Elementary algorithms and their implementations

Yiannis N. Moschovakis[1] and Vasilis Paschalis[2]

[1] Department of Mathematics, University of California, Los Angeles, CA 90095-1555, USA, and Graduate Program in Logic, Algorithms and Computation ($M\Pi\Lambda A$), Athens, Greece, ynm@math.ucla.edu
[2] Graduate Program in Logic, Algorithms and Computation ($M\Pi\Lambda A$), Athens, Greece, pasva@yahoo.com

In the sequence of articles [3, 5, 4, 6, 7], Moschovakis has proposed a mathematical modeling of the notion of *algorithm*—a set-theoretic "definition" of algorithms, much like the "definition" of real numbers as Dedekind cuts on the rationals or that of random variables as measurable functions on a probability space. The aim is to provide a traditional foundation for the theory of algorithms, a development of it within axiomatic set theory in the same way as analysis and probability theory are rigorously developed within set theory on the basis of the set theoretic modeling of their basic notions. A characteristic feature of this approach is the adoption of a very abstract notion of algorithm which takes *recursion* as a primitive operation, and is so wide as to admit "non-implementable" algorithms: *implementations* are special, restricted algorithms (which include the familiar *models of computation*, e.g., Turing and random access machines), and an algorithm is implementable if it is *reducible* to an implementation.

Our main aim here is to investigate the important relation between an (implementable) algorithm and its implementations, which was defined very briefly in [6], and, in particular, to provide some justification for it by showing that standard examples of "implementations of algorithms" naturally fall under it. We will do this in the context of (deterministic) *elementary algorithms*, i.e., algorithms which compute (possibly partial) functions $f : M^n \rightharpoonup M$ *from* (relative to) finitely many given partial functions on some set M; and so part of the paper is devoted to a fairly detailed exposition of the basic facts about these algorithms, which provide the most important examples of the general theory but have received very little attention in the general development. We will also include enough facts about the general theory, so that this paper can be read independently of the articles cited above—although many of our

The research for this article was co-funded by the European Social Fund and National Resources - (EPEAEK II) PYTHAGORAS in Greece.

choices of notions and precise definitions will appear unmotivated without the discussion in those articles.[3]

A second aim is to fix some of the basic definitions in this theory, which have evolved—and in one case evolved back—since their introduction in [3].

We will use mostly standard notation, and we have collected in the Appendix the few, well-known results from the theory of posets to which we will appeal.

1 Recursive (McCarthy) programs

A (pointed, partial) *algebra* is a structure of the form

$$\boldsymbol{M} = (M, 0, 1, \boldsymbol{\Phi}) = (M, 0, 1, \{\phi^{\boldsymbol{M}}\}_{\phi\in\boldsymbol{\Phi}}), \tag{1}$$

where 0, 1 are distinct points in the universe M, and for every *function constant* $\phi \in \boldsymbol{\Phi}$,

$$\phi^{\boldsymbol{M}} : M^n \rightharpoonup M$$

is a partial function of some arity $n = n_\phi$ associated by the *vocabulary* $\boldsymbol{\Phi}$ with the symbol ϕ. The objects 0 and 1 are used to code the truth values, so that we can include relations among the primitives of an algebra by identifying them with their characteristic functions,

$$\chi_R(\vec{x}) = R(\vec{x}) = \begin{cases} 1, & \text{if } R(\vec{x}), \\ 0, & \text{otherwise.} \end{cases}$$

Thus $R(\vec{x})$ is synonymous with $R(\vec{x}) = 1$.

Typical examples of algebras are the basic structures of arithmetic

$$\boldsymbol{N}_u = (\mathbb{N}, 0, 1, S, \mathrm{Pd}), \quad \boldsymbol{N}_b = (\mathbb{N}, 0, 1, \mathrm{parity}, \mathrm{iq}_2, \mathrm{em}_2, \mathrm{om}_2)$$

which represent the natural numbers "given" in unary and binary notation. Here $S(x) = x + 1, \mathrm{Pd}(x) = x \dot{-} 1$ are the operations of successor and predecessor, $\mathrm{parity}(x)$ is 0 or 1 accordingly as x is even or odd, and

$$\mathrm{iq}_2(x) = \text{the integer quotient of } x \text{ by } 2, \quad \mathrm{em}_2(x) = 2x, \quad \mathrm{om}_2(x) = 2x + 1$$

are the less familiar operations which are the natural primitives on "binary numbers". (We read $\mathrm{em}_2(x)$ and $\mathrm{om}_2(x)$ as *even* and *odd* multiplication by 2.) These are *total algebras*, as is the standard (in model theory) structure on the natural numbers $\boldsymbol{N} = (\mathbb{N}, 0, 1, =, +, \times)$ on $\mathbb{N}$. Genuinely partial algebras typically arise as *restrictions* of total algebras, often to finite sets: if $\{0, 1\} \subseteq L \subseteq M$, then

[3] An extensive analysis of the foundational problem of defining algorithms and the motivation for our approach is given in [6], which is also posted on http://www.math.ucla.edu/∼ynm.

$$\boldsymbol{M} \restriction L = (L, 0, 1, \{\phi^{\boldsymbol{M}} \restriction L\}_{\phi \in \Phi}),$$

where, for any $f : M^n \rightharpoonup M$,

$$f \restriction L(x_1, \ldots, x_n) = w \iff x_1, \ldots, x_n, w \in L \;\&\; f(x_1, \ldots, x_n) = w.$$

The (explicit) *terms* of the language $\mathsf{L}(\Phi)$ of programs in the vocabulary Φ are defined by the recursion

$$\begin{aligned} A :\equiv 0 \mid 1 \mid \mathsf{v}_i \mid \phi(A_1, \ldots, A_n) \mid \mathsf{p}_i^n(A_1, \ldots, A_n) \\ \mid \text{if } (A_0 = 0) \text{ then } A_1 \text{ else } A_2, \quad (\Phi\text{-terms}) \end{aligned}$$

where v_i is one of a fixed list of *individual variables*; p_i^n is one of a fixed list of n-ary (partial) *function variables*; and ϕ is any n-ary symbol in Φ. In some cases we will need the operation of *substitution* of terms for individual variables: if $\vec{\mathsf{x}} \equiv \mathsf{x}_1, \ldots, \mathsf{x}_n$ is a sequence of variables and $\vec{B} \equiv B_1, \ldots, B_n$ is a sequence of terms, then

$$A\{\vec{\mathsf{x}} :\equiv \vec{B}\} \equiv \text{the result of replacing each occurrence of } \mathsf{x}_i \text{ by } B_i.$$

Terms are interpreted naturally in any Φ-algebra $\boldsymbol{M}$, and if $\vec{\mathsf{x}}, \vec{\mathsf{p}}$ is any list of variables which includes all the variables that occur in A, we let

$$\begin{aligned} [\![A]\!](\vec{x}, \vec{p}) &= [\![A]\!]^{\boldsymbol{M}}(\vec{x}, \vec{p}) \\ &= \text{the value (if defined) of } A \text{ in } \boldsymbol{M}, \text{ with } \vec{\mathsf{x}} := \vec{x}, \vec{\mathsf{p}} := \vec{p}. \end{aligned} \quad (2)$$

Now each pair $(A, \vec{\mathsf{x}}, \vec{\mathsf{p}})$ of a term and a sufficiently inclusive list of variables defines a (partial) *functional*

$$F_A(\vec{x}, \vec{p}) = [\![A]\!](\vec{x}, \vec{p}) \quad (3)$$

on M, and the associated *operator*

$$F'_A(\vec{p}) = \lambda(\vec{x})[\![A]\!](\vec{x}, \vec{p}). \quad (4)$$

We view F'_A as a mapping

$$F'_A : (M^{k_1} \rightharpoonup M) \times \cdots \times (M^{k_n} \rightharpoonup M) \to (M^n \rightharpoonup M) \quad (5)$$

where k_i is the arity of the function variable p_i, and it is easily seen (by induction on the term A) that it is continuous.

A *recursive* (or McCarthy) *program* of $\mathsf{L}(\Phi)$ is any system of *recursive term equations*

$$A \quad : \quad \begin{cases} \mathsf{p}_A(\vec{\mathsf{x}}) = \mathsf{p}_0(\vec{\mathsf{x}}) = A_0 \\ \qquad\qquad \mathsf{p}_1(\vec{\mathsf{x}}_1) = A_1 \\ \qquad\qquad\qquad \vdots \\ \qquad\qquad \mathsf{p}_K(\vec{\mathsf{x}}_K) = A_K \end{cases} \quad (6)$$

such that $\mathsf{p}_0 \equiv \mathsf{p}_A, \mathsf{p}_1, \ldots, \mathsf{p}_K$ are distinct function variables; $\mathsf{p}_1, \ldots, \mathsf{p}_K$ are the only function variables which occur in $A_0, \ldots, A_K$; the only individual variables which occur in each A_i are in the list $\vec{\mathsf{x}}_i$; and the arities of the function variables are such that these equations make sense. The term A_0 is the *head* of A, and it may be its only *part*, since we allow $K = 0$. If $K > 0$, then the remaining parts $A_1, \ldots, A_K$ comprise the *body* of A. The *arity* of A is the number of variables in the head term A_0. Sometimes it is convenient to think of programs as (extended) *program terms* and rewrite (6) in the form

$$A \equiv A_0 \text{ where } \{\mathsf{p}_1 = \lambda(\vec{\mathsf{x}}_1)A_1, \ldots, \mathsf{p}_K = \lambda(\vec{\mathsf{x}}_K)A_K\}, \tag{7}$$

in an expanded language with a symbol for the (abstraction) λ-operator and mutual recursion. This makes it clear that the function variables $\mathsf{p}_1, \ldots, \mathsf{p}_K$ and the occurrences of the indivudual variables $\vec{\mathsf{x}}_i$ in A_i $(i > 0)$ are *bound* in the program A, and that the putative "head variable" p_A does not actually occur in A. An individual variable z is free in A if it occurs in A_0.

To interpret a program A on a Φ-structure $\boldsymbol{M}$, we consider the system of *recursive equations*

$$\begin{cases} p_1(\vec{x}_1) = [\![A_1]\!](\vec{x}_1, \vec{p}) \\ \qquad\vdots \\ p_K(\vec{x}_K) = [\![A_K]\!](\vec{x}_K, \vec{p}). \end{cases} \tag{8}$$

By the basic Fixed Point Theorem 8.1, this system has a set of least solutions

$$\overline{p}_1, \ldots, \overline{p}_K,$$

the mutual fixed points of (the body of) A, and we set

$$[\![A]\!] = [\![A]\!]^{\boldsymbol{M}} = [\![A_0]\!](\overline{p}_1, \ldots, \overline{p}_K) : M^n \rightharpoonup M.$$

Thus the *denotation* of A on $\boldsymbol{M}$ is the partial function defined by its head term from the mutual fixed points of A, and its arity is the arity of A.

A partial function $f : M^n \rightharpoonup M$ is $\boldsymbol{M}$*-recursive* if it is computed by some recursive program.

Except for the notation, these are the programs introduced by John McCarthy in [2]. McCarthy proved that *the $\boldsymbol{N}_u$-recursive partial functions are exactly the Turing-computable ones*, and it is easy to verify that these are also the $\boldsymbol{N}_b$-recursive partial functions.

In the foundational program we describe here, we associate with each Φ-program A of arity n and each Φ-algebra $\boldsymbol{M}$, a *recursor*

$$\mathrm{int}(A, \boldsymbol{M}) : M^n \rightsquigarrow M,$$

a set-theoretic object which purports to model the algorithm expressed by A on $\boldsymbol{M}$ and determines the denotation $[\![A]\!]^{\boldsymbol{M}} : M^n \rightharpoonup M$. These *referential intensions* of Φ-programs on $\boldsymbol{M}$ model "the algorithms of $\boldsymbol{M}$"; taken all together, they are the *first-order* or *elementary algorithms*, with the algebra

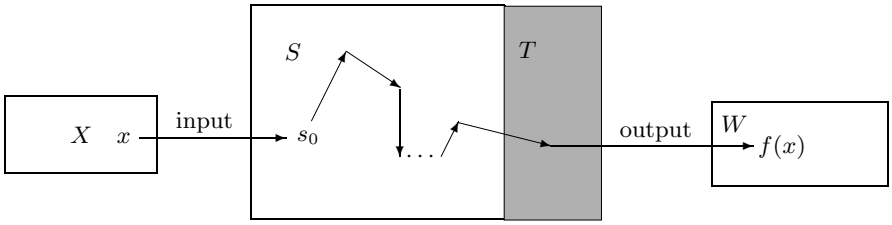

Fig. 1. Iterator computing $f : X \rightharpoonup W$.

$\boldsymbol{M}$ exhibiting the partial functions *from* (relative to) *which* any particular elementary algorithm is specified.

It naturally turns out that the $\boldsymbol{N}_u$-algorithms are not the same as the $\boldsymbol{N}_b$-algorithms on $\mathbb{N}$, because (in effect) the choices of primitives in these two algebras codify distinct ways of representing the natural numbers.

2 Recursive machines

Most of the known computation models for partial functions $f : X \to W$ on one set to another are faithfully captured by the following, well-known, general notion:

Definition 2.1 (Iterators). For any two sets X and Y, an *iterator* (or *abstract machine*) $\mathfrak{i} : X \rightsquigarrow Y$ is a quintuple $(\text{input}, S, \sigma, T, \text{output})$, satisfying the following conditions:

(I1) S is an arbitrary (non-empty) set, the *set of states* of $\mathfrak{i}$;
(I2) $\text{input} : X \to S$ is the *input function* of $\mathfrak{i}$;
(I3) $\sigma : S \to S$ is the *transition function* of $\mathfrak{i}$;
(I4) $T \subseteq S$ is the set of *terminal states* of $\mathfrak{i}$, and $s \in T \implies \sigma(s) = s$;
(I5) $\text{output} : T \to Y$ is the *output function* of $\mathfrak{i}$.

A *partial computation* of $\mathfrak{i}$ is any finite sequence $s_0, \ldots, s_n$ such that for all $i < n$, s_i is not terminal and $\sigma(s_i) = s_{i+1}$. We write

$$s \to_{\mathfrak{i}}^* s' \iff s = s', \text{ or there is a partial computation with } s_0 = s, s_n = s',$$

and we say that $\mathfrak{i}$ *computes* the partial function $\overline{\mathfrak{i}} : X \rightharpoonup Y$ defined by

$$\overline{\mathfrak{i}}(x) = w \iff (\exists s \in T)[\text{input}(x) \to_{\mathfrak{i}}^* s \ \&\ \text{output}(s) = w].$$

For example, each Turing machine can be viewed as an iterator $\mathfrak{i} : \mathbb{N} \rightsquigarrow \mathbb{N}$, by taking for states the (so-called) "complete configurations" of $\mathfrak{i}$, i.e., the triples (σ, q, i) where σ is the tape, q is the internal state, and i is the location

of the machine, along with the standard input and output functions. The same is true for random access machines, sequential machines equipped with one or more stacks, etc.

Definition 2.2 (Iterator isomorphism). An isomorphism between two iterators $\mathfrak{i}_1, \mathfrak{i}_2 : X \rightsquigarrow Y$ with the same input and output sets is any bijection $\rho : S_1 \rightarrowtail\!\!\!\!\twoheadrightarrow S_2$ of their sets of states such that the following conditions hold:

(MI1) $\rho(\text{input}_1(x)) = \text{input}_2(x) \quad (x \in X)$.
(MI2) $\rho[T_1] = T_2$.
(MI3) $\rho(\tau_1(s)) = \tau_2(\rho(s)), \quad (s \in S_1)$.
(MI4) If $s \in T_1$ and s is *input accessible*, i.e., $\text{input}_1(x) \rightarrow^* s$ for some $x \in X$, then $\text{output}_1(s) = \text{output}_2(\rho(s))$.

The restriction in (MI4) to input accessible, terminal states holds trivially in the natural case, when every terminal state is input accessible; but it is convenient to allow machines with "irrelevant" terminal states (much as we allow functions $f : X \to Y$ with $f[X] \subsetneq Y$), and for such machines it is natural to allow isomorphisms to disregard them.

For our purpose of illustrating the connection between the algorithm expressed by a recursive program and its implementations, we introduce *recursive machines*, perhaps the most direct and natural implementations of programs.

Definition 2.3 (Recursive machines). For each recursive program A and each Φ-algebra $\boldsymbol{M}$, we define the *recursive machine* $\mathfrak{i} = \mathfrak{i}(A, \boldsymbol{M})$ which computes the partial function $[\![A]\!]^{\boldsymbol{M}} : M^n \rightharpoonup M$ denoted by A as follows.

First we modify the class of Φ-terms by allowing all elements in M to occur in them (as constants) and restricting the function variables that can occur to the function variables of A:

$$B :\equiv 0 \mid 1 \mid x \mid \mathsf{v}_i \mid \phi(B_1, \ldots, B_n) \mid \mathsf{p}_i(B_1, \ldots, B_{k_i}) \\ \mid \text{if } (B_0 = 0) \text{ then } B_1 \text{ else } B_2 \quad (\Phi[A, M]\text{-terms})$$

where x is any member of M, viewed as an *individual constant*, like 0 and 1. A $\Phi[A, M]$-term B is *closed* if it has no individual variables occurring in it, so that its value in $\boldsymbol{M}$ is fixed by $\boldsymbol{M}$ (and the values assigned to the function variables $\mathsf{p}_1, \ldots, \mathsf{p}_K$).

The states of $\mathfrak{i}$ are all finite sequences s of the form

$$a_0 \ \ldots \ a_{m-1} : b_0 \ \ldots \ b_{n-1}$$

where the elements $a_0, \ldots, a_{m_1}, b_0, \ldots, b_{n-1}$ of s satisfy the following conditions:

- Each a_i is a function symbol in Φ, or one of $\mathsf{p}_1, \ldots, \mathsf{p}_K$, or a closed $\Phi[A, M]$-term, or the special symbol ?; and

- each b_j is an individual constant, i.e., $b_i \in M$.

The special symbol ':' has exactly one occurrence in each state, and that the sequences $\vec{a}, \vec{b}$ are allowed to be empty, so that the following sequences are states (with $x \in M$):

$$x : \qquad : x \qquad :$$

The *terminal states* of $\mathfrak{i}$ are the sequences of the form

$$: w$$

i.e., those with no elements on the left of ':' and just one constant on the right; and the *output* function of $\mathfrak{i}$ simply reads this constant w, i.e.,

$$\mathrm{output}(\, : w) = w.$$

The states, the terminal states and the output function of $\mathfrak{i}$ depend only on M and the function variables which occur in A. The input function of $\mathfrak{i}$ depends also on the head term of A,

$$\mathrm{input}(\vec{x}) \equiv A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\} :$$

where $\vec{\mathsf{x}}$ is the sequence of individual variables which occur in A_0.

The transition function of $\mathfrak{i}$ is defined by the seven cases in the Transition Table 1, i.e.,

$$\sigma(s) = \begin{cases} s', & \text{if } s \to s' \text{ is a special case of some line in Table 1,} \\ s, & \text{otherwise,} \end{cases}$$

and it is a function, because for a given s (clearly) at most one transition $s \to s'$ is *activated* by s. Notice that only the external calls depend on the algebra $\boldsymbol{M}$, and only the internal calls depend on the program A—and so, in particular, all programs with the same body share the same transition function.

Theorem 2.4. *Suppose A is a Φ-program with function variables $\vec{\mathsf{p}}$, $\boldsymbol{M}$ is a Φ-algebra, $\overline{p}_1, \ldots, \overline{p}_K$ are the mutual fixed points of A, and B is a closed $\Phi[A, M]$-term. Then for every $w \in M$,*

$$[\![B]\!]^{\boldsymbol{M}}(\overline{p}_1, \ldots, \overline{p}_K) = w \iff B : \to^*_{\mathfrak{i}(A,\boldsymbol{M})} : w. \tag{9}$$

In particular, with $B \equiv A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\}$,

$$[\![A]\!]^{\boldsymbol{M}}(\vec{x}) = w \iff A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\} : \to^*_{\mathfrak{i}(A,\boldsymbol{M})} : w,$$

and so the iterator $\mathfrak{i}(A, \boldsymbol{M})$ computes the denotation $[\![A]\!]^{\boldsymbol{M}}$ of A.

(pass)	$\vec{a}\ \underline{x:}\ \vec{b} \to \vec{a}\ \underline{:x}\ \vec{b} \quad (x \in M)$
(e-call)	$\vec{a}\ \underline{\phi_i : \vec{x}}\ \vec{b} \to \vec{a}\ \underline{:\phi_i^{\boldsymbol{M}}(\vec{x})}\ \vec{b}$
(i-call)	$\vec{a}\ \underline{p_i : \vec{x}}\ \vec{b} \to \vec{a}\ \underline{A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}\} :}\ \vec{b}$
(comp)	$\vec{a}\ \underline{h(A_1,\ldots,A_n):}\ \vec{b} \to \vec{a}\ \underline{h\ A_1\ \cdots\ A_n :}\ \vec{b}$
(br)	$\vec{a}\ \underline{\text{if } (A=0) \text{ then } B \text{ else } C:}\ \vec{b} \to \vec{a}\ \underline{B\ C\ ?\ A:}\ \vec{b}$
(br0)	$\vec{a}\ \underline{B\ C\ ?:0}\ \vec{b} \to \vec{a}\ \underline{B:}\ \vec{b}$
(br1)	$\vec{a}\ \underline{B\ C\ ?:y(\neq 0)}\ \vec{b} \to \vec{a}\ \underline{C:}\ \vec{b}$

- The underlined words are those which trigger a transition and change.
- $\vec{x} = x_1, \ldots, x_n$ is an n-tuple of individual constants.
- In the *external call* (e-call), $\phi_i \in \Phi$ and $\text{arity}(\phi_i) = n_i = n$.
- In the *internal call* (i-call), p_i is an n-ary function variable of A defined by the equation $p_i(\vec{\mathsf{x}}) = A_i$.
- In the *composition transition* (comp), h is a (constant or variable) function symbol with $\text{arity}(h) = n$.

Table 1. Transition Table for the recursive machine $\mathfrak{i}(A, \boldsymbol{M})$.

Outline of proof. First we define the partial functions computed by $\mathfrak{i}(A, \boldsymbol{M})$ in the indicated way,

$$\widetilde{p}_i(\vec{x}_i) = w \iff \mathsf{p}_i(\vec{x}_i) : \to^* : w,$$

and show by an easy induction on the term B the version of (9) for these,

$$[\![B]\!]^{\boldsymbol{M}}(\widetilde{p}_1, \ldots, \widetilde{p}_K) = w \iff B : \to^*_{\mathfrak{i}(A,\boldsymbol{M})} : w. \tag{10}$$

When we apply this to the terms $A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}_i\}$ and use the form of the internal call transition rule, we get

$$[\![A_i]\!]^{\boldsymbol{M}}(\vec{x}_i, \widetilde{p}_1, \ldots, \widetilde{p}_K) = w \iff \widetilde{p}_i(\vec{x}_i) = w,$$

which means that the partial functions $\widetilde{p}_1, \ldots, \widetilde{p}_K$ satisfy the system (8), and in particular

$$\overline{p}_1 \leq \widetilde{p}_1, \ldots, \overline{p}_K \leq \widetilde{p}_K.$$

Next we show that for any closed term B as above and any system $p_1, \ldots, p_K$ of solutions of (8),

$$B : \to^* w \implies [\![B]\!]^{\boldsymbol{M}}(p_1, \ldots, p_K) = w;$$

this is done by induction of the length of the computation which establishes the hypothesis, and setting $B \equiv A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}_i\}$, it implies that

$$\widetilde{p}_1 \leq p_1, \ldots, \widetilde{p}_K \leq p_K.$$

It follows that $\widetilde{p}_1, \ldots, \widetilde{p}_K$ are the least solutions of (8), i.e., $\widetilde{p}_i = \overline{p}_i$, which together with (10) completes the proof.

Both arguments appeal repeatedly to the following trivial but basic property of recursive machines: *if* $s_0, s_1, \ldots, s_n$ *is a partial computation of* $\mathfrak{i}(A, \boldsymbol{M})$ *and* $\vec{a}^*, \vec{b}^*$ *are such that the sequence* $\vec{a}^* s_0 \vec{b}^*$ *is a state, then the sequence*

$$\vec{a}^* s_0 \vec{b}^*, \vec{a}^* s_1 \vec{b}^*, \ldots, \vec{a}^* s_n \vec{b}^*$$

is also a partial computation of $\mathfrak{i}(A, \boldsymbol{M})$. □

3 Monotone recursors and recursor isomorphism

An algorithm is *directly expressed* by a definition by mutual recursion, in our approach, and so it can be modeled by the *semantic content* of a mutual recursion. This is most simply captured by a tuple of related mappings, as follows.

Definition 3.1 (Recursors). For any poset X and any complete poset W, a (*monotone*) *recursor* $\alpha : X \rightsquigarrow W$ is a tuple

$$\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_K),$$

such that for suitable, complete posets $D_1, \ldots, D_K$:

(1) Each *part* $\alpha_i : X \times D_1 \times \cdots D_K \to D_i$, $(i = 1, \ldots, k)$ is a monotone mapping.
(2) The *output mapping* $\alpha_0 : X \times D_1 \times \cdots \times D_K \to W$ is also monotone.

The number K is the *dimension* of α; the product $D_\alpha = D_1 \times \cdots \times D_K$ is its *solution set*; its *transition mapping* is the function

$$\mu_\alpha(x, \vec{d}) = (\alpha_1(x, \vec{d}), \ldots, \alpha_K(x, \vec{d})),$$

on $X \times D_\alpha$ to D_α; and the function $\overline{\alpha} : X \to W$ *computed by* α is

$$\overline{\alpha}(x) = \alpha_0(x, \vec{d}(x)) \quad (x \in X),$$

where $\vec{d}(x)$ is the least fixed point of the system of equations

$$\vec{d} = \mu_\alpha(x, \vec{d}).$$

By the basic Fixed Point Theorem 8.1,

$$\overline{\alpha}(x) = \alpha_0(x, \vec{d}^{\kappa}_{\alpha}(x)), \text{ where } \vec{d}^{\xi}_{\alpha}(x) = \mu_\alpha(x, \sup\{\vec{d}^{\eta}_{\alpha}(x) \mid \eta < \xi\}), \tag{11}$$

for every sufficiently large ordinal κ—and for $\kappa = \omega$ when α is continuous. We express all this succinctly by writing[4]

$$\alpha(x) = \alpha_0(x, \vec{d}) \,\text{where}\, \{\vec{d} = \mu_\alpha(x, \vec{d})\}, \tag{12}$$

$$\overline{\alpha}(x) = \alpha_0(x, \vec{d}) \,\overline{\text{where}}\, \{\vec{d} = \mu_\alpha(x, \vec{d})\}. \tag{13}$$

The definition allows $K = \text{dimension}(\alpha) = 0$, in which case[5] $\alpha = (\alpha_0)$ for some monotone function $\alpha_0 : X \to W$, $\overline{\alpha} = \alpha_0$, and equation (12) takes the awkward (but still useful) form

$$\alpha(x) = \alpha_0(x) \,\text{where}\, \{\ \}.$$

A recursor α is *continuous* if the mappings α_i $(i \leq K)$ are continuous, and *discrete* if X is a set (partially ordered by $=$) and $W = Y \cup \{\bot\}$ is the bottom liftup of a set. A discrete recursor $\alpha : X \rightsquigarrow Y_\bot$ computes a partial function $\overline{\alpha} : X \rightharpoonup Y$.

Definition 3.2 (The recursor of a program). Every recursive Φ-program A of arity n determines naturally the following recursor $\mathfrak{r}(A, \boldsymbol{M}) : M^n \rightsquigarrow M_\bot$ relative to a Φ-algebra $\boldsymbol{M}$:

$$\begin{aligned}\mathfrak{r}(A, \boldsymbol{M}) &= [\![A_0]\!]^{\boldsymbol{M}}(\vec{x}, \vec{p}) \\ &\quad \text{where}\, \{p_1 = \lambda(\vec{x}_1)[\![A_1]\!]^{\boldsymbol{M}}(\vec{x}_1, \vec{p}), \ldots, p_K = \lambda(\vec{x}_K)[\![A_K]\!]^{\boldsymbol{M}}(\vec{x}_K, \vec{p})\}.\end{aligned} \tag{14}$$

More explicitly (for once), $\mathfrak{r}(A, \boldsymbol{M}) = (\alpha_0, \ldots, \alpha_K)$, where $D_i = (M^{k_i} \rightharpoonup M)$ (with $\text{arity}(\mathsf{p}_i) = k_i$) for $i = 1, \ldots, K$, $D_0 = (M^n \rightharpoonup M)$, and the mappings α_i are the continuous operators defined by the parts of the program term A by (4); so $\mathfrak{r}(A, \boldsymbol{M})$ is a continuous recursor. It is immediate from the semantics of recursive programs and (11) that the recursor of a program computes its denotation,

$$\overline{\mathfrak{r}(A, \boldsymbol{M})}(\vec{x}) = [\![A]\!]^{\boldsymbol{M}}(\vec{x}) \quad (\vec{x} \in M^n). \tag{15}$$

Notice that the transition mapping of $\mathfrak{r}(A, \boldsymbol{M})$ is independent of the input $\vec{x}$, and so we can suppress it in the notation:

[4] Formally, 'where' and '$\overline{\text{where}}$' denote operators which take (suitable) tuples of monotone mappings as arguments, so that $\text{where}\,(\alpha_0, \ldots, \alpha_K)$ is a recursor and $\overline{\text{where}}\,(\alpha_0, \ldots, \alpha_n)$ is a monotone mapping. The recursor-producing operator 'where' is specific to this theory and not familiar, but the mapping producing $\overline{\text{where}}$ is one of many fairly common, recursive program constructs for which many notations are used, e.g.,

$$(\text{letrec}([d_1\ \alpha_1] \ldots [d_K\ \alpha_K])\ \ \alpha_0) \text{ or } (d_1 = \alpha_1, \ldots d_K = \alpha_K) \text{ in } \alpha_0.$$

[5] Here $D_\alpha = \{\bot\}$ (by convention or a literal reading of the definition of *product poset*), $\mu_\alpha(x, d) = d$, and (by convention again) $\alpha_0(x, \bot) = \alpha_0(x)$.

$$\mu_{\mathfrak{r}(A,\boldsymbol{M})}(\vec{p}) = \mu_{\mathfrak{r}(A,\boldsymbol{M})}(\vec{x},\vec{p}) = (\alpha_1(\vec{p}),\ldots,\alpha_K(\vec{p})). \tag{16}$$

Caution: *The recursor* $\mathfrak{r}(A,\boldsymbol{M})$ *does not always model the algorithm expressed by* A *on* $\boldsymbol{M}$, because it does not take into account the *explicit steps* which may be required for the computation of the denotation of A. In the extreme case, if $A \equiv A_0$ is an explicit term (a program with just a head and empty body), then

$$\mathfrak{r}(A,\boldsymbol{M}) = [\![A_0]\!]^{\boldsymbol{M}}(\vec{x}) \text{ where } \{\quad\}$$

is a trivial recursor of dimension 0—and it is the same for all explicit terms which define the same partial function, which is certainly not right. We will put off until Section 7 the correct and complete definition of $\mathrm{int}(A,\boldsymbol{M})$ which models more faithfully the algorithm expressed by a recursive program. As it turns out, however,

$$\mathrm{int}(A,\boldsymbol{M}) = \mathfrak{r}(\mathrm{cf}(A),\boldsymbol{M})$$

for some program $\mathrm{cf}(A)$ which is canonically associated with each A, and so the algorithms of an algebra $\boldsymbol{M}$ will all be of the form (14) for suitable A's.

Definition 3.3 (Recursor isomorphism). Two recursors $\alpha,\beta : X \rightsquigarrow W$ (on the same domain and range) are *isomorphic*[6] if they have the same dimension K, and there is a permutation $(l_1,\ldots,l_K)$ of $(1,\ldots,K)$ and poset isomorphisms $\rho_i : D_{\alpha,l_i} \to D_{\beta,i}$, such that the induced isomorphism $\rho : D_\alpha \to D_\beta$ on the solution sets preserves the recursor structures, i.e., for all $x \in X, \vec{d} \in D_\alpha$,

$$\alpha_0(x,\vec{d}) = \beta_0(x,\rho(\vec{d})),$$
$$\rho(\mu_\alpha(x,\vec{d})) = \mu_\beta(x,\rho(\vec{d})).$$

In effect, we can reorder the system of equations in the body of a recursor and replace the components of the solution set by isomorphic copies without changing its isomorphism type.

It is easy to check, directly from the definitions, that isomorphic recursors $\alpha,\beta : X \rightsquigarrow W$ compute the same function $\overline{\alpha} = \overline{\beta} : X \to W$; the idea, of course, is that *isomorphic recursors model the same algorithm*, and so we will simply write $\alpha = \beta$ to indicate that α and β are isomorphic.

4 The representation of abstract machines by recursors

We show here that the algorithms expressed by abstract machines are faithfully represented by recursors.

[6] A somewhat coarser notion of recursor isomorphism was introduced in [7] in order to simplify the definitions and proofs of some of the basic facts about recursors, but it proved not to be a very good idea. We are reverting here to the original, finer notion introduced in [5].

Definition 4.1 (The recursor of an iterator). For each iterator

$$\mathfrak{i} : (\text{input}, S, \sigma, T, \text{output}) : X \rightsquigarrow Y,$$

as in Definition 2.1, we set

$$\mathfrak{r}(\mathfrak{i}) = p(\text{input}(x)) \\ \text{where } \{p(s) = \text{if } (s \in T) \text{ then output}(s) \text{ else } p(\sigma(s))\}. \quad (17)$$

This is a continuous, discrete recursor of dimension 1, with solution space the function poset $(S \rightharpoonup Y)$ and output mapping $(x, p) \mapsto p(\text{input}(x))$, which models the *tail recursion* specified by $\mathfrak{i}$.

It is very easy to check directly that for each iterator $\mathfrak{i} : X \rightsquigarrow Y$,

$$\overline{\mathfrak{r}(\mathfrak{i})}(x) = \bar{\mathfrak{i}}(x) \quad (x \in X), \quad (18)$$

but this will also follow from the next, main result of this section.

Theorem 4.2. *Two iterators $\mathfrak{i}_1, \mathfrak{i}_2 : X \rightsquigarrow Y$ are isomorphic if and only if the associated recursors $\mathfrak{r}(\mathfrak{i}_1)$ and $\mathfrak{r}(\mathfrak{i}_2)$ are isomorphic.*

Proof. By (17), for each of the two given iterators $\mathfrak{i}_1$, $\mathfrak{i}_2$,

$$\mathfrak{r}_i = \mathfrak{r}(\mathfrak{i}_i) = (\text{value}_i, \mu_i),$$

where for $p \in D_i = (S_i \rightharpoonup Y)$ and $x \in X$,

$$\text{value}_i(x, p) = p(\text{input}_i(x)),$$
$$\mu_i(x, p) = \mu_i(p) = \lambda(s)[\text{if } (s \in T_i) \text{ then output}_i(s) \text{ else } p(\tau_i(s))] : S_i \rightharpoonup Y.$$

Part 1. Suppose first that $\rho : S_1 \rightarrowtail\!\!\!\!\!\rightarrow S_2$ is an isomorphism of $\mathfrak{i}_1$ with $\mathfrak{i}_2$, and let $f_i : S_i \rightharpoonup Y$ be the least-fixed-points of the transition functions of the two iterators, so that

$$f_i(s) = \text{output}_i(\tau_i^{|s|}(s)) \text{ where } |s| = \text{the least } n \text{ such that } \tau_i^n(s) \in T_i.$$

In particular, this equation implies easily that

$$f_1(s) = f_2(\rho(s)).$$

The required isomorphism of $\mathfrak{r}_1$ with $\mathfrak{r}_2$ is determined by a poset isomorphism of the corresponding solution sets $(S_1 \rightharpoonup Y)$ and $(S_2 \rightharpoonup Y)$, which must be of the form

$$\pi(p)(\rho(s)) = \sigma_s(p(s)) \quad (s \in S_1), \quad (19)$$

by Proposition 8.4. We will use the given bijection $\rho : S_1 \rightarrowtail\!\!\!\!\!\rightarrow S_2$ and bijections $\sigma_s : Y \rightarrowtail\!\!\!\!\!\rightarrow Y$ which are determined as follows.

We choose first for each $t \in T_1$ a bijection $\sigma_t^* : Y \rightarrowtail\!\!\!\!\!\rightarrow Y$ such that

$$\sigma_t^*(\text{output}_1(t)) = \text{output}_2(\rho(t)) \quad (t \in T_1),$$

and then we consider two cases:

(a) If $f_1(s)\uparrow$ or there is an input accessible s' such that $s \to^* s'$, let $\sigma_s(y) = y$.
(b) If $f_1(s)\downarrow$ and there is no input accessible s' such that $s \to^* s'$, let $\sigma_s = \sigma_t^*$, where $t = \tau_1^{|s|}(s) \in T_1$ is "the projection" of s to the set T_1.

Lemma. For all $s \in S_1$,

$$\sigma_{\tau_1(s)} = \sigma_s \quad (s \in S_1). \tag{20}$$

Proof. If $f_1(s)\uparrow$ or there is an input accessible s' such that $s \to^* s'$, then $\tau_1(s)$ has the same property, and so σ_s and $\sigma_{\tau_1(s)}$ are both the identity; and if $f_1(s)\downarrow$ and there is no input accessible s' such that $s \to^* s'$, then $\tau_1(s)$ has the same properties, and it "projects" to the same $t = \tau_1^n(s) \in T_1$, so that $\sigma_s = \sigma_{\tau_1(s)} = \sigma_t^*$. □ (Lemma)

Now π in (19) is an isomorphism of $(S_1 \rightharpoonup Y)$ with $(S_2 \rightharpoonup Y)$ by Proposition 8.4, and it remains only to prove the following two claims:

(i) $\text{value}_1(x, p) = \text{value}(x, \pi(p))$, i.e., $p(\text{input}_1(x)) = \pi(p)(\text{input}_2(x))$. This holds because $s = \text{input}_1(x)$ is input accessible, and so σ_s is the identity and

$$\pi(p)(\text{input}_2(x)) = \pi(p)(\rho(\text{input}_1(x)) = \sigma_s(p(\text{input}_1(x))) = p(\text{input}_1(x)).$$

(ii) $\pi(\mu_1(p)) = \mu_2(\pi(p))$, i.e., for all $s \in S_1$,

$$\pi(\mu_1(p))(\rho(s)) = \mu_2(\pi(p))(\rho(s)). \tag{21}$$

For this we distinguish three cases:

(iia) $s \in T_1$ and it is input accessible. Now (a) applies in the definition of σ_s so that σ_s is the identity and we can compute the two sides of (21) as follows:

$$\pi(\mu_1(p))(\rho(s)) = \mu_1(p)(s) = \text{output}_1(s),$$
$$\mu_2(\pi(p))(\rho(s)) = \text{output}_2(\rho(s)),$$

and the two sides are equal by (MI4) whose hypothesis holds in this case.

(iib) $s \in T_1$ but it is not input accessible, so that (b) applies. Again

$$\pi(\mu_1(p))(\rho(s)) = \sigma_s(\mu_1(p)(s)) = \sigma_s(\text{output}_1(s)),$$
$$\mu_2(\pi(p))(\rho(s)) = \text{output}_2(\rho(s)),$$

and the two sides are now equal by the choice of $\sigma_s = \sigma_s^*$.

(iic) $s \notin T_1$. In this case we can use (20):

$$\pi(\mu_1(p))(\rho(s)) = \sigma_s(\mu_1(p)(s)) = \sigma_s(p(\tau_1(s))),$$
$$\mu_2(\pi(p))(\rho(s)) = \pi(p)(\tau_2(\rho(s))) = \pi(p)(\rho(\tau_1(s))) = \sigma_{\tau_1(s)}(p(\tau_1(s)),$$

and the two sides are equal by (20).

This completes the proof of **Part 1**.

Part 2. Suppose now that $\pi : (S_1 \rightharpoonup Y) \rightarrowtail\!\!\!\!\twoheadrightarrow (S_2 \rightharpoonup Y)$ is an isomorphism of $\mathfrak{r}_1$ with $\mathfrak{r}_2$, so that by Proposition 8.4,

$$\pi(p)(\rho(s)) = \sigma_s(p(s)) \quad (p : S_1 \rightharpoonup Y),$$

where $\rho : S_1 \rightarrowtail\!\!\!\!\twoheadrightarrow S_2$ and for each $s \in S_1$, $\sigma_s : Y \rightarrowtail\!\!\!\!\twoheadrightarrow Y$. Moreover, since π is a recursor isomorphism, we know that

$$p(\mathrm{input}_1(x)) = \pi(p)(\mathrm{input}_2(x)) \tag{22}$$
$$\pi(\mu_1(p))(\rho(s)) = \mu_2(\pi(p))(\rho(s)) \tag{23}$$

We will use these identities to show that ρ is an isomorphism of $\mathfrak{i}_1$ with $\mathfrak{i}_2$.

(a) $\rho[T_1] = T_2$, (MI2).

From the definition of the transition maps μ_i, it follows immediately that $\mathrm{output}_i = \mu_i(\bot)$, and hence

$$\mathrm{output}_2 = \mu_2(\bot) = \mu_2(\pi(\bot)) = \pi(\mu_1(\bot)) = \pi(\mathrm{output}_1);$$

and then using the representation of π, for all $s \in S_1$,

$$\mathrm{output}_2(\rho(s)) = \pi(\mathrm{output}_1)(\rho(s)) = \sigma_s(\mathrm{output}_1(s)),$$

so that $\mathrm{output}_1(s)\downarrow \iff \mathrm{output}_2(\rho(s))\downarrow$, which means precisely that $\rho[T_1] = T_2$.

(b) *If $s = \mathrm{input}_1(x)$, then σ_s is the identity and $\rho(s) = \rho(\mathrm{input}_1(x)) = \mathrm{input}_2(x)$,* (MI1).

If t is such that $\rho(t) = \mathrm{input}_2(x)$, then by (22), for all p,

$$p(s) = \pi(p)(\mathrm{input}_2(x)) = \pi(p)(\rho(t)) = \sigma_t(p(t));$$

and if we apply this to

$$p(u) = \begin{cases} y, & \text{if } u = s, \\ \bot, & \text{otherwise,} \end{cases}$$

and consider the domains of convergence of the two sides, we get that $t = s$, and hence $y = \sigma_s(y)$ and $\rho(s) = \rho(t) = \mathrm{input}_2(x)$.

(c) *If s is input accessible, then for every $t \in Y$, $\sigma_s(y) = y$.*

In view of (b), it is enough to show that if σ_s is the identity, then $\sigma_{\tau_1(s)}$ is also the identity. This is immediate when $s \in T_1$, since $\tau_1(s) = s$ in this case. So assume that s is not terminal, and that σ_s is the identity, which with (23) gives, for every p,

$$\mu_2(\pi(p))(\rho(s)) = \pi(p)(\mu_1(p))(\rho(s)) = \sigma_s(\mu_1(p)(s)) = \mu_1(p)(s).$$

If $s \notin T_1$, then $\rho(s) \notin T_2$ by (a), and so this equation becomes

$$\pi(p)(\tau_2(\rho(s))) = p(\tau_1(s)).$$

If t is such that $\rho(t) = \tau_2(\rho(s))$, then this identity together with the representation of π gives

$$\sigma_t(p(t)) = \pi(p)(\rho(t)) = \pi(p)(\tau_2(\rho(s))) = p(\tau_1(s));$$

and, as above, this yields first that $t = \tau_1(s)$ and then that $\sigma_{\tau_1(s)}$ is the identity.

(d) *If s is terminal and input accessible, then* $\mathrm{output}_2(\rho(s)) = \mathrm{output}_1(s)$, (MI4).

As in (b), and using (c) this time,

$$\mathrm{output}_2(\rho(s)) = \pi(\mathrm{output}_1)(\rho(s)) = \sigma_s(\mathrm{output}_1(s)) = \mathrm{output}_1(s).$$

Finally:

(e) *For all $s \in S_1$, $\rho(\tau_1(s)) = \tau_2(\rho(s))$.*

This identity holds trivially if $t \in T_1$, in view of (a), and so we may assume that $t \notin T_1$ and apply the basic (23), which with the representation of π yields for all p and s,

$$\sigma_s(\mu_1(p)(s)) = \mu_2(\pi(p))(\rho(s)) = \pi(p)(\tau_2(\rho(s))).$$

If, as above, we choose t so that $\rho(t) = \tau_2(\rho(s))$, this gives

$$\sigma_s(p(\tau_1(s))) = \pi(p)(\rho(t)) = \sigma_t(p(t));$$

and by applying this to some p which converges only on $\tau_1(s)$, we get $t = \tau_1(s)$, so that $\tau_2(\rho(s)) = \rho(t) = \rho(\tau_1(s))$, as required. □

5 Recursor reducibility and implementations

The notion of *simulation* of one program (or machine) by another is notoriously slippery: in the relevant Section 1.2.2 of one of the standard (and most comprehensive) expositions of the subject, van Emde Boas [1] starts with

> Intuitively, a simulation of [one class of computation models] M by [another] M' is some construction which shows that everything a machine $M_i \in M$ can do on inputs x can be performed by some machine $M'_i \in M'$ on the same inputs as well;

goes on to say that "it is difficult to provide a more specific formal definition of the notion"; and then discusses several examples which show "how hard it is to define simulation as a mathematical object and still remain sufficiently general".

The situation is somewhat better at the more abstract level of recursors, where a very natural relation of *reducibility* of one recursor to another seems to capture a robust notion of "algorithm simulation". At the same time, the problem is more important for recursors than it is for machines: because this modeling of algorithms puts great emphasis on understanding *the relation between an algorithm and its implementations*, and this is defined by a reducibility.

In this section we will reproduce the relevant definitions from [6] and we will establish that the recursor $\mathfrak{r}(A, \boldsymbol{M})$ determined by a program on an algebra $\boldsymbol{M}$ is implemented by the recursive machine $\mathfrak{i}(A, \boldsymbol{M})$.

Definition 5.1 (Recursor reducibility). Suppose $\alpha, \beta : X \rightsquigarrow W$ are recursors with the same input and output domains and respective solution sets

$$D_\alpha = D_1^\alpha \times \cdots \times D_K^\alpha, \quad D_\beta = D_1^\beta \times \cdots \times D_L^\beta,$$

and transition mappings

$$\mu_\alpha : X \times D_\alpha \to D_\alpha, \quad \mu_\beta : X \times D_\beta \to D_\beta.$$

A *reduction* of α to β is any monotone mapping

$$\pi : X \times D_\alpha \to D_\beta$$

such that the following three conditions hold, for every $x \in X$ and every $d \in D_\alpha$:

(R1) $\mu_\beta(x, \pi(x, d)) \leq \pi(x, \mu_\alpha(x, d))$.
(R2) $\beta_0(x, \pi(x, d)) \leq \alpha_0(x, d)$.
(R3) $\overline{\alpha}(x) = \overline{\beta}(x)$.

We say that α is *reducible* to β if a reduction exists,

$$\alpha \leq_r \beta \iff \text{there exists a reduction } \pi : X \times D_\alpha \to D_\beta.$$

Recursor reducibility is clearly reflexive and (easily) transitive.

If $K = L = 0$ so that the recursors are merely functions, this means that they are identical, $\alpha_0(x, \bot) = \beta_0(x, \bot)$; and if $K = 0$ while $L > 0$, then these conditions degenerate to the existence of some monotone $\pi : X \to D_\beta$, such that

$$\mu_\beta(x, \pi(x)) \leq \pi(x), \quad \beta_0(x, \pi(x)) \leq \alpha_0(x, \bot), \quad \overline{\alpha}(x) = \alpha_0(x, \bot) = \overline{\beta}(x),$$

which simply means that β computes $\overline{\alpha}$—i.e., they hold of any β such that $\overline{\beta} = \overline{\alpha}$ with $\pi(x) = d_\beta^\kappa(x)$ as in (11). In the general case, (R1) and (R2) imply (by a simple ordinal recursion) that for all x and ξ,

$$d_\beta^\xi(x) \leq \pi(x, d_\alpha^\xi(x)), \quad \beta_0(x, d_\beta^\xi(x)) \leq \alpha_0(x, d_\alpha^\xi(x)),$$

and then (R3) insures that in the limit,

$$\overline{\beta}(x) = \beta_0(x, d_\beta^\kappa(x)) = \alpha_0(x, d_\alpha^\kappa(x)) = \overline{\alpha}(x);$$

thus β computes the same map as α but possibly "slower", in the sense that each iterate $d_\beta^\xi(x)$ gives us part (but possibly not all) the information in the corresponding stage $d_\alpha^\xi(x)$—and this in a uniform way. It is not completely obvious, however, that the definition captures accurately the intuitive notion of *reduction* of the computations of one recursor to those of another, and the main result in this section aims to provide some justification for it.

Definition 5.2 (Implementations). A recursor $\alpha : X \rightsquigarrow Y \cup \{\bot\}$ into a flat poset is *implemented* by an iterator $\mathfrak{i} : X \rightsquigarrow Y$ if it is reducible to the recursor representation of $\mathfrak{i}$, i.e.,

$$\alpha \leq_r \mathfrak{r}(\mathfrak{i}).$$

Theorem 5.3. *For each recursive program A and each algebra $\boldsymbol{M}$, the recursive machine $\mathfrak{i}(A, \boldsymbol{M})$ associated with A and $\boldsymbol{M}$ implements the recursor $\mathfrak{r}(A, \boldsymbol{M})$ defined by A on $\boldsymbol{M}$, i.e.,*

$$\mathfrak{r}(A, \boldsymbol{M}) \leq_r \mathfrak{r}(\mathfrak{i}(A, \boldsymbol{M})).$$

To simplify the construction of the required reduction, we establish first a technical lemma about Scott domains, cf. Definition 8.2:

Lemma 5.4. *Suppose $\alpha : X \rightsquigarrow W$ and the solution set D_α has the Scott property, and set*

$$D_\alpha^* = \{d \in D_\alpha \mid (\forall x)[d \leq \mu_\alpha(x, d)]\}. \tag{24}$$

Then D_α^* is a complete subposet of D_α, and for any recursor $\beta : X \rightsquigarrow W$, $\alpha \leq_r \beta$ if and only if there exists a monotone map $\pi^* : X \times D_\alpha^* \to D_\beta$ which satisfies (R1) – (R3) *of* Definition 5.1 *for all $x \in X, d \in D_\alpha^*$.*

Proof. We assume the hypotheses on α and $\pi^* : X \times D_\alpha^* \to D_\beta$, and we need to extend π^* so that it has the same properties on D_α.

(1) *The subposet D_α^* is complete, and for each $d \in D$, the set*

$$X_d^* = \{d^* \in D_\alpha^* \mid d^* \leq d\}$$

is directed, so that

$$\rho(d) = \sup X_d^* \in D_\alpha^*.$$

Proof. For the first claim, it is enough to show that if $X \subseteq D_\alpha^*$ is directed and $d = \sup X$, then $d \in D_\alpha^*$: this holds because for any $d^* \in X$, $d^* \leq d$, and so

$$d^* \leq \mu_\alpha(x, d^*) \leq \mu_\alpha(x, d),$$

using the characteristic property of D^*_α and the monotonicity of μ_α; and now taking suprema, we get the required $d \leq \mu_\alpha(x, d)$.

For the second claim, if $d_1, d_2 \in X^*_d$, then they are compatible, and so their supremum $d^* = \sup\{d_1, d_2\}$ exists, $d^* \leq d$, and it suffices to show that $d^* \in D^*_\alpha$; this holds because $d_i \leq \mu(x, d_i) \leq \mu_\alpha(x, d^*)$ by the characteristic property of D^*_α and the monotonicity of μ_α, and so $d^* \leq \mu_\alpha(x, d^*)$ by the definition of d^*. □ **(1)**

We now extend π^* it to all of D_α by

$$\pi(x, d) = \sup\{\pi^*(x, d^*) \mid d^* \in X^*_d\}.$$

(2) (R1): *For all x and d, $\mu_2(x, \pi(x, d)) \leq \pi(x, \mu_1(x, d))$.*

Proof. We must show that

$$\mu_2(x, \sup\{\pi^*(x, d^*) \mid d^* \in D^*_\alpha \ \&\ d^* \leq d\}) \\ \leq \sup\{\pi^*(x, e^*) \mid e^* \in D^*_\alpha \ \&\ e^* \leq \mu_1(x, d)\}. \quad (25)$$

If $d^* \in D^*_\alpha$ and $d^* \leq d$, then $d^* \leq \rho(d)$, hence $\pi^*(x, d^*) \leq \pi^*(x, \rho(d))$, and if we take suprema and apply μ_2, we get

$$\mu_2(x, \sup\{\pi^*(x, d^*) \mid d^* \in D^*_\alpha \ \&\ d^* \leq d\}) \leq \mu_2(x, \pi^*(x, \rho(d))) \\ \leq \pi^*(x, \mu_1(x, \rho(d))),$$

using the hypothesis, that (R1) holds for π^*; thus to complete the proof of (25), it is enough to verify that

$$\mu_1(x, \rho(d)) \leq \mu_1(x, d),$$

which, however, is clearly true, since $\rho(d) \leq d$ and μ_1 is monotone. □ **(2)**

(3) (R2) *For all x and d, $\beta_0(x, \pi(x, d)) \leq \alpha_0(x, d)$.*

Proof. Arguing as in the proof of **(2)**, we conclude that

$$\beta_0(x, \pi(x, d)) = \beta_0(x, \sup\{\pi^*(x, d^*) \mid d^* \in X^*_d\}) \leq \beta_0(x, \pi^*(x, \rho(d))) \\ \leq \alpha_0(x, \rho(d)) \leq \alpha_0(x, d),$$

where we appealed to (R2) for π^* in the next-to-the-last inequality. □ **(3)**

Finally, (R3) holds because the computation of $\overline{\alpha}$ takes place entirely within D^*_α on which the given π^* coincides with π and is assumed to satisfy (R1) – (R3). □

Proof of Theorem 5.3. We fix a recursive program A and an algebra $\boldsymbol{M}$ and set

$$\alpha = \mathfrak{r}(A, \boldsymbol{M}) = (\alpha_0, \alpha_1, \ldots, \alpha_K), \quad \beta = (\beta_0, \beta_1) = \mathfrak{r}(\mathfrak{i}(A, \boldsymbol{M}))$$

as these are defined in Definitions 3.2, 2.3 and 4.1, so that

$$D_0 = (M^n \rightharpoonup M), \quad D_\alpha = D_1 \times \cdots \times D_K \ \ (\text{with } D_i = (M^{k_i} \rightharpoonup M));$$

the mappings α_i are the continuous operators defined by the parts A_i of the program A by (4); the transition mapping μ_α of α (defined in (16)) is independent of the input $\vec{x}$; $D_\beta = (S \rightharpoonup M)$ with S the set of states of the recursive machine $\mathfrak{i}(A, \boldsymbol{M})$; $\beta_0(d)(\vec{x}) = d(A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\} :)$; and

$$\mu_\beta(d) = \beta_1(d) = \lambda(s)[\text{if } (s =: w' \text{ for some } w') \text{ then } w' \text{ else } d(\sigma(s))],$$

where σ is the transition mapping σ of the recursive machine and (like μ_α) is independent of the input $\vec{x}$. To complete the proof, we need to define a monotone mapping $\pi : D_\alpha \to D_\beta$ so that (tracing the definitions), the following two conditions hold:

(R1) For all $\vec{p} \in D_\alpha$, $\beta_1(\pi(\vec{p})) \leq \pi(\mu_\alpha(\vec{p}))$; this means that for all $w \in M$,

$$\pi(\mu_\alpha(\vec{p}))(: w) = w, \tag{R1a}$$

and for every non-terminal state s of the recursive machine $\mathfrak{i}(A, \boldsymbol{M})$ and any $w \in M$,

$$\pi(\vec{p})(\sigma(s)) = w \implies \pi(\mu_\alpha(\vec{p}))(s) = w. \tag{R1b}$$

(R2) For all $\vec{x} \in M^n, \vec{p} \in D_\alpha$ and $\vec{x} \in M^n, w \in M$,

$$\pi(\vec{p})(A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\} :) = w \implies [\![A_0]\!]^{\boldsymbol{M}}(\vec{x}, \vec{p}) = w.$$

The third conditions (R3) is independent of the reduction π and holds by Theorem 2.4, (18), and (15).

For each tuple $\vec{p} \in D_\alpha$, let

$$\boldsymbol{M}\{\vec{\mathsf{p}} := \vec{p}\} = (M, 0, 1, \{\phi^{\boldsymbol{M}}_{\phi \in \Phi}\} \cup \{p_1, \ldots, p_n\})$$

be the *expansion* of the algebra $\boldsymbol{M}$ by the partial functions $p_1, \ldots, p_K$ in the vocabulary $\Phi \cup \{\mathsf{p}_1, \ldots, \mathsf{p}_K\}$ of the given program A. The states of $\mathfrak{i}(A, \boldsymbol{M})$ are also states of $\mathfrak{i}(A, \boldsymbol{M}\{\vec{\mathsf{p}} := \vec{p}\})$—but differently interpreted, since $\mathsf{p}_1, \ldots, \mathsf{p}_K$ have now been fixed and so calls to them are *external* (independent of the program A) rather than *internal*. For any state $\vec{a} : \vec{b}$, we set

$$\begin{aligned} \pi(\vec{p})(\vec{a} : \vec{b}) = w &\iff \vec{a} : \vec{b} \to^*_{\boldsymbol{M}\{\vec{\mathsf{p}} := \vec{p}\}} w \\ &\iff \text{the computation of } \mathfrak{i}(A, \boldsymbol{M}\{\vec{\mathsf{p}} := \vec{p}\}) \\ &\qquad \text{which starts with } \vec{a} : \vec{b} \text{ terminates in the state } : w. \end{aligned}$$

Since $\boldsymbol{M}$ and the $\Phi[A,M]$-terms are fixed, these computations depend only on the *assignment* $\{\vec{\mathsf{p}} := \vec{p}\}$, and we will call them $\{\vec{\mathsf{p}} := \vec{p}\}$-*computations*; they are exactly like those of $\mathsf{i}(A,\boldsymbol{M})$, except when one of the p_i is called with the correct number of arguments: i.e., when p_i is n-ary, then for any $x_1,\ldots,x_n$,

$$\vec{a}\ \mathsf{p}_i : x_1 \cdots x_n\ \vec{b} \rightarrow_{\{\vec{\mathsf{p}}:=\vec{p}\}} \vec{a} : p_i(x_1,\ldots,x_n)\ \vec{b}$$

We establish that π has the required properties in a sequence of lemmas.

(1) *The mapping π is monotone.*

Proof. If $\vec{p} \leq \vec{q}$ and $\pi(\vec{p})(s) = w$, then the finite $\{\vec{\mathsf{p}} := \vec{p}\}$-computation starting with s calls only a finite number of values of the partial functions $p_1,\ldots,p_K$ and terminates in the state $: w$; the $\{\vec{\mathsf{p}} := \vec{q}\}$-computation starting with s will then call the same values of $q_1,\ldots,q_K$, it will get the same answers, and so it will reach the same terminal state $: w$. □ **(1)**

(2) *For each $\Phi[A,M]$-term B,*

$$\pi(\vec{p})(B :) = [\![B]\!]^{\boldsymbol{M}}(\vec{p}).$$

Proof is easy, by induction on B. □ **(2)**

By Lemma 5.4, it is enough to verify (R1) and (R2) for $\vec{p} \in D_\alpha^*$, and we will appeal to this in **(5)** below.

(3) (R1a): $\pi(\mu_\alpha(\vec{p}))(: w) = w$.

Proof. This is immediate from the definition of π, since the $\{\vec{\mathsf{p}} := \mu_\alpha(\vec{p})\}$-computation starting with $: w$ terminates immediately, without looking at the partial function assigned to any p_i. □ **(3)**

(4) (R1b) *for the case where s is a state of the form*

$$\vec{a}\ \mathsf{p}_i : x_1 x_2 \cdots x_n\ \vec{b}$$

with $\mathrm{arity}(\mathsf{p}_i) = n$.

Proof. By the transition table of the recursive machine $\mathsf{i}(A,\boldsymbol{M})$,

$$\vec{a}\ \mathsf{p}_i : x_1 x_2 \cdots x_n\ \vec{b} \quad \rightarrow \quad \vec{a}\ A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}\} : \vec{b}$$

and so the hypothesis of (R1b) in this case gives us that

$$\pi(\vec{p})(\vec{a}\ A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}\} : \vec{b}) = w$$

which means that w is the output of the $\{\vec{\mathsf{p}} := \vec{p}\}$-computation

$$\left[\begin{array}{r} \vec{a}\ A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}\} : \vec{b} \\ \vdots \\ : w \end{array}\right] \{\vec{\mathsf{p}} := \vec{p}\}$$

By **(2)** above,

$$\left.\begin{array}{r} \vec{a}\ A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}\} : \vec{b} \\ \vdots \\ \vec{a} : [\![A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}\}]\!](\vec{p})\ \vec{b} \end{array}\right] \{\vec{\mathsf{p}} := \vec{p}\}$$

and so, comparing outputs, we conclude that $\vec{a}$ and $\vec{b}$ are empty and

$$w = [\![A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}\}]\!](\vec{p}).$$

On the other hand, by the definition of π and μ_α, there is a $\{\vec{\mathsf{p}} := \mu_\alpha(\vec{p})\}$-computation

$$\left.\begin{array}{r} \mathsf{p}_i : x_1x_2\cdots x_n \\ : [\![A_i\{\vec{\mathsf{x}}_i :\equiv \vec{x}\}]\!](\vec{p}) \end{array}\right] \{\vec{\mathsf{p}} := \mu_\alpha(\vec{p})\}$$

which gives us precisely the required conclusion of (R1b) in this case,

$$\pi(\mu_\alpha(\vec{p}))(\vec{a}\ \mathsf{p}_i : x_1x_2\cdots x_n\ \vec{b}) = w.\Box \textbf{ (4)}$$

(5) (R1b) *otherwise, i.e., when s is not a state of the form $\vec{a}\ \mathsf{p}_i : x_1x_2\cdots x_n\ \vec{b}$.*

Proof. The transition table of $\mathfrak{i}(A, \boldsymbol{M})$ now gives

$$s \ \rightarrow\ \vec{a}' : \vec{b}'$$

for some state $\vec{a}' : \vec{b}'$, and the hypothesis of (R1b) guarantees a computation

$$\left.\begin{array}{r} \vec{a}' : \vec{b}' \\ \vdots \\ : w \end{array}\right] \{\vec{\mathsf{p}} := \vec{p}\}$$

Since the transition $s \ \rightarrow\ \vec{a}' : \vec{b}'$ does not refer to any function letter p_j, it is equally valid for $\mathfrak{i}(A, \boldsymbol{M}\{\vec{\mathsf{p}} := \vec{p}\})$, and so we have a computation

$$\left.\begin{array}{r} s \\ \vec{a}' : \vec{b}' \\ \vdots \\ : w \end{array}\right] \{\vec{\mathsf{p}} := \vec{p}\}$$

But $\vec{p} \leq \mu_\alpha(\vec{p})$ since $\vec{p} \in D^*_\alpha$, and so by **(1)**,

$$\left.\begin{array}{r} s \\ \vec{a}' : \vec{b}' \\ \vdots \\ : w \end{array}\right] \{\vec{\mathsf{p}} := \mu_\alpha(\vec{p})\}$$

which means that $\pi(\mu_\alpha(\vec{p}))(s) = w$ and completes the proof of (R1b). □ **(5)**

(6) (R2) *holds, i.e.*,

$$\pi(\vec{p})(A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\}\,:) = w \implies [\![A_0]\!]^{\boldsymbol{M}}(\vec{x},\vec{p}) = w.$$

Proof. This is an immediate consequence of **(2)**, which for $B \equiv A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\}$ gives $\pi(\vec{p})(A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\}\,:) = [\![A_0\{\vec{\mathsf{x}} :\equiv \vec{x}\}]\!]^{\boldsymbol{M}}(\vec{p}) = [\![A_0]\!]^{\boldsymbol{M}}(\vec{x},\vec{p})$. □ **(6)**

This completes the proof of Theorem 5.3. □

While the technical details of this proof certainly depend on some specific features of recursive machines, the idea of the proof is general and quite robust: it can be used to show that any reasonable "simulation" of a McCarthy program A by an abstract machine implements the recursor $\mathfrak{r}(A, \boldsymbol{M})$ defined by A—and this covers, for example, the usual "implementations of recursion" by Turing machines or random access machines of various kinds.

6 Machine simulation and recursor reducibility

On the problem of defining formally the notion of one machine simulating another, basically we agree with the comments of van Emde Boas [1] quoted in the beginning of Section 5, that it is not worth doing. There is, however, one interesting result which relates a specific, formal notion of machine simulation to recursor reducibility.

Definition 6.1 (Machine simulation, take 1). Suppose

$$\mathfrak{i}_i = (\text{input}_i, S_i, \sigma_i, T_i, \text{output}_i) : X \rightsquigarrow Y \quad (i = 1, 2)$$

are two iterators on the same input and output sets. A **formal simulation** of $\mathfrak{i}_1$ by $\mathfrak{i}_2$ is any function $\rho : S_2 \to S_1$ such that the following conditions hold:

1. For any state $t \in S_2$, if $\rho(t) \to_1 s \in S_1$, then there is some $t' \in S_2$ such that $t \to_2^* t'$ and $\rho(t') = s$.
2. If $t_0 = \text{input}_2(x)$, then $\rho(t_0) = \text{input}_1(x)$.
3. If $t \in T_2$, then $\rho(t) \in T_1$ and $\text{output}_1(\rho(t)) = \text{output}_2(t)$.
4. If $t \in S_2 \setminus T_2$ and $\rho(t) \in T_1$, then there is path

$$t \to_2 t_0 \to_1 \cdots \to_2 t_k \in T_2$$

in $\mathfrak{i}_2$ such that $\text{output}_1(\rho(t)) = \text{output}_2(t_k)$ and $\rho(t_i) = \rho(t)$ for all $i \leq k$.

This is one sort of formal notion considered by van Emde Boas [1], who (rightly) does not adopt it as his only (or basic) formal definition.

Theorem 6.2 (Paschalis [8]). *If* $\mathfrak{i}_2$ *simulates* $\mathfrak{i}_1$ *formally by Definition 6.1, then* $\mathfrak{r}(\mathfrak{i}_1) \leq_r \mathfrak{r}(\mathfrak{i}_2)$.

We do not know whether the converse of this result holds, or the relation of this notion of simulation with natural alternatives which involve maps $\rho : S_1 \to S_2$ (going the other way). The theorem, however, suggests that perhaps the robust notion $\mathfrak{r}(\mathsf{i}_1) \leq_r \mathfrak{r}(\mathsf{i}_2)$ may be, after all, the best we can do in the way of giving a broad, formal definition of what it means for one machine to simulate another.

7 Elementary (first-order) recursive algorithms

The term

$$E \equiv \text{if } (\phi_1(x) = 0) \text{ then } y \text{ else } \phi_2(\phi_1(y), x)$$

intuitively expresses an algorithm on each algebra $\boldsymbol{M}$ in which ϕ_1, ϕ_2 are interpreted, the algorithm which computes its value for any given values of x and y. We can view it as a program

$$E \quad : \quad \mathsf{p}_0(x, y) = \text{if } (\phi_1(x) = 0) \text{ then } y \text{ else } \phi_2(\phi_1(y), x) \tag{26}$$

with empty body, but the recursor $\mathfrak{r}(E, \boldsymbol{M})$ of E constructed in Definition 3.2 does not capture this algorithm—it is basically nothing but the partial function defined by E. This is because, in general, the recursor $\mathfrak{r}(A, \boldsymbol{M})$ of a program A captures only "the recursion" expressed by A, and there is no recursion in E. The theory of *canonical forms* which we will briefly outline in this section makes it possible to capture such explicit algorithms (or parts of algorithms) by the general construction $(A, \boldsymbol{M}) \mapsto \mathfrak{r}(A, \boldsymbol{M})$. In particular, it yields a robust notion of *the elementary algorithms* of any given algebra $\boldsymbol{M}$.

To ease the work on the syntax that we need to do, we add to the language a function symbol cond of arity 3 and the abbreviation

$$\text{cond}(A, B, C) \equiv_{df} \text{if } (A = 0) \text{ then } B \text{ else } C;$$

this is not correct semantically, because cond is not a *strict partial function*, but it simplifies the definition of Φ-terms which now takes the form

$$A :\equiv 0 \mid 1 \mid \mathsf{v}_i \mid \mathsf{c}(A_1, \ldots, A_n) \qquad (\Phi\text{-terms})$$

where c is any constant (ϕ_i, cond) or variable function symbol (p_i^n) of arity n.

A term is *immediate* if it is an individual variable or the "value" of a function variable on individual variables,

$$X :\equiv \mathsf{v}_i \mid \mathsf{p}_i^n(\mathsf{u}_1, \ldots, \mathsf{u}_n) \qquad \text{(Immediate terms)}$$

so that, for example, $\mathsf{u}, \mathsf{p}(u_1, u_2, u_1)$ are immediate when p has arity 3. Computationally, we think of immediate terms as "generalized variables" which can be accessed directly, like the entries $a[i], b[i, j, i]$ in an array (string) in some programming languages.

Definition 7.1 (Program reduction). Suppose A is a program as in (6),

$$\mathsf{p}_i(\vec{\mathsf{x}}_i) = \mathsf{c}(A_1, \ldots, A_{j-1}, A_j, A_{j+1}) \tag{27}$$

is one of the equations in A, *and* A_j *is not immediate*. Let q be a function variable with arity(q) = arity(p) which does not occur in A. The *one-step reduction of* A *determined by* p_i, j *and* q yields the program B constructed by replacing (27) in A by the following two equations:

$$\begin{aligned} \mathsf{p}_i(\vec{\mathsf{x}}_i) &= \mathsf{c}(A_1, \ldots, A_{j-1}, \mathsf{q}(\vec{\mathsf{x}}_i), A_{j+1}) \\ \mathsf{q}(\vec{\mathsf{x}}_i) &= A_j \end{aligned}$$

We write

$$\begin{aligned} A \Rightarrow_1 B &\iff \text{there is a one-step reduction of } A \text{ to } B, \\ A \Rightarrow B &\iff B \equiv A \text{ or } A \Rightarrow_1 A_1 \Rightarrow_1 \cdots \Rightarrow_1 A_k \equiv B, \end{aligned}$$

so that the *reduction relation* on programs is the reflexive and transitive closure of one-step reduction.

Let size(A) be the number of non-immediate terms which occur as arguments of function symbols in the parts of A. A program is *irreducible* if size(A) = 0, so that no one-step reduction can be executed on it.

Each one-step reduction lowers size by 1, and so, trivially:

Lemma 7.2. *If* $A \Rightarrow_1 A_1 \Rightarrow_1 \cdots \Rightarrow_1 A_k$ *is a sequence of one-step reductions starting with* A*, then* $k \leq \text{size}(A)$*; and* A_k *is irreducible if and only if* $k = \text{size}(A)$.

The reduction process clearly preserves the head function variable of A, and so it preserves its arity. It also (easily) preserves denotations,

$$A \Rightarrow B \implies [\![A]\!]^{\boldsymbol{M}} = [\![B]\!]^{\boldsymbol{M}},$$

but this would be true even if we removed the all-important immediacy restriction in its definition. The idea is that much more is preserved: we will claim, in fact, that if $A \Rightarrow B$, then A *and* B *express the same algorithm in every algebra*.

Caution. This notion of reduction is a syntactic operation on programs, which models (very abstractly) *partial compilation*, bringing the mutual recursion expressed by the program to a useful form before the recursion is implemented *without committing to any particular method of implementation of recursion*. No real computation is done by it.

We illustrate the reduction process by constructing a reduction sequence starting with the explicit term E in (26) and showing on the right the parameters we use for each one-step reduction:

$$E : \mathsf{p}_0(x,y) = \text{if } (\phi_1(x) = 0) \text{ then } y \text{ else } \phi_2(\phi_1(y), x) \qquad (\mathsf{p}_0, 1, \mathsf{q}_1)$$

$$E_1 : \begin{array}{l} \mathsf{p}_0(x,y) = \text{if } (\mathsf{q}_1(x,y) = 0) \text{ then } y \text{ else } \phi_2(\phi_1(y), x) \\ \mathsf{q}_1(x,y) = \phi_1(x) \end{array} \qquad (\mathsf{p}_0, 3, \mathsf{q}_2)$$

$$E_2 : \begin{array}{l} \mathsf{p}_0(x,y) = \text{if } (\mathsf{q}_1(x,y) = 0) \text{ then } y \text{ else } q_2(x,y) \\ \mathsf{q}_2(x,y) = \phi_2(\phi_1(y), x) \\ \mathsf{q}_1(x,y) = \phi_1(x) \end{array} \qquad (\mathsf{q}_2, 1, q_3)$$

$$E_3 : \begin{array}{l} \mathsf{p}_0(x,y) = \text{if } (\mathsf{q}_1(x,y) = 0) \text{ then } y \text{ else } q_2(x,y) \\ \mathsf{q}_2(x,y) = \phi_2(\mathsf{q}_3(x,y), x) \\ \mathsf{q}_3(x,y) = \phi_1(y) \\ \mathsf{q}_1(x,y) = \phi_1(x) \end{array}$$

Now E_3 is irreducible, and so the reduction process stops.

We will not take the space here to argue that E_3 expresses *as a mutual recursion* the same *explicit* algorithm which is intuitively expressed by E. But note that the order of the equations in the body of E_3 is of no consequence: Definition 3.3 insures that we can list these in any order without changing the isomorphism type of the recursor $\mathfrak{r}(E_3, \boldsymbol{M})$. This reflects our basic understanding that the intuitive, explicit algorithm expressed by E does not specify whether the evaluations that are required will be done in parallel, or in sequence, or in any particular order, except, of course, where the nesting of subterms forces a specific order for the calls to the primitives—and this is exactly what is captured by the structure of the irreducible program E_3.

To call $\mathfrak{r}(E_3, \boldsymbol{M})$ "the algorithm expressed by E", we must show that it is independent of any particular reduction of E to an irreducible term, and to do this we must abstract from the specific, fresh variables introduced by the reduction process and the order in which the new equations are added.

Definition 7.3 (Program congruence). Two programs A and B are *congruent* if B can be constructed from A by an alphabetic change (renaming) of the individual and functions variables and a permutation of the equations in the body of A. This is obviously an equivalence relation on programs which agrees with the familiar term-congruence on programs with empty body. We write:

$$A \equiv_c B \iff A \text{ and } B \text{ are congruent.}$$

It follows directly from this definition and Definition 3.3 that congruent programs have isomorphic recursors in every algebra $\boldsymbol{M}$,

$$A \equiv_c B \implies \mathfrak{r}(A, \boldsymbol{M}) = \mathfrak{r}(B, \boldsymbol{M}).$$

Theorem 7.4 (Canonical forms).

Every program A is reducible to a unique up to congruence irreducible term $\mathrm{cf}(A)$*, its canonical form.*

In detail: every program A has a canonical form $\mathrm{cf}(A)$ *with the following properties:*

(1) $\mathrm{cf}(A)$ *is an irreducible program.*
(2) $A \Rightarrow \mathrm{cf}(A)$.
(3) *If* $A \Rightarrow_1 A_1 \Rightarrow_1 \cdots \Rightarrow_1 A_k$ *is any sequence of one-step reductions and* $k = \mathrm{size}(A)$*, then* $A_k \equiv_c \mathrm{cf}(A)$.

Part (3) gives a method for computing $\mathrm{cf}(A)$ up to congruence, and also implies the first claim, that it is the unique up to congruence term which satisfies (1) and (2): because if $A \Rightarrow B$ and B is irreducible, then $A \Rightarrow_1 A_1 \Rightarrow_1 \cdots \Rightarrow_1 A_k \equiv B$ with $k = \mathrm{size}(A)$ by Lemma 7.2, and hence $B \equiv_c \mathrm{cf}(A)$ by (3).

Outline of proof. To construct canonical forms, we fix once and for all some ordering on all the function variables, and starting with A, we execute a sequence of $\mathrm{size}(A)$ one-step reductions, selecting each time the lowest $\mathsf{p}, j, \mathsf{q}$ for which a reduction can be executed. The last program of this sequence satisfies (1) and (2), and so we only need verify (3), for which it suffices to check that

$$A \Rightarrow_1 B \implies \mathrm{cf}(A) \equiv_c \mathrm{cf}(B). \tag{28}$$

For this, the basic fact is that one-step reductions commute, in the following, simple sense: if we label them by showing their parameters,

$$A \xrightarrow{\mathsf{p}_i, j, \mathsf{q}} B$$
$$\iff B \text{ results by the one-step reduction on } A \text{ determined by } \mathsf{p}_i, j, \mathsf{q}$$

then:

$$A \xrightarrow{\mathsf{p}_i, j, \mathsf{q}} B \xrightarrow{\mathsf{p}_k, l, \mathsf{r}} C, \implies \text{for some } B',\ A \xrightarrow{\mathsf{p}_k, l, \mathsf{r}} B' \xrightarrow{\mathsf{p}_i, j, \mathsf{q}} C \tag{29}$$

The obvious conditions here are that $\mathsf{q} \not\equiv \mathsf{r}$ and that either $i \neq k$ or $j \neq l$, so that all four one-step reductions indicated can be executed, but, granting these, (29) follows immediately by the definitions. Finally, (28) can be verified using (29) by an easy "permutability" argument which we will skip. □

A more general version of the Canonical Form Theorem 7.4 for *functional structures* was established in [4], and there are natural versions of it for many richer languages, including a suitable formulation of the typed λ-calculus with recursion (PCF). This version for McCarthy programs is especially simple to state (and prove), because of the simplicity in this case of the definition of reduction in Definition 5.1.

Definition 7.5 (Referential intensions). The *referential intension* of a McCarthy program A in an algebra $\boldsymbol{M}$ is the recursor of its canonical form,

$$\mathrm{int}(A, \boldsymbol{M}) = \mathfrak{r}(\mathrm{cf}(A), \boldsymbol{M});$$

it models the elementary algorithm expressed by A in $\boldsymbol{M}$.

In this modeling of algorithms then, Theorem 5.3 establishes that *every elementary algorithm* $\mathrm{int}(A, \boldsymbol{M})$ *expressed by a term* A *in an algebra* $\boldsymbol{M}$ *is implemented by the recursive machine* $\mathfrak{i}(\mathrm{cf}(A), \boldsymbol{M})$ *of the canonical form of* A.

8 Appendix

A subset $X \subseteq D$ of a poset D is *directed* if

$$x, y \in X \implies (\exists z \in X)[x \leq_D z \;\&\; y \leq_D z]$$

and D is *complete* (a *dcpo*) if every directed $X \subseteq D$ has a (necessarily unique) *supremum* (least upper bound),

$$\sup X = \min\{z \in D \mid (\forall x \in X)[x \leq_D z]\}.$$

In particular, every complete poset (with this definition) has a least element,

$$\bot_D = \sup \emptyset,$$

and for each set Y, its *bottom liftup*

$$Y_\bot = Y \cup \{\bot\}$$

is the complete *flat* poset which has just one new element $\bot \notin Y$ put below all the members of X,

$$x \leq_{Y_\bot} y \iff x = \bot \vee x = y \quad (x, y \in Y_\bot).$$

It will also be convenient to view each set X as (trivially) partially ordered by the identity relation, so that X is a subposet of $X_\bot$.

The (Cartesian) *product* $D = D_1 \times \cdots \times D_n$ of n posets is ordered componentwise,

$$x \leq_D y \iff x_1 \leq_{D_1} y_1 \;\&\; \cdots \;\&\; x_n \leq_{D_n} y_n \quad (x = (x_1, \ldots, x_n), y = (y_1, \ldots, y_n)),$$

and it is complete, if $D_1, \ldots, D_n$ are all complete.

A mapping $\pi : D \to E$ from one poset to another is *monotone* if

$$x \leq_D y \implies \pi(x) \leq_E \pi(y) \qquad (x, y \in D),$$

and *continuous* if in addition, for every directed, non-empty subset of D and every $w \in D$,

$$\text{if } w = \sup X, \text{ then } \pi(w) = \sup \pi[X].$$

If D and E are complete posets, then this conditions takes the simpler form

$$\pi(\sup X) = \sup \pi[X] \quad (X \text{ directed, non-empty}),$$

but it is convenient to allow D to be arbitrary (for example a product of complete posets and sets) in the definition.

A *poset isomorphism* is any bijection $\pi : D \rightarrowtail\!\!\!\!\twoheadrightarrow D_2$ which respects the partial ordering,

$$d_1 \leq_D d_2 \iff \pi(d_1) \leq_E \pi(d_2),$$

and it is automatically monotone and continuous.

Theorem 8.1 (Fixed Point Theorem). *Every monotone mapping $\pi : D \to D$ on a complete poset to itself has a least fixed point*

$$\overline{x} = (\mu x \in D)[x = \pi(x)],$$

characterized by the following two properties:

$$\overline{x} = \pi(\overline{x}), \quad (\forall y)[\pi(y) \leq_D y \implies \overline{x} \leq y];$$

in fact, $\overline{x} = \overline{x}^\kappa$ for every sufficiently large ordinal number κ, where the transfinite sequence $\{\overline{x}^\xi\}_\xi$ (of *iterates* of π) *is defined by the recursion*

$$\overline{x}^\xi = \pi(\sup\{\overline{x}^\eta \mid \eta < \xi\}), \tag{30}$$

and we may take $\kappa = \omega$ if π is continuous.

Moreover, if $\pi : D \times E \to D$ is monotone (respectively *continuous*) *and D is complete, then the mapping*

$$\rho(y) = (\mu x \in D)[\pi(x, y) = x] \quad (y \in E)$$

is monotone (respectively *continuous*)*, $\rho : E \to D$.*

When $D = D_1 \times \cdots \times D_n$ is a product of complete posets, then the Least Fixed Point Theorem guarantees the existence of *canonical* (least) solutions

$$\overline{x}_1 : E \to D_1 \ldots, \overline{x}_n : E \to D_n$$

for each system of monotone, recursive equations with parameters

$$\begin{array}{rcl} x_1(y) & = & \pi_1(x_1, \ldots, x_n, y) \\ x_2(y) & = & \pi_2(x_1, \ldots, x_n, y) \\ & \cdots & \\ x_n(y) & = & \pi_n(x_1, \ldots, x_n, y); \end{array} \tag{31}$$

the solutions are monotone, and if each π_i is continuous, then they are continuous.

For any sets $X_1, \ldots, X_n, Y$, a *partial function* $f : X_1 \times \cdots \times X_n \rightharpoonup Y$ is any mapping

$$f : X_1 \times \cdots \times X_n \to Y_\perp.$$

Sometimes we identify f with its obvious liftup

$$\hat{f} : X_{1,\perp} \times \cdots \times X_{n,\perp} \to Y_\perp$$

which takes the value $\perp$ if any one of its arguments is $\perp$. The *composition* of partial functions is defined using these liftups: for $x \in X_1 \times \cdots \times X_n, w \in Y$,

$$\begin{array}{l} f(g_1(x), \ldots, g_n(x)) = w \\ \iff (\exists u_1, \ldots, u_n)[g_1(x) = u_1 \ \& \cdots \& \ g_n(x) = u_n \ \& \ f(u_1, \ldots, u_n) = w]. \end{array}$$

We will use the familiar notations for *convergence* and *divergence* of partial functions,

$$f(x)\downarrow \iff f(x) \neq \perp, \quad f(x)\uparrow \iff f(x) = \perp.$$

For any poset D and any complete poset E, the function spaces

$$\mathrm{Mon}(D \to E) = \{\pi : D \to E \mid \pi \text{ is monotone}\},$$
$$\mathrm{Cont}(D \to E) = \{\pi : D \to E \mid \pi \text{ is continuous}\}$$

are complete posets with the pointwise partial ordering,

$$\pi \leq \rho \iff (\forall x \in D)[\pi(x) \leq_E \pi(y)];$$

this is also true of the partial function spaces

$$(X_1 \times \cdots X_n \rightharpoonup Y) = \{f : X_1 \times \cdots \times X_n \rightharpoonup Y\} = \mathrm{Mon}(X_1 \times \cdots X_n \to Y_\perp),$$

with which we are primarily concerned in this article. We list here two properties of them that we will need.

Definition 8.2 (Scott domains). Two points d_1, d_2 in a poset D are *compatible* if their doubleton $\{d_1, d_2\}$ has an upper bound in D; and D has the *Scott property*, if any two compatible points $d_1, d_2 \in D$ have a (necessarily unique) least upper bound:

$$(\exists e)[d_1 \leq e \;\&\; d_2 \leq e] \implies \sup\{d_1, d_2\} \text{ exists.}$$

Proposition 8.3. *Every partial function poset $(X \rightharpoonup Y)$ has the Scott property; and if $D_1, \ldots, D_n$ have the Scott property, then their product $D_1 \times \cdots \times D_n$ also has the Scott property.*

Proof. For compatible $d_1, d_2 : X \rightharpoonup Y$, clearly $\sup\{d_1, d_2\} = d_1 \cup d_2$, the least common extension of d_1 and d_2. The second statement follows by induction on n, using the obvious fact that $D_1 \times \cdots \times D_n \times D_{n+1}$ and $(D_1 \times \cdots \times D_n) \times D_{n+1}$ are isomorphic. □

The next proposition gives a normal form for all poset isomorphisms between partial function spaces, which we need in the proof of Theorem 4.2 (and it may be well-known, but we were unable to find it in the literature).

Proposition 8.4. *For any sets S_1, S_2, Y_1, Y_2, suppose $\rho : S_1 \rightarrowtail\!\!\!\!\twoheadrightarrow S_2$ and for each $s \in S_1$, $\sigma_s : Y_1 \rightarrowtail\!\!\!\!\twoheadrightarrow Y_2$ are given bijections, and set*

$$\pi(p)(\rho s) = \sigma_s(p(s)) \qquad (p : S_1 \rightharpoonup Y_1, \pi(p) : S_2 \rightharpoonup Y_2). \tag{32}$$

Then π is a poset isomorphism, and every poset isomorphism

$$\pi : (S_1 \rightharpoonup Y_1) \rightarrowtail\!\!\!\!\twoheadrightarrow (S_2 \rightharpoonup Y_2)$$

satisfies (32) *with suitable ρ, $\{\sigma_s\}_{(s \in S_1)}$.*

Proof. We skip the easy verification that the map defined by (32) is a poset isomorphism.

For the converse, fix an isomorphism $\pi : (S_1 \rightharpoonup Y_1) \rightarrowtail\!\!\!\!\twoheadrightarrow (S_2 \rightharpoonup Y_2)$, and for each $s \in S_1$ and each $y \in Y_1$, let

$$p_s^y(t) = \begin{cases} y, & \text{if } t = s, \\ \bot, & \text{otherwise,} \end{cases}$$

so that $p_s^y : S_1 \rightharpoonup Y_1$. Each p_s^y is minimal above $\bot$ (atomic) in $(S_1 \rightharpoonup Y_1)$ and, in fact, every minimal above $\bot$ point of $(S_1 \rightharpoonup Y_1)$ is p_s^y for some s and t. It follows that each π-image $\pi(p_s^y) : S_2 \rightharpoonup Y_2$ is a minimal (above $\bot$) partial function in $(S_2 \rightharpoonup Y_2)$ which converges on a single point in S_2, and so we have functions $\rho : S_1 \times Y_1 \to S_2$ and $\sigma : S_1 \times Y_1 \to Y_2$ so that

$$\pi(p_s^y)(t) = q_{\rho(s,y)}^{\sigma(s,y)}(t) = \begin{cases} \sigma(s,y), & \text{if } t = \rho(s,y), \\ \bot, & \text{otherwise.} \end{cases} \tag{33}$$

(1) *For all s, y_1, y_2, $\rho(s, y_1) = \rho(s, y_2)$.*

Proof. If $\rho(s, y_1) \neq \rho(s, y_2)$ for some $s, y_1 \neq y_2$, then the partial functions $q_{\rho(s,y_1)}^{\sigma(s,y_1)}$ and $q_{\rho(s,y_2)}^{\sigma(s,y_2)}$ are compatible (since they have disjoint domains of convergence) with least upper bound their union

$$q = q_{\rho(s,y_1)}^{\sigma(s,y_1)} \cup q_{\rho(s,y_2)}^{\sigma(s,y_2)};$$

but then the inverse image $\pi^{-1}(q)$ is above both $p_s^{y_1}$ and $p_s^{y_2}$, which are incompatible, which is absurd. □ **(1)**

We let $\rho(s) = \rho(s, y)$ for any (and all) $y \in Y_1$, and we set $\sigma_s(y) = \sigma(s, y)$, so that the basic definition (33) becomes

$$\pi(p_s^y) = q_{\rho(s)}^{\sigma_s(y)}, \qquad (\rho : S_1 \to S_2, s \in S_1, \sigma_s : Y_1 \to Y_2, y \in Y_1). \tag{34}$$

(2) *The map $\rho : S_1 \rightarrowtail\!\!\!\!\twoheadrightarrow S_2$ is a bijection.*

Proof. To see that ρ is an injection, suppose $\rho(s_1) = \rho(s_2)$ and fix some $y \in Y_1$. If $\sigma_{s_1}(y) = \sigma_{s_2}(y)$, then $q_{\rho(s_1)}^{\sigma_{s_1}(y)} = q_{\rho(s_2)}^{\sigma_{s_2}(y)}$, and so $p_{s_1}^y = p_{s_2}^y$, which implies that $s_1 = s_2$, since these two equal partial functions have respective domains of convergence the singletons $\{s_1\}$ and $\{s_2\}$; and if $\sigma_{s_1}(y) \neq \sigma_{s_2}(y)$, then the two partial functions $q_{\rho(s_1)}^{\sigma_{s_1}(y)}$ and $q_{\rho(s_2)}^{\sigma_{s_2}(y)}$ are incompatible, which means that their π-preimages $p_{s_1}^y$ and $p_{s_2}^y$ must also be incompatible—which can only happen if $s_1 = s_2$.

To see that ρ is surjective, suppose $t \in S_2$, fix some $w \in Y_2$, and set

$$r(u) = \begin{cases} w, & \text{if } u = t, \\ \bot, & \text{otherwise.} \end{cases} \qquad (u \in S_2).$$

This is a minimal point in $(S_2 \rightharpoonup Y_2)$, so there exist $s \in S_1$, $y \in Y_1$ such that $\pi(p_s^y) = r$—which means that $q_{\rho(s)}^{\sigma_s(y)} = r$, and hence $t = \rho(s)$, since the respective domains of convergence of these two partial functions are $\{\rho(s)\}$ and $\{t\}$. □ **(2)**

(3) *For each $s \in S_1$, the map $\sigma_s : Y_1 \rightarrowtail\!\!\!\!\!\rightarrow Y_2$ is a bijection.*

Proof. If $\sigma_s(y_1) = \sigma_s(y_2)$, then $q_{\rho(s)}^{\sigma_s(y_1)} = q_{\rho(s)}^{\sigma_s(y_2)}$, so that $p_s^{y_1} = p_s^{y_2}$, which implies $y_1 = y_2$; thus σ_s is an injection. And finally, for each $w \in Y_2$, let

$$r(t) = \begin{cases} w, & \text{if } t = \rho(s), \\ \perp, & \text{otherwise;} \end{cases}$$

this is a minimal point in $(S_2 \rightharpoonup Y_2)$, and so there is some s', y such that

$$\pi(p_{s'}^y) = q_{\rho(s')}^{\sigma_{s'}(y)} = r;$$

by considering the domains of convergence of these two points we conclude as above that $\rho(s') = \rho(s)$, so that $s' = s$ by **(2)**, and then considering their values we get the required $t = \sigma_s(y)$. □ **(3)**

This concludes the proof of the Proposition. □

References

1. P. van Emde Boas. Machine models and simulations. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. A, Algorithms and Complexity*, pages 1–66. Elsevier and MIT Press, 1994.
2. J. McCarthy. A basis for a mathematical theory of computation. In P. Braffort and D Herschberg, editors, *Computer programming and formal systems*, pages 33–70. North-Holland, 1963.
3. Yiannis N. Moschovakis. Abstract recursion as a foundation of the theory of algorithms. In M. M. et. al. Richter, editor, *Computation and Proof Theory*, volume 1104, pages 289–364. Springer-Verlag, Berlin, 1984. Lecture Notes in Mathematics.
4. Yiannis N. Moschovakis. The formal language of recursion. *The Journal of Symbolic Logic*, 54:1216–1252, 1989.
5. Yiannis N. Moschovakis. A mathematical modeling of pure, recursive algorithms. In A. R. Meyer and M. A. Taitslin, editors, *Logic at Botik '89*, volume 363, pages 208–229. Springer-Verlag, Berlin, 1989. Lecture Notes in Computer Science.
6. Yiannis N. Moschovakis. On founding the theory of algorithms. In H. G. Dales and G. Oliveri, editors, *Truth in mathematics*, pages 71–104. Clarendon Press, Oxford, 1998.
7. Yiannis N. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics unlimited – 2001 and beyond*, pages 929–936. Springer, 2001.
8. Vasilis Paschalis. Recursive algorithms and implementations, 2006. (M.Sc. Thesis, in Greek).