta \vdash M : g \quad y : f, \Delta \vdash N : g}{\Gamma, \Delta \vdash \mathbf{D}(S; x.M; y.N) : g} \quad +\text{-}e \\
 \frac{\Gamma \vdash L : e^* \quad \Delta \vdash M : g \quad x : e, y : g \vdash N : g}{\Gamma, \Delta \vdash \mathbf{R}(L; M; x.y.N) : g} \quad *\text{-}e \\
 \frac{\Gamma \vdash M : 1 \quad \Delta \vdash N : g}{\Gamma, \Delta \vdash \text{let } \langle \rangle := M \text{ in } N : g} \quad 1\text{-}e \\
 \frac{\Gamma \vdash M : e \rightarrow f \quad \Delta \vdash N : e}{\Gamma, \Delta \vdash MN : f} \quad \rightarrow\text{-}e \\
 \frac{\Gamma \vdash M : e_0 \cap e_1 \quad i \in \{0, 1\}}{\Gamma \vdash \mathbf{p}_i M : e_i} \quad \cap\text{-}e_i \\
 \frac{\Gamma \vdash M : e \quad \Delta \vdash N : f}{\Gamma, \Delta \vdash \langle M, N \rangle : e \cdot f} \quad \cdot\text{-}i \\
 \frac{\Gamma \vdash M : e_j}{\Gamma \vdash \mathbf{i}_j M : e_0 + e_1} \quad j \in \{0, 1\} \quad +\text{-}i_j \\
 \frac{\Gamma \vdash M : e \quad \Delta \vdash N : e^*}{\Gamma, \Delta \vdash M :: N : e^*} \quad *\text{-}i :: \\
 \frac{\Gamma \vdash M : e \quad \Delta \vdash N : e^*}{\Gamma, \Delta \vdash M :: N : e^*} \quad *\text{-}i_\epsilon \\
 \frac{\Gamma \vdash M : e \quad \Delta \vdash N : f}{\Gamma \vdash \langle \rangle : 1} \quad 1\text{-}i \\
 \frac{x : e, \Gamma \vdash M : f}{\Gamma \vdash \lambda x. M : e \rightarrow f} \quad \rightarrow\text{-}i \\
 \frac{\Gamma \vdash M : e \quad \Gamma \vdash N : f}{\Gamma \vdash \langle \langle M, N \rangle \rangle : e \cap f} \quad \cap\text{-}i
 \end{array}$$

Figure 9. Typing rules for system T.

three regular proofs $\underline{L} : \Gamma \vdash e^*$, $\underline{M} : \Delta \vdash g$ and $\underline{N} : e, g \vdash g$, and we construct the following preproof:

$$\begin{array}{c}
 \frac{\underline{L}}{\Gamma \vdash e^*} \quad \frac{\underline{M}}{\Delta \vdash g} \quad \frac{\underline{N}}{e, g \vdash g} \\
 \frac{\Gamma \vdash e^* \quad \Delta \vdash g}{\Gamma, \Delta \vdash g} \quad \frac{e^*, \Delta \vdash g \quad e, g \vdash g}{e, e^*, \Delta \vdash g} \quad \text{cut} \\
 \frac{\Gamma \vdash e^* \quad e, e^*, \Delta \vdash g}{\Gamma, \Delta \vdash g} \quad \text{cut} \quad *\text{-}l
 \end{array}$$

This preproof is regular by construction, and valid: the only infinite branch that does not eventually belong either to \underline{L} , \underline{M} or \underline{N} is the one along the constructed cycle, which it is validated by the red thread on e^* .

We use the contraction/weakening typing rule from system T only to translate contraction/weakening nodes in the starting proof, whence the second part of the statement. \square

C Proofs and details for Sect. 4

C.1 Proof of Prop. 4.3

Let π be a regular proof. We have to define a set of backpointers turning π into an ibp-proof.

We first establish a generic lemma. A *backpointer condition* P is a property of bp-proofs of the form: “for each backpointer pt , a property $P(pt)$ depending only on $\text{src}(pt)$, $\text{tgt}(pt)$, and the branch from the root of the proof to $\text{src}(pt)$ is verified”.

We say that a backpointer pt is *correct* when it verifies the first item from Def. 4.1, i.e. the subtrees rooted in $\text{src}(pt)$ and $\text{tgt}(pt)$ are isomorphic.

Lemma C.1. *Let π be a preproof and P be a backpointer condition such that for every infinite branch of π , there exists a correct backpointer pt such that $P(pt)$ is satisfied. Then π*

can be turned into a bp-preproof where all backpointers satisfy P .

Proof. For each infinite branch ρ of π , we define the backpointer pt_ρ given by the hypothesis of the Lemma.

Let $Pts_0 = \{pt_\rho \mid \rho \text{ branch of } \pi\}$, and $Pts_1 = \{pt \in Pts_0 \mid \forall pt' \in Pts_0, \text{src}(pt') \not\sqsubseteq \text{src}(pt)\}$, i.e. we only keep pointers from Pts_0 with a minimal source. We show that Pts_1 is finite. Indeed, assume Pts_1 is infinite, and let $T = \{u \mid \exists pt \in Pts_1, u \sqsubseteq \text{src}(pt)\}$. Since T contains all sources from Pts_1 , and that this sources are incomparable with each other, T is infinite. By König’s lemma, since T is finitely branching, T contains an infinite branch ρ . By definition of Pts_1 , there exists $pt \in Pts_1$ with $\text{src}(pt) \sqsubseteq \text{src}(pt_\rho)$. Let v be an address of ρ with $\text{src}(pt) \sqsubseteq v$. Since ρ is contained in T , there must be $pt' \in Pts_1$ with $v \sqsubseteq \text{src}(pt')$. We obtain $\text{src}(pt) \sqsubset \text{src}(pt')$, contradicting the fact that $pt' \in Pts_1$. We can thus conclude that Pts_1 is finite. Let $Pts_2 = \{pt \in Pts_1 \mid \forall pt' \in Pts_1, \text{src}(pt) = \text{src}(pt') \Rightarrow \text{tgt}(pt) \sqsubseteq \text{tgt}(pt')\}$, i.e. for each possible source we keep only the pointer with the smallest target. Since each pointer pt in Pts_2 is correct and satisfies $P(pt)$, and since each branch of π contains the source of exactly one pointer from Pts_2 , we obtain that $\langle \pi, Pts_2 \rangle$ is a bp-proof satisfying the backpointer condition P . \square

Thanks to Lem. C.1, in order to show Prop. 4.3 it suffices to show the following lemma:

Lemma C.2. *If π is a regular proof, every infinite branch ρ of π can be equipped with an idempotent correct backpointer.*

Proof. Let s be the maximal length of sequent antecedents in π and \mathcal{F} be the set of partial functions on $[0; s]$.

Let $eval : \mathcal{F}^* \rightarrow \mathcal{F}$ be the evaluation morphism, defined inductively by $eval(\epsilon) = id$ and $eval(\vec{u}f) = eval(\vec{u}) \circ f$. Since \mathcal{F} is a finite monoid, there exists $m \in \mathbb{N}$ such that any word $\vec{u} \in \mathcal{F}^m$ contains an infix $\vec{v} \in \mathcal{F}^+$ such that $eval(\vec{v})$ is idempotent.

We say that two $*-l$ addresses u, v have same *type* if the subtrees rooted in u, v in π are isomorphic. By extension, the type of a position is the type of its address.

Since π is valid and the number of distinct types is finite, every branch of π contains a thread going through infinitely many $*-l$ positions of the same type, and in particular it is the case for the branch ρ where we want to find an idempotent correct backpointer. Let $n \in \mathbb{N}$ such that the prefix of ρ of length n contains a thread which goes through $m + 1$ such positions $\langle v_0, 0 \rangle, \langle v_1, 0 \rangle, \dots, \langle v_m, 0 \rangle$ of the same type.

For all $i \in [1; m]$, we define $f_i = f_{v_{i-1}, v_i} \in \mathcal{F}$ as above. By choice of m , there exists $i < j \in [1; m]$ such that $f = f_i \circ f_{i+1} \circ \dots \circ f_j$ is idempotent. Moreover, as witnessed by the thread t , we have $f(0) = 0$. We define a backpointer pt with $src(pt) = v_j$ and $tgt(pt) = v_{i-1}$. \square

Together with Lem. C.1, we can conclude that every regular proof can be extended into an ibp-proof.

We now state a strengthening of Lem. C.2.

Lemma C.3. *Let π be a regular proof, and $\langle u, 0 \rangle$ be a $*-l$ position of π . Every infinite branch of π can be equipped by a correct idempotent backpointer pt such that*

- either $\langle src(pt), 0 \rangle$ and $\langle tgt(pt), 0 \rangle$ are ancestors of $\langle u, 0 \rangle$,
- or the segment $[tgt(pt), src(pt)]$ contains no $*-l$ position that is an ancestor of $\langle u, 0 \rangle$.

Proof. This is an adaptation of the proof of Lem. C.2. When the branch ρ is fixed, two cases can occur:

- if infinitely many ancestors of $\langle u, 0 \rangle$ are principal on ρ , then infinitely many of them have the same type, and we can use the proof of Lem. C.2 to define a correct idempotent backpointer between two of them.
- if only finitely many ancestors of $\langle u, 0 \rangle$ are principal on ρ , it suffices to consider a suffix ρ' of ρ containing none of these positions, and use the proof of Lem. C.2 to define a correct idempotent backpointer in this suffix. \square

C.2 Proof of Prop. 4.4

We want to show that every affine and regular proof π can be extended into a ranked proof $\langle \pi, Pts, rk \rangle$.

We describe a recursive algorithm that builds a set of backpointers and assigns ranks to all canonical star positions. We start by recalling the global proof scheme. Roughly, the idea is to consider the graph of addresses where sources and targets of backpointers are identified. Strongly Connected Components (SCCs) of this graph can then be treated independently. In each SCC, we identify a master thread: a thread that explores each canonical address infinitely many

times by going through all backpointers, and validates the corresponding infinite branch. When this thread is identified, we change the positions of backpointers to satisfy structural constraints related to rules (Thd) and (Blk), and we assign positions of this thread with the maximal rank of the SCC. We then remove addresses where this thread is principal, and recursively work on SCCs obtained on the remaining parts of the graph. When recombining SCCs together, ranks are shifted to satisfy rules (Con), (Dec), (Org), and (Blk), by avoiding overlaps of ranks and assigning higher ranks to SCCs with smaller addresses.

Let us now give a more detailed step-by-step description of this recursive algorithm:

1. Use Prop. 4.3 to obtain Pts_0 such that $\pi_{bp}^0 = \langle \pi, Pts_0 \rangle$ is an ibp-proof.
2. Consider the *canonical graph* G of canonical addresses, where sources and targets of backpointers from Pts_0 are identified. We will treat separately each strongly connected component (SCC) of G . When ranks have been assigned in each SCC, a shift is applied (i.e. all ranks of the same SCC are shifted by the same amount) so that different SCCs do not share ranks, and rules (Dec) and (Org) are respected.
3. We now describe the process of assigning ranks within a SCC of the canonical graph. By strong connectedness, we can build an infinite path visiting all nodes of this graph infinitely many times. This corresponds to an infinite branch in π , which must be validated by a *master thread* t : a thread going through all backpointers infinitely many times. All positions of this master thread are assigned with a maximal rank M . This rank M is a placeholder standing for “maximal rank in the current SCC”, and will be shifted to an appropriate value after the subsequent recursive calls are completed.
4. We now need to reorganise backpointers in order to respect rule (Thd) and (Blk) in the final bp-proof, by forbidding a $*-l$ rule of maximal rank M to occur in the scope of a backpointer linking rules of lower rank (to be assigned later). This construction is given by Lem. C.1 and C.3, where the distinguished position $\langle u, 0 \rangle$ is the origin of rank M . This shows that we can choose idempotent backpointers that are either linking addresses of rank M , or that do not contain addresses of rank M in their scope. In this last case the thread of rank M is spectator between the source and the target of the backpointer.
5. We now consider the strongly connected canonical graphs obtained by removing all $*-l$ rules of rank M , and call recursively the algorithm from step 2 on each of these strongly connected graphs. As before, ranks of each SCC will then be shifted to avoid overlaps and respect rules (Dec) and (Org).

This process terminates, because the maximal number of formulas with unassigned rank in a sequent decreases at each step. Indeed, our master thread visited every sequent of the strongly connected canonical graph, and assigned rank M to a star position in each sequent. Moreover, this algorithm generates a set of pointers Pts and a rank function rk such that $\langle \pi, Pts, rk \rangle$ is a ranked proof. Rule (BP) is ensured by the identification of sources and target of pointers in canonical proof graphs. Rules (Dec) and (Org) are ensured when shifting the ranks of SCC after internal computations. Rule (Thd) is ensured by the choice of master thread of maximal rank, that must be preserved in all paths of the canonical graph. Rule (Blk) is ensured by step 4 and by avoiding overlapping of ranks between different SCCs. Rule (Con) is ensured by step 3, where all positions assigned with the same rank are connected by a thread, and by avoiding overlapping of ranks between SCCs. Originally, only canonical star positions are assigned a rank, but it is straightforward to extend the rank function to all star positions.

C.3 Validity of proofs in ranked normal form

Lemma C.4. *Every (affine) ranked preproof is valid.*

Proof. We show this result by exhibiting a valid thread for each infinite branch of the preproof.

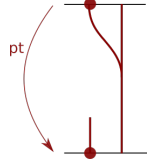
Let $\langle \pi, Pts, rk \rangle$ be an (affine) ranked preproof. Let ρ be an infinite branch of π , corresponding to an infinite path b in the canonical graph of π , staying in canonical address and following backpointers. Let Pts^∞ be the restriction of Pts to the backpointers that are seen infinitely often when going along b . This set is not empty because b is infinite and Pts is finite. Let r be the maximal rank in Pts^∞ and bp be the associated backpointer:

$$r = \max\{rk(src(pt)) \mid pt \in Pts^\infty\} = rk(src(bp))$$

There exists some node v in the infinite path b such that from this node the only backpointers that are seen form exactly the set Pts^∞ . Note that from this point every node is between $tgt(pt)$ and $src(pt)$ for some $pt \in Pts^\infty$ (depending on the current node). Let's follow (in b) the thread of the principal formula of the first occurrence of the node $src(bp)$ after v . Then the thread goes only through positions of the proof that are located between the target and the source of a backpointer of rank $r' \leq r$. If $r' < r$, the thread exists and stays spectator between those points by (Thd). If $r' = r$, the thread also exists between the target and the source of the backpointer because π being a ranked preproof implies in particular that it is an ibp-preproof. Moreover this thread is principal infinitely often because the node $src(bp)$ is visited infinitely often. Thus any branch ρ is valid, and the ranked preproof π is valid. \square

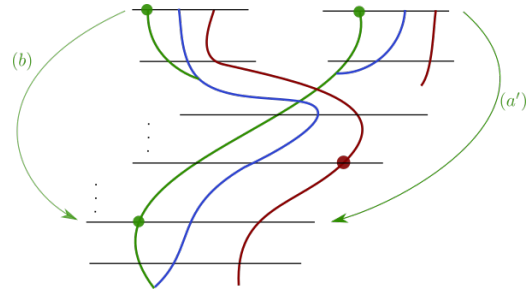
C.4 Why the ranked approach cannot be adapted with contractions

The affine construction fails with contractions, because of patterns as depicted on the right, where the potential idempotent backpointer pt is such that $f_{pt}(0)$ is defined but different from 0.



On the cycle of the pattern the red thread exists, is not valid, but is not really spectator either since it can branch to a $*-l$ rule whenever it wants.

This is exactly the phenomenon that happens in the proof for Ackermann-Péter's function given in Sect. 2.5, Fig. 5. The following picture sketches the structure and thread behaviour of the ibp-proof obtained in Ex. 4.2, ignoring some irrelevant parts.



We can recognise on the cycle formed by the backpointer (b) the pattern depicted above, that makes it impossible to assign a rank function that would not violate the (Blk) rule.

In order for the (Blk) rule to be verified by the (a') pointer, the green formulas should have a higher rank than the red one. However the green formula is not a real recursive argument of the left loop so if the backpointer (b) is left as represented on the above picture the later translation would not yield an equivalent T term. Yet if the backpointer (b) is shifted one level up so that it points to the red $*-l$ address it would again violate the (Blk) rule.

This shows that no thread can be chosen as the master thread, and the construction is stuck.

C.5 Typing derivations for the affine C to affine T translation

We give here the typing derivations needed in the simple cases of translation from affine C to affine T (Thm. 4.5). The cases for right introduction rules are given in Fig. 10; the ones for left introduction rules and cut are given in Fig. 11.

D Additional details on subsystems of second-order arithmetic

D.1 Definition of RCA_0

In the definition of RCA_0 (Def. 5.2), the available instances of comprehension and the notion of RCA_0 itself are mutually defined. It is equivalent to extending $Q2$ by Σ_1^0 -induction

$$\begin{array}{c}
 1-i \frac{}{\vdash \langle \rangle : 1} \qquad **i_e \frac{}{\vdash [] : e^*} \\
 \\
 \frac{X : E \vdash M : e \quad Y : F \vdash N : f}{\dashv\vdash \frac{}{X : E, Y : F \vdash \langle M, N \rangle : e \cdot f}} \\
 \\
 \frac{X : E \vdash M : e_i}{X : E \vdash \mathbf{i}_i M : e_0 + e_1} \quad i \in \{0, 1\} \\
 \\
 \frac{x : e, X : E \vdash M : f}{\rightarrow\vdash \frac{}{X : E \vdash \lambda x. M : e \rightarrow f}} \\
 \\
 \frac{X : E \vdash M : e \quad X : E \vdash N : f}{\cap\vdash \frac{}{X : E \vdash \langle\langle M, N \rangle\rangle : e \cap f}} \\
 \\
 \frac{X : E \vdash M : e \quad Y : F \vdash N : e^*}{**i_{::} \frac{}{X : E, Y : F \vdash M :: N : e^*}}
 \end{array}$$

Figure 10. The typing derivations for right rules where green sequents represent the results obtained through the induction.

and the following schema,

$$\forall x(\varphi \equiv \psi) \Rightarrow \exists X \forall x(X(x) \equiv \varphi)$$

where φ and ψ vary over Σ_1^0 formulas.

D.2 Definition of WKL_0

WKL_0 extends RCA_0 with *weak König’s lemma*:

“every infinite binary tree has an infinite branch”.

This statement may be formalised as follows:

- “ X is infinite” is formalised as $\forall x \exists y > x. y \in X$, stating that there are arbitrarily large elements of X .
- We may define the terms $S_1 t \triangleq 2t + 1$ and $S_2 t \triangleq 2t + 2$ to stand for the two children of a node t in a binary tree.² Note here that we are construing numbers as binary strings in dyadic notation.
- “ X is a tree” is formalised as $\forall x \in X(x = 0 \vee \exists y \in X(x = S_1 y \vee x = S_2 y))$, stating that X is prefix-closed.
- “ Y is an infinite branch” is formalised as $0 \in Y \wedge \forall x \in Y(S_1 x \in Y \equiv S_2 x \notin Y)$, stating that every node in Y has exactly one child.

While WKL_0 is strictly stronger than RCA_0 , it is conservative over RCA_0 for arithmetical formulas:

Theorem D.1 (Harrington, e.g. see [3]). *If φ is arithmetical and WKL_0 proves φ , then RCA_0 proves φ .*

To extract primitive recursive functions, we only need the rather weak specialisation of this result to $\varphi \in \Pi_2^0$. This

²Even more formally, $2x + 1$ is $S_0 \times x + S_0$ and so on.

$$\begin{array}{c}
 \frac{id \frac{}{x : 1 \vdash x : 1} \quad X : E \vdash M : e}{1-e \frac{}{x : 1, X : E \vdash \text{let } \langle \rangle := x \text{ in } M : e}} \\
 \\
 \frac{id \frac{}{z : e \cdot f \vdash z : e \cdot f} \quad x : e, y : f, X : E \vdash N : g}{\dashv\vdash \frac{}{z : e \cdot f, X : E \vdash \text{let } \langle x, y \rangle := z \text{ in } N : g}} \\
 \\
 \frac{id \frac{}{z : e + f \vdash z : e + f} \quad x : e, X : E \vdash M : g \quad y : f, X : E \vdash N : g}{+e \frac{}{z : e + f, X : E \vdash \mathbf{D}(z; x.M; y.N) : g}} \\
 \\
 \frac{\rightarrow\vdash \frac{x : f, Y : F \vdash M : g}{Y : F \vdash \lambda x. M : f \rightarrow g} \quad \dashv\vdash \frac{id \frac{}{y : e \rightarrow f \vdash y : e \rightarrow f} \quad X : E \vdash N : e}{\rightarrow\vdash \frac{}{y : e \rightarrow f, X : E \vdash yN : f}}}{\rightarrow\vdash \frac{}{y : e \rightarrow f, X : E, Y : F \vdash (\lambda x.M)(yN) : g}} \\
 \\
 \frac{id \frac{}{z : e_0 \cap e_1 \vdash z : e_0 \cap e_1} \quad \rightarrow\vdash \frac{x : e_i, X : E \vdash M : f}{X : E \vdash \lambda x. M : e_i \rightarrow f}}{\dashv\vdash \frac{}{z : e_0 \cap e_1, X : E \vdash (\lambda x.M)(\mathbf{p}z) : f}} \\
 \\
 \frac{\rightarrow\vdash \frac{x : e, Y : F \vdash M : g}{Y : F \vdash \lambda x. M : e \rightarrow g} \quad X : E \vdash N : e}{\rightarrow\vdash \frac{}{X : E, Y : F \vdash (\lambda x.M)N : g}}
 \end{array}$$

Figure 11. The typing derivations for left rules where green sequents represent the results obtained through the induction. The last derivation is the one for cut rule, as described in Sect. 4.4. The typing rules for weakening and identity are trivial and omitted here.

particular specialisation has several proofs, first by Friedman via model-theoretic methods, and then more directly in [24] using the Dialectica interpretation.

Together with Prop. 5.4, we have:

Proposition D.2 (Witnessing for WKL_0). *Suppose WKL_0 proves $\forall \vec{x} \exists y \varphi(\vec{x}, y)$, where φ is Σ_1^0 and contains no set symbols. Then there is a primitive recursive function $f(\vec{x})$ such that $\mathbb{N} \models \forall \vec{x}. \varphi(\vec{x}, f(\vec{x}))$.*

D.3 Extraction and certification

The Dialectica interpretation actually gives us more than Prop. 5.3: it also implies that a proof of correctness is extracted within a rudimentary equational theory over T^3 . However we do not concern ourselves with this additional feature in this work.

Similarly, we can get more from the assumptions of Prop. D.2, in the sense that a proof of correctness is also extracted within an equational theory over the primitive recursive functions, known as *primitive recursive arithmetic*.

³As pointed out in the introduction, strictly speaking it is this theory that is usually called T , whereas our calculus comprises only its underlying term language.

D.4 Büchi automata algorithms in RCA_0

The correctness of universality or inclusion algorithms for Büchi automata usually rely on Ramsey's theorem,⁴ which is not provable in RCA_0 even for pairs with two colours (see, e.g., [22]), but it is known that the result can also be formalised using the so-called *additive* version of Ramsey's theorem, where the colouring must be compatible with a semigroup structure; this argument was used in [25]. It is this step where the non-uniformity of the above proposition is crucial, since the result is established by a meta-level induction on the number of colours, cf. [12]. Note that the usual Ramsey theorem is not typically proved by induction on the number of colours and, as established in [12], no universality algorithm can be proved correct uniformly in RCA_0 by reduction to a form of Gödel incompleteness for cyclic theories of arithmetic.

E Additional details for Sect. 6

E.1 Reduction

We give here a more explicit definition of the reduction relation (Def. 6.1): reduction (\rightsquigarrow) is the least relation on programs which is closed under contexts (i.e., if $P \rightsquigarrow P'$ then $P :: Q \rightsquigarrow P' :: Q$, $Q :: P \rightsquigarrow Q :: P'$, and $v(\vec{Q}, P, \vec{R}) \rightsquigarrow v(\vec{Q}, P', \vec{R})$), and such the following rules are satisfied. In each case, we assume that the length of the vectors of programs match the length of the corresponding lists of formulas.

structural reductions:

- If π_u is $\text{id} \frac{}{e \vdash e}$ then $u(P) \rightsquigarrow P$.
- If π_u ends $x \frac{E, f, e, F \vdash g}{E, e, f, F \vdash g}$ then $u(\vec{P}, P, Q, \vec{Q}) \rightsquigarrow u_0(\vec{P}, Q, P, \vec{Q})$.
- If π_u ends $w \frac{E \vdash g}{e, E \vdash g}$ then $u(P, \vec{P}) \rightsquigarrow u_0(\vec{P})$.
- If π_u ends $c \frac{e, E \vdash g}{e, e, E \vdash g}$ then $u(P, \vec{P}) \rightsquigarrow u_0(P, P, \vec{P})$.
- If π_u ends $\text{cut} \frac{E \vdash e \quad e, F \vdash f}{E, F \vdash f}$ then $u(\vec{P}, \vec{Q}) \rightsquigarrow u_1(u_0(\vec{P}), \vec{Q})$.

constructor reductions:

- If π_w ends $1-r \frac{}{\vdash 1}$ then $w() \rightsquigarrow \langle \rangle$.
- If π_w ends $*-r_e \frac{}{\vdash e^*}$ then $w() \rightsquigarrow []$.
- If π_w ends $*-r. \frac{E \vdash e \quad F \vdash e^*}{E, F \vdash e^*}$ then $w(\vec{P}, \vec{Q}) \rightsquigarrow w_0(\vec{P}) :: w_1(\vec{Q})$.

left/constructor reductions:

- If π_v ends $1-l \frac{E \vdash g}{1, E \vdash g}$ then $v(\langle \rangle, \vec{P}) \rightsquigarrow v_0(\vec{P})$.
- If π_v ends $*-l \frac{E \vdash g \quad e, e^*, E \vdash g}{e^*, E \vdash g}$ then $v([], \vec{P}) \rightsquigarrow v_0(\vec{P})$
and $v(P :: Q, \vec{P}) \rightsquigarrow v_1(P, Q, \vec{P})$.

left/right reductions:

- If π_w ends $-r \frac{E \vdash e \quad F \vdash f}{E, F \vdash e \cdot f}$ and π_v ends $-l \frac{e, f, G \vdash g}{e \cdot f, G \vdash g}$ then $v(w(\vec{P}, \vec{Q}), \vec{R}) \rightsquigarrow v_0(w_0(\vec{P}), v_1(\vec{Q}), \vec{R})$.
- If π_w ends $+r_i \frac{E \vdash e_i}{E \vdash e_0 + e_1}$ and π_v ends $+l \frac{e_0, F \vdash g \quad e_1, F \vdash g}{e_0 + e_1, F \vdash g}$ then $v(w(\vec{P}), \vec{Q}) \rightsquigarrow v_i(w_0(\vec{P}), \vec{Q})$, for $i \in \{0, 1\}$.
- If π_w ends $\cap-r \frac{E \vdash e_0 \quad E \vdash e_1}{E \vdash e_0 \cap e_1}$ and π_v ends $\cap-l_i \frac{e_i, F \vdash g}{e_0 \cap e_1, F \vdash g}$ then $v(w(\vec{P}), \vec{Q}) \rightsquigarrow v_0(w_i(\vec{P}), \vec{Q})$, for $i \in \{0, 1\}$.
- If π_w ends $\rightarrow-r \frac{e, E \vdash f}{E \vdash e \rightarrow f}$ and π_v ends $\rightarrow-l \frac{F \vdash e \quad f, G \vdash g}{e \rightarrow f, F, G \vdash g}$ then $v(w(\vec{P}), \vec{Q}, \vec{R}) \rightsquigarrow v_1(w_0(v_0(\vec{Q}), \vec{P}), \vec{R})$.

Note that the choice of using left/constructor or left/right rules for a given connective corresponds to the choice of having explicit constructors for that connective in the syntax of programs. Constructors are put for all connectives in [15]; in that case, constructors for negative connectives should not be considered as evaluation contexts.

We prove the characterisation of irreducible programs, in RCA_0 :

Lemma E.1 (Lem. 6.2 in the main text). *If P is irreducible, then P is of the form*

- $\langle \rangle$, $[]$, or $P_1 :: P_2$ for some programs P_1, P_2 ; or
- $v(\vec{P})$ for some v s.t. π_v ends with $+r_i, -r, \cap-r$ or $\rightarrow-r$.

Proof. The characterisation given in the statement is computable, and so we may prove the lemma by Σ_1^0 -induction on the structure of programs. If P starts with a constructor, we are done; otherwise, if $P = v[\vec{P}]$ then v cannot be a structural rule, the identity rule, or the cut rule, otherwise P would reduce. If v is a left introduction rule then by induction P_1 (which is irreducible) must be a constructor or of the form $w[\vec{Q}]$ with w a right introduction rule, thus enabling a reduction step for P , a contradiction. \square

E.2 Reducible programs (ACA_0)

We abbreviate $P \downarrow_{\pi} P'$ as $P \downarrow P'$ in the sequel.

⁴For any function $c : \mathbb{N}^k \rightarrow \{0, \dots, n-1\}$, there is an infinite set X and $m < n$ such that $\forall x_1, \dots, x_k. c(\vec{x}) = m$.

The complete definition of the sets R_e of *reducible programs* is the following:

$$\begin{aligned} R_1 &\triangleq \{P \mid P \downarrow \langle \rangle\} \\ R_{e^*} &\triangleq \{P \mid P \downarrow Q_1 :: \dots :: Q_n, Q_1, \dots, Q_n \in R_e\} \\ R_{e \cdot f} &\triangleq \{P \mid P \downarrow v(\vec{Q}, \vec{R}), v \text{ a } \dashv\vdash\text{-}r, v0(\vec{Q}) \in R_e, v1(\vec{R}) \in R_f\} \\ R_{e \cap f} &\triangleq \{P \mid P \downarrow v(\vec{Q}), v \text{ a } \cap\text{-}r, v0(\vec{Q}) \in R_e, v1(\vec{Q}) \in R_f\} \\ R_{e_0 + e_1} &\triangleq \{P \mid P \downarrow v(\vec{Q}), v \text{ a } +\text{-}r_i, vi(\vec{Q}) \in R_{e_i}\} \\ R_{e \rightarrow f} &\triangleq \{P \mid P \downarrow v(\vec{Q}), v \text{ a } \rightarrow\text{-}r, \forall Q \in R_e, v0(Q, \vec{Q}) \in R_f\} \end{aligned}$$

(Like above, in the second case, assuming that the lengths of the vectors are consistent with the rule instances used at v .)

The key technical lemma for weak normalisation is proved below, in ACA_0 . We often use the fact that if $P \in R_e$, then $P \downarrow P'$ for some $P' \in R_e$, which we abbreviate as $P \downarrow P' \in R_e$. We also write $P \in R_e^\downarrow$ when $P \in R_e$ and P is irreducible. We use the notation \rightsquigarrow only for left-most innermost reduction steps.

Lemma E.2 (Lem. 6.6 in the main text). *For every address $w : E \vdash e$, for all $\vec{P} \in R_E$ such that $w(\vec{P}) \notin R_e$, there are v, F, f, \vec{Q} such that $|v| = |w| + 1$, $v : F \vdash f$, $v(\vec{Q}) \notin R_f$, and:*

1. for all i, j s.t. $\langle v, i \rangle \triangleleft \langle w, j \rangle$, we have $|Q_i| = |P_j|$, and
2. for all i, j s.t. $\langle v, i \rangle \triangleleft \langle w, j \rangle$, we have $|Q_i| < |P_j|$.

(Where given $P \in R_{e^*}$, we write $|P|$ for the length of the list given by the definition of R_{e^*} .)

Proof. We can assume w.l.o.g. that the elements of \vec{P} are irreducible. We reason by case analysis on the rule used at w ; we only list the most significant cases. We call the vector \vec{Q} we have to provide the *witness*.

$$\text{cut} : \pi_w \text{ ends } \text{cut} \frac{E \vdash e \quad e, F \vdash f}{E, F \vdash f}. \text{ Assume } \vec{P} \in R_E^\downarrow, \vec{Q} \in R_F^\downarrow$$

and $w(\vec{P}, \vec{Q}) \notin R_f$. There are two cases:

- if $w0(\vec{P}) \notin R_e$ then we choose $v = w0$, taking \vec{P} as witness.
- if $w0(\vec{P}) \in R_e$ then we choose $v = w1$, taking $w0(\vec{P}), \vec{Q}$ as witness since

$$w(\vec{P}, \vec{Q}) \rightsquigarrow w1(w0(\vec{P}), \vec{Q})$$

$$c : \pi_w \text{ ends } c \frac{e, E \vdash g}{e, e, E \vdash g}. \text{ Assuming } P \in R_e^\downarrow, \vec{P} \in R_E^\downarrow, \text{ we}$$

take $v = w0$ with witness P, P, \vec{P} , since

$$w(P, \vec{P}) \rightsquigarrow w0(P, P, \vec{P})$$

$$\rightarrow\text{-}r : \pi_w \text{ ends } \rightarrow\text{-}r \frac{e, E \vdash f}{E \vdash e \rightarrow f}. \text{ Assume } \vec{P} \in R_E^\downarrow \text{ and } w(\vec{P}) \notin$$

$R_{e \rightarrow f}$. $w(\vec{P})$ is irreducible, so that there must be a $R \in R_e$ such that $w0(R, \vec{P}) \notin R_f$. We choose $v = w0$ with R, \vec{P} as witness.

$$\rightarrow\text{-}l : \pi_w \text{ ends } \rightarrow\text{-}l \frac{E \vdash e \quad f, F \vdash g}{e \rightarrow f, E, F \vdash g}. \text{ Assume } P = u[\vec{R}] \in$$

$R_{e \rightarrow f}^\downarrow$, $\vec{P} \in R_E^\downarrow$, $\vec{Q} \in R_F^\downarrow$ and $w(P, \vec{P}, \vec{Q}) \notin R_g$. There are two cases:

- if $w0(\vec{P}) \notin R_e$, we take $v = w0$ with witness \vec{P} .
- if $w0(\vec{P}) \in R_e$, then $w0(\vec{P}) \downarrow P_0 \in R_e$. By definition of $R_{e \rightarrow f}$ we obtain $u0(P_0, \vec{R}) \in R_f$. We choose $v = w1$, taking $u0(P_0, \vec{R}), \vec{Q}$ as witness, since

$$\begin{aligned} w(P, \vec{P}, \vec{Q}) &\rightsquigarrow w1(u0(w0[\vec{P}], \vec{R}), \vec{Q}) \\ &\rightsquigarrow^* w1(u0(P_0, \vec{R}), \vec{Q}) \end{aligned}$$

$$*\text{-}r : \pi_w \text{ ends } *\text{-}r : \frac{E \vdash e \quad F \vdash e^*}{E, F \vdash e^*}. \text{ Assume } \vec{P} \in R_E^\downarrow, \vec{Q} \in R_F^\downarrow,$$

and $w(\vec{P}, \vec{Q}) \notin R_{e^*}$. If $w0(\vec{P}) \notin R_e$ we take $v = w0$ with \vec{P} as witness. Otherwise $w0(\vec{P}) \downarrow R_0 \in R_e$ and we take $v = w1$ with \vec{Q} as witness. Indeed, if we had $w1(\vec{Q}) \in R_{e^*}$ then we would get $w1(\vec{Q}) \downarrow R_1 :: \dots :: R_n$ with the R_i in R_e ; this would contradict the assumption about w since

$$\begin{aligned} w(\vec{P}, \vec{Q}) &\rightsquigarrow w0(\vec{P}) :: w1(\vec{Q}) \\ &\rightsquigarrow^* R_0 :: w1(\vec{Q}) \\ &\rightsquigarrow^* R_0 :: R_1 :: \dots :: R_n \end{aligned}$$

$$*\text{-}l : \pi_w \text{ ends } *\text{-}l \frac{E \vdash g \quad e, e^*, E \vdash g}{e^*, E \vdash g}. \text{ Assume } P \in R_{e^*}^\downarrow, \vec{P} \in$$

R_E^\downarrow and $w(P, \vec{P}) \notin R_g$. According to the definition of R_{e^*} we can distinguish two cases:

- if $P = []$, we take $v = w0$ with \vec{P} as witness:

$$w(P, \vec{P}) \rightsquigarrow w0(\vec{P})$$

- or $P = X :: Q$, and we take $v = w1$ with X, Q, \vec{P} as witness:

$$w(P, \vec{P}) \rightsquigarrow w1(X, Q, \vec{P})$$

We have $|P| = |Q| + 1$ in this case, so that we satisfy the condition 2/ for $i = 1$ and $j = 0$. (this is the only place where this condition is not void)

The condition 1/ is straightforward to check in all cases. \square

E.3 Alternative termination proof in the affine case

We assume a regular and affine proof π in this section. We let V, W range over finite antichains of addresses (w.r.t. the prefix ordering \sqsubseteq).

A program P is *coherent* if the sequence of addresses it contains forms an antichain, which we denote by $V(P)$.

Lemma E.3. *If P is coherent and $P \rightsquigarrow P'$ then P' is coherent and every address in $V(P')$ is either already in $V(P)$, or an immediate successor of some address in $V(P)$.*

The *run* of a program P is the sequence of addresses or pairs of addresses corresponding to the redexes fired during the (potentially infinite) leftmost innermost reduction of P .

Recall that irreducible programs of type e^* are lists of programs of type e (by Lem. 6.2). The *weight* of such a program is the length of this list.

Theorem E.4 (Weak normalisation in affine proofs). *For every coherent program P , there exists P' with $P \downarrow P'$.*

Proof. We prove that the run of P is finite. By Lem. E.3, the subset of addresses appearing in this run forms a forest rooted in $V(P)$, and every address appears at most once. Suppose by contradiction that the run is infinite. By weak König's Lemma one can extract an infinite branch of π which is contained in the run. By validity, this branch must contain a thread along a star formula f^* which is infinitely often principal. By analysis of the reduction rules, and thanks to the innermost strategy, we find an infinite sequence of irreducible programs of type f^* whose weights are strictly decreasing, which is impossible. \square

Note that the above argument requires an innermost reduction strategy so that we can compute weights and get a contradiction. It also breaks with contraction: in this case a given address may appear repeatedly in a run, so that a potential infinite run could stay below a finite prefix of π .

The above proof exploits weak König's lemma to extract an infinite branch and use the validity criterion. Unfortunately, it cannot be formalised in WKL_0 as it stands: the run of P , seen as a collection of addresses, is only recursively enumerable (until we discover that it is in fact finite). Thus we cannot define the corresponding set in RCA_0 , where set-comprehension is restricted to provably recursive formulas. This prevents us from calling weak König's lemma in WKL_0 .

In contrast to RCA_0 , WKL_0 has the ability to define non-recursive sets (e.g., an infinite branch of the Kleene tree). Nevertheless, we do not see how to use weak König's lemma to turn the run of P into a set in WKL_0 before we know it is actually finite.