Cyclic proofs, system T, and the power of contraction

Denis Kuperberg, Laureline Pinault, Damien Pous

Cyclic Proofs, System T, and the Power of Contraction (extended version, with appendices)*

DENIS KUPERBERG, Univ Lyon, CNRS, ENS de Lyon, UCBL, LIP UMR 5668, F-69342, France
LAURELINE PINAULT, Univ Lyon, CNRS, ENS de Lyon, UCBL, LIP UMR 5668, F-69342, France
DAMIEN POUS, Univ Lyon, CNRS, ENS de Lyon, UCBL, LIP UMR 5668, F-69342, France

We study a cyclic proof system C over regular expression types, inspired by linear logic and non-wellfounded proof theory. Proofs in C can be seen as strongly typed goto programs. We show that they denote computable total functions and we analyse the relative strength of C and Gödel's system T. In the general case, we prove that the two systems capture the same functions on natural numbers. In the affine case, i.e., when contraction is removed, we prove that they capture precisely the primitive recursive functions—providing an alternative and more general proof of a result by Dal Lago, about an affine version of system T.

Without contraction, we manage to give a direct and uniform encoding of C into T, by analysing cycles and translating them into explicit recursions. Whether such a direct and uniform translation from C to T can be given in the presence of contraction remains open.

We obtain the two upper bounds on the expressivity of C using a different technique: we formalise weak normalisation of a small step reduction semantics in subsystems of second-order arithmetic: ACA_0 and RCA_0.

Additional Key Words and Phrases: Cyclic proofs, system T, cyclic type system, primitive recursion, linear logic, regular expressions, second order arithmetic, reverse mathematics

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 graphs are regular trees are known as *cyclic* proofs; being finite objects, they can be communicated 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., [Brotherston and Simpson 2011]), while the converse problem is often harder [Berardi and Tatsuta 2017b; Simpson 2017], or impossible [Berardi and Tatsuta 2017a; Das 2020]. Cyclic proof systems have been recently used in the context of the μ-calculus [Afshari and Leigh 2017; Doumane 2017] and Kleene algebra [Das et al. 2018; Das and Pous 2017, 2018], in order to obtain completeness results, and in the context of linear logics [Doumane et al. 2016; Fortier and Santocanale 2013].

Here we propose a cyclic proof system which we study from the other side of the Curry-Howard correspondence: the proof system is seen as a type system, and proofs (i.e., typing derivations) as programs. Intuitively, those programs are unstructured yet strongly typed goto programs; we

*This work has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 programme (CoVeCe, agreement 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).

Authors' addresses: Denis Kuperberg, Univ Lyon, CNRS, ENS de Lyon, UCBL, LIP UMR 5668, F-69342, Lyon, France; Laureline Pinault, Univ Lyon, CNRS, ENS de Lyon, UCBL, LIP UMR 5668, F-69342, Lyon, France; Damien Pous, Univ Lyon, CNRS, ENS de Lyon, UCBL, LIP UMR 5668, F-69342, Lyon, France.

This is an author produced version of the work, which includes the appendices. The definitive Version of Record, without appendices, was published in *Proceedings of the ACM on Programming Languages*, https://doi.org/10.1145/3434282.

call them cyclic programs in the sequel. Despite the strongly typed discipline, the corresponding language is low-level, and closer in spirit to assembly or goto programs than to higher-level languages with while loops or recursion, like C or Haskell.

We import from cyclic proof theory a validity criterion which makes it possible to ensure termination of cyclic programs. This criterion is non-local, but syntactic and decidable via Büchi automata algorithms. To the best of our knowledge, we are not aware of any programming language with a cyclic type system in the literature.

We characterise the computational strength of cyclic programs 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 take as types 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 [Das and Pous 2018], 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) C_{aff} and T_{aff} are equally expressive (at all types), and their functions on natural numbers are the primitive recursive functions;
(2) C and T capture the same functions on natural numbers, those that are provably total in Peano arithmetic.

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

$$\begin{tikzcd}
\mathrm{T} \arrow[rr, bend left, "\text{Thm. 3.4}"] & & \mathrm{C} \arrow[ll, dotted, bend left, "\text{Cor. 6.7, via ACA}_0"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathrm{T}_{\mathrm{aff}} \arrow[r, bend left, "\text{Thm. 3.4}"] & \mathrm{C}_{\mathrm{aff}} \arrow[l, bend left, "\text{Thm. 4.5}"] \arrow[r, dotted, bend right, "\text{Cor. 6.14, via RCA}_0"'] & \text{prim. rec.} \arrow[l, dotted, bend right, "\text{Thm. 2.15}"']
\end{tikzcd}$$

As expected, we can translate terms of T into cyclic programs of C (Theorem 3.4); this is a compilation process, the translation is uniform and maps affine terms to affine programs. We also observe that we do not need contraction to encode primitive recursive functions into C (Theorem 2.15).

Conversely, encoding cyclic programs into T is much harder: this is a decompilation processs, we have to delineate possibly complex cycle structures in order to use the very strict recursion capabilities of T. Seen from the logic side, this is in fact an instance of the difficult problem of translating cyclic proofs into inductive ones [Berardi and Tatsuta 2017a,b; Das 2020; Simpson 2017]. We manage to provide a direct and uniform encoding in the affine case (Theorem 4.5), which we do not know how to extend in the presence of contraction.

In order to get our upper bounds on the expressivity of C and affine C for functions on natural numbers (Corollary 6.7 and Corollary 6.14), we 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 Corollary 6.7 by observing that this weak normalisation proof can be performed inside the subsystem ACA_0 of second order arithmetic [Simpson 2009], whose provably recursive functions are precisely those from system T.

For the affine case (Corollary 6.14), Dal Lago's system $\mathcal{H}(\emptyset)$ [Dal Lago 2009] 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 [Abramsky et al. 2002; Baillot and Pedicini 2001; Girard 1995; Hoshino et al. 2014]. We instead prove Corollary 6.14 by using another proof of weak normalisation for C, 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 [Avigad 1996]. This yields an alternative and more general proof of Dal Lago's result (Corollary 6.13).

On system C as a programming language. As explained above, the cyclic proof system C can be seen as a type system for a low-level programming language manipulating structured values. Even though this language is pure—no side effects—and strongly typed, we insist that it is low-level because loops are expressed using goto instructions rather than high-level constructs such as while loops or recursors. Accordingly, the (cyclic) type system ensures termination via a global yet decidable criterion (an ω-regular condition). This is in sharp contrast with other terminating programming languages such as system T (or Agda, Coq), where termination is ensured using a local notion of guardedness: there, although recursions can be nested in complicated ways, termination is ensured by imposing that each recursive call must be guarded.

Related work. System T was originally introduced by Gödel in [Gödel 1958] as an equational theory built up over a fragment of the term calculus that we identify as T here. That work introduced the 'Dialectica' *functional interpretation*, that allows T to interpret Peano Arithmetic.[1] Our work can be seen as a counterpart in T to recent work on cyclic arithmetic [Das 2020; Simpson 2017].

Other infinitary versions of system T are well-known, in particular [Tait 1965]. 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 [Das 2020; Simpson 2017].

The role of contraction w.r.t. expressivity we exhibit in the present work is reminiscent of a recent result [Kuperberg et al. 2019]: in a specific cut-free fragment of C, affine proofs capture precisely the regular languages while proofs with contraction capture the DLogSpace languages.

Although there are works on using cyclic proof systems to perform proof search and reason automatically about inductive types or program equivalence [Lucanu et al. 2009; Lucanu and Rusu 2015], those ideas do not really apply in system C because it is a programming language with a fancy (i.e., cyclic) yet simple type system. Proof search would correspond to type inhabitation—a trivial problem for closed types in our setting: the types we use are not expressive enough to serve as specifications.

Notation. We write $\mathbb{N}$ for the set of natural numbers. If $i \leq j$ are natural numbers, we write $[i; j]$ for the set $\{i, i+1, \dots, j\}$ and $[i; j[$ for the set $[i; j-1]$. 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 ith 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 sequences and tuples, and ϵ or just a blank to denote the empty sequence.

[1] Gödel 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.

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

The five first entries correspond to regular expressions; the arrow adds function spaces. As a first approximation, the intersection operator ($\cap$) can be identified with the product operator ($\cdot$): both operators are interpreted as Cartesian product in the high-level semantics we define below. Our set of rules will turn the former into an *additive* conjunction and the latter into a *multiplicative* conjunction. We use the above notations for the connectives rather than those from linear logic because:

- we want to emphasise that expressions e, f are types rather than formulas (although we shall also call them *formulas* when this is more natural);
- in the presence of contraction and weakening, the linear behaviour the various connectives disappears.

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

$$\begin{aligned} [e \cdot f] \triangleq [e \cap f] &\triangleq [e] \times [f] & [1] &\triangleq 1 \\ [e + f] &\triangleq [e] + [f] & [e^*] &\triangleq [e]^* \\ [e \rightarrow f] &\triangleq [f]^{[e]} & [a] &\triangleq D_a \end{aligned}$$

We have that $[1^*]$ is in bijection with $\mathbb{N}$, so that we can use 1^* as a type for natural numbers.

We let E, F range over finite sequences of types. Given such a sequence $E = e_0, \dots, e_{n-1}$, we write $[E]$ for $[e_0] \times \dots \times [e_{n-1}]$.

We will define a sequent proof system where sequents have the shape $E \vdash e$, and proofs of such sequents, cyclic programs, will denote functions from $[E]$ to $[e]$.

2.2 Non-Wellfounded Proofs

The rules of C are given in Figure 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 [Das and Pous 2018]. The only differences are that they can be slightly simplified here since we have the exchange rule, and that there is only one arrow, 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 $\{0, 1\}^*$, the root ϵ is represented 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 $\dfrac{\pi(v_1) \ \cdots \ \pi(v_n)}{\pi(v)}$ is an instance of a rule from Figure 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).

$$
\mathit{id}\,\frac{}{e \vdash e}
\qquad
\mathit{cut}\,\frac{E \vdash e \quad e, F \vdash g}{E, F \vdash g}
\qquad
x\,\frac{E, f, e, F \vdash g}{E, e, f, F \vdash g}
\qquad
w\,\frac{E \vdash g}{e, E \vdash g}
\qquad
c\,\frac{e, e, E \vdash g}{e, E \vdash g}
$$

$$
\cdot\text{-}l\,\frac{e, f, E \vdash g}{e \cdot f, E \vdash g}
\qquad
\cdot\text{-}r\,\frac{E \vdash e \quad F \vdash f}{E, F \vdash e \cdot f}
$$

$$
+\text{-}l\,\frac{e, E \vdash g \quad f, E \vdash g}{e + f, E \vdash g}
\qquad
+\text{-}r_i\,\frac{E \vdash e_i}{E \vdash e_0 + e_1}\; i \in \{0, 1\}
$$

$$
*\text{-}l\,\frac{E \vdash g \quad e, e^*, E \vdash g}{e^*, E \vdash g}
\qquad
*\text{-}r_\epsilon\,\frac{}{\vdash e^*}
\qquad
*\text{-}r_{::}\,\frac{E \vdash e \quad F \vdash e^*}{E, F \vdash e^*}
$$

$$
1\text{-}l\,\frac{E \vdash g}{1, E \vdash g}
\qquad
1\text{-}r\,\frac{}{\vdash 1}
$$

$$
\rightarrow\text{-}l\,\frac{E \vdash e \quad f, F \vdash g}{e \rightarrow f, E, F \vdash g}
\qquad
\rightarrow\text{-}r\,\frac{e, E \vdash f}{E \vdash e \rightarrow f}
$$

$$
\cap\text{-}l_i\,\frac{e_i, E \vdash g}{e_0 \cap e_1, E \vdash g}\; i \in \{0, 1\}
\qquad
\cap\text{-}r\,\frac{E \vdash e \quad E \vdash f}{E \vdash e \cap f}
$$

Fig. 1. The rules of C.

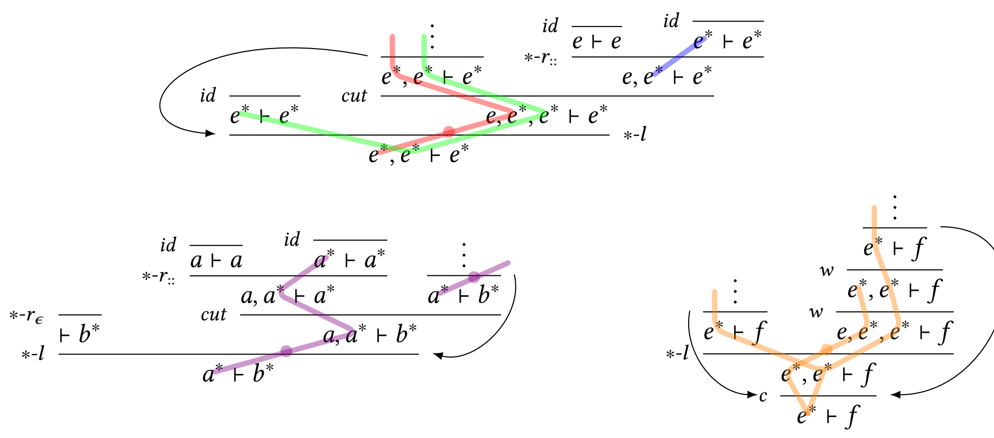


Fig. 2. Three regular preproofs.

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 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 Figure 2. The backpointers are used to denote circularity: the actual preproofs are obtained by unfolding. We define below a validity criterion for preproofs, which is satisfied only by the topmost preproof. Before doing so, we need to define threads. Those 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. A position $\langle v, i\rangle$ is a *principal* 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 $\langle v, i\rangle \lhd \langle w, j\rangle$ in the former cases, and $\langle v, i\rangle \blacktriangleleft \langle w, j\rangle$ in the latter case (in which case $i = 1$ and $j = 0$). We say that $\langle v, i\rangle$ 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 Figure 2 using shaded thick lines and an additional bullet to indicate when we pass principal steps ($\blacktriangleleft$). 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. *When working with trees, it is common in computer science to have the convention that a node in a tree has at most one parent, and many children: the root of a tree is the ancestor of all its leaves. Be careful that we use the opposite convention in the present paper, following the tradition in proof theory [Buss 1998]: the ancestors are to be found towards the leaves. This convention also matches the intuition from family trees.*

Remark 2.4. *Suppose that $u \sqsubseteq v$ are addresses in a preproof π. Then a position at v is the ancestor of at most one position at u, and it is only in the presence of contraction that a position at u may have two or more ancestors at v.*

Definition 2.5. A *thread* is a branch of the ancestry graph, i.e., a set of positions forming a linear order with respect to the ancestry relation; it is *principal* when it visits a principal position, *spectator* if it is never principal, *valid* if it is principal infinitely many often.

In the topmost preproof of Figure 2, the infinite red thread $\langle \epsilon, 0\rangle \blacktriangleright \langle 1, 1\rangle \rhd \langle 10, 0\rangle \blacktriangleright \langle 101, 1\rangle \rhd \langle 1010, 0\rangle \dots$ is valid while the infinite green thread $\langle \epsilon, 1\rangle \rhd \langle 1, 2\rangle \rhd \langle 10, 1\rangle \rhd \langle 101, 2\rangle \rhd \langle 1010, 1\rangle \dots$ is spectator. In the bottom left preproof, all threads are finite: the instances of the *cut* rule disconnect the copies of the thread $\langle \epsilon, 0\rangle \blacktriangleright \langle 1, 1\rangle$ occurring in the only infinite branch of the preproof. In the remaining preproof, all infinite threads are spectator: principal steps force the thread to terminate.

Definition 2.6. 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 Figure 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 decidable for regular preproofs: it can be formulated as a Büchi condition, and checked via standard automata algorithms. It is essentially the same as in LKA [Das and Pous 2018], which in turn is an instance of the one for μMALL [Doumane et al. 2016]: we just had to extend the notion of ancestry to cover the weakening and contraction rules. This induces an important subtlety:

Remark 2.7. *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.*

That a sequent $E \vdash e$ is provable in system C is not something interesting per se: most sequents are provable, essentially because every closed type is inhabited (see Lemma 2.13 below). Instead of provability, we do focus on proofs themselves, and on their computational content.

2.3 Computational Interpretation of System C

We now show how to interpret a proof $\pi : E \vdash e$ as a function $[\pi] : [E] \to [e]$. 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 2.8. 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 $\langle v, s\rangle \prec \langle w, t\rangle$ when $|v| = |w| + 1$ and

(1) for all i, j s.t. $\langle v, i\rangle \lhd \langle w, j\rangle$, we have $s_i = t_j$, and
(2) for all i, j s.t. $\langle v, i\rangle \mathrel{\underline{\lhd}} \langle w, j\rangle$, we have $|s_i| < |t_j|$.

(Recall that positions such as $\langle v, i\rangle$ and $\langle w, j\rangle$ in the above definition always refer to star formulas.) 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 2.9. *The relation $\prec$ 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. □

Definition 2.10. 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[2].

$$
\begin{aligned}
id &: [v](s) \triangleq s \\
cut &: [v](s,t) \triangleq [v1]([v0](s),t)
\end{aligned}
\qquad
\begin{aligned}
x &: [v](s,x,y,t) \triangleq [v0](s,y,x,t) \\
w &: [v](x,s) \triangleq [v0](s) \\
c &: [v](x,s) \triangleq [v0](x,x,s)
\end{aligned}
$$

$\cdot$-l : $[v](\langle x,y\rangle,s) \triangleq [v0](x,y,s)$

$\cdot$-r : $[v](s,t) \triangleq \langle [v0](s), [v1](t)\rangle$

$\to$-l : $[v](h,s,t) \triangleq [v1](h([v0](s)),t)$

$\to$-r : $[v](h) \triangleq (x \mapsto [v0](x,h))$

$*$-l : $[v](l,s)$ is defined by case analysis on the list l:

- $[v](\epsilon,s) \triangleq [v0](s)$
- $[v](x :: q,s) \triangleq [v1](x,q,s)$

$*$-r_ϵ : $[v]() \triangleq \epsilon$

$*$-$r_{::}$: $[v](s,t) \triangleq [v0](s) :: [v1](t)$

$+$-l : $[v](x,s)$ is defined by case analysis on $x \in [e_0 + e_1]$:

- if $x \in [e_0]$, $[v](x,s) \triangleq [v0](x,s)$
- if $x \in [e_1]$, $[v](x,s) \triangleq [v1](x,s)$

$+$-r_i : $[v](s) \triangleq [v0](s)$

1-l : $[v](\langle\rangle,s) \triangleq [v0](s)$

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

$\cap$-l_i : $[v](\langle x_0,x_1\rangle,s) \triangleq [v0](x_i,s)$

$\cap$-r : $[v](s) \triangleq \langle [v0](s), [v1](s)\rangle$

(In the *cut*, *x*, $\to$-l, $\cdot$-r and $*$-$r_{::}$ cases, the size of the tuples s and t is chosen consistently with the corresponding rule instances.)

[2] Here and elsewhere in the paper, we use commas and we omit brackets to display tuples of values such as s in a return value $[v](s)$. We also restrict our usage of brackets, ϵ and :: to display values which happen to be tuples or lists (i.e., elements of $[1]$, $[e \cdot f]$, $[e \cap f]$ or $[e^*]$ for some types e, f).

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 $\to$-l cases, the values $[v0](s)$ and $h([v0](s))$ might be arbitrarily large. This is not problematic: the corresponding positions have no children, so that those values are left unconstrained by the relation $\prec$. Similarly, in order to define the graph of the returned function in the $\to$-r-case, we call the inductive hypothesis an arbitrary number of times, with arbitrarily large values for x.

Definition 2.11. The semantics of a proof $\pi : E \vdash e$ is the function $[\pi] : [E] \to [e]$ defined by $[\pi](s) \triangleq [\epsilon](s)$.

The above semantics presents proofs of C as goto programs (the address v in a computation $[v](s)$ being the program counter) operating on a structured memory and using a strongly typed discipline to avoid runtime errors. Accordingly, we sometimes call proofs of C *cyclic programs.*

Remark 2.12. *We could have given a syntax for untyped cyclic programs (as sequences of gotos and basic instructions operating on a finite set of registers), and then presented the proof system C as a two-layers type system for those untyped cyclic programs. The first layer (preproofs) would have ensured that the values stored in the registers are manipulated along a simply typed discipline, ensuring properties such as 'progress' and 'subject-reduction'. The second layer (the global validity criterion) would have ensured termination. The syntax of those untyped programs would be quite redundant with the definition of C itself (essentially, one instruction per rule from Figure 1), whence our choice to omit it in the present paper.*

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

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

In the last equality we used the fact that $\pi_{10} = \pi_\epsilon$, so that $[10] = [\epsilon]$. 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 Figure 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 ($\to, \cap$).

Lemma 2.13. *For every closed type e, there are linear regular proofs* $\mathrm{rem}_e : e \vdash 1$ *and* $\mathrm{inh}_e : \vdash e$.

Proof. By induction on e, see Appendix A.1. □

As a consequence, weakening is admissible for closed types, by replacing it with the gadget on the left in Figure 3; moreover, every closed sequent is derivable, already in the linear fragment of C.

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

Proof. Again by induction on e, see Appendix A.1. □

$$\dfrac{\text{rem}\ \overline{\overline{e \vdash 1}} \qquad 1\text{-}l\ \dfrac{E \vdash f}{1, E \vdash f}}{e, E \vdash f}\ \mathit{cut} \qquad\qquad \dfrac{\text{dup}\ \overline{\overline{e \vdash e \cdot e}} \qquad \cdot\text{-}l\ \dfrac{e, e, E \vdash f}{e \cdot e, E \vdash f}}{e, E \vdash f}\ \mathit{cut}$$

Fig. 3. Deriving weakening and contraction.

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 Figure 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 Section 2.5 below, if we use it to replace the contraction on a star formula in the proof from Figure 5, the affine preproof we obtain is not valid: the green/blue 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 Corollary 6.14 below.

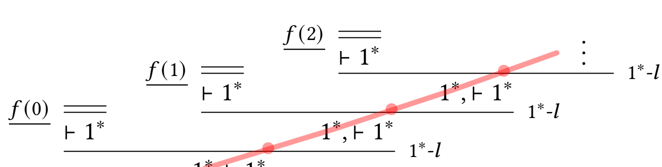
2.5 Functions on Natural Numbers

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:

$$1^*\text{-}l\ \dfrac{E \vdash g \quad 1^*, E \vdash g}{1^*, E \vdash g} \qquad\qquad 1^*\text{-}r_0\ \dfrac{}{\vdash 1^*} \qquad\qquad 1^*\text{-}r_S\ \dfrac{E \vdash 1^*}{E \vdash 1^*}$$

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) $f : \mathbb{N} \to \mathbb{N}$, we can obtain a proof $\underline{f} : 1^* \vdash 1^*$ such that $[\underline{f}] = f$: repeatedly apply the 1^*-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 2.15. *For every primitive recursive function* $f : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$, *there exists a linear and regular proof* $\pi : 1^*, \ldots, 1^* \vdash 1^*$ *such that* $[\pi] = f$.

Proof. By induction on the 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^* (Lemma 2.13). 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

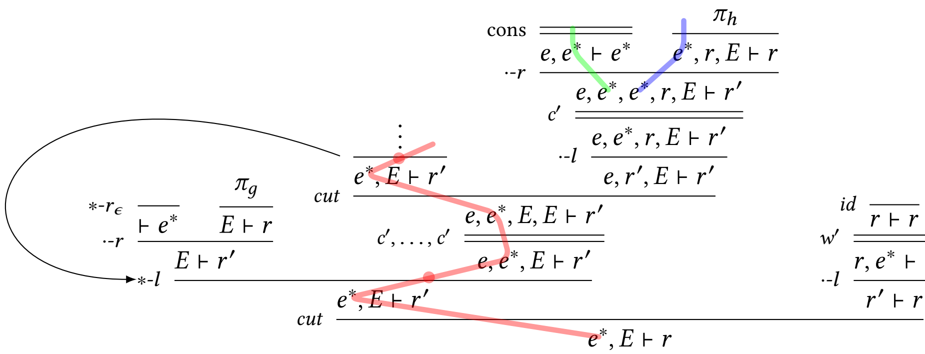
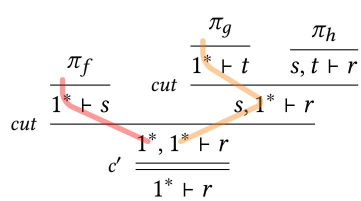


Fig. 4. Regular linear proof for primitive recursion; $e \triangleq 1$, $r \triangleq 1^*$; $r' \triangleq e^* \cdot r$.

use the following scheme:



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 (Lemma 2.14). 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 as follows:

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

where g and h are primitive recursive functions of respective arity n and $n+2$. By induction hypothesis there exist proofs π_g and π_h that encode g and h. In the recursive definition above, one can observe that both x and $\vec{y}$ 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 Figure 4, where the subproof labelled with "cons" consists of a $*\text{-}r_{::}$ step followed by two identity axioms. ◻

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 exploited only to avoid using the contraction rule and remain in the affine fragment: with contraction, we could build a proof whose sequents mention only the formula 1^*.

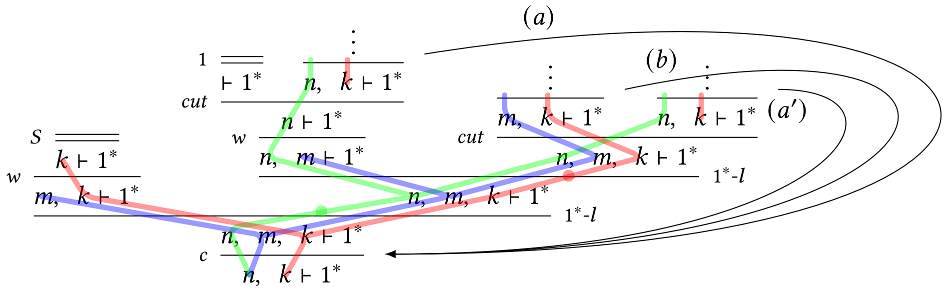


Fig. 5. A regular proof for Ackermann-Péter's function; $n \triangleq m \triangleq k \triangleq 1^*$.

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

Example 2.16. We give a regular proof whose semantics is Ackermann-Péter's function in Figure 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 and blue thread, where we go right on contraction steps (switching from green to blue) when the next visited backpointer is (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-Péter's function: we have

$$\begin{aligned}
[\epsilon](0,k) &= [0](0,0,k) = [00](0,k) = [000](k) = Sk \\
[\epsilon](Sn,0) &= [0](Sn,Sn,0) = [01](n,Sn,0) = [010](n,Sn) = [0100](n) = [01001](n,1) \\
&= [\epsilon](n,1) \\
[\epsilon](Sn,Sk) &= [0](Sn,Sn,Sk) = [01](n,Sn,Sk) = [011](n,Sn,k) = [0111](n,[0110](Sn,k)) \\
&= [\epsilon](n,[\epsilon](Sn,k))
\end{aligned}$$

We prove in the next section that we can actually represent all system T functions with regular proofs. We can also 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 Appendix A.2).

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 $0, 1$.

$$\begin{array}{rlll}
M,N,O ::= & x & \mid \lambda x.M & \mid MN \\
& & \mid \langle M,N\rangle & \mid \text{let } \langle x,y\rangle := M \text{ in } N \\
& & \mid \langle\rangle & \mid \text{let } \langle\rangle := M \text{ in } N \\
& & \mid \mathbf{i}_i M & \mid \mathbf{D}(M; x.N; x.O) \\
& & \mid [] \mid M :: N & \mid \mathbf{R}(M; N; x.y.O) \\
& & \mid \langle\!\langle M,N\rangle\!\rangle & \mid \mathbf{p}_i M
\end{array}$$

It consists of a lambda-calculus extended with pairs, singletons, sums, lists, and additive pairs. We let Γ, Δ range over *typing environments*, i.e., lists of pairs of a variable and a type. The type system is given in Figure 6. 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

$$id\,\frac{}{x : e \vdash x : e} \qquad x\,\frac{\Gamma, y : f, x : e, \Delta \vdash M : g}{\Gamma, x : e, y : f, \Delta \vdash M : g} \qquad w\,\frac{\Gamma \vdash M : f}{x : e, \Gamma \vdash M : f} \qquad c\,\frac{x : e, x : e, \Gamma \vdash M : f}{x : e, \Gamma \vdash M : f}$$

$$\cdot\text{-}e\,\frac{\Gamma \vdash M : e \cdot f \quad x : e, y : f, \Delta \vdash N : g}{\Gamma, \Delta \vdash \text{let } \langle x, y \rangle := M \text{ in } N : g} \qquad \cdot\text{-}i\,\frac{\Gamma \vdash M : e \quad \Delta \vdash N : f}{\Gamma, \Delta \vdash \langle M, N \rangle : e \cdot f}$$

$$+\text{-}e\,\frac{\Gamma \vdash S : e + f \quad x : e, \Delta \vdash M : g \quad y : f, \Delta \vdash N : g}{\Gamma, \Delta \vdash \mathbf{D}(S; x.M; y.N) : g} \qquad +\text{-}i_j\,\frac{\Gamma \vdash M : e_j}{\Gamma \vdash \mathbf{i}_j M : e_0 + e_1}\; j \in \{0, 1\}$$

$$*\text{-}e\,\frac{\Gamma \vdash L : e^* \quad \Delta \vdash M : g \quad x : e, y : g \vdash N : g}{\Gamma, \Delta \vdash \mathbf{R}(L; M; x.y.N) : g} \qquad *\text{-}i_\epsilon\,\frac{}{\vdash [] : e^*} \qquad *\text{-}i_{::}\,\frac{\Gamma \vdash M : e \quad \Delta \vdash N : e^*}{\Gamma, \Delta \vdash M :: N : e^*}$$

$$1\text{-}e\,\frac{\Gamma \vdash M : 1 \quad \Delta \vdash N : g}{\Gamma, \Delta \vdash \text{let } \langle\rangle := M \text{ in } N : g} \qquad 1\text{-}i\,\frac{}{\vdash \langle\rangle : 1}$$

$$\rightarrow\text{-}e\,\frac{\Gamma \vdash M : e \rightarrow f \quad \Delta \vdash N : e}{\Gamma, \Delta \vdash MN : f} \qquad \rightarrow\text{-}i\,\frac{x : e, \Gamma \vdash M : f}{\Gamma \vdash \lambda x.M : e \rightarrow f}$$

$$\cap\text{-}e_i\,\frac{\Gamma \vdash M : e_0 \cap e_1}{\Gamma \vdash \mathbf{p}_i M : e_i}\; i \in \{0, 1\} \qquad \cap\text{-}i\,\frac{\Gamma \vdash M : e \quad \Gamma \vdash N : f}{\Gamma \vdash \langle\!\langle M, N \rangle\!\rangle : e \cap f}$$

Fig. 6. Typing rules for system T.

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 (see Appendix B.1 for more details on the equivalence between this version of system T and the standard one).

The structural and introduction rules are term-decorated versions of the corresponding rules of C (Figure 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 ($\mathbf{R}$). This operator expects a list as first argument, and then two arguments for the cases of the empty and non-empty lists. Intuitively, we have

$$\mathbf{R}([]; M; x.y.N) = M$$
$$\mathbf{R}(X{::}Q; M; x.y.N) = N\{x \leftarrow X; y \leftarrow \mathbf{R}(Q; M; x.y.N)\}$$

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:

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

Like in Dal Lago's system $\mathcal{H}(\emptyset)$ [Dal Lago 2009], 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 Appendix B.1).

Terms should always be considered as 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 $[M] \colon [\underline{\Gamma}] \to [e]$ defined as follows by induction on the typing derivation:

$$
\begin{aligned}
id &: [x](s) \triangleq s \\
\to\text{-}e &: [MN](s,t) \triangleq [M](s)([N](t)) \\
c &: [M](v,s) \triangleq [M](v,v,s) \\
\cdot\text{-}i &: [\langle M, N\rangle](s,t) \triangleq \langle [M](s), [N](t)\rangle \\
*\text{-}e &: [\mathbf{R}(L; M; x.y.N)](s,t) \triangleq h(x_1, h(x_2, \dots h(x_n, a)\dots)),
\end{aligned}
$$

where the induction provided a list $[L](s) = x_1, \dots, x_n$, an element $a \triangleq [M](t)$, and a function $h \triangleq [N]$.

The other cases are given in Appendix B.2.

Note that in the contraction case (c), the two occurrences of M are shorthands for two distinct stages of the typing derivation: the recursive call is made on a smaller typing derivation, even though the displayed 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 $e^* \to e^* \to e^*$ 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 $\langle\rangle{::}[]$ and S for the successor function $\lambda n.\langle\rangle{::}n$, we can define Ackermann-Péter'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 $1^* \to 1^* \to 1^*$ in the empty environment. The outer recursion produces a function of type $1^* \to 1^*$. 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 $\underline{M}$.*

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 $\to$). For instance, for the $\cdot$-e case (i.e., term let $\langle x, y\rangle := M$ 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:

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

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

The $*$-e case (i.e., term $\mathbf{R}(L; M; x.y.N)$) is the only one where we introduce circularities: we obtain by induction three regular proofs $\underline{L} : \underline{\Gamma} \vdash e^*$, $\underline{M} : \underline{\Delta} \vdash g$ and $\underline{N} : e, g \vdash g$, and we construct the following preproof:

$$\dfrac{\dfrac{\underline{L}}{\underline{\Gamma}\vdash e^*} \qquad \dfrac{\dfrac{\underline{M}}{\underline{\Delta}\vdash g} \qquad \dfrac{\dfrac{\vdots}{e^*,\underline{\Delta}\vdash g} \qquad \dfrac{\underline{N}}{e,g\vdash g}}{e,e^*,\underline{\Delta}\vdash g}\,\mathit{cut}}{e^*,\underline{\Delta}\vdash g}\,*\text{-}l}{\underline{\Gamma},\underline{\Delta}\vdash g}\,\mathit{cut}$$

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 (resp. weakening) typing rule from system T only to translate contraction (resp. weakening) nodes in the starting proof, whence the second part of the statement. Moreover, we do not need to forge any new formula: all types appearing in $\underline{M}$ already appear in M. ◻

Encoding the term given in Example 3.2 for list concatenation yields the first proof in Figure 2. In contrast, encoding the term we provided for Ackermann-Péter's function (Example 3.3) does not yield the proof given in Figure 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 Figure 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 ∩ AND →)

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 Section 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 (Section 4.4).

The second step is inspired by the one sketched in [Das and Pous 2018, Theorem 33] to translate regular proofs in LAL into equational proofs in action lattices. However, the authors of [Das and Pous 2018] 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.

4.1 Proofs with Backpointers

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

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 $src(pt) \in B$.

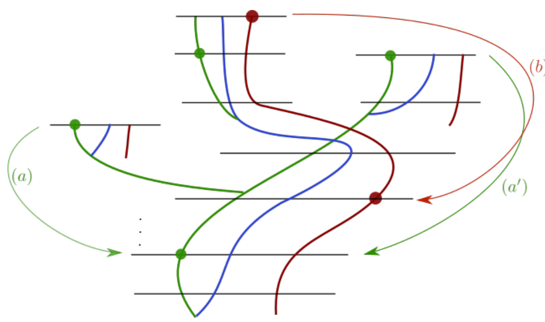


Fig. 7. Threads and backpointers of the ibp-proof for Ackermann-Péter's function.

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 [Brotherston 2005]. The backpointers define a 'bar' across the proof, and by weak König's lemma the definition implies that in every bp-proof $\langle \pi, Pts \rangle$, the set Pts must be finite. To define 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. Moreover, 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

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

Given a backpointer pt, we write f_{pt} for $f_{tgt(pt),src(pt)}$.

We say that a bp-proof is in *idempotent normal form*, or an *ibp-proof*, if for all backpointers pt, $tgt(pt)$ is a $*$-l address and f_{pt} is an idempotent with $f_{pt}(0) = 0$. This means that the branches that eventually loop only through this backpointer can be validated by the thread which is principal at $tgt(pt)$. Since there are other infinite branches in general, the validity criterion is still required.

Example 4.2. Let us go back to the proof for Ackermann-Péter's function given in Figure 5. The depicted backpointers do not point to $*$-l addresses; in order to have this property, we must shift the three backpointers one level up. We get idempotent backpointers by doing so: (a) and (a') both give rise to the idempotent partial function $0, 1 \mapsto 0$, and (b) to the idempotent $0, 1 \mapsto 1;\ 2 \mapsto 2$. However, while (a) and (a') preserve the principal position ($f_a(0) = f_{a'}(0) = 0$), this is not the case for (b): $f_b(0) = 1$. To fix this, observe that the branches that eventually visit only (b) are validated by the red thread on k. Accordingly, the backpointer (b) should thus point to the red $*$-l step on k rather than the green one on n. In order to obtain this, it suffices to shift (b) one level further up.

Doing so, we obtain an ibp-proof whose shape is depicted in Figure 7: the three backpointers are idempotent and preserve their principal position[3].

[3]The pictures can be slightly confusing here, because we do not include exchange steps. While formally, principal positions always have index 0, whence the constraint $f_{pt}(0) = 0$ in our definition of ibp-proof, the index of the principal position

Proposition 4.3. *Every regular proof π can be extended into an ibp-proof $\langle \pi, Pts\rangle$.*

We give the proof in Appendix C.1. The key idea is that since $\mathcal{F}$ is a finite monoid, any sequence containing sufficiently many elements has an idempotent infix. This makes it possible to cut every infinite branch of the starting proof by inserting an idempotent backpointer between two of the infinitely many principal positions of a thread validating the branch.

4.3 Ranked Proofs

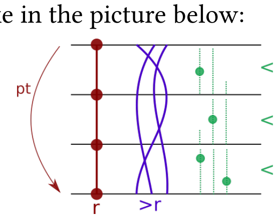
We still need one more step before translating proofs into system T terms: we use the following notion of ranks in order to organise cycles in such a way that they can be translated into recursions. Intuitively, we mark positions with natural numbers (their rank) in such a way that positions marked with the same rank give rise to a single recursive definition, and positions with highest rank give rise to the outermost recursion in the produced term.

A *ranked proof* is a tuple $\langle \pi, Pts, rk\rangle$ such that $\pi_{bp} = \langle \pi, Pts\rangle$ is an ibp-proof and rk is a function from positions of π to $\mathbb{N}$ satisfying the following properties, where we write $rk(v)$ for $rk\langle v, 0\rangle$ when v is a $*$-l address.

(BP) backpointers preserve ranks: for all $pt \in Pts$, for all i, $rk\langle src(pt), i\rangle = rk\langle tgt(pt), i\rangle$.
(Con) Positions of the same rank are strongly connected via threads and backpointers of that rank.
(Dec) Ranks decrease along threads, except when passing through $*$-l steps of higher ranks: if $\langle v, i\rangle$ is the parent of $\langle w, j\rangle$, then either we have $rk\langle v, i\rangle \leq rk\langle w, j\rangle$, or v is a $*$-l address and $rk\langle w, j\rangle, rk\langle v, i\rangle < rk(v)$.
(Thd) Backpointers preserve threads of higher ranks: for all $pt \in Pts$, for all i such that $rk\langle tgt(pt), i\rangle > rk\langle tgt(pt), 0\rangle$, there is a thread from $\langle tgt(pt), i\rangle$ to $\langle src(pt), i\rangle$.
(Blk) If $u \sqsubset v \sqsubset w$ are $*$-l addresses with $rk(u) = rk(w)$, then $rk(v) \leq rk(u)$.
(Org) A $*$-l address v is an *origin* of rank r if v is a minimal $*$-l address with $rk(v) = r$. We require that if $u \sqsubset v$ are origin addresses then $rk(u) > rk(v)$.

These conditions are meant to enforce some inductive structure on the proof. They are such that in a ranked proof, cycles in computations can be considered as nested "`for`" loops, where higher ranks correspond to outer loops. We briefly give some explanations on how to interpret these rules in light of this intuition. Rules (BP) and (Con) ensure the local coherence of ranks with respect to the structure of the proof. Rule (Dec) expresses that lower ranks correspond to innermost loops, by restricting how the computation can transition from a rank to another. Rule (Thd) and (Blk) express that computations in inner loops do not interfer with outer loops, it simply put them on pause. Finally, rule (Org) stipulates that outer loops start before inner ones in the computation.

Let us now investigate the formal consequences of these rules. By (BP) a ranked proof uses only finitely many ranks. Rule (Blk) implies that the threads enforced by condition (Thd) are actually spectator from $\langle tgt(pt), i\rangle$ to $\langle src(pt), i\rangle$. Together with (Dec), this means that threads along a backpointer with rank r behave like in the picture below:



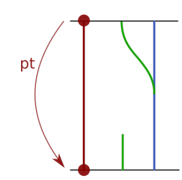
Note that the conditions on ranks imply validity, see Appendix C.2.

for the generalised $*$-l step used at address 01 (on k in Figure 5) is graphically 2, so that the constraint for the shifted backpointer (b) becomes $f_b(2) = 2$ with this graphical intuition.

Proposition 4.4. *Every affine and regular proof π can be extended into a ranked proof $\langle \pi, Pts, rk \rangle$.*

Proof. We describe a recursive algorithm that builds a set of backpointers and assigns ranks to all canonical positions. Intuitively, we start from an ibp-proof, we consider the graph of its canonical addresses where sources and targets of backpointers are identified, and we proceed with its strongly connected components (SCCs) separately. In each SCC, we identify a *master thread*: a thread that validates a branch visiting each node of the SCC infinitely many times (i.e., going through all corresponding backpointers in the starting ibp-proof). Since we are in the affine setting, this master thread identifies exactly one position in each sequent of the SCC. We reserve a maximal rank for these positions and we rearrange backpointers of the starting ibp-proof to satisfy structural constraints related to rules (Thd) and (Blk). We update the graph accordingly, remove the edges corresponding to principal steps of the master thread, and proceed recursively with its new SCCs to assign ranks to the remaining positions. When combining the ranks assigned on each SCC, we shift them to avoid conflicts (Con) and satisfy rules (Dec), (Org), and (Blk): SCCs with smaller addresses get higher ranks. We give more details in Appendix C.3. □

To see why this construction fails in the presence of contraction, consider the ibp-proof for Ackermann-Péter's function given in Figure 7. It contains the pattern depicted on the right, where the green thread is not preserved by the red backpointer, but is somehow "saved" via an auxiliary blue thread. When considering an infinite branch visiting infinitely many times the three backpointers (a, a', b) from Figure 7, we obtain a validating thread that alternates between blue and green positions (see Example 2.16). We should assign a maximal rank to all these positions, but then condition (Thd) is violated for the red backpointer, no matter how we try to shift it away.



4.4 Affine Translation

We can finally translate ranked proofs into system T terms.

Given a list of expressions $E = e_1, \ldots, e_n$ and a list of variables $X = x_1, \ldots, x_n$, we write $X : E$ for the typing environment $x_1 : e_1, \ldots, x_n : e_n$. We also write $E \to f$ for the type $e_1 \to \ldots \to e_n \to f$.

Theorem 4.5. *For every regular and affine proof $\pi : E \vdash e$ and every list X of variables of size $|E|$ there exists an affine term M such that $X : E \vdash M : e$ and $[\pi] = [M]$.*

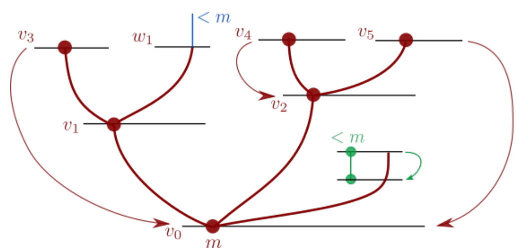
Proof. By Proposition 4.4, it suffices to prove the property for ranked proofs. We do so by lexicographic induction on the *rank* of the proof followed by its *size*, where the rank of a ranked proof is its highest assigned rank and the size of a bp-proof is its number of canonical addresses.

If the rule applied at the root of the proof is not a $*$-l rule, there are no backpointers pointing to the root, so that the subproofs rooted at its premisses are standalone and ranked proofs of strictly smaller size and at most same rank. We translate those by induction, and we combine the results to obtain the desired term. For instance, in the case of a *cut*, we obtain two terms M and N and we construct the term $(\lambda x.M)N$. Those cases are listed in Appendix C.4.

Otherwise, the root must be of the form $e^*, E_0 \vdash e_0$, and its rank m must be maximal by condition (Org). This is where we have to produce recursive terms. We explore the ancestry tree of e^* as long as its rank is m and we find:

- canonical $*$-l addresses $v_0, \ldots, v_n, \ldots, v_{n'}$ of rank m, labelled with sequents $(e^*, E_i \vdash e_i)_{i \in [0;n]}$ (with $v_0 = \epsilon$), such that $v_0, \ldots, v_n$ are not sources and $v_{n+1}, \ldots, v_{n'}$ are sources (pointing to the former ones);
- canonical addresses $w_1, \ldots, w_p$ labelled with sequents $(F_j, e^*, F'_j \vdash f_j)_{j \in [1;p]}$ such that $\langle w_j, |F_j| \rangle$ has rank $< m$.

The situation is illustrated in the following picture:



We construct a term that defines simultaneously all functions $([v_i])_{i\in[0;n]}$, by an encoding of mutual recursion. The addresses w_j correspond to points where we escape from this recursion, e.g., to enter a recursion on another argument.

Let $g \triangleq e^* \cap \bigcap_{i\in[0;n]}(E_i \to e_i)$. This type g is the 'invariant' of our recursion: it contains room for all the mutually defined functions and for a copy of the starting recursive argument.

Given a list x, X of variables for the sequence e^*, E_0, we construct a term M of the form

$$M \triangleq (\mathbf{p}_0\mathbf{p}_1\mathbf{R}(x; M^\epsilon; y.k.M^{::}))\ X_1\ \ldots\ X_l$$

with $\vdash M^\epsilon : g$ and $y : e, k : g \vdash M^{::} : g$, so that we have $x : e^*, X : E_0 \vdash M : e_0$ as expected.

This term iterates the function $\lambda yk.M^{::}$ over the list x, starting from M^ϵ, to obtain a value of type g; then it calls the first mutually defined function in that value.

Defining M^ϵ is easy. For all $i \in [0;n]$, the subproof rooted at v_i0, i.e., the left premiss of the $*$-l node at v_i, is a standalone ranked proof of $E_i \vdash e_i$, with strictly smaller rank and size. Indeed, by (Blk), backpointers whose source belongs to this subproof may not point below it. We can thus translate these subproofs by induction and obtain terms $M_i^\epsilon \vdash E_i \to e_i$ for all $i \le n$. We combine them as follows:

$$M^\epsilon \triangleq \langle\!\langle [], \langle\!\langle M_0^\epsilon, \ldots, M_n^\epsilon \rangle\!\rangle \rangle\!\rangle$$

Defining $M^{::}$ is more involved. Our goal here is to obtain for all $i \le n$ a term $M_i^{::}$ of type $E_i \to e_i$ in environment $y : e, k : g$. Then we will combine those terms as follows:

$$M^{::} \triangleq \langle\!\langle y :: \mathbf{p}_0 k, \langle\!\langle M_0^{::}, \ldots, M_n^{::} \rangle\!\rangle \rangle\!\rangle$$

As expected, we use the subproof rooted at v_i1 to define $M_i^{::}$. However, this subproof ends with $e, e^*, E_i \vdash e_i$, and is not standalone: backpointers along e^* may escape this subproof. To obtain a ranked proof of $e, g, E_i \vdash e_i$, we copy this subproof bottom up, substituting ancestors of e^* by g as long as their rank is m. Several situations appear when doing so:

- we reach a $*$-l node for which e^* is principal: an address v_{k_0} with $k_0 \le n'$. If $k_0 \le n$ we set $k \triangleq k_0$, otherwise v_{k_0} is the source of a backpointer to v_{k_1} for some $k_1 \le n$ and we set $k \triangleq k_1$. We stop copying and we insert the following finite proof:

$$\cfrac{\cfrac{}{E_k \to e_k, E_i \vdash e_k}\ \to\text{-}l, id}{g, E_k \vdash e_k}\ \cap\text{-}l_0, \cap\text{-}l_1$$

- we reach a node for which e^* is spectator and its rank decreases. This means we reached an address w_j for some $j \in [1;p]$. We insert a $\cap$-l_0 rule to transform the type g in the produced proof back into an e^*, and we copy the remainder of the ranked proof as is, without performing the substitution anymore.

- we reach a backpointer following another star formula. Since m is maximal, the target of this backpointer must be above v_i1 by (Blk). Moreover if e^* still occurs at the source of this backpointer, its thread must have been preserved by (Thd) and remained spectator, so that e^* was uniformly substituted into g. The backpointer can thus be inserted in the copied proof.

The produced object is a ranked proof (with smaller rank); in particular, the ranks of principal positions it contains must have their origins inside it by (Blk), so that condition (Org) is preserved. We can thus obtain $M_i^{::}$ by induction. □

The type g used as invariant for recursions in the above translation is reminiscent of the type r we used to encode primitive recursion (Figure 4). Its first component gives access to a copy of the current value of type e^* in those cases where we exit the recursion before exhausting this value.

It is crucial that g is defined using additive pairs in order to obtain an affine term. Indeed, while M^ϵ is typed in the empty context, the variables y and k must be provided to all components of $M^{::}$. Contraction would thus be required if we had been using multiplicative pairs. Symmetrically, having additive pairs makes it possible to avoid weakenings at the various places where values of type g are used (to perform recursive calls, to get the current value of type e^*, and to eventually call the first mutually defined function).

Remark 4.6. *Let C' be the fragment of C where contraction is allowed, except on star formulas. The above argument still works and gives us a direct and uniform encoding of C' into T: threads in C' behave exactly like in affine C. Moreover, contraction on star formulas is derivable in C' (by an easy adaptation of Lemma 2.14), so that Theorem 3.4 can be refined into an encoding of T into C'. C' and T are thus equally expressive, at all types.*

This correspondence makes C' quite appealing and we could have chosen to take it as the main system. However, C' unnecessarily rules out programs such as the implementation of Ackermann-Péter's function in Figure 5 (which does not require the arrow type, unlike the implementation we obtain in C' via Example 3.3 and Theorem 3.4—it is actually not clear that we can implement this function in C' without using arrow types).

The structure of threads is more subtle in C than in C'—cf. Remark 2.7; we find it intriguing and we would like to advocate its study.

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 [Hirschfeldt 2014; Simpson 2009]. Also, an excellent introduction to the functional interpretations of proofs, including for the theories covered here, is [Avigad and Feferman 1998].

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: $X = Y \triangleq \forall x, (X(x) \Leftrightarrow Y(x))$.

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 $<$, with S representing the successor. *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^0_{k+1} formulas are those of the form $\exists\vec{x}, \varphi$ with $\varphi \in \Pi^0_k$.
- Π^0_{k+1} formulas are those of the form $\forall\vec{x}, \varphi$ with $\varphi \in \Sigma^0_k$.

The formulas of the arithmetical hierarchy are the *arithmetical formulas*, those that do not contain second-order quantifiers. A formula is Δ^0_k (provably in a theory T) if it is equivalent to both a Σ^0_k formula and a Π^0_k formula (resp. provably in T).

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

$$(\varphi(0) \wedge \forall x, (\varphi(x) \Rightarrow \varphi(Sx))) \Rightarrow \forall x, \varphi(x) \qquad \text{(induction)}$$
$$\exists X \forall x, (X(x) \Leftrightarrow \varphi) \qquad \text{(comprehension)}$$

Definition 5.2 (ACA_0, RCA_0).

- ACA_0 extends Q2 by instances of induction and comprehension where φ is arithmetical.
- RCA_0 extends Q2 by instances of induction where $\varphi \in \Sigma^0_1$ and instances of comprehension where φ is provably Δ^0_1.

Note that ACA_0 can equivalently be defined as Q2 extended with arithmetical instances of comprehension and the following single induction axiom:

$$(\forall X, X(0) \wedge \forall x, (X(x) \Rightarrow X(Sx))) \Rightarrow \forall x, X(x) \qquad \text{(induction')}$$

Also note that in the above definition of RCA_0, the available instances of comprehension and the notion of RCA_0 itself are mutually defined. It is equivalent to extending Q2 by Σ^0_1 instances of induction and the following axiom scheme, where φ and ψ vary over Σ^0_1 formulas.

$$\forall x(\varphi \Leftrightarrow \neg\psi) \Rightarrow \exists X \forall x(X(x) \Leftrightarrow \varphi)$$

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^0_1 *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., [Avigad and Feferman 1998] for more details).

A similar characterisation of RCA_0 is known: this theory is conservative over $I\Sigma_1$, the restriction of Peano Arithmetic to Σ_1-induction, which is known to well-define only primitive recursive functions.

This result was originally established by Parsons in his *predicative* functional interpretation [Parsons 1972], though there are also direct proofs, e.g., by cut-elimination (see [Buss 1995]).

Proposition 5.4 (Witnessing for RCA_0). *Suppose* RCA_0 *proves* $\forall \vec{x} \exists y, \varphi(\vec{x}, y)$, *where* φ *is* Σ^0_1 *and contains no set symbols. Then there is a primitive recursive function* f *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 [Das 2020]. However, it is known that for any fixed preproof, RCA_0 can check whether it is valid or not:

Proposition 5.5 ([Das 2020], also implicit in [Kołodziejczyk et al. 2019]). *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.

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 [Das and Pous 2018] 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 [Das and Pous 2018]: 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 here 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.

$$\frac{}{\vdash \langle\rangle : 1} \qquad \frac{}{\vdash [] : e^*} \qquad \frac{\vdash P : e \qquad \vdash Q : e^*}{\vdash P :: Q \ : e^*} \qquad \frac{\vdash P_1 : e_1 \quad \dots \quad \vdash P_n : e_n}{\vdash v(P_1, \dots, P_n) : f} \ \pi(v) = e_1, \dots, e_n \vdash f$$

Every program has at most one typing derivation (relatively to the fixed proof π), 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:

$$[\langle\rangle] \triangleq \langle\rangle \qquad [[]] \triangleq \epsilon \qquad [P{::}Q] \triangleq [P] :: [Q] \qquad [v(P_1, \dots, P_n)] \triangleq [v]([P_1], \dots, [P_n])$$

Note that in the last case, $[v]$ is the semantics of the node v in the proof π (Definition 2.10). This semantics cannot be defined ACA_0 or RCA_0: 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 use a compact presentation of these rules here; see Appendix D.1 for an expanded definition. We use v (resp. w) to range over addresses of left (resp. right) introduction rules, and u to range over other addresses. We moreover assume that the sizes of the vectors match those that arise from the implicit typing derivations.

$$
\begin{array}{rlrl}
id: & u(P) \rightsquigarrow P & x: & u(\vec{P}, Q, R, \vec{S}) \rightsquigarrow u0(\vec{P}, R, Q, \vec{S}) \\
cut: & u(\vec{P}, \vec{Q}) \rightsquigarrow u1(u0(\vec{P}), \vec{Q}) & w: & u(P, \vec{R}) \rightsquigarrow u0(\vec{R}) \\
 & & c: & u(P, \vec{Q}) \rightsquigarrow u0(P, P, \vec{Q})
\end{array}
$$

$$
\begin{array}{rlrl}
1\text{-}l: & v(\langle\rangle, \vec{R}) \rightsquigarrow v0(\vec{R}) & 1\text{-}r: & w() \rightsquigarrow \langle\rangle \\
*\text{-}l: & v([], \vec{R}) \rightsquigarrow v0(\vec{R}) & *\text{-}r_\epsilon: & w() \rightsquigarrow [] \\
*\text{-}l: & v(P{::}Q, \vec{R}) \rightsquigarrow v1(P, Q, \vec{R}) & *\text{-}r_{::}: & w(\vec{P}, \vec{Q}) \rightsquigarrow w0(\vec{P}){::}w1(\vec{Q})
\end{array}
$$

$$
\begin{array}{rl}
\cdot\text{-}l/\cdot\text{-}r: & v(w(\vec{P}, \vec{Q}), \vec{R}) \rightsquigarrow v0(w0(\vec{P}), w1(\vec{Q}), \vec{R}) \\
+\text{-}l/+\text{-}r_i: & v(w(\vec{P}), \vec{R}) \rightsquigarrow vi(w0(\vec{P}), \vec{R}) \\
\rightarrow\text{-}l/\rightarrow\text{-}r: & v(w(\vec{P}), \vec{Q}, \vec{R}) \rightsquigarrow v1(w0(v0(\vec{Q}), \vec{P}), \vec{R}) \\
\cap\text{-}l_i/\cap\text{-}r: & v(w(\vec{P}), \vec{R}) \rightsquigarrow v0(wi(\vec{P}), \vec{R})
\end{array}
$$

As expected, subject reduction holds, so that we only work with well-typed programs in the sequel. Also note that $\rightsquigarrow$ is computable in RCA_0, and so is provably Δ^0_1. 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 address v such that π_v ends with $+\text{-}r_i$, $\cdot\text{-}r$, $\cap\text{-}r$ or $\rightarrow\text{-}r$.*

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 program of type 1^*, such that $[\underline{n}] = n$. By Lemma 6.2, the irreducible programs of type 1^* are all of this shape. This 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, in the General Case

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 every fixed regular proof π, ACA_0 proves that for all P, there exists P' with $P \downarrow_\pi P'$.*

Note that π is fixed, and that the universal quantification on P only ranges over those computations that can be performed within π. Since π is regular, those programs involve only finitely many types, and the statement we prove inside ACA_0 does not imply consistency of Peano arithmetic.

To prove this theorem, we use the following sets R_e of *reducible programs*, defined by induction on e. Those are inspired by reducibility candidates [Girard et al. 1989; Tait 1967].

$$\begin{aligned}
\mathsf{R}_1 &\triangleq \{P \mid P \downarrow_\pi \langle\rangle\} \\
\mathsf{R}_{e^*} &\triangleq \{P \mid P \downarrow_\pi Q_1 :: \cdots :: Q_n, \text{ with } Q_1, \dots, Q_n \in \mathsf{R}_e\} \\
\mathsf{R}_{e\cdot f} &\triangleq \{P \mid P \downarrow_\pi v(\vec{Q}, \vec{R}), \text{ with } v \text{ a } \cdot\text{-}r \text{ step, } v0(\vec{Q}) \in \mathsf{R}_e, \text{ and } v1(\vec{R}) \in \mathsf{R}_f\} \\
\mathsf{R}_{e\cap f} &\triangleq \{P \mid P \downarrow_\pi v(\vec{Q}), \text{ with } v \text{ a } \cap\text{-}r \text{ step, } v0(\vec{Q}) \in \mathsf{R}_e, \text{ and } v1(\vec{Q}) \in \mathsf{R}_f\} \\
\mathsf{R}_{e_0+e_1} &\triangleq \{P \mid P \downarrow_\pi v(\vec{Q}), \text{ with } v \text{ a } +\text{-}r_i \text{ step and } vi(\vec{Q}) \in \mathsf{R}_{e_i}\} \\
\mathsf{R}_{e\to f} &\triangleq \{P \mid P \downarrow_\pi v(\vec{Q}), \text{ with } v \text{ a } \to\text{-}r \text{ step and } \forall Q \in \mathsf{R}_e, v0(Q, \vec{Q}) \in \mathsf{R}_f\}
\end{aligned}$$

(Like earlier in the paper, in the third case, we assume that the lengths of the vectors are consistent with the rule instances used at v.)

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 π 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. If $\vec{P} = P_1, \dots, P_n$ and $E = E_1, \dots, E_n$, we write $\vec{P} \in \mathsf{R}_E$ when $P_i \in \mathsf{R}_{E_i}$ for all i.

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

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 contraposite:

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

(1) for all i, j s.t. $\langle v, i\rangle \lhd \langle w, j\rangle$, we have $|Q_i| = |P_j|$, and
(2) for all i, j s.t. $\langle v, i\rangle \mathrel{\ooalign{$\lhd$\cr\hidewidth$\cdot$\hidewidth}} \langle w, j\rangle$, we have $|Q_i| < |P_j|$.

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

Proof. By case analysis, see Appendix D.2. □

Proof of Proposition 6.5. Suppose by contradiction that for some address $w : E \vdash e$ we have $\vec{P} \in \mathsf{R}_E$ such that $w(\vec{P}) \notin \mathsf{R}_e$. By using Lemma 6.6 repeatedly, we can construct an infinite branch of π starting at w. We conclude like in Lemma 2.9. □

This concludes the ACA_0 proof of Theorem 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 (for unary functions). By Proposition 5.5 and Theorem 6.4 we obtain a proof in ACA_0 of "$\forall n, \exists m, \pi(\underline{n}) \downarrow_\pi \underline{m}$". By Proposition 5.3, we extract a system T term M such that for all n, $\pi(\underline{n}) \rightsquigarrow^* \underline{[M](n)}$. By Proposition 6.3, we deduce for all n, $[\pi](n) = [\pi(\underline{n})] = [\underline{[M](n)}] = [M](n)$. □

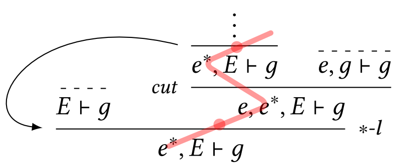
6.2 Weak Normalisation in RCA_0, in the Affine Case

Given Proposition 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^0_1 (due to the quantifier alternation in the definition of $\mathsf{R}_{e\to f}$). 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^0_1-induction.

A different termination proof, inspired from [Das and Pous 2018], can be given in the affine case, using weak König's lemma (see Appendix D.4). This lemma is not available in RCA_0, unfortunately.

Instead, we use a third termination argument, relying on the translation from Section 4.

Definition 6.8. A *simple proof* is an ibp-proof such that for every backpointer pt, $src(pt) = tgt(pt)10$ and the rule used at $tgt(pt)1$ is a *cut*, as illustrated on the left below.



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

In other words, a simple proof is a well-founded proof using the derivable rule on the right.

Our translation from T to C (Theorem 3.4) actually produces simple proofs, so that by Theorem 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. This assumption makes it possible to optimise the notion of reduction: we write $\dot{\rightsquigarrow}$ for the relation defined like in Definition 6.1, except that when v is the target of a backpointer, we use the following rule instead of the two $*$-l reduction rules:

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

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

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

Due to the backpointer from $v01$ to v, we have $\pi_{v(01)^n} = \pi_v$, so that the semantics is preserved. The main advantage of $\dot{\rightsquigarrow}$ is that when $P \dot{\rightsquigarrow} P'$, 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 $\dot{\rightsquigarrow}$ where programs only contain canonical addresses.*

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

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 $P \dot{\rightsquigarrow} P'$ 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.) □

At the meta-level, these two lemmas suffice to conclude that every leftmost innermost reduction sequence along $\rightsquigarrow$ terminates: 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 [Simpson 2009, Theorem IX.5.4]. Instead, we use the folklore fact 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. This is part of [Simpson 2009, Theorem IX.5.4], where the corresponding proof is mentionned as straightforward. We give an explicit proof in Appendix D.3. □

That we restrict to the multiset order on a finite and 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 every fixed 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 Lemma 6.9 and Lemma 6.10, we find an infinite decreasing sequence of multisets over $[0;k]$ where k is the maximal length of canonical addresses in π, contradicting Proposition 6.11. □

Corollary 6.13. *If $x_1{:}1^* \dots x_n{:}1^* \vdash M : 1^*$ is an affine term of T, then $[M]$ is primitive recursive.*

Proof. Translate M into a simple affine proof using Theorem 3.4. Then proceed like for Corollary 6.7, using Theorem 6.12 and Proposition 5.4 instead of Theorem 6.4 and Proposition 5.3. Note that Proposition 5.5 is not required here since simple proofs do not need any validity criterion. □

This corollary generalises Dal Lago's upper bound for $\mathcal{H}(\emptyset)$ [Dal Lago 2009]: our proof handles additive pairs, which we do not know how to handle using Dal Lago's method. Also note that the cyclic proof machinery is not required to obtain this corollary: we use the easy translation from T into C (Theorem 4.5) in order to obtain a small steps semantics which is convenient to work with, but this translation only produces simple proofs, which can be presented inductively, as finite trees.

Instead, the following corollary about affine C requires the machinery from Section 4 to delineate the cycle structure of affine proofs. We do not know of a more direct approach so far—see Appendix D.4 for a failed attempt.

Corollary 6.14. *If $\pi : 1^* \dots 1^* \vdash 1^*$ is an affine regular proof, then $[\pi]$ is primitive recursive.*

Proof. Translate π into an affine term using Theorem 4.5 and conclude with Corollary 6.13. □

We also recover as a corollary the following known fact [Simpson 2009, Theorem IX.5.4]:

Corollary 6.15. 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 Theorem 6.12, from which we could extract a 'universal primitive recursive function' whose complexity would bound the complexity of all primitive recursive functions (via Theorem 2.15). □

7 CONCLUSIONS AND FUTURE WORK

We proposed the cyclic sequent proof system C, which we equipped with both a denotational semantics (Definition 2.11), and a small steps operational semantics (Definition 6.1). Under this interpretation, regular proofs of system C can be seen as unstructured goto programs, whose termination is guaranteed by a (decidable) validity criterion.

We studied the expressive power of system C as a programming language, by comparing it with an appropriate version of Gödel's system T—a structured programming language. Encoding cyclic programs into recursive ones is nontrivial, but we managed to give a direct encoding from C to T in the affine case. To obtain upper bounds on 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 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 [Doumane et al. 2016; Fortier and Santocanale 2013]; whether they correspond to appropriate extensions of T is left for future work.

Our current translation of C with contraction into T works for natural number functions, but it does not scale directly to higher types. Indeed, the technique we use (usual reducibility and hereditary recursivity arguments to obtain a proof of totality in ACA_0) is restricted to computations returning finite values. It would thus be interesting to attain a 'direct' translation in the style of the one we obtained for the affine case in Section 4. As explained in Remark 4.6, higher types do not seem to be problematic *per se*, but we need a better understanding of the structure of threads with contractions on star formulas.

The type levels of recursors in T programs are closely related to the logical complexity of induction in Peano Arithmetic (in the sense of Definition 5.1). At this level of granularity, it was observed recently in [Das 2020] 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). This would be consistent with the fact that we do have an implementation of Ackermann-Péter's function in C at level 0 (Figure 5), but that remains a topic for future work.

ACKNOWLEDGMENTS

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.

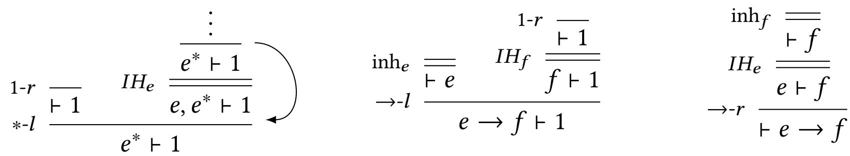


Fig. 8. Erasing star and arrow formulas, inhabiting arrow formulas.

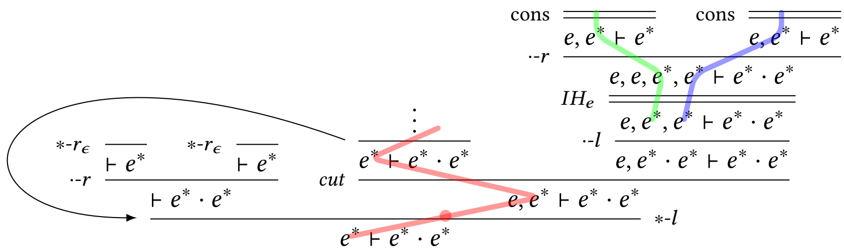


Fig. 9. Duplicating a star formula.

A ADDITIONAL DETAILS FOR SECTION 2

A.1 Weakening and Contraction

Proof of Lemma 2.13. 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 Figure 8. The rule marked IH_e is the weakening rule derived for e by induction hypothesis and the widget on the left in Figure 3. The second interesting case is the weakening of an arrow formula $e \to f$ depicted on the right of Figure 8: we use the fact that e is inhabited, by induction. The third interesting case is for inhabitation of arrow types, where use the fact that e can be erased, by induction. □

Proof of Lemma 2.14. We proceed by induction on e; the interesting case is the duplication of a star formula e^*, which is depicted in Figure 9. The subproofs labelled with 'cons' consist of an application of the $*\text{-}r_{::}$ rule followed by two identity axioms. The rule marked IH_e at address 110 is the contraction rule derived for e by induction hypothesis and the widget on the right in Figure 3. □

A.2 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 $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, 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 $(\mathbb{N} \times \mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}$. The preproof π_μ is represented in Figure 10. In this figure, x, y stand for 1^*, f stands for $1^* \cdot 1^* \to 1^*$, and k stands for 1^*: it stores the result $f(y, x)$. We note $k^{\triangleleft}$ the predecessor of k and y' the successor of y. Principal formulas may be emphasised by a red font.

$$
\dfrac{
  \dfrac{}{\vdash 1^*}\,{\scriptstyle 1^*\text{-}r_0}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{}{y, x \vdash y \cdot x}\,{\scriptstyle \cdot\text{-}r}
      \qquad
      \dfrac{
        \dfrac{\dfrac{}{y \vdash 1^*}\,{\scriptstyle id}}{y, f, x \vdash 1^*}\,{\scriptstyle w}
        \qquad
        \dfrac{
          \dfrac{
            \dfrac{}{y \vdash y'}\,{\scriptstyle S}
            \qquad
            \dfrac{\vdots}{y', f, x \vdash 1^*}
          }{y, f, x \vdash 1^*}\,{\scriptstyle cut}
        }{k^{\triangleleft}, y, f, x \vdash 1^*}\,{\scriptstyle w}
      }{k, y, f, x \vdash 1^*}\,{\scriptstyle 1^*\text{-}l}
    }{y, y, f, f, x, x \vdash 1^*}\,{\scriptstyle \rightarrow\text{-}l}
  }{y, f, x \vdash 1^*}\,{\scriptstyle c, c, c}
}{f, x \vdash 1^*}\,{\scriptstyle cut}
$$

Fig. 10. The preproof π_μ for minimisation.

The principle behind this preproof is simply to compute $k = f(y, x)$ for $y = 0, 1, 2, \ldots$, 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 Section 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 Lemma 6.2, otherwise the function is undefined at the considered point.

B ADDITIONAL DETAILS FOR SECTION 3

B.1 Equivalence with Classical System T

That our version of system T (Section 3) can be encoded into the classical one is clear: our rules for the various connectives are more restrictive, and weakening and contraction are admissible in standard system T.

Conversely, since we include explicit contraction and weakening typing rules, we can show that the typing rules of classical system T are derivable. This is straightforward for most rules. For instance, the standard typing rule for application is the following on the left, and it can be derived as on the right:

$$
\dfrac{\Gamma \vdash MN : f \qquad \Gamma \vdash M : e \to f}{\Gamma \vdash MN : f}\,{\scriptstyle \to\text{-}e'}
\qquad\qquad
\dfrac{\dfrac{\Gamma \vdash MN : f \qquad \Gamma \vdash M : e \to f}{\Gamma, \Gamma \vdash MN : f}\,{\scriptstyle \to\text{-}e}}{\Gamma \vdash MN : f}\,{\scriptstyle c, x}
$$

Similarly for the axiom:

$$
\dfrac{}{\Gamma, x : e \vdash x : e}\,{\scriptstyle id'}
\qquad\qquad
\dfrac{\dfrac{}{x : e \vdash x : e}\,{\scriptstyle id}}{\Gamma, x : e \vdash x : e}\,{\scriptstyle w}
$$

The only rule that requires more work is that for recursion. Indeed, in addition to imposing linearity constraints on w.r.t. the usage of resources, we use an iterator rather than a recursor. Accordingly, 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 $g' \triangleq E \to (e^* \cdot g)$. We use our affine recursor scheme $\mathbf{R}$ with arguments L (unchanged), $M' \triangleq \lambda\vec{u}.\langle[], M\rangle$ and $x.y'.N'$ where

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

Notice that provided $\Gamma \vdash M : g$ and $\Gamma, x : e, q : e^*, y : g \vdash N : g$, we have $\Gamma \vdash M' : g'$ and $x : e, y' : g' \vdash N' : g'$ as required by rule $*$-e for $\mathbf{R}$. Typing derivations showing this are omitted—note that we need contractions on $\vec{u}$ in the second derivation, since those variables are used both in $y'\ \vec{u}$ and in N.

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

$$\cdot\text{-}e\ \dfrac{c,x\ \dfrac{\to\text{-}e\ \dfrac{*\text{-}e\ \dfrac{\Gamma \vdash L : e^* \quad \Gamma \vdash M' : g' \quad x : e, y' : g' \vdash N' : g'}{\Gamma, \Gamma \vdash \mathbf{R}(L, M', x.y'.N') : g'} \quad id\ \dfrac{}{\Gamma \vdash \vec{u} : E}}{\Gamma, \Gamma, \Gamma \vdash \mathbf{R}(L, M', x.y'.N')\ \vec{u} : e^* \cdot g}}{\Gamma \vdash \mathbf{R}(L, M', x.y'.N')\ \vec{u} : e^* \cdot g} \quad w\ \dfrac{id\ \dfrac{}{r : g \vdash r : g}}{l : e^*, r : g \vdash r : g}}{\Gamma \vdash \mathbf{R}_T(L, M, x.q.y.N) : g}$$

B.2 Complete List for Definition 3.1

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

$$\begin{aligned} id &: [x](s) \triangleq s \\ x &: [M](s, u, v, t) \triangleq [M](s, v, u, t) \\ w &: [M](v, s) \triangleq [M](s) \\ c &: [M](v, s) \triangleq [M](v, v, s) \end{aligned}$$

- $\cdot$-i : $[\langle M, N\rangle](s, t) \triangleq \langle [M](s), [N](t)\rangle$
- $+$-i_j : $[\mathbf{i}_j M](s) \triangleq [M](s)$
- $*$-i_ϵ : $[[]]() \triangleq \epsilon$.
- $*$-$i_{::}$: $[M :: N](s, t) \triangleq [M](s) :: [N](t)$
- 1-i : $[\langle\rangle]() \triangleq 1$
- $\to$-i : $[\lambda x.M](s) \triangleq (u \mapsto [M](u, s))$
- $\cap$-i : $[\langle\!\langle M, N\rangle\!\rangle](s) \triangleq \langle [M](s), [N](s)\rangle$
- $\cdot$-e : $[\text{let } \langle x, y\rangle := M \text{ in } N](s, t) \triangleq [N](u, v)$ where the induction provided $[M](s) = \langle u, v\rangle$.
- $+$-e : $[\mathbf{D}(S; x.M; y.N)](s, t) \triangleq [M]([S](s), t)$ if $[S](s) \in [e]$ and $[N]([S](s), t)$ otherwise.
- $*$-e : $[\mathbf{R}(L; M; x.y.N)](s, t) \triangleq h(x_1, h(x_2, ..h(x_n, a)..))$ where induction provided an element $a \triangleq [M](t)$, a list $[L](s) = x_1, \ldots, x_n$, and a function $h \triangleq [N]$.
- 1-e : $[\text{let } \langle\rangle := M \text{ in } N](s, t) \triangleq [N](t)$
- $\to$-e : $[MN](s, t) \triangleq [M](s)([N](t))$
- $\cap$-e_i : $[\mathbf{p}_i M](s) \triangleq [M](s)$

C PROOFS AND DETAILS FOR SECTION 4

C.1 Proof of Proposition 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 $src(pt)$, $tgt(pt)$, and the branch from the root of the proof to $src(pt)$ is verified".

We say that a backpointer pt is *correct* when it verifies the first item from Definition 4.1, i.e., the subtrees rooted in $src(pt)$ and $tgt(pt)$ 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 $Pts_1 = \{pt \in Pts_0 \mid \forall pt' \in Pts_0, src(pt') \not\sqsubset src(pt)\}$, 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 $T = \{u \mid \exists pt \in Pts_1, u \sqsubseteq src(pt)\}$. 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 $src(pt) \sqsubseteq src(pt_\rho)$. Let v be an address of ρ with $src(pt) \sqsubseteq v$. Since ρ is contained in T, there must be $pt' \in Pts_1$ with $v \sqsubseteq src(pt')$. We obtain $src(pt) \sqsubset src(pt')$, contradicting the fact that $pt' \in Pts_1$. We can thus conclude that Pts_1 is finite. Let $Pts_2 = \{pt \in Pts_1 \mid \forall pt' \in Pts_1, src(pt) = src(pt') \Rightarrow tgt(pt) \sqsubseteq tgt(pt')\}$, 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 Lemma C.1, in order to show Proposition 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 $eval : \mathcal{F}^* \to \mathcal{F}$ 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. This is a well-known basic consequence of Ramsey's theorem.

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 principal 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 $f = f_i \circ f_{i+1} \circ \cdots \circ f_j$ 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 Lemma C.1, we can conclude that every regular proof can be extended into an ibp-proof.

We now state a strengthening of Lemma C.2.

Lemma C.3. *Let π be a regular proof, and $\langle u, 0\rangle$ be a principal 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 principal position that is an ancestor of $\langle u, 0\rangle$.*

Proof. This is an adaptation of the proof of Lemma 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 Lemma 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 Lemma C.2 to define a correct idempotent backpointer in this suffix. □

C.2 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. □

C.3 Proof of Proposition 4.4

We describe here with more details the recursive procedure we use to extend an affine and regular proof π into a ranked proof $\langle \pi, Pts, rk\rangle$.

(1) Use Proposition 4.3 to obtain Pts_0 such that $\pi^0_{bp} = \langle \pi, Pts_0\rangle$ is an ibp-proof, and consider the *canonical graph* G_0 consisting of the restriction of π to canonical addresses, and where sources and targets of backpointers from Pts_0 are identified.

(2) We consider each strongly connected component (SCC) of G_0 separately, to assign ranks in the corresponding parts of π. 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 proceed as follows to assign ranks within a SCC of the canonical graph. By strong connectedness, we build an infinite path visiting all nodes of this SCC infinitely many times. This corresponds to an infinite branch in π, which must be validated by a *master thread*: a thread going through all backpointers infinitely many times. All positions of this master thread—exactly one per sequent of the SCC since we are in the affine case—are assigned with a maximal rank M. (This rank M is actually a placeholder standing for "maximal rank in the current SCC": it has to be shifted to an appropriate value after the subsequent recursive calls are completed.)

(4) We now need to reorganise backpointers of the ibp-proof in order to respect rule (Thd) and (Blk), 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). We do so using Lemma C.1 and Lemma C.3, taking the origin of rank M as distinguished position $\langle u, 0\rangle$. The latter lemma gives us 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 spectactor between the source and the target of the backpointer.

(5) At this point, we are done with the positions of rank M. Since backpointers have been updated, we need to recompute the canonical graph. We remove from it all $*$-l steps of rank M, thus creating new SCCs, and we proceed recursively from step 2.

This process terminates, because the maximal number of positions with unassigned rank in a sequent of the considered SCC decreases at each step. Indeed, the master thread visits every sequent of the considered SCC, so that we assign the rank M to a star position in each of these sequents.

The algorithm generates a set of backpointers *Pts* and a rank function *rk* such that $\langle \pi, \mathit{Pts}, \mathit{rk}\rangle$ 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 a maximal rank for the master thread, 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.

C.4 Typing Derivations for the Affine C to Affine T Translation

We give the typing derivations needed in the simple cases for the translation from affine C to affine T (Theorem 4.5) in Figure 11 and 12. The cases for right introduction rules are given in Figure 11; the ones for left introduction rules and cut are given in Figure 12.

$$1\text{-}i\ \frac{}{\vdash \langle\rangle : 1} \qquad *\text{-}i_{\epsilon}\ \frac{}{\vdash [] : e^*} \qquad *\text{-}i_{::}\ \frac{X : E \vdash M : e \quad Y : F \vdash N : e^*}{X : E, Y : F \vdash M :: N : e^*}$$

$$\cdot\text{-}i\ \frac{X : E \vdash M : e \quad Y : F \vdash N : f}{X : E, Y : F \vdash \langle M, N\rangle : e \cdot f} \qquad +\text{-}i_i\ \frac{X : E \vdash M : e_i}{X : E \vdash \mathbf{i}_i M : e_0 + e_1}\ i \in \{0,1\}$$

$$\to\text{-}i\ \frac{x : e, X : E \vdash M : f}{X : E \vdash \lambda x.M : e \to f} \qquad \cap\text{-}i\ \frac{X : E \vdash M : e \quad X : E \vdash N : f}{X : E \vdash \langle\!\langle M, N\rangle\!\rangle : e \cap f}$$

Fig. 11. Typing derivations for translating right rules of C into T. Green sequents represent the results obtained through the induction.

$$1\text{-}e\ \dfrac{id\ \dfrac{}{x : 1 \vdash x : 1} \quad X : E \vdash M : e}{x : 1, X : E \vdash \text{let } \langle\rangle := x \text{ in } M : e}$$

$$\cdot\text{-}e\ \dfrac{id\ \dfrac{}{z : e \cdot f \vdash z : e \cdot f} \quad x : e, y : f, X : E \vdash N : g}{z : e \cdot f, X : E \vdash \text{let } \langle x, y\rangle := z \text{ in } N : g}$$

$$+\text{-}e\ \dfrac{id\ \dfrac{}{z : e + f \vdash z : e + f} \quad x : e, X : E \vdash M : g \quad y : f, X : E \vdash N : g}{z : e + f, X : E \vdash \mathbf{D}(z; x.M; y.N) : g}$$

$$\to\text{-}e\ \dfrac{\to\text{-}i\ \dfrac{x : f, Y : F \vdash M : g}{Y : F \vdash \lambda x.M : f \to g} \quad \to\text{-}e\ \dfrac{id\ \dfrac{}{y : e \to f \vdash y : e \to f} \quad X : E \vdash N : e}{y : e \to f, X : E \vdash yN : f}}{y : e \to f, X : E, Y : F \vdash (\lambda x.M)(yN) : g}$$

$$\to\text{-}e\ \dfrac{\cap\text{-}e_i\ \dfrac{id\ \dfrac{}{z : e_0 \cap e_1 \vdash z : e_0 \cap e_1}}{z : e_0 \cap e_1 \vdash \mathbf{p}_i z : e_i} \quad \to\text{-}i\ \dfrac{x : e_i, X : E \vdash M : f}{X : E \vdash \lambda x.M : e_i \to f}}{z : e_0 \cap e_1, X : E \vdash (\lambda x.M)(\mathbf{p}_i z) : f}$$

$$\to\text{-}e\ \dfrac{X : E \vdash N : e \quad \to\text{-}i\ \dfrac{x : e, Y : F \vdash M : g}{Y : F \vdash \lambda x.M : e \to g}}{X : E, Y : F \vdash (\lambda x.M)N : g}$$

Fig. 12. Typing derivations for translating left rules of C into T. Green sequents represent the results obtained through the induction. The last derivation is the one for the cut rule, as described in Section 4.4. The derivations used for weakening and identity are trivial and omitted here.

D ADDITIONAL DETAILS FOR SECTION 6

D.1 Reduction

We give a more explicit definition of the reduction relation (Definition 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 that the following rules are satisfied. We assume that the length of the vectors match the length of the corresponding lists of formulas.

structural reductions:

- If π_u is ${\scriptstyle id}\dfrac{}{e \vdash e}$ then $u(P) \rightsquigarrow P$.
- If π_u ends ${\scriptstyle x}\dfrac{E, f, e, F \vdash g}{E, e, f, F \vdash g}$ then $u(\vec{P}, P, Q, \vec{Q}) \rightsquigarrow u0(\vec{P}, Q, P, \vec{Q})$.
- If π_u ends ${\scriptstyle w}\dfrac{E \vdash g}{e, E \vdash g}$ then $u(P, \vec{P}) \rightsquigarrow u0(\vec{P})$.
- If π_u ends ${\scriptstyle c}\dfrac{e, E \vdash g}{e, e, E \vdash g}$ then $u(P, \vec{P}) \rightsquigarrow u0(P, P, \vec{P})$.
- If π_u ends ${\scriptstyle cut}\dfrac{E \vdash e \quad e, F \vdash f}{E, F \vdash f}$ then $u(\vec{P}, \vec{Q}) \rightsquigarrow u1(u0(\vec{P}), \vec{Q})$.

constructor reductions:

- If π_w ends ${\scriptstyle 1\text{-}r}\dfrac{}{\vdash 1}$ then $w() \rightsquigarrow \langle\rangle$.
- If π_w ends ${\scriptstyle *\text{-}r_\epsilon}\dfrac{}{\vdash e^*}$ then $w() \rightsquigarrow []$.
- If π_w ends ${\scriptstyle *\text{-}r_{::}}\dfrac{E \vdash e \quad F \vdash e^*}{E, F \vdash e^*}$ then $w(\vec{P}, \vec{Q}) \rightsquigarrow w0(\vec{P}) :: w1(\vec{Q})$.

left/constructor reductions:

- If π_v ends ${\scriptstyle 1\text{-}l}\dfrac{E \vdash g}{1, E \vdash g}$ then $v(\langle\rangle, \vec{P}) \rightsquigarrow v0(\vec{P})$.
- If π_v ends ${\scriptstyle *\text{-}l}\dfrac{E \vdash g \quad e, e^*, E \vdash g}{e^*, E \vdash g}$ then $v([], \vec{P}) \rightsquigarrow v0(\vec{P})$ and $v(P :: Q, \vec{P}) \rightsquigarrow v1(P, Q, \vec{P})$.

left/right reductions:

- If π_w ends ${\scriptstyle \cdot\text{-}r}\dfrac{E \vdash e \quad F \vdash f}{E, F \vdash e \cdot f}$ and π_v ends ${\scriptstyle \cdot\text{-}l}\dfrac{e, f, G \vdash g}{e \cdot f, G \vdash g}$ then $v(w(\vec{P}, \vec{Q}), \vec{R}) \rightsquigarrow v0(w0(\vec{P}), v1(\vec{Q}), \vec{R})$.
- If π_w ends ${\scriptstyle +\text{-}r_i}\dfrac{E \vdash e_i}{E \vdash e_0 + e_1}$ and π_v ends ${\scriptstyle +\text{-}l}\dfrac{e_0, F \vdash g \quad e_1, F \vdash g}{e_0 + e_1, F \vdash g}$ then $v(w(\vec{P}), \vec{Q}) \rightsquigarrow vi(w0(\vec{P}), \vec{Q})$.
- If π_w ends ${\scriptstyle \cap\text{-}r}\dfrac{E \vdash e_0 \quad E \vdash e_1}{E \vdash e_0 \cap e_1}$ and π_v ends ${\scriptstyle \cap\text{-}l_i}\dfrac{e_i, F \vdash g}{e_0 \cap e_1, F \vdash g}$ then $v(w(\vec{P}), \vec{Q}) \rightsquigarrow v0(wi(\vec{P}), \vec{Q})$.
- If π_w ends ${\scriptstyle \rightarrow\text{-}r}\dfrac{e, E \vdash f}{E \vdash e \rightarrow f}$ and π_v ends ${\scriptstyle \rightarrow\text{-}l}\dfrac{F \vdash e \quad f, G \vdash g}{e \rightarrow f, F, G \vdash g}$ then $v(w(\vec{P}), \vec{Q}, \vec{R}) \rightsquigarrow v1(w0(v0(\vec{Q}), \vec{P}), \vec{R})$.

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 [Das and Pous 2018]; 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 D.1 (Lemma 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, $\cdot$-r, $\cap$-r *or* $\to$-r.

Proof. The characterisation given in the statement is computable, and so we may prove the lemma by Σ^0_1-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. □

D.2 Reducible Programs (ACA_0)

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

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

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

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

(Where given $P \in \mathsf{R}_{e^*}$*, 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.*

cut : π_w ends $\mathit{cut}\,\dfrac{E \vdash e \quad e, F \vdash f}{E, F \vdash f}$. Assume $\vec{P} \in \mathsf{R}^\downarrow_E$, $\vec{Q} \in \mathsf{R}^\downarrow_F$ and $w(\vec{P}, \vec{Q}) \notin \mathsf{R}_f$. There are two cases:
- if $w0(\vec{P}) \notin \mathsf{R}_e$ then we choose $v = w0$, taking $\vec{P}$ as witness.
- if $w0(\vec{P}) \in \mathsf{R}_e$ 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 : π_w ends $c\,\dfrac{e, E \vdash g}{e, e, E \vdash g}$. Assuming $P \in \mathsf{R}^\downarrow_e$, $\vec{P} \in \mathsf{R}^\downarrow_E$, we take $v = w0$ with witness $P, P, \vec{P}$, since

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

$\to$-r : π_w ends $\to\text{-}r\,\dfrac{e, E \vdash f}{E \vdash e \to f}$. Assume $\vec{P} \in \mathsf{R}^\downarrow_E$ and $w(\vec{P}) \notin \mathsf{R}_{e\to f}$. $w(\vec{P})$ is irreducible, so that there must be a $R \in \mathsf{R}_e$ such that $w0(R, \vec{P}) \notin \mathsf{R}_f$. We choose $v = w0$ with $R, \vec{P}$ as witness.

$\to$-l : π_w ends $\to\text{-}l\,\dfrac{E \vdash e \quad f, F \vdash g}{e \to f, E, F \vdash g}$. Assume $P = u[\vec{R}] \in \mathsf{R}^\downarrow_{e\to f}$, $\vec{P} \in \mathsf{R}^\downarrow_E$, $\vec{Q} \in \mathsf{R}^\downarrow_F$ and $w(P, \vec{P}, \vec{Q}) \notin \mathsf{R}_g$.
There are two cases:
- if $w0(\vec{P}) \notin \mathsf{R}_e$, we take $v = w0$ with witness $\vec{P}$.

- if $w0(\vec{P}) \in \mathsf{R}_e$, then $w0(\vec{P}) \downarrow P_0 \in \mathsf{R}_e$. By definition of $\mathsf{R}_{e\to f}$ we obtain $u0(P_0, \vec{R}) \in \mathsf{R}_f$. We choose $v = w1$, taking $u0(P_0, \vec{R}), \vec{Q}$ as witness, since

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

$*$-$r_{::}$: π_w ends $*\text{-}r_{::}\dfrac{E \vdash e \quad F \vdash e^*}{E, F \vdash e^*}$. Assume $\vec{P} \in \mathsf{R}^{\downarrow}_E$, $\vec{Q} \in \mathsf{R}^{\downarrow}_F$, and $w(\vec{P}, \vec{Q}) \notin \mathsf{R}_{e^*}$. If $w0(\vec{P}) \notin \mathsf{R}_e$ we take $v = w0$ with $\vec{P}$ as witness. Otherwise $w0(\vec{P}) \downarrow R_0 \in \mathsf{R}_e$ and we take $v = w1$ with $\vec{Q}$ as witness. Indeed, if we had $w1(\vec{Q}) \in \mathsf{R}_{e^*}$ then we would get $w1(\vec{Q}) \downarrow R_1 :: \cdots :: R_n$ with the R_i in R_e; this would contradict the assumption about w since

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

$*$-l : π_w ends $*\text{-}l\dfrac{E \vdash g \quad e, e^*, E \vdash g}{e^*, E \vdash g}$. Assume $P \in \mathsf{R}^{\downarrow}_{e^*}$, $\vec{P} \in \mathsf{R}^{\downarrow}_E$ and $w(P, \vec{P}) \notin \mathsf{R}_g$. 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) □

D.3 Well-Foundedness of Multiset Ordering over a Finite Order in RCA_0

Proposition D.3 (Proposition 6.11 in the main text, also part of [Simpson 2009, Theorem IX.5.4]). *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, or -1 if m is empty. We prove the following property by (meta-level) induction on n:

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

- 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 $\forall i, \max m_i \leq n$.
  - 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 be constant from some point on: there exists j such that for all k, $m_{j+k}(n) = m_j(n)$. (This can be proved by contradiction and Π^0_1-induction, which is available in RCA_0 [Buss 1998].) Now consider the sequence $m'_i \triangleq m_{j+i}\backslash n$, where $m\backslash n$ 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.

The property entails well-foundedness over multisets on $[0; n - 1]$, so that we are done. □

D.4 Alternative Termination Proof in the Affine Case and Failed Attempt with WKL_0

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 $\mathsf{V}(P)$.

Lemma D.4. *If P is coherent and $P \rightsquigarrow P'$ then P' is coherent and every address in $\mathsf{V}(P')$ is either already in $\mathsf{V}(P)$, or an immediate successor of some address in $\mathsf{V}(P)$.*

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 Lemma 6.2). The *weight* of such a program is the length of this list.

Theorem D.5 (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 Lemma D.4, 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 ("every infinite binary tree has an infinite branch") 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. □

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.

While the above argument works for arbitrary affine proofs (not just simple affine proofs, like in Section 6.2), it exploits weak König's lemma to extract an infinite branch and use the validity criterion. As a consequence, we cannot formalise this argument in RCA_0.

It is tempting at this point to consider the system WKL_0, which is well-known in reverse-mathematics. Indeed, WKL_0 precisely extends RCA_0 with weak König's lemma, and while WKL_0 is strictly stronger than RCA_0, it is conservative over RCA_0 for arithmetical formulas:

Theorem D.6 (Harrington, see [Avigad 1996]). *If WKL_0 proves φ with φ arithmetical, then RCA_0 proves φ.*

Therefore, WKL_0 can also be used to extract primitive recursive functions:

Corollary D.7 (Witnessing for WKL_0). *Suppose WKL_0 proves $\forall\vec{x}\exists y\varphi(\vec{x}, y)$, where φ is Σ^0_1 and contains no set symbols. Then there is a primitive recursive function $f(\vec{x})$ such that $\mathbb{N} \vDash \forall\vec{x}.\varphi(\vec{x}, f(\vec{x}))$.*

Unfortunately, the above termination argument for affine proofs 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.

REFERENCES

Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. 2002. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science* 12, 5 (2002), 625–665. https://doi.org/10.1017/S0960129502003730

Bahareh Afshari and Graham E. Leigh. 2017. Cut-free completeness for modal mu-calculus. In *LiCS*. IEEE, 1–12. https://doi.org/10.1109/LICS.2017.8005088

Jeremy Avigad. 1996. Formalizing forcing arguments in subsystems of second-order arithmetic. *Annals of Pure and Applied Logic* 82, 2 (1996), 165 – 191. https://doi.org/10.1016/0168-0072(96)00003-6

Jeremy Avigad and Solomon Feferman. 1998. Gödel's Functional Interpretation. In *Handbook of Proof Theory*, Samuel R. Buss (Ed.). Elsevier.

Patrick Baillot and Marco Pedicini. 2001. Elementary complexity and geometry of interaction. *Fundamenta Informaticae* 45, 1-2 (2001), 1–31.

Stefano Berardi and Makoto Tatsuta. 2017a. Classical System of Martin-Löf's Inductive Definitions Is Not Equivalent to Cyclic Proof System. In *FoSSaCS*. Springer Verlag, 301–317. https://doi.org/10.1007/978-3-662-54458-7_18

Stefano Berardi and Makoto Tatsuta. 2017b. Equivalence of inductive definitions and cyclic proofs under arithmetic. In *LiCS*. IEEE, 1–12. https://doi.org/10.1109/LICS.2017.8005114

James Brotherston. 2005. Cyclic Proofs for First-Order Logic with Inductive Definitions. In *TABLEAUX (Lecture Notes in Artificial Intelligence, Vol. 3702)*. Springer Verlag, 78–92. https://doi.org/10.1007/11554554_8

James Brotherston and Alex Simpson. 2011. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation* 21, 6 (2011), 1177–1216. https://doi.org/10.1093/logcom/exq052

Samuel R. Buss. 1995. The witness function method and provably recursive functions of Peano arithmetic. In *Studies in Logic and the Foundations of Mathematics*. Vol. 134. Elsevier, 29–68. https://doi.org/10.1016/S0049-237X(06)80038-8

Samuel R Buss. 1998. *Handbook of proof theory*. Vol. 137. Elsevier.

Ugo Dal Lago. 2009. The geometry of linear higher-order recursion. *ACM Trans. Comput. Log.* 10, 2 (2009), 8:1–8:38. https://doi.org/10.1145/1462179.1462180

Anupam Das. 2020. On the logical complexity of cyclic arithmetic. *Logical Methods in Computer Science* 16, 1 (Jan. 2020). https://doi.org/10.23638/LMCS-16(1:1)2020

Anupam Das, Amina Doumane, and Damien Pous. 2018. Left-handed completeness for Kleene algebra, via cyclic proofs. In *LPAR (EPiC Series in Computing, Vol. 57)*. Easychair, 271–289. https://doi.org/10.29007/hzq3

Anupam Das and Damien Pous. 2017. A cut-free cyclic proof system for Kleene algebra. In *TABLEAUX (Lecture Notes in Computer Science, Vol. 10501)*. Springer Verlag, 261–277. https://doi.org/10.1007/978-3-319-66902-1_16

Anupam Das and Damien Pous. 2018. Non-wellfounded proof theory for (Kleene+action)(algebras+lattices). In *CSL (LIPIcs, Vol. 119)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:19. https://doi.org/10.4230/LIPIcs.CSL.2018.19

Amina Doumane. 2017. Constructive completeness for the linear-time μ-calculus. In *LiCS*. IEEE, 1–12. https://doi.org/10.1109/LICS.2017.8005075

Amina Doumane, David Baelde, and Alexis Saurin. 2016. Infinitary proof theory: the multiplicative additive case. In *CSL (LIPIcs, Vol. 62)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 42:1–42:17. https://doi.org/10.4230/LIPIcs.CSL.2016.42

Jérôme Fortier and Luigi Santocanale. 2013. Cuts for circular proofs: semantics and cut-elimination. In *CSL (LIPIcs, Vol. 23)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 248–262. https://doi.org/10.4230/LIPIcs.CSL.2013.248

Jean-Yves Girard. 1995. Geometry of interaction III: accommodating the additives. In *workshop on Advances in linear logic*. Cambridge University Press, 329–389.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types*. Cambridge University Press, USA.

Von Kurt Gödel. 1958. Über eine bisher noch nicht benütze erweiterung des finiten standpunktes. *Dialectica* 12, 3-4 (1958), 280–287. https://doi.org/10.1111/j.1746-8361.1958.tb01464.x

Denis R. Hirschfeldt. 2014. *Slicing the truth: On the computable and reverse mathematics of combinatorial principles*. World Scientific.

Naohiko Hoshino, Koko Muroya, and Ichiro Hasuo. 2014. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In *CSL-LiCS*. ACM, 52. https://doi.org/10.1145/2603088.2603124

Leszek Kołodziejczyk, Henryk Michalewski, Pierre Pradic, and Michał Skrzypczak. 2019. The logical strength of Büchi's decidability theorem. *Logical Methods in Computer Science* 15, 2 (May 2019). https://doi.org/10.23638/LMCS-15(2:16)2019

Denis Kuperberg, Laureline Pinault, and Damien Pous. 2019. Cyclic Proofs and Jumping Automata. In *FSTTCS (LIPIcs)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Bombay, India, 45:1–45:14. https://doi.org/10.4230/LIPIcs.FSTTCS.2019.45

Dorel Lucanu, Eugen-Ioan Goriac, Georgiana Caltais, and Grigore Rosu. 2009. CIRC: A Behavioral Verification Tool Based on Circular Coinduction. In *CALCO (Lecture Notes in Computer Science, Vol. 5728)*. Springer Verlag, 433–442. https://doi.org/10.1007/978-3-642-03741-2_30

Dorel Lucanu and Vlad Rusu. 2015. Program equivalence by circular reasoning. *Formal Aspects of Computing* 27, 4 (2015), 701–726. https://doi.org/10.1007/s00165-014-0319-6

Charles Parsons. 1972. On n-quantifier induction. *The Journal of Symbolic Logic* 37, 3 (1972), 466–482.

Alex Simpson. 2017. Cyclic Arithmetic Is Equivalent to Peano Arithmetic. In *FoSSaCS (Lecture Notes in Computer Science, Vol. 10203)*. Springer Verlag, 283–300. https://doi.org/10.1007/978-3-662-54458-7_17

Stephen G. Simpson. 2009. *Subsystems of second order arithmetic*. Vol. 1. Cambridge University Press.

William W. Tait. 1965. Infinitely Long Terms of Transfinite Type. In *Formal Systems and Recursive Functions*, J.N. Crossley and M.A.E. Dummett (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 40. Elsevier, 176 – 185. https://doi.org/10.1016/S0049-237X(08)71689-6

William W. Tait. 1967. Intensional Interpretations of Functionals of Finite Type I. *The Journal of Symbolic Logic* 32, 2 (1967), 198–212. https://doi.org/10.2307/2271658