1

On founding the theory of algorithms

Yiannis N. Moschovakis

My topic is the problem of "founding" the theory of algorithms, part of the more general problem of "founding" computer science; whether it needs founding—which, I will argue, it does; what should count as a "foundation" for it; and why a specific "mathematical foundation" which I have proposed[1] gives a satisfactory solution to the problem—better than the most commonly accepted "standard" approach. It will be impossible to completely avoid making some comments about the general problem of "founding a mathematical discipline," but I will strive (mostly) to stay away from overly broad generalities, and concentrate on the aspects of the question which are special to algorithms.

The paper splits naturally into two parts: A general introduction in Sections 1 – 4 which lays out the problem and reviews briefly the various approaches to it in the literature, and a more specific (in some places technical) outline of the proposed solution, beginning with Section 5. Before anything else, however, I will start in Section 1 with a theorem and a proof, a simple, elementary fact which is often included in a good, first course in computer science. It will be much easier to understand what I am after using this sample of "computer science talk" (and my slant towards it) as a starting point.

1 The mergesort algorithm

Suppose L is a set with a fixed (total) ordering $\leq$ on it, and let L^* be the set of all *strings* (finite sequences) of members of L. A string $v = \langle v_0, \ldots, v_{m-1}\rangle$ is *sorted* (in non-decreasing order), if $v_0 \leq v_1 \leq \ldots \leq v_{m-1}$, and for each $u \in L^*$, $\mathrm{sort}(u)$ is the sorted "rearrangement" of u,

$$\mathrm{sort}(u) =_{\mathrm{df}} \text{the unique, sorted string } v \text{ such that for some permutation}$$
$$\pi \text{ of } \{0, \ldots, m-1\}, \quad v = \langle u_{\pi(0)}, u_{\pi(1)}, \ldots, u_{\pi(m-1)}\rangle.$$

During the preparation of this paper, the author was partially supported by an NSF Grant.

[1] My first publication on this problem was [9], a broad, discursive paper, with many claims, some discussion and no proofs. This was followed by the technical papers [10, 11, 14], and also [16, 5, 12, 15, 17] on the related problems of the logic of recursion and the theory of concurrent processes. My main purposes here are (a) to return to the original, foundational concerns which motivated [9], and re-consider them in the light of the technical progress which has been achieved since then; and (b) to propose (in Section 7) a modeling of the connection between an algorithm and its implementations, which, in some sense, completes the foundational frame of this program. I have tried hard to make this paper largely independent of the earlier technical work and as broadly accessible as possible, but I have, almost certainly, failed.

The efficient computation of sort(u) is of paramount importance in many computing applications. Most spell-checkers, for example, view a given manuscript as a finite sequence of words and start by "alphabetizing" it, i.e., sorting it with respect to the lexicographic ordering. The subsequent lookup of these words in the dictionary can be done very quickly, so that this initial sorting is the most critical (expensive) part of the spell-checking process.

Among the many sorting algorithms which have been studied in the literature, the **mergesort** is (perhaps) simplest to define and analyze, if not the easiest to implement. It is based on the fact that the sorting function satisfies the equation

$$\mathrm{sort}(u) = \begin{cases} u & \text{if } |u| \leq 1, \\ \mathrm{merge}(\mathrm{sort}(h_1(u)), \mathrm{sort}(h_2(u))) & \text{otherwise,} \end{cases} \tag{1.1}$$

where $|u|$ is the length of u; $h_1(u)$ and $h_2(u)$ are the first and second halves of the sequence u (appropriately adjusted when $|u|$ is odd); and the function merge(v, w) is defined recursively by the equation

$$\mathrm{merge}(v, w) = \begin{cases} w & \text{if } v = \emptyset, \\ v & \text{else, if } w = \emptyset, \\ \langle v_0 \rangle * \mathrm{merge}(\mathrm{tail}(v), w) & \text{else, if } v_0 \leq w_0, \\ \langle w_0 \rangle * \mathrm{merge}(v, \mathrm{tail}(w)) & \text{otherwise.} \end{cases} \tag{1.2}$$

Here $u * v$ is the *concatenation* operation,

$$\langle u_0, \ldots, u_{m-1} \rangle * \langle v_0, \ldots, v_{n-1} \rangle = \langle u_0, \ldots, u_{m-1}, v_0, \ldots, v_{n-1} \rangle,$$

and tail(u) is the "beheading" operation on non-empty strings,

$$\mathrm{tail}(\langle u_0, u_1, \ldots, u_{m-1} \rangle) = \langle u_1, \ldots, u_{m-1} \rangle \quad (\text{for } m > 0).$$

We establish these facts and the main property of the mergesort algorithm in four, simple propositions.

1.1. Lemma. *Equation* (1.2) *determines a unique function on strings, and such that if v and w are sorted, then*

$$\mathrm{merge}(v, w) = \mathrm{sort}(v * w), \tag{1.3}$$

i.e., merge(v, w) is the "merge" of v and w in this case.

Proof is by induction on the sum $|v| + |w|$ of the lengths of the given sequences. If either $u = \emptyset$ or $v = \emptyset$, then (1.2) determines the value merge(v, w) and also implies (1.3), since $\emptyset * u = u * \emptyset = u$. If both v and w are non-empty, then by induction hypothesis

$$\mathrm{merge}(v, \mathrm{tail}(w)) = \mathrm{sort}(v * \mathrm{tail}(w)), \quad \mathrm{merge}(\mathrm{tail}(v), w) = \mathrm{sort}(\mathrm{tail}(v) * w),$$

and then (1.2) yields immediately that merge$(v, w) = $ sort$(v * w)$, as required. $\dashv$

1.2. Lemma. *For each v and w,* merge(v, w) *can be computed from* (1.2) *using no more than $|v| + |w| - 1$ comparisons of members of L.*

Proof is again by induction on $|v| + |w|$, and at the basis, when either $v = \emptyset$ or $w = \emptyset$, (1.2) gives the value of $\text{merge}(v, w)$ using no comparisons at all. If both v and w are non-empty, then we need to compare v_0 with w_0 to determine which of the last two cases in (1.2) applies, and then (by the induction hypothesis) no more than $|v| + |w| - 2$ additional comparisons to complete the computation. ⊣

1.3. Lemma. *The sorting function* $\text{sort}(u)$ *satisfies equation* (1.1).

Proof. If $|u| \leq 1$, then u is sorted, and so $\text{sort}(u) = u$, in agreement with (1.1). If $|u| \geq 2$, then the second case in (1.1) applies, and by Lemma 1.1,

$$\text{merge}(\text{sort}(h_1(u)), \text{sort}(h_2(u))) = \text{sort}(\text{sort}(h_1(u)) * \text{sort}(h_2(u))) = \text{sort}(u),$$

as required. ⊣

1.4. Lemma. *If* $|u| = 2^n$, *then* $\text{sort}(u)$ *can be computed from* (1.1) *using no more than* $n \cdot 2^n$ *comparisons of members of* L.

Proof. By induction on n, the result is immediate when $n = 0$, since (1.1) yields $\text{sort}(u) = u$ using no comparisons when $u = \langle u_0 \rangle$ has length $2^0 = 1$. If $|u| = 2^{n+1}$, then each of the halves of u has length 2^n, and the induction hypothesis guarantees that we can compute $\text{sort}(h_1(u))$ and $\text{sort}(h_2(u))$ by (1.1) using no more than $(n-1) \cdot 2^{n-1}$ comparisons for each, i.e., $(n-1) \cdot 2^n$ comparisons in all; by Lemma 1.2 now, the computation of $\text{merge}(\text{sort}(h_1(u)), \text{sort}(h_2(u)))$ can be done by (1.3) using no more than $2^n - 1 < 2^n$ additional comparisons, for a grand total of $n \cdot 2^n$. ⊣

If we define the "binary logarithm" of a positive number by

$$\log_2(m) = \text{the least } n \text{ such that } m \leq 2^n,$$

then Lemma 1.4 (with a bit of arithmetic) yields easily the main result we have been after:

1.5. Theorem. *The mergesort algorithm sorts a string of length* n *using no more than* $n \cdot \log_2(n)$ *comparisons.* ⊣

It is an important result, because the number of required comparisons is a very reasonable measure of complexity for a sorting algorithm, and it can be shown that $n \log_2(n)$ *is asymptotically the least number of comparisons required to sort a string of length* n.

1.6. Programming considerations. The mergesort is a recursive algorithm, and so it is easiest to express in a relatively rich programming language which understands recursion, like `Pascal`, `C`, or `Lisp`—in fact, all that is needed is to re-write equations (1.1) and (1.2) in the rigid syntax of these languages;[2] it is correspondingly difficult to express it directly in the assembly language of

[2] I am cheating just a bit here: this re-write is easy if the language can deal with *strings* (as `Lisp` and some extensions of the others do), but a bit cumbersome if we must first "teach" the language the basic operations on strings.

some machine, because in that case we must first implement recursion, which is not a simple matter. In addition, whether produced by the compiler of a high-level language or by hand, the implementation of the mergesort requires a good deal of space and (as with all implementations of recursive algorithms), it may be slow. Because of these reasons, the mergesort is not often used in practice, despite its simplicity and optimality.

2 Deconstruction

Before going on to learn that most of the preceding section was really meaningless gibberish, the conscientious reader should re-read it and make sure that, in fact, it makes perfect sense—except, perhaps, for the last paragraph which turned the computerese up a bit.

Lemmas 1.1 and 1.3 make straight-forward, mathematical assertions about the merging and sorting functions, and their proofs are standard. Not so with Lemmas 1.2 and 1.4: they proclaim that the values of these functions *can be computed from equations* (1.2) *and* (1.1) *using no more* than some *number of comparisons*. Evidently, these lemmas are not just about the merging and sorting functions, but also about *computations*, *numbers of comparisons*, and (more significantly) about the specific equations (1.2) and (1.1). We understand the proof of Lemma 1.2, for example, by reading equation (1.2) as an (implicit) definition of a *computation procedure*:

2.1. The merging algorithm. *To compute* $\mathrm{merge}(v, w)$, *look first at* v*; if* $v = \emptyset$, *give output* w*; otherwise, if* $w = \emptyset$, *give output* v*; otherwise, if* $v_0 \leq w_0$, *compute* $z = \mathrm{merge}(\mathrm{tail}(v), w)$ *and give output* $\langle v_0 \rangle * z$*; and if none of the preceding cases applies, compute* $z = \mathrm{merge}(v, \mathrm{tail}(w))$ *and give output* $\langle w_0 \rangle * z$.

And here is the corresponding reading of (1.1) which we need for the proof of 1.4:

2.2. The mergesort algorithm. *To sort a string* u, *check first if* $|u| \leq 1$, *and if this is true, give output* u*; otherwise, sort separately the first and the second half of* u, *and then merge the values by the procedure* 2.1.

But these elaborations are not enough: We also made in the proofs of 1.2 and 1.4 certain assumptions about the "making" and "counting" of "comparisons" by the computation procedure we extracted from equations (1.2) and (1.1). In the proof of 1.4, for example, we assumed that *if we need* C_1 *comparisons to sort* $h_1(u)$ *and* C_2 *comparisons to sort* $h_2(u)$, *then, altogether we need* $C_1 + C_2$ *comparisons to* (separately) *sort both of these strings.* These are very natural assumptions, to be sure, as are the interpretations of equations (1.2) and (1.1)—which is why the proofs in Section 1 appear to be solid. Suppose, however, that in the middle of a mathematical seminar talk about some operator $T(f)$ on Hilbert space, the lecturer appeals to the equation

$$T(f + g) = T(f) + T(g);$$

he or she would be immediately challenged to prove that $T(f)$ is additive, starting (presumably) with a precise definition of $T(f)$, if one has not been given. What is missing in Section 1 are precise (mathematical) definitions of *algorithms*, *uses of comparisons*, etc., and rigorous proofs, from the definitions, of the basic properties of algorithms on which the arguments were grounded.

I have called **algorithms** these purposeful interpretations of equations (1.2) and (1.1), but *computation procedures* or *effective, deterministic instructions* could do as well (for now)—all these words are used in computer science literature, more-or-less interchangeably.

2.3. Implementations. The second paragraph of Section 1 starts with the comment that [among sorting algorithms]

> ... the mergesort is (perhaps) simplest to define and analyze, if not the easiest to implement,

and the last paragraph 1.6 elaborates on the issue. Lots of new words and claims are thrown around in 1.6: It is asserted that "the mergesort is a recursive algorithm" which can be "expressed in `Pascal` or `Lisp`"; that "it is not a simple matter to implement recursion [in an assembly language]"; that "the implementation of the mergesort requires a lot of space", etc. The innocent reader should take it on faith that all of this makes perfect, common sense to an experienced programmer, and also that very little of it has ever been defined properly. Now "not the easiest" and "a lot of space" will never be made precise, to be sure, but this kind of talk suggests that programmers understand and (generally) affirm the following:

(1) A given algorithm can be expressed (programmed, implemented) in different programming languages, and so (in particular), an algorithm has many implementations.

(2) Implementations have important properties, e.g., the time and space needed for their execution.

2.4. Moral. To found the theory of algorithms, we must define precisely its basic notions, starting with *algorithms*, *implementations*, and the relation between a given algorithm and its various implementations; and it is important that this be done so that the arguments in Section 1 are endowed with precise meaning very nearly in their present form, because these simple, intuitive ideas are so natural and appealing as to cast doubt on the necessity for rigor.

3 How do we define basic notions?

The Moral 2.4 declares that we should give *precise definitions* of algorithms and implementations, but there is more than one way to go about this. Consider the following three, different approaches (one with two flavors), starting with the "standard" one, which, in fact, I will adopt.

3.1. (I) Define them in set theory. This is certainly the "orthodox" method of making notions precise in modern mathematics: To "found" number theory, we define the whole numbers and the operations on them in set theory; to "found" analysis, we give rigorous, set-theoretic definitions of the real numbers, functions, limits, derivatives, etc.; to "found" probability theory, we declare that "a random variable is a measurable function on a probability space," right after we give precise, set-theoretic definitions of all the words within the quotes.

Despite its wide acceptability by working mathematicians, this kind of "set-theoretic foundation" for a mathematical theory has been attacked by many philosophers, most seriously Benacerraf [1], and also by some mathematicians; Saunders MacLane has entertained generations of audiences by asking plaintively in countless lectures,

> does anybody, *seriously* think that $2 = \{\emptyset, \{\emptyset\}\}$?

Probably not, but the von Neumann ordinal $\{\emptyset, \{\emptyset\}\}$ clearly "codes" all the properties of two-element sets which depend only on their cardinality; somewhat more fancifully, $\{\emptyset, \{\emptyset\}\}$ *models faithfully* the number 2 (whatever that is) up to *equinumerosity*—as, in fact, does any two-element set. For some less trivial examples, any *Peano system* $(M, 0, S)$ models faithfully "the natural numbers" (whatever they are), up to first-order isomorphism;[3] and any countable, dense linear ordering without endpoints models faithfully "the order type" η of the rational numbers (whatever that is), up to order isomorphism.[4,5]

The proper role of a "set-theoretic definition" of a mathematical notion C is not to tell us in ultimate, metaphysical terms exactly *what* the C-objects (those which fall under C) *are*, but to identify and delineate their fundamental, mathematical properties. Typically, we do this by specifying a class of sets M_C and an equivalence relation $\sim_C$ on M_C, with the intention that each $\alpha \in M_C$ *faithfully represents* (codes) some C-object α_C, and that two members $\alpha, \beta \in M_C$ code the same C-object exactly when $\alpha \sim_C \beta$. A modeling of this type is successful if the $\sim_C$-invariant properties of the members of M_C capture exactly the fundamental properties of the C-objects—which implies that every fundamental property of a C-object can be "read off" any of its codes.[6]

---

[3] A triple $(M, 0, S)$ is a Peano system if M is a set; $0 \in M$; $S : M \to M \setminus \{0\}$ is an injection; and every subset X of M which contains 0 and is closed under S exhausts M. All foundations of the natural numbers start with the facts that (a) *there exists a Peano system*, and (b) *any two Peano systems are isomorphic*, and differ only in what they do with them.

[4] Any two countable, dense linear orderings with no endpoints are order isomorphic (Cantor).

[5] In fact, I believe that most mathematical theories (and all the non-trivial ones) can be clarified considerably by having their basic notions *modeled faithfully in set theory*; that for many of them, a (proper) set-theoretic foundation is not only useful but necessary—in the sense that their basic notions cannot be satisfactorily explicated without reference to fundamentally set-theoretic notions; and that set-theoretic foundations of mathematical theories can be formulated so that they are compatible with a large variety of views about truth in mathematics and the nature of mathematical objects. Without explaining it in detail or defending it, the textbook [13] applies this view consistently to the presentation of the elementary theory of sets and its applications. The brief remarks here are only meant to clarify what I aim to do with algorithms in the more technical sections of the paper, following this one.

[6] Sometimes we can do more and choose C so that $\sim_C$ is the identity relation on C, notably in the case of Cantor's *ordinal numbers* where the class of von Neumann ordinals has this property. In other cases this is not possible: For example, Cantor dealt with *linear order types* exactly as he dealt with ordinal numbers, but (apparently) there is no way to define in Zermelo-

For the case of algorithms, I will first introduce the class of *recursors*, which model the "mathematical structure of algorithms" (much like measurable functions on probability spaces model random variables), and the relation of *recursor isomorphism* between them, which models "algorithm identity". Algorithms, however, do not make sense absolutely, but only with respect to certain "data" and certain "given" (possibly higher-order) operations on these data, relative to which they are "effective"; for the full modeling, then, I will also introduce the appropriate *structures* which model such data+givens contexts (up to *structure isomorphism*), and finally claim that the recursors which are *explicitly and immediately definable* (in a specific, precise sense) on each structure $\mathfrak{M}$ model faithfully "the algorithms of $\mathfrak{M}$".

3.2. (II) Deny that they exist. In the original, "naive" development of the calculus, there were real numbers, variables, limits, infinitesimals, differentials and many other things. Some of these were eventually given rigorous, set-theoretic definitions, perhaps not always completely faithful to their naive counterparts, but close enough; for example, a real-valued *function* is not exactly the same thing as a *dependent variable* and the modern notion of a *differential* is far removed from the classical one, but we can still recognize the old objects in their precise counterparts. There are, however, no *infinitesimals* in (standard) modern analysis; classical statements about infinitesimals are viewed as informal (and vague) "ways of speaking" about real numbers, functions and limits, and they must be replaced by precise statements which make no reference to them and (roughly) mean the same thing.

There are two, wildly different approaches to the foundations of computer science which treat algorithms as "pre-mathematical" notions, to be denied rather than defined.

3.3. (IIa) Algorithms as implementations. By this "standard view", especially popular among complexity theorists, there are no algorithms, only *implementations*, variously called *machines* or *models of computations*;[7] these are modeled in set theory; and assertions about algorithms like those in Section 1 are understood as informal "ways of speaking" about implementations. I will discuss this approach in detail Section 4.

3.4. (IIb) Algorithms as constructive proofs. Another, more radical proposal which also denies independent existence to algorithms is the claim that *algorithms are implicitly defined by constructive proofs.* Consider, for example, an assertion of the form

$$\phi \equiv (\forall x \in A)(\exists y \in B)P(x, y). \tag{3.1}$$

Fraenkel set theory a class of linearly ordered sets which contains exactly one representative from each order isomorphism class. Because of this, "linear order types" can be "defined in set theory" only in the minimal way described here, but their study does not appear to have suffered because of this defect.

[7] Not all who adopt it will approve of my description of this view: In his classic [7], for example, Knuth dubs "algorithms" (essentially) what I call "implementations" and avoids altogether the second word. It amounts to the same thing.

A constructive proof of ϕ should naturally yield an algorithm for computing a function $f : A \to B$, such that

$$(\forall x \in A)P(x, f(x)),$$

and there exists a considerable body of work verifying this for formalized systems of constructive mathematics, typically using various notions of *realizability* or (considerably deeper) applications of the Gentzen *cut elimination* procedure. To pursue the reduction suggested here, however, one needs to argue the converse: that statements about algorithms (in general) are really assertions about constructive proofs, and that they can be re-formulated so that all references to "algorithms" are eliminated.[8]

One problem with this view is that algorithms "support" many auxiliary notions, like "number of comparisons," "length of computation," etc., which are not usually associated with proofs. Girard, who is its foremost expositor, has introduced *linear logic* partly in an attempt to associate with proofs some of these notions, especially an account of *use of resources* which is often important in algorithm analysis. I suppose one could re-prove the results of Section 1 in some dialect of linear logic, and show that *no more than* $n\cdot\log_2(n)$ *"assumptions" of comparisons are needed to prove that* $\mathrm{sort}(u)$ *is defined*, if u has length n. This, or something very much like it, would be the assertion about constructive proofs which captures the meaning of Theorem 1.5. Now, some considerable effort is required to do this proof-theoretic analysis, and, in the end (I believe) one will again need to write down and argue from the all-important equations (1.2) and (1.1); but the *mere* (classical) *truth* of these equations suffices to "yield the algorithm" and its basic property, and so I do not see the foundational significance of constructing the linear logic proof.

Although I doubt seriously that algorithms will ever be eliminated in favor of constructive proofs (or anything else, for that matter), I think that this view is worth pursuing, because it leads to some very interesting problems. With specific, precise definitions of algorithms and constructive proofs at hand, one could investigate whether, in fact, every algorithm can be extracted (in some concrete way) from some associated, constructive proof. Results of this type would add to our understanding of the important connection between *computability* and *constructivity*.

3.5. (III) Axiomatize their theory. This is what we do for set theory: Can we similarly take "algorithms", "implementations", and whatever else we need as *primitive notions* and formulate a reasonable axiomatic theory which will make sense out of computer science talk such as that in Section 1?

I am trying to ask a methodological question here, one which could be answered without making a commitment to any specific philosophy of mathematics.

[8]Still more radical would be to simply *define* "algorithm" to be *constructive proof of an assertion of the form* (3.1), but I cannot recall seeing this view explained or defended.

We can understand a proposed set of axioms for a theory T *formally*,[9] as being "all there is to T"; *realistically*, as expressing some important truths about the fundamental objects and notions of T, which exist independently of what we choose to say about them; and, surely, in many more, subtler ways. It seems, however, that the foundational value of a specific axiomatization (how much it helps us to understand T) is independent of our general view of the axiomatic method. It has more to do with the choice of primitives and axioms, and what the development of T from them reveals about T.[10]

I will also exclude from this option the kind of "second-order axiomatizations" which accept (uncritically, as part of logic) quantification over *all* subsets of the domain. It is often claimed, for example, that the Peano axioms provide a foundation of arithmetic in second order logic, because of the "categoricity" theorem (b) in Footnote 3. This is true, as far as it goes, but we cannot account for all uses of whole numbers in mathematics by appealing to such an *external* (metamathematical) interpretation of (b): In many important applications we need to understand (b) *internally* (as part of our mathematics), for example, to prove that "every two complete, ordered fields are isomorphic".[11] This problem is even more severe for complex notions like algorithms (or topological spaces, random variables, etc.) whose basic properties are explicitly and irreducibly set-theoretic: Second order "axiomatizations" can yield (at most) a poor shadow of the account of them that we need to understand their uses in mathematics.

What remains is the possibility of an axiomatization of computer science whose natural formalization would be in first-order logic, or (at least) in a many-sorted, first order logic, where some of the basic sets are fixed to stand for *numbers* (so we can talk of "the number of comparisons" or "the number of steps" in a computation) and a few other, mathematical objects. The trouble now is that the theory is too complex: There are too many notions competing for primitive status (algorithms, implementations and computations, at the least) and the relations between them do not appear to be easily expressible in first-order terms. I doubt that the project can be carried through, and, in any case, there are no proposals on the table suggesting how we might get started.

3.6. Syntax vs. semantics. Finally, I should mention—and dismiss outright—various, vague suggestions in computer science literature that *algorithms*

[9] Here "formally" means "without regard to meaning" and not (necessarily) "in a formal language". A coherent axiomatization in ordinary language can always be "formalized," in the trivial sense of making precise the syntax of the relevant fragment of English and the logic; whether (and *how*) the formal version corresponds to the naive one is hard to talk about, and involves precisely the philosophical issues about axiomatizations which I am trying to avoid.

[10] Zermelo's axiomatization of set theory is a good example of this. It was first proposed in Zermelo [22] quite formally, as an expedient for avoiding inconsistency, and only much later in Zermelo [21] was it justified on the basis of a realistic, intuitive understanding of the cumulative hierarchy of sets. By the time this happened, the axioms (augmented with replacement) had been well established and there was no doubt of their value both in developing (technically) and in understanding the theory of sets.

[11] This and the fundamentally set-theoretic nature of (b) in Footnote 3 are part of the argument for the "necessity" of set-theoretic foundations alluded to in Footnote 5.

are syntactic objects, e.g., *programs*. Perhaps Frege [2] said it best:

> This connection [between a sign and its denotation] is arbitrary. You cannot forbid the use of an arbitrarily produced process or object as a sign for something else.

In the absence of a precise semantics, `Pascal` programs are just meaningless scribbles; to read them as algorithms, we must first interpret the language—and it is then the *meanings* attached to programs by this interpretation which are the algorithms, not the programs themselves.[12]

4 Abstract machines and implementations

The first definition of an *abstract machine* was given by Turing, in the classic [20]. Without repeating here the well-known definition (e.g., see [6]),[13] we recall that each *Turing machine* M is equipped with a "semi-infinite tape" which it uses both to compute and also to communicate with its environment: To determine the value $f(n)$ (if any) of the partial function[14] $f : \mathbb{N} \rightharpoonup \mathbb{N}$ computed by M, we put n on the tape in some standard way, e.g., by placing $n+1$ consecutive 1s at its beginning; we start the machine in some specified, initial, internal state q_0 and looking at the leftmost end of the tape; and we wait until the machine stops (if it does), at which time the value $f(n)$ can be read off the tape, by counting the successive 1s at the left end. Turing argued that *the number-theoretic functions which can* (in principle) *be computed by any deterministic, physical device are exactly those which can be computed by a Turing machine*, and the corresponding version of this claim for partial functions has come to be known as the *Church-Turing Thesis*, because an equivalent claim was made by Church at about the same time. Turing's brilliant analysis of "mechanical computation" in [20] and a huge body of work in the last sixty years has established the truth of the Church-Turing Thesis beyond reasonable doubt; it is of immense importance in the derivation of foundationally significant *undecidability results* from technical theorems about Turing machines, and it has been called "the first natural law of pure mathematics."

Turing machines capture the notion of *mechanical computability of number-theoretic functions*, by the Church-Turing Thesis, but they do not model faith-

[12] It has also been suggested that we do not need algorithms, only the equivalence relation which holds between two programs P and Q (perhaps in different programming languages) when they (intuitively) *express the same algorithm*. It is difficult to see how we can do this for all programming languages (current and still to be invented) without a separate notion of algorithm; and, in any case, if we have a good notion of "program equivalence", we can then "define" algorithms to be the equivalence classes of this equivalence and solve the basic problem.

[13] Turing machines are modeled in set theory by finite sets of tuples of some form, but their specific representation does not concern us here.

[14] A *partial function* $f : X \rightharpoonup W$ is an (ordinary, total) function $f : D_f \to W$, from some subset $D_f \subseteq X$ of X into W; *or* (equivalently) a (total) function $f : X \to W \cup \{\bot\}$, where $\bot \notin W$ is some fixed object "objectifying" the "undefined," so that "$f(x)$ is undefined" is the same as "$f(x) = \bot$". For most of what we do here it does not matter, but the official choice for this paper is the second one, so that "$f : X \rightharpoonup W$" is synonymous with "$f : X \to W \cup \{\bot\}$".

fully the notion of *mechanical computation*. If, for example, we code the input by putting the argument n on the tape in *binary*[15] (rather than *unary*) notation (using no more than $\log_2(n)$ 0s and 1s), then the time needed for the computation of $f(n)$ can sometimes be considerably shortened; and if we let the machine use two tapes rather than one, then (in some cases) we may gain a quadratic speedup of the computation, see [8]. This means that important aspects of the complexity of computations are not captured by Turing machines. We consider here a most general notion of *model of computation*, which (in particular) makes the mode of input and output part of the "machine".

4.1. Iterators. For any two sets X and W, an **iterator** $\phi : X \rightsquigarrow W$ is a quadruple $(\text{input}, S, \sigma, T, \text{output})$, where:

(1) S is an arbitrary (non-empty) set, the *set of states* of ϕ;
(2) $\text{input} : X \to S$ is the *input function* of ϕ;
(3) $\sigma : S \to S$ is the *transition function* of ϕ;
(4) $T \subseteq S$ is the set of *terminal states* of ϕ, and $s \in T \Longrightarrow \sigma(s) = s$; and
(5) $\text{output} : T \to W$ is the *output function* of ϕ.

The **computation** of ϕ for a given $x \in X$ is the sequence of states $\{s_n(x)\}_{n\in\mathbb{N}}$ defined recursively by

$$\begin{aligned} s_0(x) &= \text{input}(x), \\ s_{n+1}(x) &= \begin{cases} s_n(x) & \text{if } s_n(x) \in T, \\ \sigma(s_n(x)), & \text{otherwise;} \end{cases} \end{aligned}$$

the **computation length** on the input x (if it is finite) is

$$\ell(x) = (\text{the least } n \text{ such that } s_n(x) \in T) + 1;$$

and the partial function $\overline{\phi} : X \rightharpoonup W$ **computed by** ϕ is defined by the formula

$$\overline{\phi}(x) = \text{output}(s_{\ell(x)}(x)).$$

Each Turing machine M can be viewed as an iterator $M : \mathbb{N} \rightsquigarrow \mathbb{N}$, by taking for states the (so-called) "complete configurations" of M, 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.

It is generally conceded that this broad notion of iterator can model the manner in which every conceivable (deterministic, discrete, digital) mechanical device computes a function, and so it captures the *structure of mechanical computation*. It is too wide to capture the *effectivity of mechanical computation*, because it allows an arbitrary set of states and arbitrary input, transition and output functions, but (for the moment) I will disregard this problem; it is easy enough to solve by imposing definability or finiteness assumptions on the components of

[15] The binary representation of a natural number n is the unique sequence $a_k a_{k-1} \cdots a_0$ of 0s and 1s (with $a_k = 1$, unless $n = 0$), such that $n = a_0 + 2a_1 + 2^2 a_2 + \cdots + 2^k a_k$.

iterators, similar to those of Turing machines, see 8.4. The question I want to address now is whether the notion of iterator is *wide enough* to model faithfully *algorithms*, as it is typically assumed in complexity theory;[16] put another way,

$$\text{are algorithms the same as mechanical computation procedures?} \tag{4.1}$$

A positive answer to this question expresses more precisely the view (IIa) in 3.3, and it might appear that it is the correct answer, especially as we have been using the two terms synonymously up until now. There are, however, at least two serious problems with this position.

4.2. Recursion and iteration. If all algorithms are modeled by iterators, then which iterator models the mergesort algorithm of Section 1? This was defined implicitly by the *recursive equations* (1.1) and (1.2) (or so we claimed in Section 1), and so we first need to transform the intuitive computation procedure which we extracted from these equations into a precise definition of an iterator. The problem is not special to the mergesort, which is just one of many important examples of *recursive algorithms* defined by systems of recursive equations.

To clarify the situation, consider the following description of an arbitrary iterator $\phi = (\text{input}, S, \sigma, T, \text{output})$ by a *while-program* in a pidgin, `Pascal`-like programming language:

$s := \text{input}(x);$
`while` $(s \notin T)\ s := \sigma(s);$
$w := \text{output}(s);$
`return` w.

We do not need any elaborate, precise definitions of the semantics of while-programs to recognize that this one (naturally understood) defines ϕ, and that, conversely, the algorithm expressed by any program built with assignments and while-loops can be directly modeled by an iterator. The first problem, then, is how to construct while-programs which express the intuitive, computation procedures implicit in systems of recursive equations like (1.1) and (1.2).

This can be done, in many different ways generally called *implementations of recursion*.[17] These methods are not simple, but they are precise enough so that they can be automated: For example, one of the most important tasks of

[16] Knuth [7] (essentially, in the present terminology) defines an *algorithm* to be an iterator $\phi : X \rightsquigarrow W$, which also satisfies the additional hypothesis that *for every* $x \in X$, *the computation terminates*. This termination restriction is reminiscent of the view (IIb) in 3.4, and it is hard to understand in the context of Knuth's own (informal) use of the notion. Suppose, for example, that Prof. Prewales had proposed in 1990 a precise, mechanical procedure which searched (in a most original and efficient way) for a minimal counterexample to Fermat's last theorem; would we not have called this an "algorithm," just because Prewales could not produce a proof of termination? And what would be the "meaning" of the `Pascal` program produced by Prewales, which (by general agreement) implemented his procedure? It seems more natural to say that Prewales had, indeed, defined an algorithm, and to say this even now, when we know that the execution of his program is doomed to diverge.

[17] In the simplest of the classical, "sequential" methods for implementing recursion, the most important part of the state is a "stack", a finite sequence of pieces of information which (roughly) reminds the machine what it was doing before starting on the "recursive call" just

a *compiler* for a "higher level" language like `Pascal` is exactly this conversion of *recursive programs* to *while-programs*, in the assembly language of a specific processor (a concrete, physical iterator, really), which can then run them.

Assume then that we associate with each system E of recursive equations (like (1.1) and (1.2)) an iterator ϕ_E, using some fixed "compilation process", and we make the view (IIa) in 3.3 precise by calling ϕ_E *the algorithm defined by* E. Now the **first problem** with this view is that ϕ_E is far removed from E and the resulting, rigorous proofs of the important properties of ϕ_E are complex and only tenuously related to the simple, intuitive arguments outlined in Section 1.

The complaint is not so much about the mere complexity of the rigorous proofs, because it is not unusual for technical complications to crop up when we insist on full rigor in mathematics. It is the artificiality and irrelevance of many of the necessary arguments which casts doubt on the approach, as they deal mostly with the specifics of the compilation procedure rather than the salient, mathematical properties of algorithms. Still, this is not a fatal objection to (IIa), only an argument against it, on the grounds that the loss of elegance and simplicity which it requires is out of proportion with the gain in rigor that it yields.

4.3. The non-uniqueness of compilation. The **second problem** with the view (IIa) is that there are many ways to "compile" recursive programs—to assign an iterator ϕ_E to each system of recursive equations E—and there is no single, natural way to choose any one of them as "canonical". This is a most serious problem, I think, which makes it very unlikely that we can usefully identify algorithms with computational procedures, or iterators.

Take the mergesort, for example, express it formally in `Pascal`, `C` and `Lisp`, and suppose ϕ_P, ϕ_C and ϕ_L are the iterators which we get when we compile these programs in some specific way for some specific processor. Each of these three iterators has equal claim to be "the mergesort algorithm" by (IIa), and there is no obvious way to choose among them. More significantly (because we might allow ourselves some arbitrary choice here), these three iterators, obviously, have something in common, but exactly

$$\text{what is the relation between } \phi_P,\ \phi_C \text{ and } \phi_L? \qquad (4.2)$$

The natural answer is that

$$\text{they are all implementations of the mergesort algorithm,} \qquad (4.3)$$

but, of course, we cannot say this without an independent notion of *the mergesort algorithm.* Even if we give up on making precise and answering fully Question (4.2), we would still like to say that

completed. There are also "parallel" implementations, in which the "stack" is replaced by a "tree" (or other structure) of "processes" which "communicate" among themselves in predetermined ways. This listing of buzzwords is as far as I can go here in suggesting to the knowledgable reader the reduction procedures to which I am alluding.

> every computational procedure extracted from the recursive equations (1.1) and (1.2) satisfies Lemmas 1.2 and 1.4

(suitably formulated for iterators), and it is hard to see how we can express this without making reference to some one, semantic object, assigned directly to (1.1) and (1.2) and with a prior claim to model *the mergesort algorithm*.

4.4. Proposal I: Implementations are iterators. From this discussion, it seems to me most natural to assume that *iterators model implementations*, which are special, "iterative algorithms," and that results such as Lemmas 1.2 and 1.4 are about more abstract objects, whatever we decide to call them; each of these objects, then, may admit many implementations, and codes the "implementation independent" properties of algorithms.

5 The theory of recursive equations

To motivate our choice of set-theoretic representations of algorithms in the next section, let us first outline rigorous formulations and proofs of the results in Section 1 in the context of the *theory of recursive equations*. This is a simple, classical theory, whose basic results are very similar in flavor to those of the *theory of differential equations*.

A *poset* (partially ordered set)[18] $(D, \leq_D)$ is *inductive* or *complete* if every chain (linearly ordered subset) $A \subseteq D$ has a least, upper bound, $\sup A$, and a mapping (function)

$$\pi : D \to E$$

on one poset to another is *monotone* if

$$d \leq_D d' \Longrightarrow \pi(d) \leq_E \pi(d').$$

The basic fact about complete posets is that monotone mappings have least fixed points, in the following, strong sense:

5.1. The monotone, least fixed point theorem. *If $\pi : X \times D \to D$ is a monotone mapping on the poset product $X \times D$ to D, and if D is inductive, then, for each $x \in X$, the equation*

$$d = \pi(x, d) \quad (x \in X, d \in D)$$

[18] A poset is a structure $(D, \leq_D)$, where the binary relation $\leq_D$ on D satisfies the conditions (a) $d \leq_D d$; (b) $d_1 \leq_D d_2 \;\&\; d_2 \leq_D d_3] \Longrightarrow d_1 \leq_D d_3$; and (c) $[d_1 \leq_D d_2 \;\&\; d_2 \leq_D d_1] \Longrightarrow d_1 = d_2$. Every set X is a *discrete* poset with the identity relation, $x_1 \leq_X x_2 \iff x_1 = x_2$; and for every W and $\perp \notin W$, the set $W \cup \{\perp\}$ is a *flat* poset, with $x \leq y \iff x = \perp \vee x = y$. Since the empty set is (trivially) a chain and its least upper bound (when it exists) is easily the least element of D, every inductive poset has a least element $\perp_D = \sup \emptyset$. It can be shown that a poset $(D, \leq_D)$ is inductive exactly when it has a least element and every non-empty, directed subset of D has a supremum. There is a tendency in recent computer science literature to widen the notion by omitting the requirement that D has a least element, which is why I am avoiding the common term *dcpo* for these structures. Computer scientists also tend to study only *continuous* (in the appropriate, *Scott topology*) rather than the more general *monotone* mappings, which makes the theory easier but not general enough to cover all the applications that we need here. The basic facts about inductive sets and monotone mappings can be found in most textbooks on *denotational semantics* and in some set theory books, e.g., [13].

has a least solution

$$d(x) = (\mu d \in D)[d = \pi(x, d)],$$

characterized by the conditions

$$d(x) = \pi(x, d(x)), \quad (\forall e \in D)[e \leq_D \pi(x, e) \Longrightarrow d(x) \leq_D e];$$

in addition, the function $x \mapsto d(x)$ *is monotone on* X *to* D.[19]

The simplest, interesting inductive posets are the partial function spaces

$$(A \rightharpoonup B) = \{p \mid p : A \rightharpoonup B\} \quad (= \{p \mid p : A \to B \cup \{\bot\}\})$$

partially ordered "pointwise,"

$$\begin{aligned} p \leq q \iff & (\forall x \in A)[p(x) \leq q(x)] \\ \iff & (\forall x \in A, y \in B)[p(x) = y \Longrightarrow q(x) = y], \end{aligned}$$

and products of these, i.e., spaces of pairs (or tuples) of partial functions. To apply Theorem 5.1 to the sorting problem of Section 1, for example, we need the posets $(L^* \rightharpoonup L^*)$ and $(L^* \times L^* \rightharpoonup L^*)$ which contain the functions sort and merge, and also the poset

$$(L \times L \rightharpoonup \{\mathit{ff}, \mathit{tt}\}),$$

where $\{\mathit{ff}, \mathit{tt}\}$ is some arbitrary set of two, distinct objects standing for *falsity* and *truth* and which contains the *characteristic function*

$$\chi_{\leq}(s, t) = \begin{cases} \mathit{tt}, & \text{if } s \leq t, \\ \mathit{ff}, & \text{if } t < s \end{cases}$$

of the given ordering on L. In general, a partial function $c : L \times L \rightharpoonup \{\mathit{ff}, \mathit{tt}\}$ can be viewed as the *characteristic partial function*, of a *partial, binary relation* on L. The idea is to generalize the problem, and try to find (partial) "merging" and "sorting" functions, relative to an arbitrary partial relation $c : L \times L \rightharpoonup \{\mathit{ff}, \mathit{tt}\}$, which stands for some approximation to a total ordering. We can get this very easily from Theorem 5.1: *for each* $c : L \times L \rightharpoonup \{\mathit{ff}, \mathit{tt}\}$, *there exist partial functions*

$$\mathrm{sort}(c) : L^* \rightharpoonup L^* \quad \text{and} \quad \mathrm{merge}(c) : L^* \times L^* \rightharpoonup L^*,$$

which are (least) *solutions of the recursive equations*

$$\mathrm{sort}(c)(u) = \begin{cases} u & \text{if } |u| \leq 1, \\ \mathrm{merge}(c)(\mathrm{sort}(c)(h_1(u)), \mathrm{sort}(c)(h_2(u))) & \text{otherwise,} \end{cases} \tag{5.1}$$

$$\mathrm{merge}(c)(v, w) = \begin{cases} w & \text{if } v = \emptyset, \\ v & \text{else, if } w = \emptyset, \\ \langle v_0 \rangle * \mathrm{merge}(c)(\mathrm{tail}(v), w) & \text{else, if } c(v_0, w_0) = \mathit{tt}, \\ \langle w_0 \rangle * \mathrm{merge}(c)(v, \mathrm{tail}(w)) & \text{else, if } c(v_0, w_0) = \mathit{ff}; \end{cases} \tag{5.2}$$

[19] Various versions of this basic fact have been attributed to different mathematicians, but a special case (with a proof which suffices for the full result) is already a subroutine of Zermelo's first proof of the Wellordering Theorem in [23].

and which depend monotonically on $c : L \times L \rightharpoonup \{ff, tt\}$. If $\leq$ is the given ordering on L, then merge$(\chi_\leq)$ and sort$(\chi_\leq)$ are obviously the merging and sorting functions we need; and on the other hand, using *exactly* the arguments (by induction on $|v| + |w|$ and $|u|$) for Lemmas 1.2 and 1.4, we can show the following:

5.2. Theorem. *Suppose* sort(c) *and* merge(c) *are monotonic functions of* $c : L \times L \rightharpoonup \{ff, tt\}$ *which satisfy the recursive equations* (5.1) *and* (5.2).

(a) *If* merge$(c)(v, w) = z \in L^*$, *then there exists a partial function* $c' \leq c$ *which is defined on at most* $|v| + |w| - 1$ *pairs, and such that* merge$(c')(v, w) = z$.

(b) *If* $|u| = 2^n$ *and* sort$(c)(u) = z \in L^*$, *then there exists a partial function* $c' \leq c$ *which is defined on at most* $n \cdot 2^n$ *pairs, and such that* sort$(c')(u) = z$.

There is no mention of "algorithms" or "uses of comparisons" in Theorem 5.2, but it is not hard to find in it the heart of the claims of Lemmas 1.2 and 1.4. The key move is from the equations (1.1) and (1.2) (which *we know* to hold of the sorting and merging functions), to the "parametrized" equations (5.1), (5.2), whose meaning is unclear, for arbitrary c, but which have least solutions sort(c) and merge(c) by Theorem 5.1, and these solutions depend monotonically on "the parameter" c. Let us now make the natural assumption that any method for extracting a computation procedure (perhaps an iterator) ϕ from the equations (1.1) and (1.2), should also apply to (5.1), (5.2) and yield a generalized computation procedure $\phi(c)$, for each c, which computes sort(c)—simply by replacing each instruction to *check if* $s \leq t$ by *compute* $c(s, t)$. If sort$(u) = z$, so that ϕ applied to u computes z, then sort$(c)(u) = z$, for some small $c \leq \chi_\leq$ by Theorem 5.2, and hence $\phi(c)$ applied to u should also compute z—but it cannot "ask" for comparisons outside the domain of c, because then it would diverge.

This simple method of *varying the parameter* (here the ordering $\leq$ on L) and then applying Theorem 5.1, is a powerful tool for deriving properties of functions which are (least) solutions of recursive equations.

6 Functionals and recursors

What do we learn from the rigorous arguments of the preceding section about choosing a set-theoretic object to model "the mergesort algorithm"? It seems that all we needed was the "semantic content" of equations (1.1) and (1.2), i.e., the pair (f, g) of operations defined by their right-hand-sides,

$$f(u, p, q) = \begin{cases} u & \text{if } |u| \leq 1, \\ q(p(h_1(u)), p(h_2(u))) & \text{otherwise,} \end{cases} \tag{6.1}$$

$$g(v, w, p, q) = \begin{cases} w & \text{if } v = \emptyset, \\ v & \text{else, if } w = \emptyset, \\ \langle v_0 \rangle * q(\text{tail}(v), w) & \text{else, if } v_0 \leq w_0, \\ \langle w_0 \rangle * q(v, \text{tail}(w)) & \text{otherwise.} \end{cases} \tag{6.2}$$

Formally, these are *functionals on* L^*, in a technical sense which is basic and useful enough to deserve special billing.

6.1. Functionals. A **functional on** a collection of sets $\mathcal{M}$ is any monotone, partial function

$$h : X_1 \times \cdots \times X_n \rightharpoonup W,$$

where $W \in \mathcal{M}$ or $W = \{ff, tt\}$; and each X_i is either a set in $\mathcal{M}$, or a partial function space $X_i = (U \rightharpoonup V)$, with $U = U_1 \times \cdots \times U_l$ a product of sets in $\mathcal{M}$ and $V \in \mathcal{M}$ or $V = \{ff, tt\}$. For example, the operation of *m-ary partial function application*

$$\mathrm{ap}_m(x_1, \ldots, x_m, p) = p(x_1, \ldots, x_m) \quad (x_1, \ldots, x_m \in M, p : M^m \rightharpoonup W) \tag{6.3}$$

is a functional on the sets M, W; and the operation

$$\exists_M(p) = \begin{cases} tt & \text{if } (\exists x \in M)[p(x) = tt], \\ ff & \text{if } (\forall x \in M)[p(x) = ff], \end{cases} \tag{6.4}$$

is a functional on M which "embodies" (in Kleene's expression) existential quantification on M. Note also that, by this definition, all partial functions and partial relations on M are functionals.

It was (essentially) *systems of functionals* like (f, g) that I chose initially in [9, 11] to model algorithms, and these are the concrete objects which come up in the most interesting applications. To develop the general theory simply and smoothly, however, it is best to use a class of more abstract objects, which includes suitable representations of these systems.[20]

6.2. Recursors. A **recursor** $\alpha : X \rightsquigarrow W$ on a poset X (perhaps discrete, just a set) to a set W is a triple $(D, \tau, \mathrm{value})$, where:

(1) D is an inductive poset, the *domain* or *solution set* of α;

(2) $\tau : X \times D \to D$ is a monotone mapping, the *transition mapping* of α; and

(3) $\mathrm{value} : X \times D \rightharpoonup W$ is a monotone, partial mapping, the *value mapping* of α.[21]

The partial function $\overline{\alpha} : X \rightharpoonup W$ determined (computed) by α is defined by

$$\overline{\alpha}(x) = \mathrm{value}(x, (\mu d \in D)[d = \tau(x, d)]),$$

where, for each $x \in X$, $(\mu d \in D)[d = \tau(x, d)]$ is the least, fixed point of the recursive equation

$$d = \tau(x, d) \quad (x \in X, d \in D);$$

and it is monotone, by Theorem 5.1. We say that α is a *recursor on* a collection of sets $\mathcal{M}$, if $\overline{\alpha} : X \rightharpoonup W$ is a functional on $\mathcal{M}$ as in 6.1.

[20] The present version yields, in particular, a natural and comprehensible formulation of *recursor isomorphism*, a notion whose original definition (in [11]) is quite opaque.

[21] This means that $d_1 \leq d_2 \Longrightarrow \mathrm{value}(x, d_1) \leq \mathrm{value}(x, d_2)$, or, equivalently,

$$d_1 \leq d_2 \;\&\; \mathrm{value}(x, d_1) \text{ is defined } \Longrightarrow \mathrm{value}(x, d_1) = \mathit{value}(x, d_2).$$

See Footnotes 14 and 18 for the precise conventions about partial functions.

Two recursors $\alpha_1 = (D_1, \tau_1, \text{value}_1), \alpha_2 = (D_2, \tau_2, \text{value}_2) : X \rightsquigarrow W$ (on the same sets) are **isomorphic**, if there exists an order-preserving bijection

$$\pi : D_1 \to D_2$$

which respects the transition and value mappings, i.e., for all $x \in X$ and $d \in D_1$,

$$\begin{aligned} \pi(\tau_1(x, d)) &= \tau_2(x, \pi(d)), \\ \text{value}_1(x, d) &= \text{value}_2(x, \pi(d)). \end{aligned}$$

Isomorphic recursors (easily) determine the same partial functions, i.e., $\overline{\alpha}_1 = \overline{\alpha}_2$.

6.3. Proposal II: Algorithms are recursors. *The mathematical structure of every algorithm on a poset X to a set W is modeled faithfully by some recursor $\alpha : X \rightsquigarrow W$*; and *two recursors model the same algorithm if they are isomorphic.*

6.4. The where notation. Defining and manipulating recursors becomes much easier with the following, compact where notation, one of several variants of the notation for recursive definitions used in programming languages: To specify that $\alpha = (D, \tau, \text{value}) : X \rightsquigarrow W$, we write

$$\alpha(x) = \text{value}(x, d) \ \mathsf{where}\ \{d = \tau(x, d)\}, \tag{6.5}$$

suggesting that to compute the value $\overline{\alpha}(x)$ using α, we first take the least solution of the equation within the braces $\{\ \ \}$ and then plug it into the "head" partial mapping in the front. We can have more than one equations within the braces in this notation,

$$\begin{aligned} \alpha(x) = &\quad \text{value}(x, d_1, d_2) \ \mathsf{where}\ \{d_1 = \tau_1(x, d_1, d_2), d_2 = \tau_2(x, d_1, d_2)\} \\ &=_{\text{df}} \text{value}(x, \langle d_1, d_2\rangle) \ \mathsf{where}\ \{\langle d_1, d_2\rangle = \langle \tau_1(x, d_1, d_2), \tau_2(x, d_1, d_2)\rangle\}, \end{aligned}$$

where the angled brackets indicate that the domain of α is the product poset $D_1 \times D_2$; and we can also allow recursive equations involving (partial) functions within the braces,

$$\begin{aligned} \alpha(x) = &\quad \text{value}(x, p) \ \mathsf{where}\ \{p(u) = \tau(x, u, p)\} \\ &=_{\text{df}} \text{value}(x, p) \ \mathsf{where}\ \{p = \lambda(u)\tau(x, u, p)\}, \end{aligned}$$

in which case the domain of α is the partial function poset $(U \rightharpoonup W)$, the range of the variable p.[22]

The judicious application and combination of these conventions facilitates significantly the definition and manipulation of recursors. For example, each monotone partial function $f : X \rightharpoonup W$ (and, in particular, each functional) is naturally represented by the "degenerate" recursor

$$\mathbf{r}_f(x) =_{\text{df}} f(x) \ \mathsf{where}\ \{d = d\},$$

[22] If $t(u)$ is an expression which takes values in $W \cup \{\perp\}$ and in which the variable u occurs, ranging over U, then $\lambda(u)t(u)$ stands for the partial function p, where $p(u) = t(u)$.

with domain $\{\perp\}$ and such that (obviously) $\overline{\mathbf{r}}_f = f$. Less trivially, each iterator $\phi = (\text{input}, S, \sigma, T, \text{output})$ on X to W, is represented by the recursor

$$\mathbf{r}_\phi(x) =_{\text{df}} p(\text{input}(x)) \textsf{ where } \{p(s) = \text{if } s \in T \text{ then output}(s) \text{ else } p(\sigma(s))\} \quad (6.6)$$

with domain the partial function poset $(S \rightharpoonup W)$, which computes the same partial function $\overline{\mathbf{r}}_\phi(x) = \overline{\phi}(x)$ as ϕ and codes ϕ up to iterator isomorphism.[23] Finally, the "systems of functionals" which arise in the study of recursive equations can also be represented by recursors, e.g., we set

$$\text{mergesort}_1(u) = p(u) \textsf{ where } \{p(u) = f(u,p,q), q(v,w) = g(v,w,p,q)\}, \quad (6.7)$$

where f and g are defined by (6.1), (6.2).[24]

It is natural and convenient to "identify" monotone partial functions, iterators and systems of functionals with these recursors which represent them, so that the class of recursors may be said to include these objects.

6.5. Algorithm identity. Suppose A is an (intuitive) algorithm which computes (say) the first one billion prime numbers, and you define A' by saying

> *first add* $2+2$ *and then do* A;

or, you let A'' be

> *do* A *two times* (simultaneously, in parallel) *and give as output just one of the results*:

Are A, A' and A'' *different* algorithms, or are they all *identical*? They are, clearly, very closely related, but most people would call them different—or grant, at least, that any rigorous representation of algorithms would model them by non-isomorphic objects; and, indeed, if α, α' and α'' are their natural recursor representations, then no two of these three recursors are isomorphic.

In fact, recursor isomorphism is a very fine equivalence relationship which is not preserved by many useful algorithm transformations (optimizations, refinements, etc.), and we must often introduce "coarser" equivalences (or reductions) to express important facts of informal algorithm analysis.[25] Rather than a defect, this is a virtue, in most cases, as it forces out a precise version of exactly what it is which is being proved.

[23] An isomorphism between two iterators $\phi_i = (\text{input}_i, S_i, \sigma_i, T_i, \text{output}_i)$ $(i = 1, 2)$ on X to W is a bijection $\rho : S_1 \to S_2$ between the sets of states, such that $\rho[T_1] = T_2$; $\rho(\text{input}_1(x)) = \text{input}_2(x)$; $\rho(\sigma_1(s)) = \sigma_2(\rho(s))$, for every $s \in S_1$; and $\text{output}_1(s) = \text{output}_2(\rho(s))$, for every $s \in T_1$ which is *input-accessible*, i.e., such that for some $x \in X$ and some n, $s = \sigma_1^n(\text{input}_1(x))$. The precise result is that *the recursors* $\mathbf{r}_1$ *and* $\mathbf{r}_2$ *associated with two iterators* ϕ_1 *and* ϕ_2 *are isomorphic if and only if* ϕ_1 *and* ϕ_2 *are isomorphic iterators.*

[24] This is only approximate, see 8.6 below. Note, also, that we might equally well have set

$$\text{mergesort}_2(u) = p(u) \textsf{ where } \{q(v,w) = g(v,w,p,q), p(u) = f(u,p,q)\},$$

but mergesort_1 and mergesort_2 are isomorphic: It is an easy, general fact, that re-ordering the listing of the **parts** within the braces of a where expression produces an isomorphic recursor.

[25] By the simple result quoted in Footnote 24, however, changing the order in which we specify computations which are to be executed in parallel "preserves the algorithm".

6.6. Infinitary recursors; graph connectivity. It is clear that not every recursor models an algorithm,[26] because we have allowed the transition and value mappings to be completely arbitrary, as they are for iterators. We will deal with this question of "effective definability" of algorithms in Section 8. In contrast to iterators, however, a recursor may fail to determine an "effective computation" in a more dramatic way, as in the following example.

Suppose that (G, R) is a *graph*, i.e., a non-empty set of *nodes* G together with a symmetric, binary *edge relation* R on G, and consider the *recursive equivalence*

$$p(x, y) \iff x = y \vee (\exists z)[R(x, z) \,\&\, p(z, y)] \quad (p \subseteq G \times G). \tag{6.8}$$

Quite easily, the least binary relation $\overline{p}$ on G which satisfies (6.8) is

$$\overline{p}(x, y) \iff \text{there is a path which joins } x \text{ with } y, \tag{6.9}$$

and from this it follows that if we set

$$\begin{aligned} \mathbf{conn} \Leftrightarrow (\exists x)q(x) \text{ where } \{&q(x) \Leftrightarrow (\forall y)p(x, y), \\ &p(x, y) \Leftrightarrow x = y \vee (\exists z)[R(x, z) \,\&\, p(z, y)]\}, \end{aligned} \tag{6.10}$$

then $\mathbf{conn} : \mathbf{I} \rightsquigarrow \{ff, tt\}$ is a nullary[27] recursor which "verifies" the connectedness of the graph G, i.e.,

$$\overline{\mathbf{conn}} \Leftrightarrow tt \iff G \text{ is connected},$$

We can also extract from the recursive equivalence (6.8) a computation procedure (of sorts) for verifying whether an arbitrary $x \in G$ can be joined with some arbitrary $y \in G$, much as we did in Section 1: *If $x = y$, give output tt, and if not, check* (simultaneously) *for each immediate neighnor z of x, if it can be joined with y, and give tt only if one does.* So far, so good, but *how long—how many basic, computation steps—does it take to verify that G is connected*, i.e., to carry out all the verifications required to show that *every x* can be joined with *every y* in G? Well, it depends on the so-called *diameter* of G, the supremum of shortest paths connecting its points: If this is finite (and, in particular, if G is finite), then we can clearly do all the verifications in a finite number of steps, but if G is connected with infinite diameter, then it seems that we need to use inifinitely many steps to check that every point in G can be joined with every other one, and so the total "computation" of $\overline{\mathbf{conn}}$ requires at least ω (= the least infinite ordinal) steps.

Whether (in the proper context) we can take **conn** to represent an "algorithm" is an interesting question, to which I will return in 9—but, if it does, then that should be some sort of (absolutely) non-implementable, *infinitary* al-

[26] Well, maybe not so clear, see the remarks following 8.4.

[27] Here $\mathbf{I}$ is some fixed set with a single element (say $\emptyset$), so that a recursor $\alpha : \mathbf{I} \rightsquigarrow \{ff, tt\}$ has no real arguments, and simply computes an object $\overline{\alpha} = \overline{\alpha}(\emptyset) \in \{ff, tt, \perp\}$. I am also using "$\Leftrightarrow$" for the equality relation on $\{\perp, ff, tt\}$ in the definition of **conn**, since $\overline{\mathbf{conn}}$ is a partial relation.

gorithm, since "real," terminating computations cannot take infinitely many steps for their completion.

The "number of steps" required by a recursor α to "compute" a value $\overline{\alpha}(x)$ is an important quantity associated with α, part of a bundle of notions with which the mathematical theory of recursors starts.

6.7. Recursor iteration. Fix a recursor $\alpha = (D, \tau, \text{value}) : X \rightsquigarrow W$ and some $x \in X$, let

$$\tau_x(d) = \tau(x, d),$$

and for each ordinal number ξ, set (by ordinal recursion)

$$\begin{aligned} d^\xi_\alpha(x) &=_{\text{df}} \tau_x(\sup\{d^\eta_\alpha(x) \mid \eta < \xi\}) \quad (\text{with } \sup\emptyset = \bot), \\ \overline{\alpha}^\xi(x) &=_{\text{df}} \text{value}(x, d^\xi_\alpha(x)), \\ ||\alpha|| &=_{\text{df}} \text{the least } \xi \ (\forall x \in X)[d^\xi_\alpha(x) = \sup\{d^\eta_\alpha(x) \mid \eta < \xi\}]. \end{aligned} \tag{6.11}$$

It is not hard to show that these definitions make sense[28] and that they determine the partial function computed by α, i.e.,

$$\overline{\alpha}(x) = \sup_\xi \overline{\alpha}^\xi(x).$$

We call α **finitary** if $||\alpha|| \le \omega$, and **infinitary** if $||\alpha|| > \omega$.

The **closure ordinal** $||\alpha||$ and the (partial) **stage assignment**

$$|\alpha|(x) =_{\text{df}} \mu\xi \ [\overline{\alpha}^\xi(x) \in W] < ||\alpha||, \tag{6.12}$$

(defined exactly when $\overline{\alpha}(x)$ is defined) are fundamental invariants of α: For the recursor $\mathbf{r}_\phi$ associated with an iterator ϕ by (6.6), for example, $||\mathbf{r}_\phi|| = \omega$, and

$$|\mathbf{r}_\phi|(x) = \ell(x) - 1 = (\text{the computation length on } x) - 1.$$

In the case of 6.6,

$$||\mathbf{conn}|| = \text{the diameter of } G + 2,$$

so that if G has infinite diameter, then $||\mathbf{conn}|| = \omega + 2$.

One may choose to view the iteration sequence $\{d^\xi(x) \mid \xi < ||\alpha||\}$ as some sort of very abstract, "logical computation" of $\overline{\alpha}(x)$, whose length (if it terminates) is the possibly infinite ordinal $|\alpha|(x)$. More loosely, but closer to the truth, we may say that each iterate $d^\xi(x)$ codes some "information" about the value $\overline{\alpha}(x)$, which can be extracted by the value mapping and increases with ξ; and when enough such information is available, then $\overline{\alpha}(x) = \overline{\alpha}^\xi(x) = \text{value}(x, d^\xi(x))$ becomes defined.

These iterates are also the key tool for "rigorizing" many informal arguments about algorithms extracted from recursive equations. I will not go into this here, and I will also avoid any further discussion of the mathematical theory of recursors, whose basic facts are presented in [10, 11, 14].

[28] This is an outline of the standard proof of Theorem 5.1.

7 Implementations

Imagine a world (presumably run by mathematicians) where one could patent algorithms, so that each time you used Prof. X's "superfast multiplication" α you should pay him a fee.[29] Now to use α, you must first *implement* it, i.e., write a program P which (in some sense) expresses α, and which can be understood and "run" by some actual machine; and Prof. Y has written just such a program P, but he claims that it has nothing to do with X's α, it is actually an implementation of some other algorithm β, unrelated to α and invented by himself: What are the relevant objective criteria—the mathematical relations which hold or do not hold between α and P or β and P—for settling the dispute? The humor is dubious, but the problem of making precise exactly what it means to say that *the program P implements the algorithm α* is very serious, one of the basic (I think) foundational problems in the theory of algorithms.

Having already resolved in 4.4 that implementations are iterators, and that each iterator ϕ can be identified with a recursor $\mathbf{r}_\phi$, (6.6), I will propose an answer which follows from a general theory of *reduction* among recursors: First I will define a relation $\alpha \leq_r \beta$ between recursors, which (roughly) means that "the abstract computations of α are faithfully simulated by those of β", and then I will say that ϕ *implements* α if $\alpha \leq_r \mathbf{r}_\phi$.

7.1. Recursor reduction. A recursor $\alpha_1 = (D_1, \tau_1, \mathrm{value}_1) : X_1 \rightsquigarrow W$ is **reducible** to another $\alpha_2 = (D_2, \tau_2, \mathrm{value}_2) : X_2 \rightsquigarrow W$ (on the same set of values), and we write $\alpha_1 \leq_r \alpha_2$, if there exist monotone mappings

$$\rho : X_1 \to X_2, \quad \pi : X_1 \times D_1 \to D_2,$$

so that:

(1) For all $x \in X_1$ and $d \in D_1$, $\tau_2(\rho(x), \pi(x,d)) \leq \pi(x, \tau_1(x,d))$;
(2) for all $x \in X_1$ and $d \in D_1$, $\mathrm{value}_2(\rho(x), \pi(x,d)) \leq \mathrm{value}_1(x,d)$; and
(3) for each $x \in X_1$, $\overline{\alpha}_1(x) = \overline{\alpha}_2(\rho(x))$.

It is easy to show that the reduction relation is reflexive and transitive on the class of all recursors.

7.2. Proposal III: "To implement" means "to reduce". An **implementation of** a recursor α is any iterator ϕ such that $\alpha \leq_r \mathbf{r}_\phi$; and α is (abstractly) **implementable**, if it admits an implementation.

In the concrete examples of this very abstract notion, the universe X_2 of α_2 is an expansion of the universe X_1 of α_1 by new "data structures," e.g., stacks, caches, etc. To understand how the abstract computations of the two recursors are related, imagine (as at the end of Section 6) that each $d \in D_1$ represents some information about the value $\overline{\alpha}_1(x)$, which by (3) is the same as $\overline{\alpha}_2(\rho(x))$; for each x, now, $\pi(x,d)$ gives us a corresponding piece of information about $\overline{\alpha}_2(\rho(x))$, and

[29] In our world, the law is vague and still not fully formed, but (as I understand it) it denies patents to algorithms but grants Copyrights to programs.

(1) and (2) prescribe that each step of α_2 "increments no more" the available information and "contributes no more" to the computation of the common value $\overline{\alpha}_1(x) = \overline{\alpha}_2(\rho(x))$ than the corresponding step of α_1.[30] Technically, (1) and (2) yield that for all ordinals ξ,

$$\pi(x, d_1^\xi(x)) \geq d_2^\xi(\rho(x)),$$

from which it follows that

$$\overline{\alpha}_1^\xi(x) \geq \overline{\alpha}_2^\xi(\rho(x)), \tag{7.1}$$

and (3), then, says simply that the iteration of α_2 eventually "catches up" with that of α_1, so that, in the limit, the same partial function is computed.

From (7.1) we also get

$$|\alpha_1|(x) \leq |\alpha_2|(\rho(x)) \quad (\text{if } \overline{\alpha}_1(x) \text{ is defined}),$$

so that in particular

$$||\alpha_1|| \leq ||\alpha_2||; \tag{7.2}$$

and this with 6.7 implies that *if α is abstractly implementable, then $||\alpha|| \leq \omega$.* It is not hard to verify that the converse also holds,[31] so that:

7.3. Fact. *The abstractly implementable recursors are exactly those with closure ordinal $\leq \omega$.*

To justify this modeling of "ϕ implements α", one must (at least) show that it covers simply and naturally the standard reductions of recursion to iteration, and that it extends the precise definitions which already exist for simulating one iterator by another. This can be done, quite easily, but the inevitable technicalities are not suitable for this paper.

8 Algorithms

It is tempting to assume that the successor operation $S(n) = n + 1$ on the natural numbers is "immediately computable," an absolute "given," presumably because of the trivial nature of the algorithm for constructing the unary (tally) representation of $S(n)$ from that of n—just *add one tally*; if we use binary notation, however, then the computation of $S(n)$ is not so trivial, and may require the examination of all $\log_2(n)$ binary digits of n for its construction—while multiplication by 2 becomes trivial now—*just add one* 0. The point was already made in Section 4, to argue that the mode of input must be part of any model of computation, but it also shows that while there is one, absolute notion

[30] It is not always true for a monotone τ that $d \leq \tau_x(d)$, but τ_x is only applied to such d's in the construction (6.11) of the iteration sequence $\{d^\xi(x) \mid \xi < ||\alpha||\}$, so that, where it is relevant, applying the transition mapping, indeed, does not decrease our information about the value.

[31] You cook up an iterator whose computation for each x is the sequence of iterates $d^0(x), d^1(x), \ldots$, and then check that α is reducible to it. It is, in general, an inefficient implementation, but it is used routinely in finite model theory, and (I have been told) it is also used for some very special database applications.

of *computability* on $\mathbb{N}$ (by the Church-Turing Thesis), there is no corresponding absolute notion of "algorithm" on the natural numbers—much less on arbitrary sets. Algorithms make sense only *relative* to operations which we wish to admit as *immediately given* on the relevant sets of *data*. Any set can be a data set; as for the given operations, we may have partial functions, functionals, or, in the most general case, recursors.

8.1. Structures. A (recursor) **structure** is a pair $\mathfrak{M} = (\mathcal{M}, \mathcal{F})$ such that:

(1) Each $M \in \mathcal{M}$ is a set and at least one such M is non-empty; these are the *basic* or *data sets* of $\mathfrak{M}$.
(2) Each $\alpha \in \mathcal{F}$ is a *recursor on* the data sets of $\mathfrak{M}$, i.e., $\overline{\alpha} : X \rightharpoonup W$ is a functional on $\mathcal{M}$ as in 6.1.

$\mathfrak{M}$ is a *first order structure* if every given is a (total) function, and a *functional structure* if every given is a functional.

Simplest among these are the usual (single- or many-sorted) first-order structures of model theory, e.g.,

$$\mathfrak{N} = (\mathbb{N}, 0, S, P, \chi_0), \tag{8.1}$$

where 0 is the (nullary) constant, $S(n) = n + 1$, $P(n) = n - 1$ (with $P(0) = 0$), and $\chi_0 : \mathbb{N} \to \{ff, tt\}$ is the characteristic function of $\{0\}$; the choice of givens in this simplest *structure of arithmetic* implies (in effect) that we take the numbers to be finite sequences of tallies. The expansion

$$(\mathfrak{N}, \exists_{\mathbb{N}}) = (\mathbb{N}, 0, S, P, \chi_0, \exists_{\mathbb{N}}), \tag{8.2}$$

of $\mathfrak{N}$ by the existential quantifier (6.4) is an important example of a functional structure, and every first-order structure $\mathfrak{M}$ has an analogous expansion $(\mathfrak{M}, \exists_M)$. In the most general case, the "given" recursors of a structure represent algorithms which we accept as "black boxes," right from the start, and they are the basic ingredients with which we build the algorithms of a structure.

8.2. Formal definability. With each structure $\mathfrak{M} = (\mathcal{M}, \mathcal{F})$, we can associate a *vocabulary* (signature), with variables over the data sets of $\mathfrak{M}$ and the partial functions and partial relations on them, and with constant, function symbols for the given recursors; and using such a vocabulary, we can then build formal languages, which codify various notions of definability on $\mathfrak{M}$. Simplest among these is the *Formal Language of Recursion* FLR, a language of terms and λ-terms, built up from the vocabulary using (partial) *function application* (6.3), *definition by cases* (conditional), *calls* to the givens and *recursion*, in the general form of 6.4. We will also need the fragment FLI of FLR, obtained by replacing the general where construct by its special case used in iteration. Table 1 gives a summary definition of the syntax of these languages, computer science style.[32]

[32]The only non-obvious side condition in the syntax is that in the recursion construct for FLI, the "recursion variable" p occurs only where it is explicitly shown.

Term:	$A \;:\equiv\; \mathit{ff} \mid \mathit{tt}$	
		$\mid p(Z_1, \ldots, Z_n)$
		$\mid \mathsf{f}(Z_1, \ldots, Z_n, \pi_1, \ldots, \pi_m)$
		$\mid \mathsf{if}\ A_0\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2\ \mathsf{fi}$
(for FLR)		$\mid A_0\ \mathsf{where}\ \{p_1(\vec{u}_1) = A_1, \ldots, p_n(\vec{u}_n) = A_n\}$
(for FLI)		$\mid p(\vec{I})\ \mathsf{where}\ \{p(\vec{s}) = \mathsf{if}\ T\ \mathsf{then}\ O\ \mathsf{else}\ p(\vec{Z})\ \mathsf{fi}\}$
	$Z \;:\equiv\; u \mid A$	
λ-term:	$\pi \;:\equiv\; p \mid \lambda(u_1, \ldots, u_n)A$	

TABLE 1. The syntax of FLR and FLI.

8.3. Algorithmic semantics. The **algorithmic** or **intensional semantics** of FLR which concern us here are defined in [11], using the main result of [10]: Roughly, a recursor

$$\mathbf{int}_A = \mathbf{int}_A^{\mathfrak{M}} \tag{8.3}$$

(the **intension of** A in $\mathfrak{M}$) is naturally associated with each structure $\mathfrak{M}$ and term A, in such a way that the domain of $\mathbf{int}_A$ is a product of the domains of the givens of $\mathfrak{M}$ and its data sets, and its transition and value mappings are defined *explicitly and immediately* using application, conditionals and calls.[33]

8.4. Proposal IV: Algorithms are definable recursors. *Every algorithm relative to given recursors $\mathcal{F}$ is* (modeled faithfully by) *the intension* $\mathbf{int}_A$ *of some* FLR-*term A on the structure $\mathfrak{M}$ with the givens $\mathcal{F}$; and, conversely, each* $\mathbf{int}_A$ *on $\mathfrak{M}$ is an algorithm relative to $\mathcal{F}$.*

The iterative algorithms of a first order structure $\mathfrak{M}$ are the $\mathfrak{M}$-intensions of FLI-*terms.*[34]

A functional on the data sets is *recursive* or *computable* relative to $\mathcal{F}$ if it is computed by an algorithm of $\mathfrak{M}$, and it is *iteratively computable* if it is computed by an iterative algorithm of $\mathfrak{M}$. These are exactly the functionals definable by FLR- and FLI-terms on $\mathfrak{M}$, in the natural, denotational semantics of FLR.

Notice that, in a trivial sense, every recursor $\alpha : X \rightsquigarrow W$ models an algorithm of some structure, e.g., the structure (X, W, α)! On the other hand, no function or relation on an arbitrary set M is automatically computable by some algorithm, not the equality relation $x = y$ on M, not even the identity function $f(x) = x$.

[33] The language FLR "evolved" somewhat between [10] and [16], and the intensional semantics are constructed in [11] for a more restricted class of recursors, but none of this is very important or affects the present discussion. The mapping $A \mapsto \mathbf{int}_A$ is defined (basically) by recursion on the structure of A, as one might expect. It is not a difficult construction, but it does involve some subtleties and technicalities (mostly in making precise this "explicitly and immediately") which make it impractical to give a useful summary of it here. In addition to the papers already cited, [14] discusses some applications of intensional semantics to the philosophy of language and establishes the *decidability of algorithm identity on any fixed structure with finitely many givens.*

[34] The notion of an *iterative algorithm* does not have a clear meaning, except on first order structures.

It is usual, of course, to include such simple functions among the givens of a structure, but it is not necessary—and there are examples where it is not natural.

8.5. Implementations of algorithms. Let us say that a structure $\mathfrak{M}$ is **implementable in** a first order structure $\mathfrak{M}'$ and write $\mathfrak{M} \leq_i \mathfrak{M}'$, if every $\mathfrak{M}$-algorithm is reducible to some iterative algorithm of $\mathfrak{M}'$; in standard, computer science terminology, $\mathfrak{M} \leq_i \mathfrak{M}'$ is expressed by saying that *every recursive program in* $\mathfrak{M}$ *can be simulated by a while-program in* $\mathfrak{M}'$. For the integers, $\mathfrak{N} \leq_i \mathfrak{N}$, but it is not generally true of first order structures that $\mathfrak{M} \leq_i \mathfrak{M}$, and, in fact, there are natural (infinite) examples in which not every $\mathfrak{M}$-recursive function can be computed by a while-program of $\mathfrak{M}$.[35] The standard reductions of recursion to iteration establish $\mathfrak{M} \leq_i \mathfrak{M}^*$, where $\mathfrak{M}$ is first order and $\mathfrak{M}^*$ is an expansion of $\mathfrak{M}$ by a stack or other, auxiliary data structures.

A serious attempt to defend and support Proposals I – IV requires a detailed examination of several examples and a comparison of the rigorous theory built upon them with the "naive" theory of algorithms, as it has been developed (especially) by complexity theorists, and this I cannot do here. I will confine myself to a final re-examination of the mergesort, and a brief discussion, in the next Section, of the infinitary algorithms which arise naturally in this theory.

8.6. The mergesort algorithm. The natural structure for the mergesort has data sets L and L^*, and the obvious functions for the manipulation of strings for its givens:

$$\mathfrak{L} = (L, L^*, \emptyset, \mathrm{eq}_\emptyset, \mathrm{head}, \mathrm{tail}, \mathrm{append}, \chi_\leq), \tag{8.4}$$

where $\emptyset$ is the empty string (as a nullary function); $\mathrm{eq}_\emptyset : L^* \to \{\mathit{ff}, \mathit{tt}\}$ checks for equality with $\emptyset$,

$$\mathrm{eq}_\emptyset(u) = \begin{cases} \mathit{tt}, & \text{if } u = \emptyset, \\ \mathit{ff}, & \text{otherwise;} \end{cases}$$

head(u) and tail(u) are as in Section 1 (with $\mathrm{head}(\emptyset) = \mathrm{tail}(\emptyset) = \emptyset$ to make them total functions); append : $L \times L^* \to L^*$ prefixes a sequence with an arbitrary element of L,

$$\mathrm{append}(x, \langle a_1, \ldots, a_{n-1}\rangle) = \langle x, a_1, \ldots, a_{n-1}\rangle,$$

so that, in particular, $\mathrm{append}(x, \emptyset) = \langle x\rangle$; and $\chi_\leq$ is the characteristic function of the given ordering on L. The basic equations (1.2) and (1.1) refer directly to several, additional operations on strings, but these can be defined (computed) from those of $\mathfrak{L}$, e.g.,

$$\mathrm{test}_1(u) = \text{if } \mathrm{eq}_0(\mathrm{tail}(u)) \text{ then } \mathit{tt} \text{ else } \mathit{ff}$$

tests if $|u| \leq 1$ or not. This is an explicit definition, while $h_1(u)$ and $h_2(u)$ are (easily) defined by recursion, but FLR has a recursive construct and it is quite

[35] There is a large literature on the reduction of recursion to iteration under various conditions, see for example [19] and the papers cited there.

trivial to write in the end a single FLR term A for the mergesort, with one free variable, a formal version of (6.7) with imbedded, recursive terms for the parts $f(u,p,q)$ and $g(v,w,p,q)$. Now the official "model" for the mergesort is the recursor

$$\mathbf{msort} = \mathbf{int}_A : L^* \rightsquigarrow L^*$$

which is assigned to this term by the algorithmic semantics of FLR, and it codes not only how the mergesort depends on the ordering, but the whole "flow of computation" determined by the equations (1.1), (1.2). By the general theory,

$$\mathbf{msort}(u) = f_0(u,\vec{p}) \text{ where } \{p_1(\vec{z}_1) = f_1(\vec{z}_1,\vec{p}), \ldots, p_n(\vec{z}_n) = f_n(\vec{z}_n,\vec{p})\}, \quad (8.5)$$

where each of the functionals f_i is defined *explicitly and immediately* from the givens of $\mathfrak{L}$. I have not repeated this full definition here, but in this case[36] it means that each f_i is an application with nesting no more than one-deep, for example

$$f_i(z_1,z_2) = p_j(p_k(z_1), z_2, z_1);$$

or an immediate conditional, for example

$$f_i(z_1,z_2) = \text{if } p_i(z_1) \text{ then } p_j(z_1,z_1,z_2) \text{ else } p_k(z_2);$$

or a truth value $f_i(z_1,z_2) = \mathit{ff}$; or, finally, a direct call to the givens, for example

$$f_i(z_1,z_2) = \text{append}(z_1,z_2).$$

Because the critical given $\chi_\leq$ occurs only once in the equations (1.1), (1.2) (when we write them carefully, using the conditional), only one of the f_i's depends on it; and from this it follows that

$$\mathbf{msort}(u) = \alpha(u,c) \text{ where } \{c(s,t) = \chi_\leq(s,t)\}, \quad (8.6)$$

where, c varies over the set of partial relations $(L \times L \rightharpoonup \{\mathit{ff}, \mathit{tt}\})$; $\alpha(u,c)$ is an algorithm of the reduct of $\mathfrak{L}$ which does not include the ordering $\chi_\leq$; and (most significantly) *the* where *notation has been extended to make sense when it is applied to arbitrary recursors*, not just functionals.[37] Notice that (8.6) is a *recursor identity*; and once we have it, it is natural to define the *dependence* of $\mathbf{msort}(u)$ on $\chi_\leq$ in terms of the dependence of $\alpha(u,c)$ on the partial relation c, from which point on the proof of the basic property of the mergesort follows by the arguments of Section 1, as we made them precise in Section 5.

An important aspect of this "finished" analysis of the mergesort is that the application of the method of *parameter variation*, which (maybe) seemed a bit ad hoc in Section 5, arises now naturally from the move from (8.5) to (8.6), one of the basic, general transformations of the theory.

[36] In fact, the forms listed describe fully the *normal form* for intensions in first order structures.

[37] This follows from the basic, general facts of the theory of recursors, a natural (and not very difficult) extension of the fixed-point theory of monotone mappings which keeps track (in effect) of the recursive definitions, not just their fixed points.

9 Infinitary algorithms

It is clear, from the discussion so far, that, in this approach, there is no absolute notion of *effectively implementable algorithm*, just as there is no absolute notion of *algorithm*, independent of any givens: We can only talk of *implementing an $\mathfrak{M}$-algorithm* $\mathbf{int}_A$ *in* $\mathfrak{M}'$, meaning that we can find an implementation of $\mathbf{int}_A$ among the iterative algorithms of $\mathfrak{M}'$. At the same time, the theory makes room for the infinitary algorithms of 6.6, those with closure ordinal greater than ω, which cannot be implemented in any structure whatsoever: What are we to make of them, and do they serve any useful purpose, do they help illuminate our intuitive notion of *algorithm*? I will consider here, briefly, two ways in which infinitary algorithms arise naturally, as generalizations of concretely implementable algorithms and as interesting mathematical objects of study, in their own right.

9.1. Algorithms on finite structures. If we read (6.10) as the definition of an FLR-term C on the expansion $(G, R, =, \exists_G)$ of an arbitrary graph (G, R) by $=$ and the existential quantifier, it yields a nullary algorithm of this structure

$$\mathbf{conn}_d = \mathbf{int}_C : \mathbf{I} \rightsquigarrow \{ff, tt\} \tag{9.1}$$

which like **conn** of (6.10), computes the connectivity of G,

$$\overline{\mathbf{conn}_d} \Leftrightarrow tt \iff G \text{ is connected.}$$

Now $\mathbf{conn}_d$ is somewhat more "detailed" than **conn** (because it also accounts for the explicit steps in the computation), but still, there is some number m such that

$$||\mathbf{conn}_d|| = \text{diameter of } G + m; \tag{9.2}$$

and so, again, $\mathbf{conn}_d$ is implementable if and only if G has finite diameter, by 7.3. For finite G, we can easily build real, practical implementations of $\mathbf{conn}_d$ which can be programmed on a physical processor—even the trivial implementation suggested in Footnote 31 is not too bad in this case. These implementations are useful in database theory, but the fact of G's finiteness is hardly used in their construction—typically we only appeal to it at the very last moment, to build up an implementation of the quantifiers. So, would it help to build a conceptual wall between the implementable and the infinitary definable recursors, fix the terminology so that the latter would be denied the honorable title of *algorithm*? I would argue that the crucial fact about $\mathbf{conn}_d$ is (9.2) and its Corollary (by (7.2)) that *every implementation of* $\mathbf{conn}_d$ *has closure ordinal* $\geq$ *diameter of* $G + m$, and this has nothing to do with the cardinality of G.

The same considerations apply to much of the work in *structural complexity*, a flourishing area of research in theoretical computer science. It is traditional in this field to study only finite structures, but its basic questions are about algorithms; they often make perfectly good sense on infinite structures as well; and, it seems to me, the field might gain much in clarity (and perhaps even some interesting mathematical results) if people seek answers, at least initially, on arbitrary structures, and put off imposing finiteness conditions until they need them—typically not until the very end of the argument.

9.2. The Gentzen algorithm. In one of the most celebrated and seminal results of modern logic, Gentzen [3] showed that every proof[38] of predicate logic can be transformed into a *cut free* proof of the same conclusion, in a canonical proof system based on a few, intuitive *natural deduction rules*. The Gentzen *cut elimination operation* is defined recursively, by an equation not unlike that of the mergesort:

$$\begin{aligned}\gamma(d) = \ & \text{if } T_1(d) \text{ then } f_1(d) \\ & \text{else if } T_2(d) \text{ then } f_2(\gamma(\tau(d))) \\ & \text{else } f_3(\gamma(\sigma_1(d)), \gamma(\sigma_2(d))),\end{aligned}$$

where d varies over the set Π of (formal) proofs. The conditions T_1, T_2 and the transformations $f_1 - f_3$, τ, and σ_1, σ_2 are complex, but (at least, in principle) they can be defined explicitly in a natural, first order structure $\mathfrak{G}$ with data sets for formulas, variables, proofs, etc., and the usual syntactic operations on these objects as givens, so that the construction yields a $\mathfrak{G}$-algorithm $\boldsymbol{g} : \Pi \rightsquigarrow \Pi$ with $\overline{\boldsymbol{g}} = \gamma$. An implementation of $\boldsymbol{g}$ (or the similar algorithm invented by Herbrand) is one of the basic routines of every theorem prover.

For classical proof theory, the most important fact about the Gentzen algorithm is that it yields a (cut-free) value $\gamma(d)$ of the conclusion of every proof d, which, together with the special properties of cut-free proofs has numerous metamathematical consequences. Not far behind it is the upper bound of the necessary blowup in the *size* of proofs:

$$|\gamma(d)| \leq e(|d|, |d|), \tag{9.3}$$

where the size $|d|$ of a proof is (say) the number of symbols in it and $e(n, k)$ is the iterated exponential,[39] defined by the recursion

$$e(0, k) = k, \quad e(n+1, k) = 2^{e(n,k)}.$$

Some time after [3], Gentzen [4] extended this theorem to Peano arithmetic, where, however, matters are considerably more complex, because the Gödel Incompleteness Theorem rules out the possibility of a finitary cut elimination result. In a modern version of this construction, we introduce an infinitary version of the Gentzen proof system for arithmetic, whose set Π^* of "formal" proofs includes now some infinite objects and admits an infinitary operation, the so-called *ω-rule*: (roughly) *from the infinitely many premises* $\phi(\mathrm{n})$, one for each numeral n

[38] The conclusion of the given proof cannot have free and bound occurrences of the same variable, but such details are of no consequence for the point I want to make and I will steer clear of precise definitions and sharp, optimal statements of facts. A good reference for this discussion is Schwichtenberg's [18], which explains clearly all the results I will allude to—and much more.

[39] The precise result is much better than this, see [18].

naming a number n, *infer* $(\forall x)A(x)$. The extended Gentzen operation is defined again on Π^* very much like γ, by a recursive equation of the form

$$\begin{aligned}\gamma^*(d) = {} & \text{if } T_1(d) \text{ then } f_1(d)\\ & \text{else if } T_2(d) \text{ then } f_2(\gamma^*(\tau(d)))\\ & \text{else if } T_3(d) \text{ then } f_3(\gamma^*(\sigma_1(d)), \gamma^*(\sigma_2(d)))\\ & \text{else } f_4(\lambda(n)\gamma^*(\rho(n,d))),\end{aligned}$$

where f_4 is a functional embodying the ω-rule; and this equation, as before, defines a $\mathfrak{G}^*$-algorithm $\boldsymbol{g}^*$, where $\mathfrak{G}^*$ is very much like $\mathfrak{G}$, except that it has a functional embodying the ω-rule. Proofs now have infinite, ordinal length but—and this is the important fact—*the upper bound estimate* (9.3) *persists*, with the extended, iterated exponential on the ordinals; and so Gentzen shows that *every theorem of Peano arithmetic admits a cut free, infinitary proof of size no more than*

$$\varepsilon_0 = \text{the least ordinal closed under } + \text{ and } \alpha \mapsto \omega^\alpha.$$

There is a large number of metamathematical consequences and by-products of the proof of this fundamental theorem, which rivals the basic Cut Elimination Theorem for its importance.

Now, much of this can be done without ever mentioning the word "algorithm", by dealing directly with the defining, recursive equations, much as we did in Section 5. But it is not a natural thing to do, and the literature on Gentzen's theorems is full of references to "computational procedures," "constructions." and, in fact, "algorithms." It seems to me that the finitary, implementable, $\boldsymbol{g}$ and its infinitary extension $\boldsymbol{g}^*$ share so many common properties, that it is natural and profitable to study the two of them together, within one, unified theory which takes *recursor structure* and *effective definability* rather than *implementability* as the key, characteristic features of "algorithms".

Bibliography

1. Paul Benacerraf. What numbers cannot be. In Paul Benacerraf and Hilary Putnam, editors, *Philosophy of Mathematics, Second Edition*, pages 272–294. Cambridge University Press, 1983. Originally published in *Philosophical Review*, 74 (1965).
2. Gottlob Frege. On sense and denotation. In Peter Geach and Max Black, editors, *Translations from the Philosophical Writings of Gottlob Frege.* Basil Blackwell, Oxford, 1952. Originally published in 1892.
3. Gerhard Gentzen. Untersuchungen über das logisches Schliessen. *Mathematische Zeitschrift*, pages 176–210, 405–431, 1934–1935.
4. Gerhard Gentzen. Beweisbarkeit und Undbeweisbarkeit von anfangsfällen des transfiniten Induktion in der reinen Zahlentheorie. *Mathematische Annallen*, 119:140–161, 1943.
5. A. J. C. Hurkens, Monica McArthur, Yiannis N. Moschovakis, Lawrence

Moss, and Glen T. Whitney. The logic of recursive equations. To appear in the *Journal of Symbolic Logic.*

6. Stephen Cole Kleene. *Introduction to metamathematics.* van Nostrand, Princeton, 1952.
7. D. E. Knuth. *The Art of Computer Programming. Fundamental Algorithms*, volume 1. Addison-Wesley, second edition, 1973.
8. Wolfgang Maass. Combinatorial lower bound arguments for deterministic and nondeterministic turing machines. *Transactions of the American Mathematical Society*, 292:675–693, 1985.
9. Yiannis N. Moschovakis. Abstract recursion as a foundation of the theory of algorithms. In M. M. Richter et al., editors, *Computation and Proof Theory*, volume 1104, pages 289–364. Springer-Verlag, Berlin, 1984. Lecture Notes in Mathematics.
10. Yiannis N. Moschovakis. The formal language of recursion. *The Journal of Symbolic Logic*, 54:1216–1252, 1989.
11. 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.
12. Yiannis N. Moschovakis. A model of concurrency with fair merge and full recursion. *Information and Computation*, 93:114–171, 1991.
13. Yiannis N. Moschovakis. *Notes on set theory.* Undergraduate Texts in Mathematics. Springer-Verlag, New York, Berlin, 1994.
14. Yiannis N. Moschovakis. Sense and denotation as algorithm and value. In J. Väänänen and J. Oikkonen, editors, *Logic Colloquium '90*, volume 2, pages 210–249. Springer-Verlag, Berlin, 1994. Lecture Notes in Logic.
15. Yiannis N. Moschovakis. Computable concurrent processes. *Theoretical Computer Science*, 139:243–273, 1995.
16. Yiannis N. Moschovakis. The logic of functional recursion. To appear in the Proceedings of the International Congress for Logic, Methodoloy and the Philosophy of Science held in Florence, 1995.
17. Yiannis N. Moschovakis and Glen T. Whitney. Powerdomains, powerstructures and fairness. In L. Pacholski and J. Tiuryn, editors, *Computer Science Logic*, number 933 in Lecture Notes in Computer Science, pages 382–396. Springer-Verlag, Berlin, 1995.
18. Helmut Schwichtenberg. Proof theory: Some applications of cut-elimination. In Jon Barwise, editor, *Handbook of Mathematical Logic*, pages 867–895. North -Holland, Amsterdam, New York, London, 1977.
19. Jerzy Tiuryn. A simplified proof of ddl $<$ dl. *Information and Computation*, 81:1–12, 1989.
20. Mathison Turing, Alan. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*,

42:230–265, 1936–37.

21. Ernst Zermelo. Über Grenzzahlen und Mengenbereiche: Neue Untersuchungen über die Grundlagen der Mengenlehre. *Fundamenta Mathematicae*, 16:29–47, 1930.
22. Ernst Zermelo. Investigations in the foundations of set theory. In *From Frege to Gödel*. Harvard University Press, Cambridge, Massachusetts, 1967. Originally published in 1908.
23. Ernst Zermelo. Proof that every set can be well-ordered. In *From Frege to Gödel*, pages 139–141. Harvard University Press, Cambridge, Massachusetts, 1967. Originally published in 1904.

email: `ynm@math.ucla.edu`