Primitive recursion in the abstract

Daniel Leivant, Jean-Yves Marion

Primitive recursion in the abstract

Daniel Leivant[1] *and Jean-Yves Marion*[2]

June 12, 2019

Abstract

Recurrence can be used as a function definition schema for any non-trivial free algebra, yielding the same computational complexity in all cases. We show that primitive-recursive computing is in fact independent of free algebras altogether, and can be characterized by a generic programming principle, namely the control of iteration by the depletion of finite components of the underlying structure.

1 Introduction

1.1 Abstract delineation of PR

Recall that the schema of *recurrence over* $\mathbb{N}$ consists of the two equations

$$\begin{array}{rcl} f(0,\vec{x}) & = & g_0(\vec{x}) \\ f(\mathtt{s}n,\vec{x}) & = & g_{\mathtt{s}}(n,\vec{x},f(n,\vec{x})) \end{array} \tag{1}$$

More generally, given a free algebra $\mathbb{A} = \mathbb{A}(C)$ generated from a finite set C of constructors (where $\mathtt{c} \in C$ has arity $\mathfrak{r}(\mathtt{c})$), the schema of *recurrence over* $\mathbb{A}$ has one equation per constructor $\mathtt{c}$:

$$f(\mathtt{c}(z_1,\cdots,z_k),\vec{x}) \quad = \quad g_{\mathtt{c}}(\vec{z},\vec{x},f_1,\ldots,f_k) \tag{2}$$

where $f_i = f(z_i,\vec{x})$ and $k = \mathfrak{r}(\mathtt{c})$.

The recurrence schema for $\mathbb{N}$ originates with Dedekind's interest in formalizing arithmetic, was first articulated by Skolem [19], and was studied extensively (see e.g. [17]). The set $\mathbf{PR}(\mathbb{A})$ of *primitive recursive functions* over $\mathbb{A}$ is generated from the constructors of $\mathbb{A}$ by recurrence over $\mathbb{A}$ and explicit definitions.[3]

We show here that primitive recursive computing is independent from free algebras altogether, and is rooted instead in fundamental programming constructs alone. Namely, **PR** is the set of mappings between structured (finite) data-objects that are computed by imperative programs whose loops are governed by the depletion of the structure's functions, dubbed here "variants". To show that a program terminates using time and space resources primitive-recursive in the input's size, it therefore suffices to identify for each loop a variant, which is usually germane to the algorithm. This characterization also encompasses in one fell swoop various variations of primitive recursion. Moreover, the fact that variants are second-order entities makes them amenable to methods of implicit computational complexity, as we show elsewhere.

[1]SICE (Indiana University) and IRIF (Université Paris-Diderot)

[2]LORIA, Université de Lorraine and CNRS

[3]The phrase "primitive recursion" was triggered by Ackermann's and Sudan's discoveries of computable ("recursive") functions that are not in $\mathbf{PR}(\mathbb{N})$. Given present-day usage of "recursion" for a broader notion of recursive procedures, it seems preferable to refer to the schemas above as "recurrence" rather than "recursion."

1.2 Inductive data-objects as finite structures

We identify data-objects, such as elements of a free algebra, with finite partial-structures. For example, binary strings are finite partial-structure over the vocabulary with a constant `e` and unary function identifiers `0` and `1`. Thus, the string `011` is taken to be the following partial-structure with four atoms, and where the partial-function denoted by `0` is defined for only one of the four:

$$\mathtt{e} \circ \xrightarrow{\mathbf{0}} \circ \xrightarrow{\mathbf{1}} \circ \xrightarrow{\mathbf{1}} \circ$$

A function over $\mathbb{A}$ can thus be construed as a mapping between such finite structures.

Computation by recurrence on $\mathbb{A}$ terminates because the recurrence argument is being depleted. A more generic form of depletion, adapted to loops of imperative programs, is obtained by assigning to each loop a set of finite partial-functions, which we dub the loop's *variant,* and requiring that each pass through the loop contracts the variant. Our variants are analogous to the variants used in Hoare-style program verification [9, 6, 21], but whereas the latter decrease along a prescribed well-ordering, our variants are depleted by function-contractions, i.e. function re-assignment to *undefined*. The distinction between positive and negative forms of assignment is thus fundamental in our approach.

1.3 Main results

Our programming language **STV** is a basic imperative language for the transformation of structures. We focus on finite partial-structures, following the the approach of [16] and the imperative language **ST** defined there. **ST**, which is a variant of Gurevich's ASMs [4, 11, 12], focuses on finite structures, but also supports computing over infinite data-structures, such as free algebras, once their elements are construed as finite structures. **ST** is Turing complete, and therefore a suitable framework for identifying syntactic conditions that characterize complexity classes.

The programming language **STV** defined here differs from **ST** only in having loop *variants,* which convey in a more generic and abstract sense the resource depletion implicit in recurrence. Our main technical result is that **STV** characterizes an abstract notion of primitive recursion, in the strongest possible sense. On the one hand, all **STV**-programs run in time and space that are primitive-recursive in the size of their input structure (Theorem 2). For the converse, we show that for each free algebra $\mathbb{A}$ the functions in $\mathbf{PR}(\mathbb{A})$ are computable by **STV**-programs (Theorem 4). Moreover, recurrence is embedded directly in **STV**, using no extraneous concepts or coding schemes.

The equivalence above is extensional, in the sense that it refers to computability, and not to particular algorithms. However, if we take **ST** as our reference Turing-complete computation model, then every **ST**-program that runs in PR time can be augmented with variants to become an equivalent **STV**-program.

We caution against confounding our approach with unrelated prior research addressing seemingly related themes. Recurrence and recursion over finite structures have been shown to characterize logarithmic space and polynomial time queries, respectively [13, 18], but the programs in question do not allow inception of new structure

elements, and so remain confined to linear space complexity, and are inadequate for the broad approach we seek. On the other hand, unbounded recurrence over arbitrary structures has been considered by a number of authors [1, 2, 20], but always in the traditional sense of computing within an infinite structure. Also, while the meta-finite structures of [8] merge finite and infinite components, both of those are considered in the traditional framework, whereas we deal with purely finite structures, referring to infinity only in relation to *collections* of such structures. Finally, the functions we consider are from structures to structures (as in [18]), and are thus unrelated to the global functions of [10, 7], which are (isomorphism-invariant) mappings that assigns to each structure a function over it.

As for generalizations of primitive recursion to computing over abstract structures, [5] refers to the computation model of [3], which incorporates primitive recursive from the outset, and therefore does not examine the abstract contents of **PR** as we do here.

The paper is sectioned along the outline above: §2 defines the programming language **STV**, and §3 gives examples of programs, most of which we use in the sequel. In §4 we prove that **STV** characterizes primitive recursion, and that it includes, modulo augmenting loops with variants, all **ST**-programs that terminate in PR time.

2 STV: Programs with loop variants

The imperative programming language **ST** we defined in [16] is designed to be a Turing-complete language for the transformation of finite partial-structures, whose building blocks are as fundamental as possible. It is a variant of Gurevich's abstract state machines (ASMs) [4, 11, 12] that focuses on finite structures, distinguishes between constructive and destructive assignments, and give a prominent place to recurrence. (ASMs strive to generalize directly hardware models, and are based on global iteration.)

The imperative programming language **STV** proposed here refines **ST** programs with a restrictive condition on loops which guarantees termination, foregoing in the act Turing completeness. We refer the reader to [16] for a broader discussion of **ST**.

2.1 Finite partial-structures

The programs of **STV** operate over a single data-type, namely finite partial-structures, defined as follows. We posit a fixed denumerable set A of *atoms.* An *A-function* is a finite k-ary partial-function over A, where $k \geqslant 0$; thus, the nullary A-functions are the atoms. To accommodate non-denoting terms we extend A to a flat domain $A_\perp$, which has, in addition to the atoms, a fresh object $\perp$, intended to denote "undefined." The atoms are the *standard* elements of $A_\perp$. We identify a k-ary A-function F with the strict total function $\tilde{F} : A_\perp^k \to A_\perp$ that for input $\vec{a}$ returns $F(\vec{a})$ if it is defined, and $\perp$ otherwise. An *entry* of an A-function F is a tuple $\langle a_1 \dots a_k, b\rangle$ where $b = F(a_1, \dots, a_k) \neq \perp$. The *scope* of F is the set of atoms occurring in its entries. The *range* of F is the set of atoms obtained as values of F. The *size* of F is the number $|F|$ of entries in it.

Function partiality provides a natural representation of finite relations over A by partial functions, without recourse to booleans: we identify a finite k-ary relation R over A ($k > 0$) with the partial-function

$$\xi_R(a_1, \ldots, a_k) \;=\; \text{if } R(a_1, \ldots, a_k) \text{ then } a_1 \text{ else } \perp$$

Conversely, any partial k-ary function F over A determines the k-ary relation

$$R_F \;=\; \{\langle \vec{a} \rangle \in A^k \mid \; F(\vec{a}) \text{ is defined } \}$$

A *vocabulary* is a finite list V of function-identifiers, with each $\mathbf{f}$ in V assigned an *arity* $\mathfrak{r}(\mathbf{f}) \geqslant 0$. We superscript an identifier with its arity when convenient, and refer to nullary function-identifiers as *tokens* and to unary ones as *pointers.* The distinction is fundamental, because a pointer is potentially an unbounded memory, whereas a token is not. The order of identifiers in V will matter here, but we nonetheless use the membership notation $\mathbf{f} \in V$.

A *V-structure* is a mapping σ that to each $\mathbf{f}^k \in V$, assigns a k-ary A-function $\sigma(\mathbf{f})$, said to be a *component* of σ. The *scope* of σ is the union of the scopes of its components, and the *size* $|\sigma|$ of σ is the sum of the sizes of its components. Note that if $\langle \vec{a}, b \rangle$ occurs as entry of multiple functions, then those occurrences are counted separately in $|\sigma|$.

If σ is a V-structure, and τ a W-structure where $W \supseteq V$, then we say that τ is an *expansion* of σ (to W), and that σ a *reduct* of τ (to V), if $\sigma(\mathbf{f})$ is identical to $\tau(\mathbf{f})$ for every $\mathbf{f} \in V$. Note that the scope of τ may be strictly larger than that of σ, due to the identifiers in $W - V$. We say that a V-structure σ is an *under-structure* of a V-structure τ if for every $\mathbf{f} \in V$, $\sigma(\mathbf{f}) \subseteq \tau(\mathbf{f})$; that is, every entry of $\sigma(\mathbf{f})$ is an entry of $\tau(\mathbf{f})$. Note that the definition does not require that $\sigma(\mathbf{f})$ be $\tau(\mathbf{f})$ restricted to the scope of σ, which is why we avoid the phrase "sub-structure."

If σ_i are structures for V_i ($i = 1 \ldots k$), where the V_i's are disjoint and the scopes of the $\sigma_i's$ are disjoint, then the list $\langle \sigma_1, \ldots, \sigma_k \rangle$ can be identified with the single structure $\cup_i \sigma_i$, for the concatenated vocabulary $V_1 * \cdots * V_k$.

Given a vocabulary V, the set $\mathbf{Tm}_V$ of *V-terms* is generated by $\boldsymbol{\omega} \in \mathbf{Tm}_V$; and if $\mathbf{f}^k \in V$, and $\mathbf{t}_1, \ldots, \mathbf{t}_k \in \mathbf{Tm}_V$ then $\mathbf{f}\mathbf{t}_1 \cdots \mathbf{t}_k \in \mathbf{Tm}_V$. Terms without $\boldsymbol{\omega}$ are *standard.* Note that we write function application in formal terms without parentheses and commas. We implicitly posit that the arity of a function matches the number of arguments displayed. Given a V-structure σ the *value* of a V-term $\mathbf{t}$ in σ, denoted $\sigma(\mathbf{t})$, is obtained by recurrence on $\mathbf{t}$: $\sigma(\boldsymbol{\omega}) = \perp$ and, for $\mathbf{f}^k \in V$, $\sigma(\mathbf{f}\mathbf{t}_1 \cdots \mathbf{t}_k) = \sigma(\mathbf{f})(\sigma(\mathbf{t}_1), \ldots, \sigma(\mathbf{t}_k))$

An atom $a \in A$ is *V-accessible* in σ if $a = \sigma(\mathbf{t})$ for some $\mathbf{t} \in \mathbf{Tm}_V$. A V-structure σ is *accessible* if every atom in the scope of σ is V-accessible. The *accessible under-structure* of a structure σ consists of the entries $\langle a_1 \ldots a_k, b \rangle$ where $a_1 \ldots a_k, b$ are all accessible.

If every atom in the scope of an accessible V-structure σ is the value of a *unique* V-term we say that σ is *free.* For example, every element of a free algebra is a free structure, as is any tuple of such elements.

2.2 Structure revisions

We define the following three basic transformations of V-structures. In each case we indicate how an input structure σ is transformed by the operation into a structure σ' that differs from σ only as indicated.

1. An *extension* is a phrase $\mathbf{f}\,\mathbf{t}_1 \cdots \mathbf{t}_k \downarrow \mathbf{q}$ where the $\mathbf{t}_i$'s and $\mathbf{q}$ are all standard terms. The intent is that σ' is identical to σ, except that if $\sigma(\mathbf{f}\,\mathbf{t}_1 \cdots \mathbf{t}_k) = \bot$ then $\sigma'(\mathbf{f}\,\mathbf{t}_1 \cdots \mathbf{t}_k) = \sigma(\mathbf{q})$. Thus, σ' is identical to σ if $\sigma(\mathbf{f}\,\mathbf{t}_1 \cdots \mathbf{t}_k)$ *is* defined.

2. A *contraction,* the dual of an extension, is a phrase of the form $\mathbf{f}\mathbf{t}_1 \cdots \mathbf{t}_k \uparrow$. The intent is that $\sigma'(\mathbf{f})(\sigma(\mathbf{t}_1), \ldots, \sigma(\mathbf{t}_k)) = \bot$. Note that this removes the entry $\langle \sigma(\mathbf{t}_1), \ldots, \sigma(\mathbf{t}_k), \sigma(\mathbf{f}\mathbf{t}_1 \cdots \mathbf{t}_k) \rangle$ if defined, from $\sigma(\mathbf{f})$, but not from $\sigma(\mathbf{g})$ for other identifiers $\mathbf{g}$.

3. An *inception* is a phrase of the form $\mathbf{c}\Downarrow$, where $\mathbf{c}$ is a token. A common alternative notation is $\mathbf{c} := \mathbf{new}$. The intent is that σ' is identical to σ, except that if $\sigma(\mathbf{c}) = \bot$, then $\sigma'(\mathbf{c})$ is an atom not in the scope of σ.

   We have no atom-removal operation dual to inception, since atoms can be removed from the scope of a structure by repeated contractions.

We refer to extensions, contractions, and inceptions as *revisions.* An extension or inception is *executed* if it adds an entry. That is, $\mathbf{f}\vec{\mathbf{t}} \downarrow \mathtt{b}$ executes when $\mathbf{f}\vec{\mathbf{t}} = \bot$, and similarly for an inception. The identifiers $\mathbf{c}$ and $\mathbf{f}$ in the templates above are the revision's *eigen-identifier.*

A more general form of inception, with a fresh atom assigned to an arbitrary term $\mathbf{t}$, is obtained as the composition

$$\mathbf{b}\Downarrow;\ \ \mathbf{t}\downarrow\mathbf{b};\ \ \mathbf{b}\Uparrow$$

where $\mathbf{b}$ is a fresh (and reserved) token.

An extension and a contraction can be combined into an *assignment*, i.e. a phrase of the form $\mathbf{f}\vec{\mathbf{t}} := \mathbf{q}$. This can be viewed as an abbreviation, with $\mathbf{b}$ a fresh token, of the composition

$$\mathbf{b}\downarrow\mathbf{q};\ \ \mathbf{f}\vec{\mathbf{t}}\uparrow;\ \ \mathbf{f}\vec{\mathbf{t}}\downarrow\mathbf{b};\ \ \mathbf{b}\uparrow$$

The atom $\sigma(\mathbf{q})$ is memorized here by $\mathbf{b}$, in case $\mathbf{q}$ becomes inaccessible through the contraction $\mathbf{f}\vec{\mathbf{t}}\uparrow$.

Although assignments are common and useful, we take the revisions above as our basic constructs, for two reasons. Conceptually, they are truly elemental; and concretely, the contrast between extensions and contraction is central to our characterization of complexity classes.

2.3 STV programs

Fix a vocabulary V. A *V-equation* is a phrase $\mathbf{t} \simeq \mathbf{q}$ where $\mathbf{t}$ and $\mathbf{q}$ are V-terms, intended to state that $\mathbf{t}$ and $\mathbf{q}$ are equal in $A_\bot$ (i.e. both undefined or both defined

and equal). A V-*guard* is a boolean combination of V-equations. We write $!\,\mathbf{t}$ for the guard $\mathbf{t} \not\simeq \boldsymbol{\omega}$.[4]

A V-*variant* is a finite set T of identifiers in V, to which we refer as T's *components.*

Our intent is to bound iteration via variant depletion. This can be achieved simply by requiring that the total size of the variant is reduced with each iteration cycle, much like the traditional function-variants [9, 6, 21]. However, we seek syntactic conditions that guarantee such a behavior, or at least semantic conditions that can be easily enforced by syntactic flags. We consider separately the non-increasing of a variant, and its decrease. Non-increasing of a variant T can be enforced by prohibiting expansions of T's components within the loop body; this is a syntactic condition that applies to the body as a whole. We enforce variant *depletion* by halting iteration if variant depletion does not occur; this is a semantic condition that applies locally, and can be enforced by conjoining the guard with an appropriate boolean flag.

Formally, the programs of **STV** are generated inductively in tandem with the syntactic notion of a variant being *non-inflating* in a program.

1. A revision is a program. A variant T is *non-inflating* in a revision unless it is an extension whose eigen-function is in T.

2. If P and Q are **STV**-programs, and T non-inflating in both, then so is P; Q.

3. If G is a guard and P, Q are **STV**-programs, and T is non-inflating in both, then so is $\mathbf{if}\,[G]\,\{P\}\,\{Q\}$.

4. If G is a guard, P is an **STV**-program, and S, T are non-inflating in P, then $\mathbf{do}\,[G]\,[T]\,\{P\}$ is an **STV**-program, in which S is non-inflating.

The formal denotational semantics of programs is defined as a binary *yield relation* $\Rightarrow_P$ between V-structures by recurrence on the syntax of a program P. It is routine (see [16]), except for loops. If P is $\mathbf{do}\,[G]\,[T]\,\{Q\}$, then $\sigma \Rightarrow_P \tau$ if for some $k \geqslant 0$ one of the two options below holds. Let us write $\xi \stackrel{-}{\Rightarrow} \xi'$ [respectively $\xi \stackrel{0}{\Rightarrow} \xi'$] for the conjunction of $\xi \models G$, $\xi \stackrel{0}{\Rightarrow} \xi'$, and the condition that $\xi \Rightarrow_Q \xi'$ executes [respectively, fails to execute] a contraction of T.

1. $\sigma = \sigma_0 \stackrel{-}{\Rightarrow} \sigma_1 \stackrel{-}{\Rightarrow} \cdots \stackrel{-}{\Rightarrow} \sigma_k = \tau$, where $\tau \not\models G$; or

2. $\sigma = \sigma_0 \stackrel{-}{\Rightarrow} \sigma_1 \stackrel{-}{\Rightarrow} \cdots \stackrel{-}{\Rightarrow} \sigma_k \stackrel{0}{\Rightarrow} \sigma_{k+1} = \tau$

That is, a loop $\mathbf{do}\,[G]\,[T]\,\{Q\}$ is entered if G is true in the current V-structure, and is re-entered if G is true in the current V-structure, *and* the previous pass executes at least one contraction for some component of the variant T. Thus, as $\mathbf{do}\,[G][T]\,\{Q\}$ is executed, T grows within P, by the syntactic condition that T is non-inflating in P, and is decreased by a contraction at least once for each iteration, save the last, by the semantic condition on loop execution. When no variant component shrinks within a pass through P, the execution of the loop is terminated.

Note that the depletion condition we impose on loops can be conveyed by a built-in syntactic controller: for each variant T take a fresh token c_T to serve as a "controller"

[4] The notations $\simeq$ and ! are due to Kleene [15].

for T. Each loop $\mathbf{do}[G][T]\{P\}$ is preceded by $\mathtt{c}_T \Downarrow$, the test $!\mathtt{c}_T$ is conjuncted to the guard G, and P is preceded by the contraction $\mathtt{c}_T \uparrow$. Finally, each revision in P, whose eigen-function is in T, is coupled with an inception $\mathtt{c}_T \Downarrow$.

A V-structure is *accepted* by an **STV**-program P if P terminates for σ as input. A class $\mathfrak{C}$ of V-structures is *recognized* by P if $\mathfrak{C}$ consists of the V-structures accepted by P.

We say that a k-ary relation between structures is *accepted* by P if P terminates for input $\langle \sigma_1, \ldots, \sigma_k \rangle$; and P *recognizes* a class $\mathfrak{C}$ of k-ary relations between structures as above if $\mathfrak{C}$ consists of the k-tuples accepted by P.

We define as well the interpretation of programs as transducers, as follows. Let $\Phi : \mathfrak{C} \rightharpoonup \mathfrak{C}'$ be a partial-mapping from a class $\mathfrak{C}$ of V-structures to a class $\mathfrak{C}'$ of V'-structures. A W-program P *computes* Φ if for every $\sigma \in \mathfrak{C}$, $\sigma^W \Rightarrow_P \mathcal{Q}$ for some W-expansion $\mathcal{Q}$ of $\Phi(\sigma)$. Note that the vocabulary V' of the output structure need not be related to the input vocabulary V.[5]

We shall focus mostly on programs as transducers. Note that all structure revisions refer only to accessible structure nodes. It follows that non-accessible nodes play no role in the computational behavior of **STV**-programs.

Note that the depletion condition we impose on loops can be conveyed syntactically, as follows. For each loop L present, say an instance of the program $P = \mathbf{do}[G][T]Q$, let $\mathtt{c}_L$ be a reserved token, to serve as a toggle for the depletion of L's variant.

- Precede L by $\mathtt{c}_L \Downarrow$.
- Conjunct the test $!\,\mathtt{c}_L$ to G.
- Precede Q by $\mathtt{c}_L \uparrow$.
- Replace each contraction $\vec{\mathbf{ft}} \uparrow$ in Q, where $\mathbf{f} \in T$, by $\mathtt{c}_L \downarrow \vec{\mathbf{ft}}; \vec{\mathbf{ft}} \uparrow$.

For a program P over V we define the binary *yield relation* $\Rightarrow_P$ between V-structures by recurrence on the syntax of P. When P is a revision the definition follows the semantics given above.

3 Examples of STV programs

3.1 String duplication

The following program duplicates a structure σ representing a binary string; that is, the output structure has the same scope as the input, but with functions appearing in duplicate. The algorithm has two phases: a first loop, whose variant consists of all pointers in V, creates two new copies of the string, while depleting the input functions. A second loop restores one of the two copies to the original identifiers, thereby allowing the duplication to be useful within a larger program that refers to those original identifiers.

[5] Of course, if $\mathfrak{C}$ is a proper class (in the sense of Gödel-Bernays set theory), then the mapping defined by P is a proper-class.

```
a := e;
do [!0a ∨ !1a] [0, 1]                          % 0/1 copied to 0̄/1̄ and 0̂/1̂
    { b ↓ a;                                   %    while being consumed (via b) as variant
    if [ !0a ]
     { 0̄(a) ↓ 0a; 0̂(a) ↓ 0a; a ↓ 0a; 0b ↑ }
     { 1̄(a) ↓ 1a; 1̂(a) ↓ 1a; a ↓ 1a; 1b ↑ }
    };
a := e;                                        % 0̂/1̂ restored to 0/1
do [ !0̂a ∨ !1̂a ] [ 0̂, 1̂ ]
    {if [ !0̂a ]
      { 0a↓0̂a; 0̂a↑; a ↓ 0a; }
      { 1a↓1̂a; 1̂a↑; a ↓ 1a; }
    }
```

3.2 Generating large output

Let $V = \{\mathtt{z}^0, \mathtt{s}^1\}$ be the vocabulary for the natural-number structures, i.e. the free structures for the terms $\mathtt{s}^{[n]}\mathtt{z}$. The addition of V-structures (z_0, s_0) and (z_1, s_1), representing natural numbers n_0, n_1, is computed by an **STV** program that duplicates the second input, and uses one of the two copies as a loop variant for splicing the other copy over the first input.

A program for multiplication is obtained by duplicating the second input, initializing the output to z, and then using the first input as variant of a loop whose body splices the second argument on the output-so-far.

A program E for *exponentiation,* transforming the structure $\mathtt{s}^{[n]}\mathtt{z}$ to the structure $\mathtt{s}^{2^n}\mathtt{z}$, is constructed similarly to the multiplication program above, except that the output is initialized to the structure for sz. and the loop's body duplicates the output-so-far and adds up the two copies.

3.3 Enumerators

A pair (a, e), with $a \in A$ and $e : A \to A$, is an *enumerator* for a V-structure σ if for some n the sequence $a, e(a), e(e(a)), \ldots, e^{[n]}(a)$ consists of all *accessible* atoms of σ, and $e^{[n+1]}(a) = \bot$.

The following program L builds, in each V-structure σ taken as input, an enumerator (a, e) for σ. That is, for some fresh identifiers $\mathtt{a}^0$, $\mathtt{e}^1$, the output τ of L for input σ is an expansion τ of σ with an enumerator $(\tau(\mathtt{a}), \tau(\mathtt{e}))$ for σ (whence for τ as well). L initializes e to a list of the atoms denoted by V's tokens. L's main loop, with body C, collects new accessible elements into an auxiliary unary pointer p, used as a cache and re-initialized to empty at the start of C. For each $\mathbf{f}^k \in V$ in turn, C creates k new copies of p. Using the set of these copies as a variant, C then cycles through k-tuples

$\vec{a}$ of elements in $\mathtt{p}$ (using auxiliary tokens) and appends $\sigma(\mathtt{f})\vec{a}$ to $\mathtt{p}$ if it is not already in $\mathtt{p}$. When this process is completed for all $\mathbf{f} \in V$, C concatenates $\mathtt{p}$ to $\mathtt{e}$. The loop is exited by the depletion condition on the semantics, when the cache $\mathtt{p}$ remains empty at the end of C, i.e. when no new atom has been found.

Note that if the input structure σ is free, then the construction above clearly yields an enumerator $\mathtt{e}$ that is *monotone,* in the following sense: for each term $\mathbf{q} = \mathbf{f}^k\mathbf{t}_1 \cdots \mathbf{t}_k$ the enumerator lists $\mathbf{t}_i$ before $\mathbf{q}$.

3.4 Duplicating the accessible under-structure

The program above for string duplication implicitly relies on the presence of a trivial enumerator for the string-structure. Using the program L above for constructing an enumerator for all V-structures, we can now outline a program that duplicates the accessible under-structure of any V-structure. Thus, the program duplicates completely any accessible V-structure.

A program D_m to create for each pointer $\mathtt{f} \in V$ (of any arity) m copies $\mathtt{f}_1 \ldots \mathtt{f}_m$ of $\mathtt{f}$ (over the same atoms as $\mathtt{f}$) can be obtained as follows. D_m first constructs an enumerator $(\mathtt{a}, \mathtt{e})$ for the input structure. Recall that the identifiers $\mathtt{f}_1 \ldots \mathtt{f}_m$ for the duplicates to be created are all initially empty, by our semantic conventions.

For each of the (finitely many) identifiers $\mathtt{f}^k \in V$ in turn, D_m then creates k copies of $\mathtt{e}$, and uses them to cycle through all k-tuples $\vec{a}$ of accessible atoms in σ, extending each $\mathtt{f}_i$ with the entry $\langle \vec{a}, \sigma(\mathtt{f})\vec{a} \rangle$. The k copies of $\mathtt{e}$ are also used collectively as the loop's variant. The loop ends when the variant is depleted, leaving no unchecked tuple $\vec{a}$. Note that the original enumerator $\mathtt{e}$ is left alone during the process, remaining available for the program segment dealing with the next pointer in V.

3.5 Quasi-inverses

In inductive data the constructors are injective, but a lax form of function-inversion, namely *quasi-inversion*, can be defined for arbitrary functions, as follows.[6] For a relation $R \subseteq A \times B$ and $a \in A$, define $R'a =_{\mathrm{df}} \{b \in B \mid aRb\}$.[7] Say that a partial-function $f : A \rightharpoonup B$ is a *choice-function for* R if $f \subseteq R$ and $f(a)$ is defined whenever $R'a \neq \emptyset$. A partial-function $g : A \rightharpoonup B$ is a *quasi-inverse* of f if it is a choice function for the relation f^{-1}. When f is r-ary, i.e. $A = \times_{i=1}^{r} A_i$, g can be construed as an r-tuple of partial-functions $\langle g_1 \ldots g_r \rangle$. We write f^{-i} for g_i. Evidently, if a unary function f is injective then its (unique) quasi-inverse is the usual inverse f^{-1}.

Let V be a vocabulary. We construct an **STV**-program J that for each V-structure σ as input yields an expansion of σ with quasi-inverses for the accessible portion of each non-nullary $\sigma(\mathbf{f})$, $\mathbf{f} \in V$. J is similar to the program D above for duplicating the accessible portion of all functions. However, whereas D examines all entries $\langle \vec{a}, \sigma(\mathtt{f})\vec{a} \rangle$, and extends $\mathtt{f}_1, \ldots, \mathtt{f}_m$ whenever that entry is defined, J extends, for $i = 1 \ldots k$, the function $\mathtt{f}^{-i}$ with the entry $\langle \sigma(\mathtt{f})\vec{a}, a_i \rangle$.

[6] Quasi-inverses are often defined algebraically: g is a quasi-inverse of f when $f \circ g \circ f = f$.

[7] We use infix notation for binary relations.

For a vocabulary V we can now easily define, using quasi-inverses, a program *Sub* over V that maps any free accessible structure σ, and atom $y = \sigma\mathbf{t}$ in the scope of σ, to the restriction of σ to atoms denoted by sub-terms of $\mathbf{t}$.

4 The abstract core of primitive recursion

4.1 Soundness of STV-programs for PR

Recall (§2.1) that the *size* $|\sigma|$ of a structure σ is the sum of sizes of its components. In fact this is in tune with our use of variants, which are consumed by eliminating function entries, not atoms. Moreover, the size of functions seems to be an appropriate measure in general, since it conveys the information contents of a structure more faithfully than the number of atoms.

Note that for word-structures, i.e. $\sigma(w)$ for $w \in \Sigma^*$ (Σ an alphabet) the total size of the structure's functions is precisely the length of w, so in this important case our measure is identical to the count of atoms.

We say that a program P runs *within time* $t : \mathbb{N} \to \mathbb{N}$ if for all structures σ, the number of configurations in the execution trace of P on input σ is finite and $\leqslant t(|\sigma|)$. P runs *within space* $s : \mathbb{N} \to \mathbb{N}$ if for all σ, all configurations in the execution trace of P on input σ are of size $\leqslant s(|\sigma|)$. We say that P *runs in PR* if it runs within time t, for some PR function t. This is trivially equivalent to P running in PSpace, since s cannot exceed t, t cannot exceed $2^{O(s)}$, and PR is closed under exponentiation.

We assign to each **STV**-program P a primitive-recursive function $b_P : \mathbb{N} \to \mathbb{N}$ as follows.

- If P is an extension then $b_P(n) = 1$; if P is a contraction or an inception then $b_P(n) = 0$.
- If P is $S; Q$ then $b_P(n) = b_Q(b_S(n))$
- If P is $\mathbf{if}[G]\{S\}\{Q\}$ then $b_P(n) = \max[b_S(n), b_Q(n)]$.
- If P is $\mathbf{do}[G][T]\{Q\}$ then $b_P(n) = b_Q^{[n]}(n)$.

LEMMA **1** *If P is an* **STV***-program computing a mapping Φ_P between structures, then for every structure σ*

$$|\Phi_P(\sigma)| \leqslant b_P(|\sigma|)$$

PROOF. Structural induction on P.

- If P is a revision, then the claim is immediate by the definition of b_P.
- If P is $S; Q$ then

$$\begin{array}{rcll} |\Phi_P(\sigma)| & = & |\Phi_Q(\Phi_S(\sigma))| & \\ & \leqslant & b_Q(|\Phi_S(\sigma)|) & \text{(IH for } Q\text{)} \\ & \leqslant & b_Q(b_S(|\sigma|)) & \text{(IH for } S\text{, } b_Q \text{ is non-decreasing)} \\ & = & b_P(|\sigma|) & \end{array}$$

- The case for P of the form $\mathbf{if}[G]\{S\}\{Q\}$ is immediate.
- If P is $\mathbf{do}[G][T]\{Q\}$ then $\Phi_P(\sigma)$ is $\Phi_Q^{[m]}(\sigma)$ for some m. By the definition of variants, and the semantics of looping, m is bounded by the size of T, which is bounded by $|\sigma|$. So

$$\begin{array}{rcll} |\Phi_P(\sigma)| & = & |\Phi_Q^{[m]}(\sigma)| & \text{for some } m \leqslant |\sigma| \\ & \leqslant & b_Q^{[m]}(|\sigma|) & \text{IH, } b_Q \text{ is non-decreasing} \\ & \leqslant & b_Q^{[n]}(|\sigma|) & \text{where } n = |\sigma| \text{ by the comment above,} \\ & & & \quad \text{since } b_Q \text{ is non-decreasing} \\ & = & b_P(|\sigma|) & \text{by the dfn of } b_P \end{array}$$

□

From Lemma 1 we obtain the soundness of **STV**-programs for PR:

THEOREM **2** *Every* **STV***-program runs in PR space, and therefore in PR time.*

4.2 Completeness of STV-programs for PR

Turning to the completeness of **STV** for primitive recursion, we could prove that **STV** is complete for $\mathbf{PR}(\mathbb{N})$, and invoke the coding of primitive recurrence over any free algebra in $\mathbf{PR}(\mathbb{N})$. This, however, would fail to establish a direct representation of generic recurrence by **STV**-programs, which is one of the *raisons d'être* of **STV**. We show instead that, for any free algebra $\mathbb{A}$, every function primitive-recursive f over $\mathbb{A}$ is computed by an **STV**-program that conveys directly the PR definition of f.

For a free algebra $\mathbb{A} = \mathbb{A}(C)$, and an element $a \in \mathbb{A}$, let σ_a be a given as a C-structure.

LEMMA **3** *For each free algebra $\mathbb{A} = \mathbb{A}(C)$ and each instance of the schema (2) above of recurrence over $\mathbb{A}$ (with $\vec{x} = x_1, \ldots x_m$), the following holds. Given* **STV***-programs for the functions $g_{\mathtt{c}}$, there is an* **STV***-program P that, for each $y, x_1, \ldots, x_m \in \mathbb{A}$, maps the structure $\langle \sigma_y, \sigma_{x_1}, \ldots, \sigma_{x_m} \rangle$ to σ_t where $t = f(y, x_1, \ldots, x_m)$.*

PROOF. Assume that $g_{\mathtt{c}}$ in (2) is computed by an **STV**-program P_c, for each $\mathtt{c} \in C$. Our program P builds up a unary pointer $\mathtt{r}$ that maps each atom in the structure σ_y for y to the structure σ_t where $t = \mathbf{f}(\mathbf{q}, x_1, \ldots, x_m)$. I.e. for each sub-term-occurrence $\mathbf{p}$ of y $\sigma\mathtt{r}(\mathtt{sp}) = \sigma(\mathbf{f}(\mathbf{q}, x_1, \ldots, x_m))$.

P starts by invoking the program L above to expand $\sigma(y)$ with an enumerator $\mathtt{e}$ and quasi-inverses for each $\mathtt{c} \in C$. Note that since σ_y is a free structure, each quasi-inverse is an inverse; also, $\mathtt{e}$ never lists $\sigma\mathbf{q}$ before listing $\sigma\mathbf{p}$ for a sub-term $\mathbf{p}$ of $\mathbf{q}$.

P's main loop examines the atoms listed by $\mathtt{e}$, using $\mathtt{e}$ itself as variant (after saving a copy). For each $a = \sigma\mathbf{q}$ listed, the constructor-inverses are used to identify the main

constructor of $\mathbf{q}$, say $\mathtt{c}$ of arity k, as well as the atoms denoted by $\mathbf{q}$'s immediate-sub-terms, namely $b_1 = \mathtt{c}^{-1}a,\ \ldots\ b_k = \mathtt{c}^{-k}a$. P then invokes P_c for the following structure as input, where β_i is a copy of the structure for the sub-term of $\mathbf{q}$ rooted at b_i, obtained by the program *Abs* of §3.5.

$$\langle \beta_1, \ldots, \beta_k, \sigma_{x_1}, \ldots, \sigma_{x_m}, \mathtt{r}(\mathtt{c}^{-1}a), \ldots, \mathtt{r}(\mathtt{c}^{-k}a)\rangle$$

It then defines $\mathtt{r}(a)$ to be the root of the output structure of P_c.

By our definition of the enumerator, its last entry is the recurrence argument $y = \sigma\mathbf{q}$ itself, and by the definition of P, $\mathtt{r}(y)$ is the root of $\sigma(f(y, x_1, \ldots, x_m))$. The last phase of P uses contractions to eliminate all atoms and entries other than that substructure, leaving as output σ_t where $t = f(y, x_1, \ldots, x_m)$. □

THEOREM **4** *For each free algebra $\mathbb{A}$, the collection of* **STV***-programs is complete for* $\mathbf{PR}(\mathbb{A})$.

PROOF. Let $f \in \mathbf{PR}(\mathbb{A})$. We show that f is computable in **STV** by discourse-level induction on the definition of f as a PR function over $\mathbb{A}$. The cases where f is a constructor are trivial. For explicit definitions, and more particularly composition, we need to address the need of duplicating substructures, which is obtained by the duplication program of 3.4. Finally, the case of recurrence is treated in Lemma 3. □

4.3 Completeness of STV for PR-bounded ST-programs

Theorem 4 establishes, for any free algebra $\mathbb{A}$ a simple and direct mapping from definitions in $\mathbf{PR}(\mathbb{A})$ to **STV**-programs, If we take the **ST**-programs of [16] as the Turing-complete computation model of reference, the question remains as to whether every **ST**-program P running within primitive-recursive time and space is directly mapped to an equivalent **STV**-program Q. Indeed, it suffices to take Q of the form $E; B; P'$, where:

1. E expands σ with an enumerator for σ. (Recall that an enumerator for a V-structure σ generates the V-accessible elements of σ.)
2. B invokes Theorem 4 for $\mathbb{A} = \mathbb{N}$, using E as input, to compute the function f, i.e. further expanding σ with (b^0, t^1) representing a chain of length $f(n)$, where n is the size of σ.
3. P' is P with each loop preceded with duplicating $\mathtt{t}$, and using the copy as variant for the loop.

5 Conclusion

We've followed here [16], where we introduced programming over finite partial-structures as an approach for the analysis and certification of resources in an abstract setting.

The key insight is that inductive data-objects, such as natural numbers, strings and lists, can be construed as finite partial-structures, and as such are amenable to programming for the transformation of finite partial-structures. We showed there that the underlying theory of finite partial-structure is mutually interpretable with Peano Arithmetic, and noted that the corresponding programming language **ST**, for finite partial-structure transformation, is Turing complete.

Here we presented a variation **STV** of **ST**, that requires each loop to be assigned a "variant" in the guise of a set of the structure's components, with each pass through the loop consuming at least one variant's entry. We showed that this generic construct yields an abstract delineation of primitive recursive computing: On the one hand recurrence over any free algebra is captured directly in **STV**, and on the other hand any function computed by **STV** programs is primitive recursively bounded, and is therefore primitive recursive.

References

[1] Philippe Andary, Bruno Patrou, and Pierre Valarcher. About implementation of primitive recursive algorithms. In Danile Beauquier, Egon Brger, and Anatol Slissenko, editors, *Proceedings of the 12th International Workshop on Abstract State Machines*, pages 77–90, 2005.

[2] Philippe Andary, Bruno Patrou, and Pierre Valarcher. A representation theorem for primitive recursive algorithms. *Fundam. Inform.*, 107(4):313–330, 2011.

[3] Lenore Blum, Mike Shub, and Steve Smale. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bulletin of the American Mathematical Society*, 21:1–46, 1989.

[4] Egon Börger. The origins and the development of the ASM method for high level system design and analysis. *J. UCS*, 8(1):2–74, 2002.

[5] Olivier Bournez, Felipe Cucker, Paulin Jacobé de Naurois, and Jean-Yves Marion. Computability over an arbitrary structure. sequential and parallel polynomial time. In *Foundations of Software Science and Computational Structures*, pages 185–199, 2003.

[6] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[7] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer-Verlag, Berlin, 1995.

[8] Erich Grädel and Yuri Gurevich. Metafinite model theory. In Daniel Leivant, editor, *Logic and Computational Complexity*, volume 960 of *Lecture Notes in Computer Science*, pages 313–366. Springer, 1995.

[9] David Gries. *The Science of Programming.* Texts and Monographs in Computer Science. Springer, 1981.

[10] Yuri Gurevich. Logic in computer science column. *Bulletin of the EATCS*, 35:71–81, 1988.

[11] Yuri Gurevich. Evolving algebras: an attempt to discover semantics. In Grzegorz Rozenberg and Arto Salomaa, editors, *Current Trends in Theoretical Computer Science*, volume 40, pages 266–292. World Scientific, 1993.

[12] Yuri Gurevich. The sequential ASM thesis. In *Current Trends in Theoretical Computer Science*, pages 363–392. World Scientific, 2001.

[13] Juris Hartmanis. On non-determinancy in simple computing devices. *Acta Inf.*, 1:336–344, 1972.

[14] Jean van Heijenoort. *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press, Cambridge, MA, 1967.

[15] S.C. Kleene. *Formalized Recursive Functionals and Formalized Realizability*. Memoirs of the AMS. American Mathematical Society, Providence, RI, 1969.

[16] Daniel Leivant. A theory of finite structures. *CoRR*, abs/1808.04949, 2018.

[17] Rosza Peter. *Rekursive Funktionen*. Akadémia Kiadó, Budapest, 1951.

[18] Vladimir Yu. Sazonov. Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik*, 16(7):319–323, 1980.

[19] Thoralf Skolem. Einige bemerkungen zur axiomatischen begründung der mengenlehre. In *Matematikerkongressen in Helsingfors Den femte skandinaviske matematikerkongressen, 1922* [14], pages 217–232. English translation in [14].

[20] Thomas Strahm and Jeffery I. Zucker. Primitive recursive selection functions for existential assertions over abstract algebras. *J. Log. Algebr. Program.*, 76(2):175–197, 2008.

[21] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA, 1993.