Ramified Structural Recursion and Corecursion
Extended Abstract

Norman Danner¹ and James S. Royer²

¹ Department of Mathematics and Computer Science, Wesleyan University, Middletown, CT 06459, USA; ndanner@wesleyan.edu
² Department of Electrical Engineering and Computer Science, Syracuse University, Syracuse, NY 13210, USA; jsroyer@syr.edu

Abstract. We investigate feasible computation over a fairly general notion of data and codata. Specifically, we present a direct Bellantoni-Cook-style normal/safe typed programming formalism, $RS_1$, that expresses feasible structural recursions and corecursions over data and codata specified by polynomial functors. (Lists, streams, finite trees, infinite trees, etc. are all directly definable.) A novel aspect of $RS_1$ is that it embraces structure-sharing as in standard functional-programming implementations. As our data representations use sharing, our implementation of structural recursions are memoized to avoid the possibly exponentially-many repeated subcomputations a naïve implementation might perform. We introduce notions of size for representations of data (accounting for sharing) and codata (using ideas from type-2 computational complexity) and establish that type-level $RS_1$-functions have polynomial-bounded runtimes and satisfy a polynomial-time completeness condition. Also, restricting $RS_1$ terms to particular types produces characterizations of some standard complexity classes (e.g., ω-regular languages, linear-space functions) and some less-standard classes (e.g., log-space streams).

1 Introduction

What counts as polynomial-time (much less “feasible”) computation over general forms of data is not a settled matter. The complexity-theoretic literature of higher-type computability is still thin, it is spotty on computation over codata (infinite lists and trees) with some notable exceptions,³ and even in the case of inductively defined data there are there remain issues that are not that well explored (see the end of §2 below). We develop a notion of polynomial-time computation over data and codata using a fairly simple implicit complexity formalism, $RS_1$, that satisfies poly-time soundness and completeness properties. $RS_1$ is constructed in stages. We first introduce $S^-$, a formalism for computing over inductively defined data by classical structural (aka primitive) recursion. $S^-$ has roughly the computational power of Gödel’s primitive recursive functionals.

³ Hartmanis and Stearns’ paper [11] that founded computational complexity largely focuses on the time-complexity of infinite streams as the authors directly adapted Turing’s original machine model [20] which, recall, concerns stream-computation.
To tame this power, we impose a form of Bellantoni and Cook normal/safe ramification on $S$’s structural recursions and obtain $RS^{-1}$, a system that satisfies appropriate poly-time soundness and completeness properties. We next introduce $S$, an extension of $S$ to include codata definitions and classical structural (aka primitive) corecursions. We extend the safe/normal ramification to corecursions and obtain $RS^{1}$ that also satisfies poly-time soundness and completeness properties. The subscript on $RS^{-1}$ and $RS^{1}$ is a reminder that these formalisms focus on type-level 1 computation, even though $RS^{-1}$ and $RS^{1}$ allow higher-type terms. It turns out that by restricting types in $RS^{1}$-terms, one can characterize other complexity classes, e.g., $\omega$-regular languages, log-space streams of characters, linear-space streams of strings, etc. These seem to be related to the two-sorted complexity class characterizations studied by Cook and Nguyen [6].

Related Work. The Pola project of Burrell, Cockett, and Redmond [4,5] has aims similar to ours, but Pola forbids any structure-sharing of safe-data or safe-codata. $RS^{-1}$ and $RS^{1}$, in contrast, embrace structure-sharing and adjust the implementation of structural recursions to accommodate it. As a result $RS^{1}$ and Pola describe different notions of polynomial-time over data and codata. How deep these differences go is an intriguing question. Pola also has a well-developed categorical semantics that, at present, $RS^{1}$ notably lacks. Ramyaa and Leivant [17,18] explore feasible first-order stream programming formalisms. In [17], they use infinite binary trees with string-labels to give a partial proof-theoretic characterization of the type-2 basic feasible functionals (BFF$_2$) of Mehlhorn [16] and Cook and Urquhart [7]. In [18], they give a definition of logspace stream computation and a schema of ramified co-recurrence which parallels Leivant’s ramified recurrence of [14], and characterize logspace streams as those definable using 2-tier co-recurrences. Férré et al. [10] also consider stream computation, but primarily as a technical tool in characterizing BFF$_2$ as the functions computed by a rewrite system over streams that has a second-order polynomial interpretation.

Background. Pointer Machines. We assume that the underlying model of computation is along the lines of Kolmogorov and Uspenskii’s “pointer machines” or Schönhage’s storage modification machines [21].

Types. The simple types over a set of base types $B$ are given by: $Ty^B := B | \text{unit} | Ty^B + Ty^B | Ty^B \times Ty^B | Ty^B \rightarrow Ty^B$, where $\text{unit}$ (which counts as a base type) is the type of the empty product ($\cdot$). Let $\text{level}(\text{a base type}) = 0$, $\text{level}(\sigma + \tau) = \text{level}(\sigma \times \tau) = \text{max}(\text{level}(\sigma), \text{level}(\tau))$, $\text{level}(\sigma \rightarrow \tau) = \text{max}(1 + \text{level}(\sigma), \text{level}(\tau))$, and $Ty^B_\leq \{ \sigma \in Ty^B \mid \text{level}(\sigma) \leq i \}$. We call level-$0$ types ground types. A type judgment $\Gamma \vdash e : \sigma$ asserts that $e$ can be assigned type $\sigma$ under type context $\Gamma$, where a type context is a finite function from variables to types.

Algebraic Notions. Set denotes the category of sets and total functions. Below we are mainly concerned with total functions and lower type-levels, so Set suffices as the setting for the semantics of our programming formalisms. Types are thus interpreted as sets where coproduct ($+$), product ($\times$), and exponentiation ($\rightarrow$) have their standard Set-interpretations. Let $\iota_i : A_i \rightarrow A_1 + A_2$ ($i = 1, 2$) be the canonical coproduct injections and $\pi_i : A_1 \times A_2 \rightarrow A_i$ ($i = 1, 2$) be the
canonical product projections. A polynomial functor is a functor inductively built from identity and functors and coproducts and products, e.g., $F_0 X = \text{unit} + (\text{nat} \times X)$ with $F_0 f = \text{id}_{\text{unit}} + (\text{id}_{\text{nat}} \times f)$, where $\text{nat}$ is the type of natural numbers introduced below in Example 4. The constant-objects in our polynomial functors will always be types. Convention: For $F$, a polynomial function given by $FX = e$, and $\sigma$, a type, read $F\sigma$ as the type $e[X := \sigma]$. E.g., $F_0 \text{nat} = \text{unit} + \text{nat} \times \text{nat}$.

The Base Formalism. This paper’s programming formalism are built atop $L$, a standard, simply-typed, call-by-value lambda calculus. The $L$-types are $Ty^0$. Figs. A.1 and A.2 give $L$’s syntax and typing rules. We use the standard syntactic sugar: (i) let $x_1 = e_1; \ldots; x_m = e_m \text{ in } e_0 \equiv (\lambda x_1, \ldots, x_m \cdot e_0) \ e_1 \ldots \ e_m$ and (ii) let* $x_1 = e_1; \ldots; x_m = e_m \text{ in } e_0 \equiv \text{let } x_1 = e_1 \text{ in } (\ldots \text{ let } x_m = e_m \text{ in } e_0 \ldots )$.

Semantics. The denotational semantics of $L$ is standard. As $\text{unit}$ is the sole base type of $Ty^0$, for each $\sigma \in Ty^0$, $[\sigma]$ is a finite set. $L$’s operational semantics is also fairly standard as specified by the evaluation relation, $\Downarrow$, described in Fig. A.3. Terminology: An evaluation relation relates closures to values. A closure $(\Gamma \vdash e : \tau)\theta$ consists of a term $\Gamma \vdash e : \tau$ and an environment $\theta$ for $\Gamma \vdash e : \tau$. (We write $e\theta$ for $(\Gamma \vdash e : \tau)\theta$ when $e$’s typing is understood.) An environment $\theta$ for $\Gamma \vdash e : \tau$ is a finite map from variables to values with $fv(e) \subseteq \text{dom}(\theta) \subseteq \text{dom}(\Gamma)$ and, for each $x \in \text{dom}(\theta)$, $\theta(x)$ is a type-$\Gamma(x)$ value. A value $\varepsilon \theta$ is a closure in which $z$ (the value term) is either an abstraction or else an internal representation of () or $\iota_v \iota_v$ or $(\nu_1, \nu_2)$, where $\nu_1$ and $\nu_2$ are value terms. By internal representation we mean the “machine” representation of value terms, the details of which are not important for $L$, but vital for the $RS^-$ and $RS$ formalisms below.

2 Structural Recursions

The Classical Case. We extend $L$ to $S^-$, a formalism that computes, roughly, Gödel’s primitive recursive functionals [15] over inductively-defined data types. Later we introduce $RS^+_t$, a ramified, “feasible” version of $S^-$. Fig. 1 gives the revised raw syntax [1], typing rules ($c_t$-I, $d_t$-I, fold$_t$-I) and evaluation rules ($\text{Const}_t$, $\text{Destr}_t$, Fold$_t$) for $S^-$. A declaration, $\text{data } \tau = \mu \tau, \sigma$, introduces a data-type $\tau$. The polynomial functor $F_t \tau = \sigma$ is called $\tau$’s signature functor. The declaration also implicitly introduces: $\tau$’s constructor function $c_t : F_t \tau \rightarrow \tau$, $\tau$’s destructor function $d_t : \tau \rightarrow F_t \tau$, and $\tau$’s creator $\text{fold}_t : (\forall \sigma)(F_t \sigma \rightarrow \sigma) \rightarrow \tau \rightarrow \sigma$. We require that the $\sigma$ in $\text{data } \tau = \mu \tau, \sigma$ be a ground type with constituent base types are drawn from $t$, $\text{unit}$, and previously declared types. Semantically, the data type $\tau$ is the least fixed point of $F_t$: it is a smallest set $X$ isomorphic to $F_t(X)$, where $c_t$ and $d_t$ witness this isomorphism. It is standard that polynomial functors have such least fixed points. In examples we use syntactically-sugared versions of $\text{data } \tau = \mu \tau, \sigma$ of the form: $\text{data } \tau = C_1 \text{ of } \sigma_1 \mid \ldots \mid C_n \text{ of } \sigma_n$, where $F_t(\tau) = \sigma_1 + \sigma_2 + \ldots + \sigma_n$ and, for each $i$, if $\sigma_i = \text{unit}$, then $C_i \equiv c_t \circ i_t^i(\tau)$.

4 Other authors (e.g., [19]) use broader notions of polynomial functor.
4 Danner and Royer: Ramified Structural Recursion and Corecursion

\[ Dcl ::= \text{data } T = \mu T'. T_0 \]  
\[ P ::= \text{declare } Dcl (\cdot; Dcl)^* \text{ in } E \]  

(i)

\[
\begin{align*}
\Gamma \vdash c : F \tau & \quad \text{Const}_c : \frac{e \in \tau}{(c, e) \in \tau} \\
\Gamma \vdash (c, e) : \tau & \quad \text{Destr}_c : \frac{e \in \tau}{(c, e) \theta \downarrow \theta'} \\
\Gamma \vdash e \in \tau & \quad \text{Fold}_e : \frac{\theta \downarrow (\theta', \tau)}{f(X \times (\text{fold}_f x)) \theta \downarrow (\theta', \tau)}
\end{align*}
\]

Fig. 1. Extensions for \( S^\tau \), where \( \tau \) is a data-type with signature functor \( F \).

and if \( \sigma_i \neq \text{unit} \), then \( C_i \equiv c_r \circ i^\tau_i : \sigma_i \rightarrow \tau \) Type-\( \tau \) data can then be identified with the elements of the free algebra over the sugared constructors \( C_1, \ldots, C_n \) and the other constituent data-types’ constructors.

Example 1. The declaration, \( \text{data } \text{nat} = \text{Zero of unit} \mid \text{Succ of nat} \), introduces the type \( \text{nat} \) with signature functor \( F_{\text{nat}} \circ X = \text{unit} \times X \) and sugared constructors \( \text{Zero} : \text{nat} \) and \( \text{Succ} : \text{nat} \rightarrow \text{nat} \). Type-\( \text{nat} \) data thus corresponds to the terms of the free algebra over \( \text{Zero} \) and \( \text{Succ} \), i.e., \( \text{Zero} \), \( \text{Succ(Zero)} \), \( \text{Succ(Succ(Zero))} \), etc.

Example 2. The declaration, \( \text{data } \text{tree} = \text{Leaf of unit} \mid \text{Fork of tree} \times \text{tree} \), introduces the type \( \text{tree} \) with signature functor \( F_{\text{tree}} \circ X = \text{unit} \times X \times X \) and sugared constructors \( \text{Leaf} : \text{tree} \) and \( \text{Fork} : \text{tree} \times \text{tree} \rightarrow \text{tree} \). Type-\( \text{tree} \) data thus corresponds to the terms of the free algebra over \( \text{Leaf} \) and \( \text{Fork} \), i.e., \( \text{Leaf} \), \( \text{Leaf(Leaf, Leaf)} \), \( \text{Fork(Leaf, Leaf, Leaf)} \), etc.

The recursor for type-\( \tau \) data, \( \text{fold}_\tau \), has its operational semantics given by Fig. 1 and satisfies: \( (\text{fold}_\tau g \circ c_r) = g \circ F(\text{fold}_\tau g) \). This last equation expresses structural (aka primitive) recursion over \( \tau \). For example, given an \( f : F_{\text{nat}} \rightarrow \text{nat} \) with \( f(t1()) = \text{Zero} \) and \( f(t2(x, y)) = \text{Succ(max(x, y))} \), then \( \text{fold}_\text{tree} f t \) computes the height of \( \text{tree} t \). As the \( g \) in \( (\text{fold}_\tau g x) \) can be of any positive type level, one can show that \( S^\tau \) computes a version of Gödel’s primitive recursive functionals. To rein in the power of fold-recursions to express just low complexity computations, we apply a standard tool of implicit complexity, ramification. First, however, we need to consider how data is represented and how the size of a representation is measured.

Representation, Size, and Memoization. Representing Data. Our internal representation of data follows standard practice in implementations of functional languages. Each invocation of a constructor function: (i) allocates a fresh cons-cell that stores the values of the invocation’s arguments and (ii) returns, as its

\[ 1 \leq i < n, \text{ define: } i^n_i = t_4 \circ i^{(1-1)}_i \text{ and } i^n_i = i^{(n-1)}_i. \text{ Also, define: } i^1_i = \text{id}. \]

\[ \text{In the rule } \text{Fold}_c, \text{ the use of } F \text{ should be read as shorthand for a } \lambda\text{-term that expresses, in } S^\tau, \text{ the polynomial function } F \text{ (specialized to the appropriate types).} \]
value, a pointer to this new cons-cell. **N.B.** The product and coproduct constructors also create cons-cells. As our formalism is purely functional, it follows that all data is represented by directed acyclic graphs (*dags*) on cons-cells.

**Measuring The Size of Data Representations.** A data-representation’s size is simply the number of *data* cons-cells in the representation. For example, consider:

\[
\text{let } t_0 = \text{Leaf}; \; t_1 = \text{Fork}(t_0, t_0); \ldots; t_n = \text{Fork}(t_{n-1}, t_{n-1}) \text{ in } t_n \tag{2}
\]

The size of \(t_n\)’s representation is \(n + 1\) (one *Leaf*-cell and \(n\) *Fork*-cells). This notion of size depends on the operational semantics. Denotationally, \(t_n\) names a proper tree which is also named by \(t'_n\), a sized-(\(2^n + 1\)) *tree* consisting of \(2^n\) *Leaf*-cells and \((2^n - 1)\) *Fork*-cells.

**Definition 3.** Suppose \(e_0\theta, \ldots, e_k\theta\) are ground-type closures. The apparent size of \{ \(e_0\theta, \ldots, e_k\theta\) \} (written: \(|e_0\ldots, e_k\theta|\) is the number of data cons-cells in the representation the values of \(e_0\theta, \ldots, e_k\theta\). **N.B.** This takes account of sharing. E.g., if \(\theta\) is the environment in force in the body of \(\theta\), then \(|t_0, \ldots, t_n|\theta = n+1\.)

**Memoized Structural Recursions.** Two of our goals for our feasible programming formalisms are: (i) to have the run-time of programs to be polynomial-bounded in the size of the representations they compute over; and (ii) to have our programs to return equivalent results on equivalent inputs (e.g., \(t_n\) and \(t'_n\) as above). These goals would seem to conflict given our conventions on data-representations and sizes. This is resolved via the standard programming trick of memoization. Computing \((\text{fold}_x f x)\) can be treated as a linear programming problem with \(x\)'s data representation as the underlying dag, there is, then, an exact match between the \(\text{fold}\)-recursion’s steps and \(x\)'s cons-cells, moreover, the result of each step is stored for possible reuse later in the recursion. We assume that our structural-recursion implementation uses memoization for just branching data types (e.g., *tree*); for nonbranching data-types (e.g., *nat*) it is not needed.

**The Ramified Case.** \(RS^-\), our ramification of \(S^-\), uses Bellantoni and Cook’s normal/safe distinction that splits data into two sorts: *normal* data that drive recursions and *safe* data over which recursions compute. E.g., in \((\text{fold}_g g x)\) we want \(x\) normal and \(g\);(safe data) \rightarrow (safe data). Typing constraints enforce this distinction, which is roughly the idea behind Bellantoni and Cook’s *BC* function algebra [3 §5] (and Leivant’s formalism from [14]), but not Bellantoni and Cook’s better known *B* function algebra. **Normal types:** The *normal base types* consist of *unit* and the types directly introduced by data-definitions. The *normal ground types* are the closure of the normal base types under + and \(\times\). In data \(\tau = \mu \cdot \sigma\), we require that \(\sigma\) be normal. A declaration \(\text{data } \tau = \mu \cdot \sigma\) introduces \(c_\tau\) and \(d_\tau\) as before, but \(\text{fold}_\tau\) is replaced with \(\text{fold}_\tau^5\) as explained shortly. **Safe types:** By convention, \(\text{data } \tau = \mu \cdot \sigma\) implicitly introduces a parallel type \(\tau^5\). We extend \(-^5\) to all normal ground types by: \(\text{unit}^5 = \text{unit}, (\sigma_1 \times \sigma_2)^5 = \sigma_1^5 \times \sigma_2^5\).

---

7 For simplicity, we do not count the cons-cells of product and coproduct constructors in representations as the asymptotics are the same whether we count these or not.
\(\text{fold}_\tau^S\)-I: \[
\frac{\Gamma \vdash f: F \sigma \rightarrow \sigma \quad \Gamma \vdash e: \tau}{\Gamma \vdash \text{fold}_\tau^S f e: \sigma} \quad (\dagger) \quad \text{lower:} \quad \frac{\Gamma \vdash e: \tau^S}{\Gamma \vdash e: \tau} \quad (\ddagger)
\]

Fig. 2. Key additions for RS\(^-\). \((\dagger)\) \(\tau\) is a normal and \(\sigma\) is safe. \((\ddagger)\) \(\text{sfv}(e) = \emptyset\).

\((\sigma_1 + \sigma_2)^\tau = \sigma_1^\tau + \sigma_2^\tau\). (Note: \texttt{unit} is the sole normal and safe base type.) \(\tau^S\) has constructor \(c_1^\tau: (F_1 \tau)^S \rightarrow \tau^S\) and destructor \(d_2^\tau: \tau^S \rightarrow (F_2 \tau)^S\). In examples, we use sugared constructors for \(\tau^S\), e.g., \(\text{Succ}^S: \text{nat}^S \rightarrow \text{nat}^S\). The elements of \(\tau^S\) are essentially “safe” copies of the elements of \(\tau\). Let \(\text{sfv}(\Gamma \vdash e: \tau) = \{ x \in \text{fv}(e) \mid \Gamma(x) \text{ is safe} \}\), which we write as \(\text{sfv}(e)\) when the judgment is understood.

The new recursor for \(\tau\)-data, \(\text{fold}_\tau^S\), has the same operational semantics as \(\text{fold}_\tau\) (Fig. 1 \text{Fold}_\tau\) and the same typing rule as \(\text{fold}_\tau\) except for the new side-condition, see Fig. 2. Examples: \(u_\tau = \lambda x. (\text{fold}_\tau^S x)^S\) translates each type-\(\tau\) datum to the corresponding type-\(\tau^S\) datum; \(\text{plus} = \lambda x, y. (\text{let } g = \lambda z. \text{case } z \text{ of } (1w) \Rightarrow y; (2w) \Rightarrow (\text{Succ}^S g x)\) in \((\text{fold}_\text{nat}^S g x): \text{nat} \rightarrow \text{nat}^S \rightarrow \text{nat}^S\) adds its arguments and \(\text{times} = \lambda x, y. (\text{let } h = \lambda z. \text{case } z \text{ of } (1w) \Rightarrow (\text{Succ}^S \text{Zero}); (1w) \Rightarrow (\text{plus} x w)\) in \(\text{fold}_\text{nat}^S h y): \text{nat} \rightarrow \text{nat} \rightarrow \text{nat}^S\) multiplies its arguments.

Ramified type systems have a perennial difficulty: certain natural compositions can be untypable, e.g., \(\text{cube} = \lambda x. \text{times} x (\text{times} x x)\) fails to type using the rules stated so far. As a mitigation, we introduce the \text{lower} typing rule (Fig. 1) which is an adaptation to \(\lambda\)-calculi of Bellantoni and Cook’s Raising Rule [3]. Using \text{lower} on the \(\text{times} x x\) subterm yields a \text{nat} \rightarrow \text{nat}^S\) version of \text{cube} and an second application of \text{lower} yields a \text{nat} \rightarrow \text{nat} version. When we say a type-1 function is RS\(^-\) computable, we usually mean it is computed by a type-\((\sigma \rightarrow \tau)\) RS\(^-\)-term where both \(\sigma\) and \(\tau\) are normal.

RS\(^-\) is thus the modification of \(S^-\) as sketched above with one last change: \(+\)-I and \(\times\)-I now have the side-condition that the component types, \(\sigma_1\) and \(\sigma_2\), are both either normal ground types or safe ground types. (Thus each ground-type RS\(^-\)-term is of either of normal- or safe-type. This simplifies size bounds.)

Poly-Heap Size Bounds. Bellantoni and Cook proved poly-max size bounds for their formalisms, e.g., if \(e\) is a base-type (string-valued) \(BC\)-expression, then, for all \(\theta, \|e\theta\| \leq (p + \max_{y \in \text{sfv}(e)} |y|)\theta\), where \(p\) is a \text{normal polynomial}, i.e., \(p\) is polynomial over \(\{ \|x\| \mid x \in \text{fv}(e) \} \) and \(x\) has a normal type. Because of sharing we replace poly-max with \text{poly-heap} bounds, i.e., those of the form \(p + |y_1, \ldots, y_n|\) (recall Definition 3) where \(p\) is a normal polynomial and \(\{ y_1, \ldots, y_n \} = \text{sfv}(e)\).

(Convention: We write bounds as \(|e| \leq p + |y_1, \ldots, y_n|\), keeping the universal quantification over \(\theta\) implicit and in place of \(|y_1, \ldots, y_n|\) we write \(|\text{sfv}(e)|\).

**Theorem 4 (RS\(^-\) Poly-Heap Size-Boundness).** Given an RS\(^-\) judgment \(\Gamma \vdash e: \tau\) in which \(\tau\) and each \(\sigma \in \text{image}(\Gamma)\) is a ground type, one can effectively find a normal polynomial \(p\) with \(|e| \leq p + |\text{sfv}(e)|\).

A partial proof of Theorem 4 is given in the Technical Appendix. RS\(^-\) also satisfies \text{poly-cost boundness} (the computation tree of \(\Gamma \vdash e: \tau\) has a poly-size
of computation and cost, \( RS^\tau_1 \) can compute all poly-time computable type-1 functions. For want of space we omit these results, but their proofs are similar to analogous results in [3][9]. N.B. While the completeness result’s proof is standard, the result itself is a little subtitle. Typically, complexity classes concern the purely representational level and not extensionality constraints imposed by the things represented. In contrast, the \( RS^\tau_1 \)-computable (tree \( \rightarrow \) tree)-functions form a nonstandard complexity class: all the poly-time (in the dag-size) computable functions over tree-representations which functions are extensional with respect to tree-data. Type Restricted \( RS^\tau_1 \). Let \( RS^\tau_1[\text{nat}] \) be the restriction of \( RS^\tau_1 \) to terms with types in \( Ty\{\text{nat},\text{nat}^1\} \). It follows from [2][14] that the \( RS^\tau_1[\text{nat}] \)-computable (\( \text{nat} \times \cdots \times \text{nat} \rightarrow \text{nat} \))-functions = \( \mathcal{E}_2 \), the second Grzegorczyk class (aka, the linear-space computable functions). \( \mathcal{E}_2 \) plays a key rôle in “two-sorted complexity” characterizations [12 Chapter 4]. We shall make similar use of it below.

3 Structural Corecusions

**The Classical Case.** We extend \( S^- \) to \( S \), a formalism that computes, roughly, Gödel’s primitive recursive functionals over inductively- and coinductively-defined data. \( RS^\tau_1 \) will be our ramified, “feasible” version of \( S \). Fig. 3 gives the revised syntax [3] and evaluation rules (Destr', Unfold'). The typing rules for \( \hat{c}_\tau \), \( \hat{d}_\tau \), and \( \text{unfold}_\tau \) are given implicitly below. A declaration, \( \text{codata} \tau = \nu t.\sigma \), introduces a codata-type \( \tau \). The polynomial functor \( F, t = \sigma \) is called \( \tau \)'s signature functor. The declaration also implicitly introduces: \( \tau \)'s constructor function \( \hat{c}_\tau : F, \tau \rightarrow \tau \), \( \tau \)'s destructor function \( \hat{d}_\tau : \tau \rightarrow F, \tau \), and \( \tau \)'s corecursor \( \text{unfold}_\tau : (\nu \sigma)((\sigma \rightarrow F\sigma) \rightarrow \sigma \rightarrow \tau) \). The \( \sigma \) in \( \text{codata} \tau = \nu t.\sigma \) must be a ground type with constituent base types drawn from \( \text{t} \), \( \text{unit} \), and previously declared types. Type-\( \tau \)'s corecursor, \( \text{unfold}_\tau \), has its operational semantics given by Fig. 3’s Unfold',-rule and satisfies: \( \hat{d}_\tau \circ (\text{unfold}_\tau f) = F, (\text{unfold}_\tau f) \circ f \). N.B. Codata constructors and unfolds are lazy: \( \hat{c}_\tau \)- and \( \text{unfold}_\tau \)-expression values and hence are not evaluated unless forced by a \( \hat{d}_\tau \)-application per Destr' and Unfold'. Semantically, a codata type \( \tau \) is the greatest fixed point of \( F, \tau \): it is a largest set \( X \) isomorphic to \( F, (X) \), where \( \hat{d}_\tau \) and \( \hat{c}_\tau \) witness the isomorphism. Polynomial Set-functors are known to have such greatest fixed points [19 Theorem 10.1]. In examples, we use sugared codata-declarations along the lines of the sugared data-declarations.

**Example 5.** The declaration, \( \text{codata} \ \text{nats} = \text{Cons of} \ \text{nats} \times \text{nats} \), introduces the type \( \text{nats} \) with signature functor \( F_{\text{nats}}X = \text{nats} \times X \) and constructor \( \text{Cons} : \text{nats} \times \text{nats} \rightarrow \text{nats} \). Each element of \( \text{nats} \) corresponds to an infinite sequence of \( \text{nats} \)'s. Given an \( f : \text{nats} \rightarrow \text{nats} \), let \( ms = \text{unfold}_{\text{nats}} (\lambda x. \begin{cases} \text{case} x \text{ of} (i_1 y) & \Rightarrow (f \text{ Zero}, \text{ Succ Zero}); (i_2 y) & \Rightarrow (f \text{ Succ} y, \text{ Succ(Succ} y)), \end{cases} \), so \( ms \equiv \) the sequence \( f(0), f(1), f(2), \cdots \). Given an \( ns : \text{nats} \), let \( g = \lambda n. \begin{cases} \text{fold}_{\text{nats}} (\lambda x. \begin{cases} \text{case} x \text{ of} (i_1 y) & \Rightarrow ns; (i_2 y) & \Rightarrow (\hat{d}_{\text{nats}} y)) n, \end{cases} \), so \( g(n) = \) the \( n \)th \( \text{nats} \) in \( ns \)'s sequence.
The Ramified Case.

Normal and Safe Types:

First, we bring in all the definition of \( ms \) and the restriction of \( S \) to types of levels \( \leq 1 \) and ranks \( \leq k \). Not surprisingly, the \( S_k \)-functions of types \( \text{nat} \times \cdots \times \text{nat} \rightarrow \text{nat} \) correspond to Péter’s \( (k + 1) \)-primitive recursive functions \([15]\). We shall show how normal/safe ramification can rein in the power of these corecursions. First, we consider codata representations and their size.

**Representation and Size.** A type-\( \tau \) codatum \( x \) is represented via lazy \( \varepsilon_{\tau} \)- and/or \( \text{unfold}_{\tau} \)-expressions; if we probe \( x \) with ever-longer series of destructor applications, a possibly infinite structure unfurls. A codatum is thus a function-like object that must be queried (via destructor applications) to be computed over. To measure codata-size we adapt Kapron and Cook’s notion of the length of a type-1 function \([13]\). Measuring just rank-0 codata suffices for this paper.

**Definition 6.** Suppose \( e\theta \) is of type \( \tau \), a rank-0 codata-type.

(a) The apparent size of \( e\theta \) (written: \( |e\theta| \)) is 1.

(b) The observed size of \( e\theta \) (written: \( \|e\theta\| \)) is the function over natural numbers: \( n \mapsto \max\{ |d(e)\theta| : d \text{ varies over sequences of compositions of destructors with (i) } d(e) \text{ type correct and (ii) at most } n \text{ occurrences of } d_{\tau} \}. \)

Roughly, \( |\|e\theta\||(n) \) is the maximum apparent-size of the data in \( \tau \)-cons-cells along any path from the head of \( e\theta \) that includes at most \( n \) type-\( \tau \) links. **Example:** For \( ns \) of Example\([5]\) \( \|\|ns\|\|\)(n) = 1 + \max_{i<n}(\text{the }i\text{th element of }\|ns\|\text{'s sequence}). \)

**The Ramified Case.** \( RS_I \), our ramification of \( S \), extends the normal/safe distinction to codata. **Key Points:** As the value of \( (\text{unfold}_{\tau} g) \) is the result of a (co)recursion, it should be safe, as \( g \) gives the computation step, we should have \( g:\text{safe} \rightarrow \text{safe} \), and as \( \text{unfold} \)'s are lazy, destructs drive the computation.

**Normal and Safe Types:** First, we bring in all the \( RS_{I}^- \) conventions to this setting to ramify data. Second, a declaration codata \( \tau = u \cdot \sigma \) introduces the normal type \( \tau \) with constructor \( \varepsilon_{\tau} \) and destructor \( d_{\tau} \) as before, a safe type \( \tau^S \) with constructor \( \varepsilon_{\tau}^S : (F(\tau)^S) \rightarrow \tau^S \) and destructor \( d_{\tau}^S : \tau^S \rightarrow (F(\tau)^S, \sigma^S) \), and \( \text{unfold}^S : (\forall \sigma)(\sigma^S \rightarrow (F(\tau)^S, \sigma^S) \rightarrow \sigma^S \rightarrow \tau^S) \) where \( \text{unfold}^S \) has the same operational semantics as \( \text{unfold} \). **Example:** Replace \( \text{unfold}_{\text{nats}}, \text{Zero}, \text{Succ}, \) and \( f : \text{nats} \rightarrow \text{nats} \) with \( \text{unfold}_{\text{nats}}^S, \text{Zero}^S, \text{Succ}^S, \) and \( f : \text{nats}^S \rightarrow \text{nats}^S \) in Example\([5]\)'s definition of \( ms \), then \( ms \) can be assigned assigned type \( \text{nats}^S \). **N.B.** Given an

\[
\begin{align*}
\text{Decl} & ::= \text{data } T = \mu X. T y_0 \mid \text{codata } T = \nu X. T y_0 \\
\text{Destr}_{\tau} & ::= \frac{e\theta \downarrow \|e\theta\|}{(d_{\tau}(\varepsilon_{\tau} e))\theta \downarrow \|e\theta\|} \\
\text{Unfold}_{\tau} & ::= \frac{\text{(unfold}_{\tau} f)(f e\theta \downarrow \|e\theta\|)}{d_{\tau}(\text{unfold}_{\tau} f e\theta \downarrow \|e\theta\|)}
\end{align*}
\]

**Fig. 3.** Key Additions for \( S \).
RS $^-1$-computable $f \colon \text{nat} \to \text{nat}$, there may not be an RS $^-1$-definable analogue of $ms$ from Example 5.

**Poly-Heap Size Bounds.** To adapt poly-heap bounds to take account of observed sizes we use Kapron and Cook’s notion of second-order polynomials \[13\]; these are roughly ordinary polynomials with applied type-1 function symbols included (e.g., $x^2 + f(y + 2)$). Now $|e| \leq p + |\text{sfv}(e)|$ is a poly-heap bound on apparent size when $p$ is a normal second-order polynomial (i.e., over $\{ |x| \mid \Gamma(x) \text{ is normal} \}$ and $\{ ||x|| \mid I(x) \text{ is a normal codata type} \}$) and $||e|| \leq \lambda n.(p + |\text{sfv}(e)|)$ is a poly-heap bound on observed size where now $p$ can have $n$ as a type-0 variable.

**Theorem 7 (RS $^-1$ Poly-Heap Size-Boundness).** For an RS $^-1$-judgment $\Gamma \vdash e : \tau$ where $\tau$ and each $\sigma \in \text{image}(\Gamma)$ is a ground type, one can effectively find a normal second-order polynomial $p$ such that, if $\tau$ is a data-type, then $|e| \leq p + |\text{sfv}(e)|$ and, if $\tau$ is a codata-type, then $||e|| \leq \lambda n.(p + |\text{sfv}(e)|)$.

RS $^-1$ satisfies appropriate poly-cost boundness and poly-completeness properties with proofs similar to the analogous (type-2) results in \[8,9\]; but, as with RS $^-1$, we have not the space to describe, much less prove, these results.

**Type Restricted RS $^-1$.** Let $RS_1(\sigma \to \tau|B)$ denote the functions of type $\sigma \to \tau$ computable by RS $^-1$-terms with types from $Ty^B$ where $B$ = the normal and safe versions of the base types occurring in $\sigma$, $\tau$, and $B$. For $\text{data bit} = \text{Nought}$ and $\text{codata stream} = \text{Cons of bit} \times \text{stream}$, one can show: (i) $RS_1(\text{unit} \to \text{stream}|\emptyset) = \omega$-regular languages; (ii) $RS_1(\text{stream} \to \text{stream}|\emptyset) = \text{finite-state stream maps}$; (iii) $RS_1(\text{unit} \to \text{stream}|\{\text{nat}\}) = \text{logspace streams}$; and (iv) $RS_1(\text{stream} \to \text{stream}|\{\text{nat}\}) = \text{logspace stream-functions}$.

4 Conclusions

RS $^-1$ characterizes a notion of poly-time computation over data and codata. As a formalism, RS $^-1$ is not much more complicated than the original ones of Bel- lantoni and Cook \[3\] and Leivant \[14\], although a few of RS $^-1$’s additions involve subtleties. The above work suggests many paths for exploration. Here, briefly, are a few.

**Pola vs. RS $^-1$**. Pola restricts sharing for its notion of poly-time over data and codata. RS $^-1$ essentially forces sharing to obtain its notion of poly-time over data and codata. How different are these two notions? Can one notion “simulate” the other in some reasonable sense? Is there a good notion of poly-time over data and codata that sits above both the Pola and RS $^-1$ notions?

**Higher-types.** Higher-type functions over data-realm and higher-rank streams and trees in the codata-realm are roughly two different perspectives on the same thing. In investigating true higher-type extensions of RS $^-1$, having these two views may help puzzling out sensible approaches to higher-type feasibility.

**Programming in RS $^-1$ is clumsy.** One problem is that RS $^-1$-recursions carry out their computations using safe $\to$ safe functions, but there are very few of
these that have *closed* definitions in $RS_1$. E.g., there is no closed $RS_1$-function that gives the $\text{nat}^5$-maximum of two $\text{nat}^5$-values, even though adding such a function would be a complexity-theoretic conservative extension. Based on an insight first pointed out and studied by Hofmann [12], any polynomial-time computable $f: \text{safe} \rightarrow \text{safe}$ with $|f(x)| \leq |x|$ for all $x$, would be a similarly conservative extension to $RS_1$. Finding a simple scheme to add to $RS_1$ that allows the definition of more such functions over data (and the dual notion, $\|x\| \leq \|f(x)\|$, for functions over codata) is a nice problem.

References

1. Abelson, H., Sussman, G.: Structure and Interpretation of Computer Programs. MIT Press, 2nd edn. (1996)
2. Bellantoni, S.: Predicative Recursion and Computational Complexity. Ph.D. thesis, University of Toronto (1992), Univ. of Toronto Computer Science Dept. TR 264/92
3. Bellantoni, S., Cook, S.: A new recursion-theoretic characterization of the polytime functions. Computational Complexity 2, 97–110 (1992)
4. Burrell, M., Cockett, R., Redmond, B.: Pola: A language for ptime programming. LCC’09 pp. 1–15 (Jun 2009)
5. Cockett, R., Redmond, B.: A categorical setting for lower complexity. Electronic Notes in Theoretical Computer Science 265, 277–300 (2010)
6. Cook, S., Nguyen, P.: Logical Foundations of Proof Complexity. Cambridge University Press (2010)
7. Cook, S., Urquhart, A.: Functional interpretations of feasibly constructive arithmetic. Annals of Pure and Applied Logic 63, 103–200 (1993)
8. Danner, N., Royer, J.: Adventures in time and space. Logical Methods in Computer Science 3(1:9), 1–53 (2007)
9. Danner, N., Royer, J.: Two algorithms in search of a type system. Theory of Computing Systems 45, 787—821 (2009)
10. Féreé, H., Hainry, E., Hoyrup, M., Péchoux, R.: Interpretation of stream programs: Characterizing type 2 polynomial time complexity. In: Cheong, O., Chwa, K.Y., Park, K. (eds.) Algorithms and Computation, LNCS, vol. 6506, pp. 291–303. Springer (2010)
11. Hartmanis, J., Stearns, R.: On the computational complexity of algorithms. Transactions of the American Mathematical Society 117, 285–306 (1965)
12. Hofmann, M.: Linear types and non-size increasing polynomial time computation. Information and Computation 183, 57–85 (2003)
13. Kapron, B., Cook, S.: A new characterization of type 2 feasibility. SIAM Journal on Computing 25, 117–132 (1996)
14. Leivant, D.: Ramified recurrence and computational complexity I: Word recurrence and poly-time. In: Clote, P., Remmel, J. (eds.) Feasible Mathematics II, pp. 320–343. Birkhäuser (1995)
15. Longley, J.: Notions of computability at higher types, I. In: Cori, R., Razborov, A., Torcevic, S., Wood, C. (eds.) Logic Colloquium 2000, LNL, vol. 19, pp. 32–142. A. K. Peters (2005)
16. Mehlhorn, K.: Polynomial and abstract subrecursive classes. Journal of Computer and System Science 12, 147–178 (1976)
17. Ramyaa, R., Leivant, D.: Feasible functions over co-inductive data. In: Dawar, A., Queiroz, R. (eds.) Logic, Language, Information and Computation. LNCS, vol. 6188, pp. 191–203. Springer (2010)

18. Ramyaa, R., Leivant, D.: Ramified corecurrence and logspace. In: Mislove, M., Ouaknine, J. (eds.) Twenty-seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII). ENTCS, vol. 276, pp. 247–261 (2011)

19. Rutten, J.: Universal coalgebra: A theory of systems. Theoretical Computer Science 249, 3–80 (2000)

20. Turing, A.: On computable numbers, with an application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 42, 230–265 (1936)

21. van Emde Boas, P.: Machine models and simulations. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. A, pp. 1–66. Elsevier (1990)
\[ E ::= X \mid (E_1 \ E_2) \mid (\lambda x. E) \mid () \mid (E_1, E_2) \mid (\pi_1 E) \mid (\pi_2 E) \mid (\iota_1 E) \mid (\iota_2 E) \mid \text{case } E_0 \text{ of } (\iota_1 X_1) \Rightarrow E_1; (\iota_2 X_2) \Rightarrow E_2 \]

**Fig. A.1.** L raw syntax, where \( X ::= \text{identifiers} \).

**Technical Appendix**

Notes

1. The “S” in \( S^\perp \) and \( S \) stands for \textit{structure} and the “R” in \( RS^\perp \) and \( RS_1 \) stands for \textit{ramified}.
2. Internal representations of constructors are underlined as in Figs. 1 and A.3.
3. The side-condition of Pair-rule in Fig. A.3: If \( e_1 \theta \downarrow v_1 \theta_1 \) and \( e_2 \theta \downarrow v_2 \theta_2 \), then \( \theta_1 \) and \( \theta_2 \) may be inconsistent. Hence in Pair, \( e_2 \) is alpha-reduced to \( e'_2 \) so that the \( e_1 \)- and \( e'_2 \)-evaluations introduce distinct variables into their value’s environments.
4. The unsugared version of Succ(Succ Zero) is \( c_{\text{nat}}(\iota_2(c_{\text{nat}}(\iota_2(c_{\text{nat}}(\iota_1()))))) \).
5. Call-by-value and growth. Note that for \( e' \) of ground-type, \( |((\lambda x. \text{Fork } x) \ e')| = 1 + |e'| \) because, by the call-by-value semantics, \( e' \) is evaluated to a value \( v \theta \) (i.e., a reference to a data-representation) which becomes the value of \( x \) used in \( \text{Fork}(x, x)[x \mapsto v \theta] \).
6. Dodging exponential growth. If one could define a function $f : \text{nat} \rightarrow \text{tree}$ such that $f \text{ Zero } = \text{ Leaf}$ and $f(\text{Succ } x) = \text{Fork}(f(x))(f(x))$, then $|f(x)|$ could be exponentially larger than $|x|$. Theorem 6 implies that no such $f$ is $RS_1$-definable, but intuitively the reason is that a $\text{fold}_{\text{nat}}$ definition provides one reference to the result of the recursive call since the $\text{Succ}$-constructor is unary. This one reference can be used multiple times, but always representing links to the same result, and hence not increasing the size. The function that is $RS_1$-definable is (in effect) $f' \text{ Zero } = \text{ Leaf}$ and $f'(\text{Succ } x) = \text{let } r = f'(x) \text{ in } \text{Fork}(r, r)$. 

7. Bellantoni and Cook’s Raising Rule. It amounts to a (sound!) specialization of Whitehead and Russell’s Axiom of Reducibility. Compare the end of the first paragraph of [3, §5] and +12.1 of Principia Mathematica, Vol. 1, 1/e, Cambridge University Press, 1910, available from http://name.umdl.umich.edu/AAT3201.0001.001.

8. Consider $\text{leaves} : \text{tree} \rightarrow \text{nat}$ where $\text{leaves}(t) =$ the number of leaves of $t$. $RS_1$ cannot compute this because $|\text{leaves}(t)|$ can exponentially-larger than $|t|$. In contrast, Pola can compute this as Pola allows some forms of change-of-parameter in recursions and, under Pola, $t$ is always a strict tree.

9. Codata, Memoization, and Sharing. Corecursions (unfolds) are not memoized, but structure sharing is allowed in codata.

10. Codata and poly-completeness. Since type-level 1 $RS_1$ functions can have codata inputs and outputs, we can translate some standard examples from type-2 complexity to show that $RS_1$ is missing some functions over codata, where these functions’ runtime complexity is comparable to that of $RS_1$-computable functions. The cure to this problem is to introduce an analogue of Bellantoni’s $\text{Mod}$ function [2, Chapter 8] $(\text{Mod } mn = m \text{ mod } n)$ or the authors’ $\text{Down}$ function [3, §4] $(\text{Down } x y = x$, if $|x| \leq |y|; \epsilon$, otherwise) both of which are (safe $\rightarrow$ normal $\rightarrow$ normal) functions. As to the motivations for such functions and their odd typing we refer the reader to [3]. Adding such a function to $RS_1$ is not a major change.

The next lemma is a key property of terms with normal types. Its proof is a simple induction on type derivations.

**Lemma A.1.** If $\Gamma \vdash e : \tau$ where $\tau$ is normal, then $\text{sfv}(e) = \emptyset$.

**Lemma A.2 (Basic Poly-Heap Bounds Arithmetic).** Suppose $\Gamma \vdash e : \sigma$, $|e| \leq p + |\text{sfv}(e)|$, $\Gamma \vdash e' : \sigma'$, and $|e'| \leq p' + |\text{sfv}(e')|$, where $p$ and $p'$ are polynomials over $\{ |x| : \Gamma(x) \text{ is normal} \}$. Also suppose $x \in \text{fv}(e)$ with $\Gamma(x) = \sigma'$. Then:

(a) $|e[x := e']| \leq (p|x| : p') + |\text{sfv}(e[x := e'])|$, if $\sigma'$ is normal.

(b) $|e[x := e']| \leq p + p' + |\text{sfv}(e[x := e'])|$, if $\sigma'$ is safe.

(c) $|(e, e')| \leq p + p' + |\text{sfv}((e, e'))|$.

**Proof (Sketch).** Part (a): By $\text{sfv}(e') = \emptyset$. Hence, by the monotonicity of our polynomials, (a) follows.

Part (b): By monotonicity again (and some abuse of notation): $|e[x := e']| \leq p + |e'|, (\text{sfv}(e) - \{ x \})| \leq p + (p' + |\text{sfv}(e')|, |\text{sfv}(e) - \{ x \})| \leq p + p' + |e[x := e']|$. 


Part \(\blacksquare\): A naïve upper bound on \(|(e, e')|\) is \(p + p' + 2|\text{sfv}(e, e')|\), but this double counts the structure shared by \(e\) and \(e'\). So by eliminating the double counting, we have the required bound. \(\Box\)

Poly-Heap vs. Poly-Max Bounds. The analogue of parts \(\text{(b)}\) and \(\text{(c)}\) of Lemma \(\mathbf{A.2}\) hold for poly-max bounds. Bounds of the form of part \(\text{(b)}\) are key in poly-boundedness arguments for forms of “safe” recursions. The analogue of Lemma \(\mathbf{A.2}\) \(\text{(c)}\) fails for poly-max bounds. However, if one requires \((\text{à la Pola})\) that \(e\) and \(e'\) have no safe variables in common, then the poly-max-analogue of Lemma \(\mathbf{A.2}\) \(\text{(c)}\) does hold. These two alternative ways of counting are at the heart of the \(\text{RS}_I/\text{Pola}\) split. Note that what is at stake in how one bounds a pair is how, in general, one bounds the size of branching structures.

Theorem A.3 (Theorem \(\text{A}\) Restated). Given an \(\text{RS}_I^\tau\) judgment \(\Gamma \vdash e : \tau\) in which \(\tau\) and each \(\sigma \in \text{image}(\Gamma)\) is a ground type, one can effectively find a normal polynomial \(p\) with \(|e| \leq p + |\text{sfv}(e)|\).

Proof (Partial sketch). Our first problem in exhibiting the upper bound is that \(e\) may well contain higher-type subterms. Let \(\tilde{e}\) be the normalized version of \(e\). Note that \(|e| \leq |\tilde{e}|\), where \(|\tilde{e}|\) can be much larger than \(|e|\). But a poly-heap bound on \(|\tilde{e}|\) serves as a bound on \(|e|\). Thus, we assume without loss of generality that \(e\) is normalized. Since \(e\) is normalized, the only place a \(\lambda\)-expression can occur in \(e\) is as the first argument of a \(\text{fold}^S\)-construct, moreover, these \(\lambda\)-expressions have level-1 types. Also note that each variable occurring in \(e\) must be of ground type.

The proof is a structural induction on the derivation of \(\Gamma \vdash e : \tau\). We consider the last rule used in this derivation.

All of the cases, save one, are standard, straightforward arguments—adjusting for the change from poly-max to poly-heap bounds. So we omit these. The interesting case is the one for \(\text{fold}^S\). We treat this case which, for simplicity and concreteness, we further narrow to the case for \(\text{fold}^S_{\text{tree}}\), which touches on the key issues in the general \(\text{fold}^S\)-case. Recall that \(F_{\text{tree}}X = \text{unit} + X \times X\), \(F_{\text{tree}}f = \text{id}_{\text{unit}} + f \times f = \lambda u.\ (\text{case } u \text{ of } (\_1 v) \Rightarrow \_1(\_1); (\_2 v) \Rightarrow \_1(f(\pi_1(v)), f(\pi_2(v))))\), and \((\text{fold}^S_{\text{tree}} g) \circ \varepsilon_{\text{tree}} = g \circ F_{\text{tree}}(\text{fold}^S_{\text{tree}} g)\).

Some conventions: To cut down on clutter, when \(y\) is of ground type \(\sigma\) and \(v\) is a type-\(\sigma\) value (i.e., a pointer to an internal representation of a type-\(\sigma\) object), we shall rewrite \(e\theta[y \mapsto v\theta']\) to \(e[y := v]\theta\), provided the value named by \(v\theta'\) is a function of \(\theta\). The substitution of the (pointer) \(v\) for the variable \(y\) in \(e\) is, in essence, just cutting out one level of indirection and thus simplifies reasoning about the value of \(e\theta[y \mapsto v\theta']\). Similarly, in “heap” expressions \(|e_1, \ldots, e_k|\) we allow value terms (i.e., pointers to representations) among the \(e_i\)’s with the obvious meaning of \(|e_1, \ldots, e_k|\), again we are simply cutting out a level of indirection. Finally, if \(E\) a set of \(k\)-many expressions \(e_1, \ldots, e_k\), then \(|E| = |e_1, \ldots, e_k|\).

Case: \(\text{fold}^S_{\text{tree}}\). Thus, \(e = (\text{fold}^S_{\text{tree}} (\lambda z. e_0) e_1), \) where \(\Gamma \vdash e : \tau, \Gamma, z : F_{\text{tree}}\tau \vdash e_0 : \tau, \Gamma \vdash e_1 : \text{tree}\), and \(\tau\) is a safe base type. By the induction hypothesis, there are normal polynomials \(p_0\) and \(p_1\) that \(|e_0| \leq p_0 + |\text{sfv}(e_0)|\), and \(|e_1| \leq p_1\). Fix
an environment $\theta$ and suppose $e_1 \theta \downarrow t_1 \theta'$. Recall that $t_1$ is a pointer to the dag-representation of $e_1$’s value. (Since $t_1$ is a data-constant, it suffices to take $\theta' = \theta$.) Let $t_2, \ldots, t_n$ be pointers to the other tree-cons-cells in the representation, ordered so that, for all $i$ and $j$, if $t_i$ is a dag-ancestor of $t_j$, then $i \leq j$. Suppose, for $i = 1, \ldots, n$, $(\text{fold}_{\text{tree}}^5(\lambda x \cdot e_0) t_i) \theta \downarrow r_i \theta$, where $r_i$ is a pointer to the dag-representation of the result of the $\text{fold}_{\text{tree}}^5$-recursion. N.B. The $t_i$’s and $r_i$’s are functions of $\theta$. So, as a reminder of this, in our bounds calculations, we shall make explicit the usually suppressed $\theta$.

**Claim 1:** Suppose the hypotheses of Lemma A.2 and suppose $\sigma'$ is safe. Then $|\{ e[x := e'], e_1, \ldots, e_k \}| \theta \leq (p + |\{ e', e_1, \ldots, e_k \}| \cup \text{sfv}(e[x := e'])\}| \theta$, for all $\theta$.

**Proof:** This is just an extension of the proof of Lemma A.2(b).

**Claim 2:** For each $i = 1, \ldots, n$:

(a) If $t_i = \text{Leaf} = \lambda x \cdot (\text{Leaf}(\lambda \cdot ))$, then $(\text{fold}_{\text{tree}}^5(\lambda x \cdot e_0) t_i) \theta = e_0 \theta[z \mapsto \lambda \cdot ]$.

(b) If $t_i = \text{Fork} t_j t_k = \lambda x \cdot (\text{Leaf}(\lambda \cdot ))$, then $(\text{fold}_{\text{tree}}^5(\lambda x \cdot e_0) t_i) \theta = e_0 \theta[z \mapsto \lambda \cdot ]\theta$.

(c) $|\{ r_1, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta \leq p_0 + |\{ r_{i+1}, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta$.

**Proof:** Part (a) is a straightforward calculation.

Part (b) is another straightforward calculation, taking into account that the implementation of $\text{fold}_{\text{tree}}^5$ is memoizing.

Part (c). Case: $t_i$ is a leaf. Then

$$|\{ r_1, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta = |\{ e_0[z := \lambda \cdot ]\}, r_{i+1}, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta \quad \text{(by part (a))}$$

$$= p_0 + |\{ \lambda \cdot \}, r_{i+1}, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta \quad \text{(by Claim 1)}$$

$$= p_0 + |\{ r_{i+1}, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta \quad \text{(since } |\lambda \cdot | = 0).$$

Case: $t_i$ is a fork. Then

$$|\{ r_1, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta = |\{ e_0[z := \lambda \cdot \langle r_j, r_k \rangle], r_{i+1}, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta \quad \text{(by part (b))}$$

$$= p_0 + |\{ \lambda \cdot \langle r_j, r_k \rangle, r_{i+1}, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta \quad \text{(by Claim 1)}$$

$$= p_0 + |\{ r_{i+1}, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta \quad \text{(since } j, k > i).$$

Thus by Claim 2(c), $|\langle \text{fold}_{\text{tree}}^5(\lambda x \cdot e_0) e_1 \rangle \theta| = |\langle \text{fold}_{\text{tree}}^5(\lambda x \cdot e_0) t_1 \rangle \theta| = |\{ r_1 \} \cup \text{sfv}(e)\}| \theta \leq |\{ r_1, \ldots, r_n \} \cup \text{sfv}(e)\}| \theta \leq (p_0 \cdot n + |\text{sfv}(e)\}| \theta$. Recall that $|e_1| \leq p_1$. Therefore, $p = p_0 \cdot p_1$ suffices for this case.

The effectiveness part of the theorem follows from the fact that the induction argument essentially describes a recursive algorithm for constructing $p$. □

**Theorem A.4 (Theorem 7 Restated).** For an RS$_1$-judgment $\Gamma \vdash e : \tau$ where $\tau$ and each $\sigma \in \text{image}(\Gamma)$ is a ground type, one can effectively find a normal second-order polynomial $p$ such that, if $\tau$ is a data-type, then $|e| \leq p + |\text{sfv}(e)|$ and, if $\tau$ is a codata-type, then $|e| \leq \lambda n. (p + |\text{sfv}(e)|)$. 
Proof (Partial sketch). As in the proof of Theorem 4, we may without loss of generality assume that \( e \) is normalized. Thus the only place a \( \lambda \)-expression can occur in \( e \) is as the first argument of a \text{fold}^S \)- or an \text{unfold}^S \)-construct, and moreover, these \( \lambda \)-expressions have level-1 types. Also note that each variable occurring in \( e \) must be of ground type. Our proof is a structural induction on the derivation of \( \Gamma \vdash e : \tau \). We consider the last rule used in this derivation.

Now, as in our sketch of the proof of Theorem 4, here we shall present just one key case (\text{unfold}^S _\text{nats} -I), and in fact, a specialization of that (\text{unfold}^S _\text{nats} -I). Unlike the situation for the proof of Theorem 4, the omitted cases here are less standard and a few involve some fine points. However, almost all of these omitted cases parallel problems we dealt with our work on feasible type-level 2 programming formalisms [8, 9].

Case: \text{unfold}^S _\text{nats} -I. We consider the case where \( \sigma \) is a \text{data} type. Thus, \( e = (\text{unfold}^S _\text{nats} (\lambda z . e_0) e_1) \), where \( \sigma \) is a safe ground data-type, \( \Gamma, z : \sigma^S \vdash e_0 : \text{nat}^S \times \sigma^S \), and \( \Gamma \vdash e_1 : \sigma^S \). Recall \( F_{\text{nats}} \cdot \text{X} = \text{nat} \times \text{X} \), \( F_{\text{nats}} f = \text{id} \circ \text{nat} \times f = \lambda u . (\pi_1 u, f(\pi_2 u)) \), and \( \hat{d}_{\text{nats}} \circ (\text{unfold}^S _\text{nats} g) = F_{\text{nats}} (\text{unfold}^S _\text{nats} g) \circ g = \lambda u . (\pi_1 (g(u)), \text{unfold}^S _\text{nats} g (\pi_2 (g(u)))) \). Let \( g_1 = \pi_1 \circ g \) and \( g_2 = \pi_2 \circ g \), then for all \( n \geq 1 \):

\[
\hat{d}^{(n)} _\text{nats} (\text{unfold}^S _\text{nats} g u) = \left( g_1 (g_2^{(n-1)} u), \text{unfold}^S _\text{nats} g (g_2^{(n)} u) \right). \tag{4}
\]

Now, by the induction hypothesis, there are normal polynomials \( p_0 \) and \( p_1 \) such that \( |e_0| \leq p_0 + |z|, \text{sfv}(e)| \) and \( |e_1| \leq p_1 + |\text{sfv}(e)| \). By (4), to bound \( ||e||(n) \) for \( n \geq 1 \), it suffices to bound \( |g_1 (g_2^{(n-1)} e_2)| \) for \( g_1 = \pi_1 \circ (\lambda z . e_0) \) and \( g_2 = \pi_2 \circ (\lambda z . e_0) \).

For \( n = 1 \), \( |g_1 (g_2^{(n-1)} e_1)| = |e_0[z := e_2]| \leq p_0 + |e_1, \text{sfv}(e)| \leq p_0 + p_1 + |e| \). Iterating this, we have for \( n \geq 1 \), \( ||e||(n) \leq |g_1 (g_2^{(n-1)} e_1)| \leq p_0 + p_1 \cdot n + |\text{sfv}(e)| \). Hence, \( p = p_0 + p_1 \cdot n \) suffices.

In the case where \( \sigma \) is a \text{codata} type, the basic structure of the argument stays the same but the \text{second-order polynomial} algebra becomes more involved.

The induction above essentially describes a recursive algorithm for constructing \( p \). Hence, the effectiveness part of the theorem follows. \( \square \)