Chapter 0

On the Choice of Correct Notions for the General Theory

This is a book on general recursion theory. The approach is axiomatic, and the aim is to present *a* coherent framework for the manifold developments in ordinary and generalized recursion theory.

The starting point is an analysis of the relation

$$\{a\}(\sigma) \simeq z,$$

which is intended to assert that the "computing device" named or coded by a and acting on the input sequence $\sigma = (x_1, \ldots, x_n)$ gives z as output.

The history of this notion goes back to the very foundation of the theory of general recursion in the mid 1930's. It can be traced from the theory of Turing on idealized machine computability via Kleene's indexing and normal form theorems to present-day generalizations.

It was Kleene who in 1959 took this relation as basic in developing his theory of recursion in higher types [83]; subsequently it was adopted by Moschovakis [112] in his study of prime and search computability over more general domains.

Indexing was also behind various other abstract approaches. We mention the axiomatics of Strong [166], Wagner [169], and H. Friedman [33], and the computation theories of Y. Moschovakis [113]. The latter theory, in particular, has been very influential on our own thinking.

Historical development is one thing, conceptual analysis another: it is the purpose of this introduction to present some theoretical grounds for basing our approach to general recursion theory on the axiomatic notion of a "computation theory". The discussion here is not part of the systematic development of the theory of computation theories and so proofs will not be given, but we urge the reader to go to the various sources cited.

0.1 Finite Algorithmic Procedures

We shall start out by analyzing how to compute in the context of an arbitrary algebraic system

$$\mathfrak{A} = \langle A, \sigma_1, \ldots, \sigma_l, S_1, \ldots, S_k \rangle,$$

where the operations σ and the relations S are finitary but not necessarily total. Whenever necessary we assume that equality on A is among the basic relations S.

The notion of a *finite algorithmic procedure* (fap) is one of a number of abstract algorithms introduced by H. Friedman [34] and further studied by J. C. Shepherdson [148]. A comparative study of fap's with inductive definability and computation theories is carried through in Moldestad, Stoltenberg-Hansen and Tucker [108, 109].

To be precise a fap P is an ordered list of instructions $I_1, \ldots, I_k$ which are of two kinds:

Operational Instructions

(i) $\quad r_\mu \coloneqq \sigma(r_{\lambda_1}, \ldots, r_{\lambda_m}),$

i.e. apply the m-ary operation σ to the contents of registers $r_{\lambda_1}, \ldots, r_{\lambda_m}$ and replace the contents of register r_μ by this value.

(ii) $\quad r_\mu \coloneqq r_\lambda,$

i.e. replace the content of register r_μ with that of r_λ.

(iii) $\quad H,$

i.e. stop.

Conditional Instructions

(iv) $\quad$ if $S(r_{\lambda_1}, \ldots, r_{\lambda_m})$ then i else j,

i.e. if the m-ary relation S is true of the contents of $r_{\lambda_1}, \ldots, r_{\lambda_m}$ then the next instruction is I_i, otherwise it is I_j.

(v) $\quad$ if $r_\mu = r_\lambda$ then i else j,

i.e. if registers r_μ and r_λ contain the same element then the next instruction is I_i, otherwise it is I_j.

By convention, a fap P involves a finite list of registers $r_0, r_1, \ldots, r_{n-1}$ where the first few registers $r_1, \ldots, r_m$ are reserved as *input registers* and r_0 as *output register*; the remaining registers $r_{m+1}, \ldots, r_{n-1}$ are called *working registers*.

A partial function $f\colon A^m \to A$ is called fap-*computable* if there exists a fap P over the system $\mathfrak{A}$ such that for all $(a_1, \ldots, a_m)$ if $a_1, \ldots, a_m$ are loaded into registers $r_1, \ldots, r_m$, respectively, and P applied, then $f(a_1, \ldots, a_m) = a$ iff P halts and the content of the output register r_0 is a.

The class of fap-computable functions over $\mathfrak{A}$ is denoted by $\mathrm{FAP}(\mathfrak{A})$.

The general notion of a fap may be too poor to support a reasonable theory of

computing over an algebraic system. There are two natural extensions of the general notion: *a finite algorithmic procedure with stacking* (fap S) first defined in [108], and *a finite algorithmic procedure with counting* (fap C) which first appeared in Friedman's paper.

In a fap S we have two new *operational instructions*

(vi) $s \coloneqq (i; r_0, \ldots, r_{n-1})$,

i.e. place a copy of the contents of the registers $r_0, \ldots, r_{n-1}$ as an n-tuple in the stack register s together with the marker i; i is a natural number.

(vii) restore $(r_0, r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_{n-1})$,

i.e. replace the contents of the registers $r_0, r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_{n-1}$ by those of the last (i.e. *topmost*) n-tuple placed in the stack.

There is one new *conditional instruction*

(viii) if $s = \varnothing$ then i else j,

with the obvious meaning.

In a fap S program stacking is only introduced through a *stacking block* of instructions

$s \coloneqq (i; r_0, \ldots, r_{n-1})$
I_{i_1}
$\vdots$
I_{i_l}
goto k
$*: r_j \coloneqq r_0$
restore $(r_0, r_1, \ldots, r_{j-1}, r_{j+1}, \ldots, r_{n-1})$.

The meaning of this block is as follows. Let the fap S contain the registers $r_0, \ldots, r_{n-1}, s$. At the start of the block we *store* the information in $r_0, \ldots, r_{n-1}$ on top of the stack s together with its marker i. Then we use a sequence $I_{i_1}, \ldots, I_{i_l}$ to *reload* some but not necessarily all of the registers $r_0, \ldots, r_{n-1}$. This done, we are ready for a subcomputation within the total program. For this we use the *return instruction* goto k (which is an abbreviation of "if $r_\mu = r_\mu$ then k else k"). Note that I_k must be either an ordinary fap instruction outside all the blocks in the program or the first instruction of any stacking block in the program. "$*: r_j \coloneqq r_0$" is called the *exit instruction* of the block. This means that if the subcomputation is successful, i.e. that it stops and leaves r_0 non-empty, then place the content of r_0 into register r_j. We then restore the contents of $r_0, \ldots, r_{j-1}, r_{j+1}, \ldots, r_{n-1}$ and proceed with the main computation using the new content of r_j supplied by the subcomputation.

Just as above we have a notion of fap S-*computable* and a class of functions FAPS($\mathfrak{A}$).

In a fap C we add to the *algebra registers* $r_0, r_1, \ldots$ certain *counting registers* $c_0, c_1, \ldots$ which are to contain natural numbers. There are three new *operational instructions*

(ix) $c_\mu \coloneqq c_\lambda + 1,$

i.e. add one to the contents of c_λ and place that value in c_μ.

(x) $c_\mu \coloneqq c_\lambda \dot{-} 1,$

i.e. if c_λ contains 0 place 0 in c_μ, else subtract one from the contents of c_λ and place that value in c_μ.

(xi) $c_\mu \coloneqq 0,$

i.e. make the contents of c_μ zero.

We may add one new *conditional instruction*

(xii) if $c_\mu = c_\lambda$ then i else j,

again with the obvious meaning.

With a fap C program we can have computable functions of two kinds, viz. f can be a partial function with values in A,

$$f\colon \omega^n \times A^m \to A$$

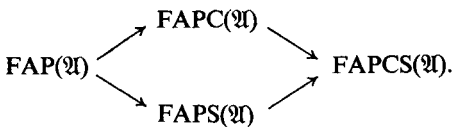
or f can have values in ω,

$$f\colon \omega^n \times A^m \to \omega.$$

In both cases we use $c_1, \ldots, c_n, r_1, \ldots, r_m$ as input registers. In the first case r_0 is the output register, in the second c_0. We thus get a notion of fap C-*computable* and a class of functions FAPC($\mathfrak{A}$).

Stacking and counting can be combined to produce a notion of fap CS-*computable* and a class FAPCS($\mathfrak{A}$) over $\mathfrak{A}$.

These are the basic classes and we have the following immediate diagram of inclusions



It should be clear that when $\mathfrak{A}$ includes enough arithmetic then the classes coincide. In a more general algebraic context there are interesting differences. We shall relate some basic results from [108] and [109], and, in particular, we shall

discuss how inductive definability over $\mathfrak{A}$ and computation theories over $\mathfrak{A}$ fit into the above diagram.

0.2 FAP *and Inductive Definability*

The class $\mathrm{Ind}(\mathfrak{A})$ of inductively defined functions over the structure $\mathfrak{A}$ can be introduced in various but related ways. We follow the syntactic description in [105] which itself is patterned upon Platek's equational calculus [133]. It is convenient in working with partial functions on A to extend A by a new element u for undefined. We introduce a class of terms by the following clauses:

(i) Variables $x_1, x_2, \ldots$ are terms of type 0.
(ii) For each m, the m-ary partial function variables $p_1^m, p_2^m, \ldots$ are terms of type $1 \cdot m$.
(iii) For each m-ary operation σ of $\mathfrak{A}$ the function symbol $\boldsymbol{\sigma}$ is a term of type $1 \cdot m$.
(iv) For each m-ary relation S of $\mathfrak{A}$ the function symbol $\mathbf{DC}_S$ is a term of type $1 \cdot m + 2$.
(v) $\mathbf{u}$ is a term of type 0.
(vi) If t is a term of type $1 \cdot m$ and $t_1, \ldots, t_m$ are terms of type 0, then $t(t_1, \ldots, t_m)$ is a term of type 0.
(vii) If t is a term of type 0 then $\mathbf{FP}[\lambda p_1^m, y_1, \ldots, y_m \cdot t]$ is a term of type $1 \cdot m$.

If S is an m-ary relation in $\mathfrak{A}$ then

$$DC_S(a_1, \ldots, a_m, x, y) = \begin{cases} x & \text{if } S(a_1, \ldots, a_m) \\ y & \text{if } \neg S(a_1, \ldots, a_m). \end{cases}$$

And FP is the fixed-point operator. It should be clear how to interpret terms in the algebra $\mathfrak{A}$. A partial function $f: A^m \to A$ will be *inductively definable* if there is an algebra term (i.e. a term of type 0) with $y_1, \ldots, y_m$ as its only free variables such that for all $a_1, \ldots, a_m \in A$, $f(a_1, \ldots, a_m) = t(a_1, \ldots, a_m)$. $\mathrm{Ind}(\mathfrak{A})$ is the class of functions inductively definable over $\mathfrak{A}$.

As an example let us consider the term

$$\mathbf{FP}[\lambda p', y \mathbf{DC}_S(y, y, \mathbf{DC}_S(\boldsymbol{\sigma}_1(y), y, \boldsymbol{\sigma}_2(p'(\boldsymbol{\sigma}_1(y)), \boldsymbol{\sigma}_1(y))))](x).$$

The reader may want to verify that the following is a fap S program which computes the function defined by the above term. The program has one input register r_1 and two working registers r_2, r_3.

1. if $S(r_1)$ then 2 else 4
2. $r_0 \coloneqq r_1$
3. if $s = \varnothing$ then H else *
4. $r_2 \coloneqq \sigma_1(r_1)$
5. if $S(r_2)$ then 6 else 8
6. $r_0 \coloneqq r_1$
7. if $s = \varnothing$ then H else *
8. $s \coloneqq (1; r_0, r_1, r_2, r_3)$
9. $r_1 \coloneqq \sigma_1(r_1)$
10. goto 1
11. *: $r_2 \coloneqq r_0$
12. restore (r_0, r_1, r_3)
13. $r_3 \coloneqq \sigma_1(r_1)$
14. $r_0 \coloneqq \sigma_2(r_2, r_3)$
15. if $s = \varnothing$ then H else *

We have a single block I_8–I_{12}, and we see how we have to store previous information and use this block in the iterative computation of the value of the fixed-point function on the input value in r_1.

0.2.1 Theorem. $\mathrm{Ind}(\mathfrak{A}) = \mathrm{FAPS}(\mathfrak{A})$.

The example is no proof; see [108] for details which are far from trivial. The example should also suggest that $\mathrm{FAP}(\mathfrak{A})$ is in general a proper subclass of $\mathrm{Ind}(\mathfrak{A})$. In fact, in [108] a subclass of "direct terms" of the class of algebra terms is distinguished such that the corresponding class of *directly inductively definable* functions, $\mathrm{DInd}(\mathfrak{A})$, exactly corresponds to $\mathrm{FAP}(\mathfrak{A})$.

0.3 FAP *and Computation Theories*

Let there be given a notion $\{a\}(\sigma) \simeq z$ over some domain A, and from this let us abstract the set of all computation tuples $\Theta = \{(a, \sigma, z); \{a\}(\sigma) \simeq z\}$. A function f on A is computable under the given notion if there is an index or code a such that for all σ

$$f(\sigma) \simeq z \quad \text{iff} \quad (a, \sigma, z) \in \Theta.$$

The axiomatic approach reverses this procedure. Let there be given a set Θ of tuples (a, σ, z) over A. A function f on A is called Θ-computable if there is a code a such that the equivalence above holds.

Not every set Θ is a reasonable computation theory. We must put in some basic functions and require closure of Θ under some reasonable properties; the details are given in the first few paragraphs of Chapter 1.

As the reader will see from the general development in Chapter 1, in considering computation theories over a structure we are almost forced to also include recursive (sub-) computations on the natural numbers; having given a code set C we can inside this code set reconstruct a copy of the integers and thus have access to the recursive functions over this "successor set" (see Section 1.4).

It is therefore natural given a structure $\mathfrak{A} = \langle A; \boldsymbol{\sigma}, \mathbf{S}\rangle$ to expand it to a structure $\mathfrak{A}_\omega = \langle A \cup \omega; \boldsymbol{\sigma}, \mathbf{S}, s, p, 0\rangle$ where $s, p, 0$ are the successor, predecessor, and constant zero function on ω, respectively. We have the following relationship [109].

0.3.1 Proposition. *Let* $f\colon \omega^n \times A^m \to A$ *or* $f\colon \omega^n \times A^m \to \omega$. *Then*

(i) $f \in \mathrm{FAP}(\mathfrak{A}_\omega)$ iff f *is* fap C-*computable over* $\mathfrak{A}$.
(ii) $f \in \mathrm{FAPS}(\mathfrak{A}_\omega)$ iff f *is* fap CS-*computable over* $\mathfrak{A}$.

The question is now whether counting alone or both counting and stacking are necessary to give a computation theory for an arbitrary $\mathfrak{A}$?

To give the answer we need to introduce the notion of a *term evaluation function*. The class $T[X_1, \ldots, X_n]$ is inductively defined by the clauses:

(i) $X_1, \ldots, X_n$ belong to $T[X_1, \ldots, X_n]$;
(ii) if $t_1, \ldots, t_m$ belong to $T[X_1, \ldots, X_n]$ and σ is an m-ary operation of $\mathfrak{A}$, then $\sigma(t_1, \ldots, t_m) \in T[X_1, \ldots, X_n]$.

Each term $t(X_1, \ldots, X_n)$ defines a function $A^n \to A$ by substitution of algebra elements for indeterminates. Terms in $T[X_1, \ldots, X_n]$ can be numerically coded uniformly in n, in the sense that there is a recursive subset $\Omega \subseteq \omega$ such that for each $i \in \Omega$ there is a unique term $[i]$ in $T[X_1, \ldots, X_n]$, and the correspondence $i \to [i]$ is a surjection. And there are recursive functions which given $i \in \Omega$ allow us to effectively write down the corresponding term $[i]$. Define $E_n\colon \Omega \times A^n \to A$ by $E_n(i, \mathbf{a}) = [i](\mathbf{a})$.

0.3.2 Proposition. $\mathrm{FAP}(\mathfrak{A}_\omega)$ *is a computation theory iff* E_n *is uniformly* fap C-*computable*.

This is proved in [109]. One way is rather straightforward. The difficult part comes in verifying that $\mathrm{FAP}(\mathfrak{A}_\omega)$ has a universal function. The problem is that in the absence of a computable pairing scheme a machine with a fixed number of registers may not be able to simulate a machine with an arbitrarily large number of registers. One way of getting around this problem is by letting the simulating machine manipulate codes for terms instead of actually executing the simulated operations. For codes for terms are natural numbers for which we do have the required pairing function. At some points there is a need to evaluate terms and it is exactly here the computability of the E_n-functions is needed.

But E_n is uniformly fap CS-computable. This is, in fact, by 0.2.1 and 0.3.1 reducible to showing that E_n belongs to $\mathrm{Ind}(\mathfrak{A}_\omega)$. E_n has the following inductive definition

$$E_n(i, \mathbf{a}) = \begin{cases} a_j & \text{if } i \text{ codes } X_j \\ \sigma_j(E_n(i_1, \mathbf{a}), \ldots) & \text{if } [i] = \sigma_j([i_1], \ldots) \\ u & \text{if } i \text{ does not code a term or codes the empty term.} \end{cases}$$

Pulling the results together we arrive at the following characterization.

0.3.3 Theorem. FAPS($\mathfrak{A}_\omega$) *is the minimal computation theory over* $\mathfrak{A}_\omega$ *with code set* ω.

The mathematical part of the theory is clear, we have located the exact position of Ind($\mathfrak{A}$) and the minimal computation theory in the diagram of Section 0.1. We showed in 0.2.1 that Ind($\mathfrak{A}$) = FAPS($\mathfrak{A}$), and it follows from 0.3.3 that FAPSC($\mathfrak{A}$) is the class of functions computable in the minimal computation theory over $\mathfrak{A}$ with code set ω.

Going beyond is a matter of personal taste and preference. Our opinion is that computations in general should be allowed to use both elements of the structure and natural numbers (e.g. the *order* of an element in a group). Term evaluation should also be computable, hence we seem to be led to FAPCS($\mathfrak{A}$), i.e. to a computation theory over $\mathfrak{A}$.

0.3.4 Example. We conclude with an example due to J. V. Tucker (see his [168] for a more general result). He first shows that if $\mathfrak{A}$ is locally finite, then the halting problem for FAPS($\mathfrak{A}$) is fap CS-decidable.

The argument is based upon the simple fact that the number of state descriptions in a fap S computation is effectively bounded by a fap CS-computable function (because the order of an element in $\mathfrak{A}$ is fap CS-computable).

Let $\mathfrak{A}_0$ be the group of all roots of unity. This is surely a locally finite structure. We have the following relationships:

$$\text{Ind}(\mathfrak{A}_0) = \text{FAPS}(\mathfrak{A}_0) \subsetneq \text{FAPC}(\mathfrak{A}_0) = \text{FAPCS}(\mathfrak{A}_0).$$

The last equality is typical in algebraic contexts, term evaluation E_n will be computable. It remains to prove that FAPS($\mathfrak{A}_0$) $\neq$ FAPCS($\mathfrak{A}_0$). This follows from the fact that $\mathfrak{A}_0$ is a computable group in the sense of Rabin-Mal'cev [99], i.e. it has a recursive coordination $\alpha\colon \Omega_\alpha \to \mathfrak{A}_0$, where we can choose α to be a bijection on a recursive subset $\Omega_\alpha \subseteq \omega$, and the "pull-back" of the group operation in $\mathfrak{A}_0$ is recursive on Ω_α.

We now observe that any fap CS-function pulls back to a recursive function. Since there are r.e. sets which are not recursive, it follows from the initial observation that the halting problem for FAPS($\mathfrak{A}_0$) is fap CS-decidable, that FAPS($\mathfrak{A}_0$) $\neq$ FAPCS($\mathfrak{A}_0$).

We shall in the main part of the book be interested in recursion theories over domains which include the natural numbers. From this point of view the preceding

discussion gives "sufficient" theoretical grounds for basing the general theory on the axiomatic notion of a computation theory.

There have, however, been recent discussions of an inductive definability approach to recursion in higher types. It may be useful to compare these approaches to the computation-theoretic point of view.

0.4 Platek's Thesis

The aim of this approach is to study definability/computability over an arbitrary domain Ob using the fixed-point operator. But fixed-points at one level may be obtained from fixed-points of higher levels. This leads to the hierarchy HC of hereditarily consistent functionals over Ob as the natural domain of the general theory.

Remark. Platek's thesis was never published, we follow the discussion in J. Moldestad *Computations in Higher Types* [105].

The hierarchy HC is defined inductively as follows. Any type symbol $\tau \neq 0$ can be written in the form $\tau_1 \to (\tau_2 \to \ldots (\tau_k \to 0) \ldots)$. The level of τ is defined by $l(\tau) = \max\{l(\tau_i) + 1\}$. Monotonicity for partial functions is defined as usual. Then $HC(\tau)$ is defined to be the set of all partial monotone functions defined on a subset of $HC(\tau_1) \times \ldots \times HC(\tau_k)$ and with values in Ob.

The fixed-point operator at type τ is an element $FP \in HC((\tau \to \tau) \to \tau)$, such that when FP is applied to an element $f \in HC(\tau \to \tau)$ it produces the least fixed-point $FP(f)$ of f, which will be an element of $HC(\tau)$.

Platek's index-free approach to recursion theory can now be introduced. Let $\mathscr{B} \subseteq HC$. Then the recursion theory generated by $\mathscr{B}$, which we will denote by $\mathscr{R}_\omega(\mathscr{B})$, is the least set extending $\mathscr{B}$, closed under composition, and containing the function DC (definition by cases), the combinators $I(f) = f$, $K(f, g) = f$, and $S(f, g, h) = f(h)(g(h))$, and the fixed-point operator FP.

To set out the relationship with computation theories we quote two results from Moldestad's study. The first is Platek's reduction theorem.

0.4.1 Reduction Theorem. *Let $\mathscr{B}$ contain some basic functions (a coding scheme, the characteristic function of the natural numbers, the successor and predecessor functions). If $\mathscr{B} \subseteq HC^{l+2}$, then*

$$\mathscr{R}_\omega(\mathscr{B})^{l+3} = \mathscr{R}_{l+1}(\mathscr{B})^{l+3}.$$

If we are interested in objects of level at most $l + 3$, then we need only apply the fixed-point operator up to type $l + 1$.

0.4.2 Equivalence Theorem. *There exists a computation theory Θ (derived from Kleene's schemata* S1–S9 *in* [83]*) such that*

$$\mathscr{R}_\omega(\{h_1, \ldots, h_k\}) = \Theta[h_1, \ldots, h_k]$$

for all finite lists of HC objects $h_1, \ldots, h_k$.

We shall in a moment return to the reduction theorem. But first we spell out the content of the equivalence theorem. It seems that we can draw the following conclusions. Let $\mathscr{B} \subseteq HC$.

1 $\mathscr{R}_\omega(\mathscr{B}) = \bigcup \{\mathscr{R}_\omega(\mathscr{B}_0) : \mathscr{B}_0$ finite subset of $\mathscr{B}\}$.
2 For finite $\mathscr{B}_0$, $\mathscr{R}_\omega(\mathscr{B}_0) = \Theta[\mathscr{B}_0]$.
3 $\mathscr{R}_\omega(\mathscr{B}) \subseteq \Theta[\mathscr{B}]$, but in general $\subsetneq$.

Only the third assertion requires a comment. The notation $\Theta[\mathscr{B}]$ is somewhat ambiguous. We must assume that $\mathscr{B}$ is given as a *list*, i.e. with a specific enumeration. This means that in any precise version of $\Theta[\mathscr{B}]$ we have the enumeration function of the list $\mathscr{B}$. But this enumeration function is not necessarily in $\mathscr{R}_\omega(\mathscr{B})$.

Back to the reduction theorem. This result shows that the framework of computation theories is adequate if the aim is to study computability/definability over some given domain. We need not climb up through the hierarchy *HC*. A computation theory Θ can be considered as a set of functions $\Theta \subseteq HC^1$. Then, by the reduction theorem

$$\mathscr{R}_\omega(\Theta)^1 = \mathscr{R}_1(\Theta)^1 = \Theta,$$

the last equality being true since Θ satisfies the first recursion theorem; see Theorems 1.7.8 and 1.7.9 of Chapter 1.

Remark. Platek obtains Kleene's theory of recursion in higher types as a "pull-back" from his theory on *HC*. We shall return to this matter in Chapter 4.

0.5 Recent Developments in Inductive Definability

The index-free approach of Platek is conceptually of great importance in the development of generalized recursion. The theory has, however, some weak points. Recently improved and largely equivalent versions have been published independently by Y. Moschovakis and S. Feferman.

The relevant papers of Moschovakis are the joint contribution with Kechris, *Recursion in higher types* [77], and the paper *On the basic notions in the theory of induction* [117]. Feferman's version is presented in *Inductive schemata and recursively continuous functionals* [25].

Feferman summarizes his criticism of Platek in the following points.

a The structure of natural numbers is included as part—there could be more general situations, e.g. applications in algebra.
b Inductive definability of relations is not accounted for.