Mathematical Structures in Computer Science (2020), **30**, pp. 33–43
doi:10.1017/S0960129519000112

PAPER

Primitive recursion in the abstract

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

[1]Indiana University and Université Paris-Diderot and [2]Université de Lorraine, CNRS and LORIA
*Corresponding author. Email: leivant@indiana.edu

(Received 18 November 2018; accepted 14 April 2019; first published online 21 January 2020)

Abstract
Recurrence can be used as a function definition schema for any nontrivial 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.

Keywords: Program termination; implicit computational complexity; primitive recursion; program variants; finite structures; structure transformation

1. Introduction

1.1 Abstract delineation of PR

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

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

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

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

where $f_i = f(z_i,\vec{x})$ and k is the arity of c.

The recurrence schema for $\mathbb{N}$ originates with Dedekind's interest in formalizing arithmetic, articulated by Skolem (1923), and studied extensively (see, e.g., Peter (1951)). The set primitive-recursive $\mathbf{PR}(\mathbb{A})$ of *primitive-recursive functions* over $\mathbb{A}$ is generated from the constructors of $\mathbb{A}$ by recurrence over $\mathbb{A}$ and explicit definitions.[1]

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.

1.2 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:

$$\begin{tikzcd} \mathtt{e}\ \circ \arrow[r, "\mathbf{0}"] & \circ \arrow[r, "\mathbf{1}"] & \circ \arrow[r, "\mathbf{1}"] & \circ \end{tikzcd}$$

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 (Dijkstra 1976; Gries 1981; Winskel 1993), but whereas the latter decrease along a prescribed well ordering, our variants are depleted by function-contractions, that is, reassigning a defined value of a function to *undefined*. The distinction between positive and negative forms of assignment is thus fundamental in our approach.

1.3 Main results

We refer to a basic imperative programming language **STV** for the transformation of structures. We focus on finite partial-structures, following the approach of Leivant (2018) and the imperative language **ST** defined there. **ST**, which is a variant of Gurevich's abstract state machines (ASMs) (Börger 2002; Gurevich 1993, 2001), focuses on finite structures, and also supports computing over infinite data-structures, such as free algebras, once their elements are construed as finite structures. **ST** is Turing complete and is 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 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 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 **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 (Hartmanis 1972; Sazonov 1980), 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 (Andary et al. 2005, 2011; Strahm and Zucker 2008), but always in the traditional sense of computing within an infinite structure. Also, while the meta-finite structures of (Grädel and Gurevich, 1995) 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 Sazonov (1980)), and are thus unrelated to the global functions of Gurevich (1988) and Ebbinghaus and Flum (1995), which are (isomorphism-invariant) mappings that assign to each structure a function over it.

As for generalizations of primitive recursion to computing over abstract structures, Bournez et al. (2003) refer to the computation model of Blum et al. (1989), which incorporates primitive recursion from the outset, and therefore does not examine the abstract contents of **PR** as we do here.

This paper is sectioned along the outline above: Section 2 defines the programming language **STV**, and Section 3 gives examples of programs, most of which we use in the sequel. In Section 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 Leivant (2018) 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 variation of Gurevich's ASMs (Börger 2002; Gurevich 1993, 2001), one that focuses on finite structures, distinguishes between constructive and destructive assignments, and treats iteration as a core concept. (ASMs strive to generalize directly hardware models and consequently refer to a single loop iterating an entire program.)

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 Leivant (2018) 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 the non-denoting terms, we extend A to a flat domain $A_\perp$, which has, in addition to the atoms, a fresh object $\perp$, the "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 $\langle a_1 \ldots a_k, b\rangle$, where $b = F(a_1, \ldots, a_k) \neq \perp$. 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 the ones of arity > 0 as *pointers*. This 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), 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 $\tau(\mathbf{f})$.

If σ_i is a V_i-structure ($i = 1 \ldots k$), where the V_i's are disjoint and the scopes of the σ_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 $V_1 * \cdots * V_k$.

Fix a reserved identifier $\boldsymbol{\omega}$, intended to denote $\perp$. 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 $\sigma(\mathbf{f}\,\mathbf{t}_1 \cdots \mathbf{t}_k) = \perp$, 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 $\sigma'(\mathbf{f})(\sigma(\mathbf{t}_1), \ldots, \sigma(\mathbf{t}_k)) = \perp$. 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 $\sigma(\mathbf{c}) = \perp$, 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, $\mathbf{f}\vec{\mathbf{t}}\downarrow \mathrm{b}$ is executed when $\mathbf{f}\vec{\mathbf{t}} = \perp$, 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 token.

An extension and a contraction can be combined into an *assignment*, that is, 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 $\vec{\mathbf{ft}}\uparrow$.

Although assignments are common and useful, we take the revisions above as our basic constructs, because they are truly elemental, and the contrast between extensions and contractions 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 $A_\perp$ (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 $\mathbf{t} \not\simeq \boldsymbol{\omega}$.[2]

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 (Dijkstra 1976; Gries 1981; Winskel 1993). 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-increase of a variant, and its decrease. Non-increase of a variant T can be enforced by prohibiting extensions 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.

Formally, the programs of **STV** are generated by:

(1) A revision is a program.
(2) If P and Q are **STV**-programs, then so is $P; Q$.
(3) If G is a guard and P, Q are **STV**-programs, then so is **if** $[G]\,\{P\}\,\{Q\}$.
(4) If G is a guard, T a variant, and Q an **STV**-program with no extension whose eigen-identifier is in T, then **do** $[G]\,[T]\,\{Q\}$ is an **STV**-program.

The semantics of **STV** is defined as a binary *yield relation* $\Rightarrow_P$ between V-structures by recurrence on the syntax of the program P. The definition is obvious, except for iteration, which proceeds unless the guard fails, or the variant fails to be depleted, in the following sense. If P is **do** $[G]\,[T]\,\{Q\}$, let us write $\xi \overset{-}{\Rightarrow} \xi'$ [respectively $\xi \overset{0}{\Rightarrow} \xi'$] for the conjunction of $\xi \models G$, $\xi \Rightarrow_Q \xi'$, and the condition that $\xi \Rightarrow_Q \xi'$ executes [respectively, fails to execute] a contraction of T. Then $\sigma \Rightarrow_P \tau$ if for some $k \geqslant 0$ one of the following holds.

(1) $\sigma = \sigma_0 \overset{-}{\Rightarrow} \sigma_1 \overset{-}{\Rightarrow} \cdots \overset{-}{\Rightarrow} \sigma_k = \tau$, where $\tau \not\models G$; or
(2) $\sigma = \sigma_0 \overset{-}{\Rightarrow} \sigma_1 \overset{-}{\Rightarrow} \cdots \overset{-}{\Rightarrow} \sigma_k \overset{0}{\Rightarrow} \sigma_{k+1} = \tau$

Thus, **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. As **do** $[G][T]\,\{Q\}$ is executed, T is not extended, by the syntax of looping, and is subject to at least one contraction at least once for each iteration, save the last, by the semantic condition on loop execution.

Remarks

(1) 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 c_L be a reserved token, to serve as a toggle for the depletion of L's variant.
– Precede L by $\mathsf{c}_L\Downarrow$.

  - 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 $\mathtt{c}_L\downarrow\vec{\mathbf{ft}};\vec{\mathbf{ft}}\uparrow$.

(2) All structure revisions refer only to accessible structure nodes. It follows that non-accessible atoms play no role in the computational behavior of **STV**-programs.

We shall focus mostly on programs as transducers. 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.[3]

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.

```
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, that is, 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 for *exponentiation*, transforming each structure $\mathtt{s}^{[n]}\mathtt{z}$ to the structure $\mathtt{s}^{2^n}\mathtt{z}$, is constructed similar 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

Let σ be a V-structure. A pair (a, e), with $a \in A$, and e a partial unary function over A, is an *enumerator* for 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 $e^{[n+1]}(a) = \perp$.

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

Clearly, if the input structure σ is free, then the construction above yields an enumerator 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.

We first define, for $m > 0$, a program D_m that yields for each pointer $\mathtt{f} \in V$ m copies $\mathtt{f}_1 \ldots \mathtt{f}_m$ of f (over the same atoms as f). To begin, D_m constructs an enumerator (a, 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 eand 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 e are also used collectively as the loop's variant. The loop is exited when the variant is depleted, leaving no unchecked tuple $\vec{a}$. Note that the original enumerator 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, and therefore have inverses. A lax form of function-inversion, namely *quasi-inversion*, can be defined for arbitrary functions, as follows.[4] For a relation $R \subseteq A \times B$ and $a \in A$, define $R'a =_{\text{df}} \{b \in B \mid aRb\}$.[5] 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, that is, $A = \times_{i=1}^{r} A_i$, 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 $\langle \sigma(\mathtt{f})\vec{a}, a_i \rangle$.

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. Soundness and Completeness of STV for PR

4.1 Soundness of STV*-programs for* PR

Recall (Section 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, that is, $\sigma(w)$ 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 PR space, 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 **if**$[G]\{S\}\{Q\}$, then $b_P(n) = \max\,[b_S(n), b_Q(n)]$.
- If P is **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

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

- The case for $P \equiv$ **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

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

□

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 **PR**($\mathbb{N}$), and invoke the numeric coding of any free algebra. 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**. Consequently, we show for any free algebra $\mathbb{A}$that every $f \in$ **PR**($\mathbb{A}$) 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 $\mathbb{A}$-recurrence schema (2), the following holds. Given* **STV***-programs for the functions g_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_c in (2) is computed by an **STV**-program P_c, for each $\mathtt{c} \in C$.

Since $y \in \mathbb{A}$, each atom in σ_y is the root of σ_z for some sub-term z of y. Our program P builds up a unary pointer $\mathtt{r}$ that maps each such z to the structure for $f(z, x_1, \ldots, x_m)$.

P starts by invoking programs L and J above to expand σ_y with an enumerator $\mathtt{e}$ and quasi-inverses for each $\mathtt{c} \in C$. Since σ_y is free, each such quasi-inverse is an inverse, and $\mathtt{e}$ lists any sub-term w of z before z.

P's main loop examines the atoms listed by $\mathtt{e}$, using $\mathtt{e}$ itself as variant (after saving a copy). For each z listed, the constructor-inverses are used to identify the main constructor of z, say $\mathtt{c}$ of arity k, as well as the the immediate sub-terms of z, namely $w_1 = \mathtt{c}^{-1}z, \ \ldots \ w_k = \mathtt{c}^{-k}z$. P then invokes P_c for the structure

$$\langle \sigma_{w_1}, \ \ldots, \ \sigma_{w_k}, \ \sigma_{x_1}, \ \ldots, \ \sigma_{x_m}, \ \mathtt{r}(w_1), \ \ldots, \mathtt{r}(w_k) \rangle$$

as input, and sets $\mathtt{r}(z)$ to be the root of P_c's output. By our definition of the enumerator, its last entry is the recurrence argument y itself, and by the definition of P, $\mathtt{r}(y)$ is the root of σ_t.

The last phase of P uses contractions to eliminate all atoms and entries other than σ_t. □

Theorem 4. *For each free algebra $\mathbb{A}$, the collection of* **STV***-programs is complete for* **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, the duplication program of Section 3.4 takes care of structure duplication. 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 (Leivant, 2018) as the Turing-complete computation model of reference, the question remains as to whether every **ST**-program P running within primitive-recursive resources 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.
(2) B invokes Theorem 4 for $\mathbb{A} = \mathbb{N}$, using E as input, to compute the function f, that is, 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 `t`, and using the copy as variant for the loop.

5. Conclusion

We have followed here Leivant (2018), where we introduced programming over finite partial-structures as an approach for the analysis and certification of resources in an abstract setting (Heijenoort, 1967). 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.

Here we presented a variation **STV** of **ST**, which 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.

Notes

1 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."
2 The notations $\simeq$ and ! are due to Kleene (1969).
3 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.
4 Quasi-inverses are often defined algebraically: g is a quasi-inverse of f when $f \circ g \circ f = f$.
5 We use infix notation for binary relations.

References

Andary, P., Patrou, B. and Valarcher, P. (2005). About implementation of primitive recursive algorithms. In: Beauquier, D., Börger, E. and Slissenko, A. (eds.) *Proceedings of the 12th International Workshop on Abstract State Machines*, 77–90.
Andary, P., Patrou, B. and Valarcher, P. (2011). A representation theorem for primitive recursive algorithms. *Fundamenta Informaticae* **107** (4) 313–330.
Blum, L., Shub, M. and Smale, S. (1989). 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.

Börger, E. (2002). The origins and the development of the ASM method for high level system design and analysis. *Journal of UCS* **8** (1) 2–74.

Bournez, O., Cucker, F., de Naurois, P. J. and Marion, J.-Y. (2003). Computability over an arbitrary structure. Sequential and parallel polynomial time. In: *Foundations of Software Science and Computational Structures*, 185–199.

Dijkstra, E. W. (1976). *A Discipline of Programming*. Prentice-Hall.

Ebbinghaus, H.-D. and Flum, J. (1995). *Finite Model Theory*. Springer-Verlag.

Grädel, E. and Gurevich, Y. (1995). Metafinite model theory. In: Leivant, D. (ed.) *Logic and Computational Complexity*, Lecture Notes in Computer Science, vol. 960, Springer, 313–366.

Gries, D. (1981). *The Science of Programming*, Texts and Monographs in Computer Science. Springer.

Gurevich, Y. (1988). Logic in computer science column. *Bulletin of the EATCS* **35** 71–81.

Gurevich, Y. (1993). Evolving algebras: An attempt to discover semantics. In: Rozenberg, G. and Salomaa, A. (eds.) *Current Trends in Theoretical Computer Science*, vol. 40, World Scientific, 266–292.

Gurevich, Y. (2001). The sequential ASM thesis. In: *Current Trends in Theoretical Computer Science*, World Scientific, 363–392.

Hartmanis, J. (1972). On non-determinancy in simple computing devices. *Acta Informatica* **1** 336–344.

van Heijenoort, J. (1967). *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931*. Harvard University Press.

Kleene, S. C. (1969). *Formalized Recursive Functionals and Formalized Realizability*. Memoirs of the AMS. American Mathematical Society.

Leivant, D. (2018). A theory of finite structures. *CoRR*, abs/1808.04949.

Peter, R. (1951). *Rekursive Funktionen*. Akadémia Kiadó.

Sazonov, V. Y. (1980). Polynomial computability and recursivity in finite domains. *Elektronische Informationsverarbeitung und Kybernetik* **16** (7) 319–323.

Skolem, T. (1923). Einige bemerkungen zur axiomatischen begründung der mengenlehre. In *Matematikerkongressen in Helsingfors Den femte skandinaviske matematikerkongressen, 1922* Heijenoort (1967), 217–232. English translation in (Heijenoort, 1967).

Strahm, T. and Zucker, J. I. (2008). Primitive recursive selection functions for existential assertions over abstract algebras. *Journal of Logical and Algebraic Methods* **76** (2) 175–197.

Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.

Cite this article: Leivant D and Marion J-Y (2020). Primitive recursion in the abstract. *Mathematical Structures in Computer Science* **30**, 33–43. https://doi.org/10.1017/S0960129519000112