Computability on Continuous Higher Types and its role in the semantics of programming languages

R. L. Constable[†] and H. Egli*

TR 74-209

July 1974

Department of Computer Science
Cornell University
Ithaca, New York 14850

† Part of this work was supported by NSF contract GJ-579

* Present address: Forschungsinstitut fur Mathematik,
ETH - Zurich
CH-8006 Zurich

Page

COMPUTABILITY ON CONTINUOUS HIGHER TYPES

and its role in the semantics of programming languages

R.L. Constable and H. Egli

Department of Computer Science
Cornell University
Ithaca, N.Y.

Abstract:

This paper is about mathematical problems in programming language semantics and their influence on recursive function theory. In the process of constructing computable Scott models of the lambda calculus we examine the concepts of deterministic and non-deterministic effective operators of all finite types and continuous deterministic and non-deterministic partial computable operators on continuous inputs of all finite types. These are new recursion theoretic concepts which are appropriate to semantics and were inspired in part by Scott's work on continuity.

KEY WORDS: effective operators of all finite types, continuous functions, partial computable operators, lambda calculus, Scott models of lambda calculus, recursive functions, non-determinism, parallelism, mathematical semantics of programming languages.

COMPUTABILITY ON CONTINUOUS HIGHER TYPES

and its role in the semantics of programming languages

1. INTRODUCTION

Recent work on the semantics of programming languages has led computer scientists to reformulate and extend certain basic ideas in classical recursive function theory. As a simple example, we no longer exclusively use the convention that a partial recursive function $\phi(\ )$ is undefined if any of its arguments is. This paper is about mathematical problems in the study of programming language semantics and their influence on basic recursive function theory, in particular on the treatment of non-determinism and parallelism and the role of continuity in higher type objects.

Our main work consists first of definitions of continuous deterministic and non-deterministic partial computable operators on arbitrary continuous inputs of all finite types and of deterministic and non-deterministic effective operators of all finite types, and consists second of a theorem which relates these objects.

We were led to these objects by an attempt to construct computable Scott models of the lambda calculus adequate for programming language semantics. Such constructions are straightforward once the objects mentioned are in hand (as we show in §10), but the existing recursive function theory literature does not provide them.

None of the definitions or theorems is very complicated (and that is one of their virtues). In fact, one consequence of this work is a delineation of the simple parts of the somewhat arcane subject of computability theory of all finite types on partial inputs. Another consequence is a set of simple observations about parallelism (and non-determinism), about call by value and about representation of data which illustrate the ways in which a computer oriented account of computability differs from a purely mathematical account.

Most of the ideas of this paper can be demonstrated on the low types: 0,1,2 and 3, which is done in §§2 through 5. Therefore the paper is organized to present most definitions and theorems first for those low types where they are more widely familiar. Then in §6-9the results are given in general. Except for an intuitive description of the results, the presentation of the general definitions and theorems in §§6-10 is self-contained. For some readers it might be helpful to read the general account first and fill in the motivation and special instances later.

Aside from the consequences mentioned above, we feel that the main results of this paper may become interesting for two reasons. One, the computable lambda calculus model may be an interesting vehicle for basic recursive function theory. Two, the objects we define and the model built from them may answer interesting questions about programming language semantics. It has been widely accepted that the semantics of higher type programming language constructs such as arbitrary procedures and jumps can be given in spirit of mathematical semantics only if typeless function spaces are available [22], [24], [4]. If such function spaces are necessary, we would like to understand them computationally. For example we ask, must such powerful function spaces contain non-recursive objects if they contain "all computable finite type objects" (perhaps because of their necessarily high cardinality)? We show that they need not.

Our results also reveal an interesting relationship between the operational semantics of languages with arbitrary procedures (as represented mathematically by the effective operators) and the mathematical semantics of such languages (as represented by the computable operators on continuous inputs).

We might also wonder whether programming language features which are described computably can be represented in the model. Our results show that they can be.

Furthermore we hope that our treatment of semantics in terms of recursive function theory will further clarify the relationship of the two subjects to each other. It is our view that just as the theory of computational complexity leaves a rich residue in pure recursion theory, so semantics too will leave such a residue.

2. TYPE ZERO

2.1 integers and effective integers

We will be considering computable functions of all finite types. The ground type is denoted 0. If τ and σ are types, then

$\tau \to \sigma$ is the type of a function from type τ objects to type σ objects. Thus $0 \to 0$ is the type of a function from individuals to individuals while $(0 \to 0) \to 0$ is the type of a function from functions to individuals. The domain of objects of type τ will be denoted by D^τ. Thus the domain of individuals is D^0.

There are two common ways to treat the type of multi-argument functions (or equivalently, functions from a Cartesian product of domains). We can introduce a new class of types, say $\tau\times\sigma$, to represent the domain $D^\tau \times D^\sigma$. We can also interpret a multi-argument function, $\phi(x,y)$, as two applications of a single argument function; namely $\phi(x)(y)$ where $\phi(x)$ produces a name for a function whose argument is y. In this case a function $\phi: D^0 \times D^0 \to D^0$ would have type $0 \to (0 \to 0)$. This device is due to Schönfinkel and is used extensively in the λ-calculus and combinatory logic.

In this paper we are mainly interested in computing over the integers as the base type. Let $\mathbb{N} := \{0,1,2,\ldots\}$ be the integers in decimal representation.

In computing as opposed to mathematics it is frequently important to consider the representation of objects. For example, the classical mathematician will accept the following representation of an integer:

<u>if</u> Fermat's conjecture is true <u>then</u> 1 <u>else</u> 0 <u>fi</u>.

But this notation suffers defects compared with the decimal representation.

A particularly important set of representations for integers are the arithmetic expressions of programming languages, e.g. from the

the simple expressions such as INTEGER(SIN(x*2)) + x ÷ F(x) to complex forms involving loops. These are integer representations actually manipulated in computing. These particular expressions for integers depend on the programming language under discussion. But we can capture the important mathematical properties of such notations in terms of recursive function theory. Let $\phi_i : \mathbb{N} \to \mathbb{N}$ be an indexing of the partial recursive functions, say as defined in Kleene [7] or Rogers [17]. The arithmetization of algorithms used to define the indexing provides a number for every "arithmetic expression", i.e. the expression $\phi_i(10)$ is denoted by the index of the algorithm for evaluating this expression (formally the index of $\phi_i(10)$ is $S_0^1(i,10)$ where S_n^m is Kleene's s-m-n functio These indices can be called <u>effective</u> <u>representations</u> of integers. Such notations will be important in describing the semantics of programming languages.

For our purposes the most convenient effective representation of integers arises from an indexing of the recursively enumerable sets, $\{W_i\}$. Let

$$W_i := \{y \mid \exists x . \phi_i(x) = y\}^{\dagger}$$

Then if $W_i := \{n\}$ we can think of i and the indexing ϕ as an effective representation of n.

<u>2.2 extended integers</u>

In computing over the domain of integers, $\mathbb{N}$, we are led to objects other than integers. Consider for example a composition of algorithms of type $(0 \to 0)$, say $\phi(\psi(x))$. If algorithm ψ on

† This notation is similar to that in Rogers. He uses $W_i := \{x \mid \exists y\ \phi_i(x) = y\}$. The two definitions are recursively equivalent.

input x is not defined, e.g. the computation runs forever, then the input to ϕ <u>is not an integer</u>. If we want to think of the composition of algorithms as function composition, then we need some non-integer as input value.

Suppose we introduce the undefined value, $\perp$, read "bottom". Then if $\psi(x)$ runs forever we say $\psi(x) = \perp$, and we say that the input to ϕ is the value $\perp$. In classical recursive function theory, the value of any function on $\perp$ is again $\perp$, i.e. $\phi(\perp) = \perp$ for all partial recursive functions. But in computing it is possible to have algorithms such as

ϕ_1	ϕ_2
TYPE(0 → 0) READ x PRINT 3	TYPE(0 → 0) PRINT 3

The first algorithm, ϕ_1, computes a function whose value on $\perp$ is $\perp$. The second algorithm satisfies $\phi_2(x) = 3$ for all x including $x = \perp$.

Thus even for the task of describing the behavior of algorithms over the integers, $\mathbb{N}$, we are led to consider an <u>extended domain</u>, $\mathbb{D} := \{\perp, \mathbb{N}\}$ and extended functions $\bar{\Phi}_i : \mathbb{D} \to \mathbb{D}$. Note, these extended functions are not <u>computed</u> by the algorithms (since they are not computable), only <u>defined by them</u>. We will say much more about them in the section on type 1.

Given the domain $\mathbb{D}$ we are interested in the "extended arithmetic expressions" for the same reason that we considered arithmetic expressions. Again as before we can treat these expressions abstractly using indexings. We let $\mathbb{E}$ denote the indices of <u>extended integers</u>, that is, $\mathbb{E} := \{i \mid W_i = \{x\}\ \&\ x \in \mathbb{N} \text{ or } W_i = \phi\}$.

(For technical reasons this definition is slightly different in the general theory in §§8,9.)

The appropriate notion of equality on $\mathbb{E}$ is $i \equiv_\circ j$ iff $W_i = W_j$. We sometimes write $\mathbb{E}^\circ$ for $\mathbb{E}$. Also the objects denoted by the indices in $\mathbb{E}^\circ$ are denoted $\mathbb{E}\mathbb{D}^\circ$. Thus $\mathbb{E}\mathbb{D}^\circ = \mathbb{D}$.

2.3 non-deterministic extended integers

When we consider parallel or non-deterministic algorithms over $\mathbb{D}$, individual objects other than $\perp, \mathbb{N}$ appear necessary. For example, consider an algorithm using a "parallel or" such as:

$$* \quad Y := (1 \dot{-} (0 \cdot \phi(x))) \mid \mathrm{OR} \mid (1 \dot{-} (0 \cdot \phi(x+1)))$$

where $x \mid \mathrm{OR} \mid y$ has the following meaning:

x	y	$x \mid \mathrm{OR} \mid y$	
$\perp$	$\perp$	$\perp$	
$\perp$	n	n	
n	$\perp$	n	
n	n	n	
n	m	$\{n,m\}$	$m \neq n$

This operation, "parallel or", can be considered to map $\mathbb{D} \times \mathbb{D}$ into subsets of $\mathbb{N}$ (where $\perp$ is the null set and n is the unit set, $\{n\}$). If we intended our algorithm to map from $\mathbb{D} \times \mathbb{D}$ into $\mathbb{D}$, then we must treat $\{n,m\}$ as an undefined value.

If we define proper subtraction and multiplication on $\mathbb{D}$ to mean $1 \dot{-} \perp = \perp$ and $x \cdot \perp = \perp \cdot x = \perp$, then the algorithm * above assigns the value 1 to y exactly when one or both of $\phi(x)$ and $\phi(x+1)$ is defined. This assignment defines a unique function of type $(0 \to 0) \times 0 \to 0$. The function being computed requires non-determin

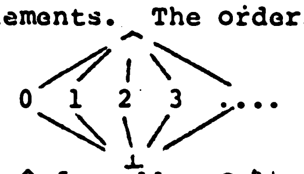
When we consider the related algorithm

** $\quad Y := \phi(x) \mid \text{OR} \mid \phi(x+1)$

we notice that it has value $\perp$ when both $\phi(x) = \perp$ and $\phi(x+1) = \perp$ but has an "undefined value" when $\phi(x)$ and $\phi(x+1)$ are different numbers.

In order to distinguish this new sort of undefined function value we introduce a new symbol, ^, say "top" or "overdefined". Then when treating parallel or non-deterministic procedures we are interested in computing over the domain $\hat{\mathbb{D}} := \{\perp, \mathbb{N}, \hat{}\}$ of <u>non-deterministic</u> <u>extended</u> <u>integers</u>.

For reasons of continuity to be discussed later we want an ordering on these elements. The ordering is given by the diagram



That is, $\perp \sqsubseteq n \sqsubseteq n \sqsubseteq \hat{}$ for all $n \in \mathbb{N}$ and no other relationships hold.

As before we want to consider effective representations of these domain elements. We are thus led to a class of <u>non-deterministic</u> <u>effective</u> <u>extended</u> <u>integers</u> $\hat{\mathbb{E}}\mathbb{D}^\circ = \hat{\mathbb{D}}$. The index i denotes the extended integer enumerated by W_i. If $W_i = \{n,m,\ldots\}$ $n \neq m$, then i represents ^.

Again as before we introduce the notation $\hat{\mathbb{E}}^\circ$, $\hat{\mathbb{E}\mathbb{D}}^\circ$ and $i \,\hat{\equiv}_\circ\, j$ iff W_i is empty or contains one element and $W_i = W_j$ or if W_i contains two or more elements, then so does W_j.

3. TYPE ONE

3.1 recursive partial functions

We have already seen how an attempt to describe numerical computation leads us from the domain $\mathbb{N}$ to the extended domains $\mathbb{D}$ and $\hat{\mathbb{D}}$. When we want to mention either of them we will use the notation D°.

We now want to examine the type 1 functions $D^{\circ} \to D^{\circ}$ more carefully. We are interested in the abstract problems associated with describing programming languages, and the type 1 objects are especially illuminating for this task because the most common algorithms are those for computing number theoretic functions.

The original goals of classical recursive function theory were to describe the computable functions, not to model programming languages. Therefore we cannot expect all the basic ideas of that theory to suit our purposes. However, recursion theory provides an abstract setting in which to formulate many of our problems, and it provides a precise basis for new concepts and definitions. As we mentioned earlier, the central notion in this theory is that of the partial recursive functions

$$\phi_i : \mathbb{N} \to \mathbb{N}$$

defined via some formalism such as recursion equations or Turing machines, etc.,which formalism is then arithmetized to provide an indexing, $\{\phi_i\}$. For technical reasons, however, we shall assume that we begin with some formalism for r.e. sets, $\{W_i\}$. From this we define the partial computable functions $\psi_i : \mathbb{N} \to \mathbb{D}$ by single-valuing the W_i and regarding them as enumerations of the graph of ψ_i

This is one of the most technically convenient ways to extend the definition of computability to higher types. Here are the details.

Let $p^2: \mathbb{N}^2 \to \mathbb{N}$ be a pairing function with inverses s_1^2, s_2^2. Thus $s_1^2(p^2(x,y)) = x$, $s_2^2(p^2(x,y) = y)$. In the presence of a set of pairing functions every integer n can be considered as a pair, $<s_1^2(n), s_2^2(n)>$. Thus an r.e. set W_i can be considered as a set of ordered pairs. For convenience we let $<x,y> := p^2(x,y)$. Such a set will represent the graph of a function $\mathbb{N} \to \mathbb{D}$ iff it is <u>single-valued</u>, i.e. if $<x,y> \in W_i$, then $<x,z> \in W_i$ iff $z = y$. Given any r.e. set W_i we can describe a procedure to <u>single-value</u> it which produces another r.e. set $W_{s(i)}$. The procedure is to enumerate W_i and cancel any pair x,y iff a pair $<x,z>$ with $z \neq y$ was previously enumerated.

<u>Definition</u> 1: Given any single-valued r.e. set $W_{s(i)}$, it defines a partial recursive function $\psi_i: \mathbb{N} \to \mathbb{D}$ by the rule that $\psi_i(x) = y \in \mathbb{N}$ iff $<x,y> \in W_{s(i)}$ and $\psi_i(x) = \perp$ iff there is no pair $<x,y>$, $y \in \mathbb{N}$ in $W_{s(i)}$.

This formalism can be extended to multi-argument functions $\psi_i^n: \mathbb{N}^n \to \mathbb{D}$ by the use of n-tupling functions $p^n: \mathbb{N}^n \to \mathbb{D}$.

For purposes of illustrating the relationship of these ideas to programming languages, we shall consider a simple programming language which can also be used to define an indexing of partial computable functions $\phi_i: \mathbb{N} \to \mathbb{N}$ or r.e. sets W_i.

The language possesses type declarations, TYPE 0 and TYPE(0 → 0) and the following statements.

input/output	arithmetic	control
READ x_i		IF $x_i \neq 0$ THEN GOTO L_i
PRINT x_i	$x_i \leftarrow x_i+1$	HALT
	$x_i \leftarrow x_i \dot{-} 1$	

Each program is a type declaration followed by a finite list of statements. The ordering of programs is denoted $\{\phi_i\}$. This formalism is similar enough to the literature (say Minsky [4] or [2]) that we do not discuss it in detail.

3.2 extended partial recursive functions.

Now with all the preliminaries aside we can turn to the problem of the semantics of programs. The examples of §2 suggest that in order to describe the behavior of programs of type $\mathbb{N} \to \mathbb{N}$ we need extended functions $\Phi_i : \mathbb{D} \to \mathbb{D}$. We can now define them.

Definition 2: Given a recursive partial function $\phi_i : \mathbb{N} \to \mathbb{N}$ it defines an extended partial recursive function $\Phi_i : \mathbb{D} \to \mathbb{D}$ by the conditions

(a) $\phi_i(x) = y \Rightarrow \Phi_i(x) = y$

(b) $\phi_i(x)\uparrow \Rightarrow \Phi_i(x) = \perp$

(c) if program ϕ_i of type $0 \to 0$ returns a value y without reference to the input, then $\Phi_i(\perp) = y$, otherwise $\Phi_i(\perp) = \perp$.

Remarks: The defined function Φ_i characterizes the algorithm i better than the computable function ϕ_i. Furthermore in the case of deterministic programs the semantics of composition are described precisely by composition of the Φ_i.

The definition of the Φ_i given above is somewhat unsatisfactory because it relies so heavily on the formalism. We would like a characterization in the spirit of the $W_{s(i)}$ definition of ψ_i.

Suppose we try the definition that $\bar{\Phi}_i$ is specified by single-valued subsets of $\mathbb{D} \times \mathbb{D}$. Then a function like

$$\bar{\Phi}(x) := \text{if } x = \bot \text{ then } 0 \text{ else } 1$$

is computable since its graph is simply $\{\langle\bot,0\rangle,\langle x,1\rangle \mid x \in \mathbb{N}\}$.

Unfortunately the function $\bar{\Phi}$ does not arise as the defining function of any numerical algorithm i, for if it did it would be the defining function of a computable function which solves the halting problem over $\mathbb{N}$.

The restriction which is missing is that functions $\bar{\Phi}_i$ defined by algorithms from $\mathbb{N}$ to $\mathbb{N}$ are monotomic on $\mathbb{D}$ in the sense that $\bar{\Phi}_i(\bot) \sqsubseteq \bar{\Phi}_i(x)$ for all x. This is because any algorithm which outputs a value on input $\bot$ cannot query its input, therefore it must give the same value on all inputs (we prove this in Theorem 1 below).

These observations lead us to define the partial recursive monotone functions over $\mathbb{D}$. We will actually use the term <u>continuous</u> rather than monotone because the appropriate concept for higher types is <u>continuity</u> which degenerates to monotonicity at type 1.

First we describe a procedure for creating r.e. subsets of $\mathbb{D} \times \mathbb{D}$ which define continuous functions. We could present subsets of $\mathbb{D} \times \mathbb{D}$ directly, call them $\mathbb{W}_i$ and single value them by canceling any pair $\langle c,d\rangle$ if a pair $\langle a,b\rangle$ was previously enumerated and $a \sqsubseteq c$ but not $b \sqsubseteq d$ or $c \sqsubseteq a$ but not $d \sqsubseteq b$.

We prefer however to define these objects $\bar{\Phi}_i$ in terms of the r.e. sets W_i over $\mathbb{N}$. We can do this by arithmetizing $\mathbb{D}$. To conform to our future terminology we will speak of a basis for the domain $\mathbb{D}$. The basis is the r.e. family of sets $B_0=\perp$, $B_1= 0,\ldots,B_{n+1}= n,\ldots$, Using pairing functions we will interpret an r.e. set W_i as a set of pairs, $\{\langle n, m\rangle\}$ which represent pairs of basis elements $\{\langle B_n, B_m\rangle \mid \langle n, m\rangle \in W_i\}$. Then we can make W_i monotone and single-valued by enumerating W_i and guaranteeing that the pair $\langle c,d\rangle$ is cancelled from the enumeration if a pair $\langle a,b\rangle$ was previously enumerated and $B_a \sqsubseteq B_c$ but not $B_a \sqsubseteq B_b$ or $B_c \sqsubseteq B_a$ but not $B_b \sqsubseteq B_d$. Call the resulting modified set $W_{s(i)}$.

Definition 3: Given any single-valued r.e. subset of $\mathbb{D} \times \mathbb{D}$, as above, say $W_{s(i)}$, it defines (or partially computes) the continuous partial recursive extended function $\Phi_i : \mathbb{D} \to \mathbb{D}$ such that $\Phi_i(B_x) = B_y$ if $\langle x,y\rangle \in W_{s(i)}$ or if $\langle \perp, B_y\rangle \in W_{s(i)}$, and $\Phi_i(B_x) = \perp$ if no pair $\langle x,y\rangle \in W_{s(i)}$.

Notice for future reference that we can write this definition as $\Phi_i(x) := \bigsqcup\{B_y \mid \langle z,y\rangle \in W_{s(i)} \text{ and } B_z \sqsubseteq x\}$.

Remarks: We speak of the total function $\Phi_i : \mathbb{D} \to \mathbb{D}$ as partial recursive because we can compute part of it, namely the part with numerical range. We can say that the set $W_{s(i)}$ "partially computes the function Φ_i". We also imagine that some algorithm for a function $\mathbb{N} \to \mathbb{N}$ defines Φ_i, but in this account we are not interested in that algorithm.

The functions $\Phi_i : \mathbb{D} \to \mathbb{D}$ are not computable in the usual sense because they never return $\perp$ as a value (i.e., $\perp$ is never printed on the output tape). But we can however compute with the indices of inputs and outputs, i.e. from i and index j for input x we can compute a k such that $\Phi_k^{\circ} = \Phi_i^1(\Phi_j^{\circ})$. This is similar to what we do when treating effective operators as we next see.

3.3 effective operators of type 1

One of the ways in which real programming languages deal with the undefined value $\perp$ is by passing unevaluated expressions as arguments to procedures. Thus in the case of a composite function $\Phi_0(\Phi_1(x),\Phi_2(y))$, the arguments to Φ_0 may come as decimal representations of integers, if $\Phi_1(x)$ and $\Phi_2(y)$ are evaluated, or they may come as effective representations of integers, i.e. as expressions to be evaluated. In the case of a purely numerical language, expressions such as $\Phi_1(x)$ may be considered as indices via the S-m-n function; thus $\Phi_1(x)$ has the index $S_0^1(1,x)$.

In classical recursive function theory this treatment of expressions and indices as arguments is usually not introduced until effective operations of type 2 are defined. But for this paper we want an account of effective operations of all types, including 0 and 1 because all types are important in modeling programming languages.

Let us denote effective operations of type 1 by Ψ_j: $\Sigma D \to \Sigma D$. We can define them using a programming language formalism which makes provision for passing names as values and evaluating names. We can also define them in terms of the ψ_i. The latter is the more mathematically concise approach, but it does not of course reveal the programming language issues.

Definition 4: An effective operation of type 1, Ψ_j^1: $\Sigma D^{\circ} \to \Sigma D^{\circ}$ is defined by the partial recursive function $\psi_j : \mathbb{N} \to \mathbb{D}$ iff for all

$x,y \in \Sigma^\circ$, $x \equiv_\circ y$ implies $\psi_j(x) = \psi_j(y)$, in which case $\Psi_j(\Psi^\circ_x) := \psi_j(x)$.

Remark: Given an index j it is of course undecidable whether it defines an effective operation on $\Sigma\mathbb{D}^\circ$. The set of indices of type 1 effective operations is denoted Σ^1. It is a non-recursive subset of $\mathbb{N}$.

3.4 relationship between effective operations and recursive functions

It is natural to wonder whether the effective operations and the extended partial recursive functions define the same functions from $\mathbb{D}$ to $\mathbb{D}$. We answer this question with the following theorem.

Theorem 1:

(a) Given any $\Phi^1_i: \mathbb{D}^\circ \to \mathbb{D}^\circ$ we can find a $\Psi_j: \Sigma\mathbb{D} \to \Sigma\mathbb{D}$ such that * holds:

$$* \quad \Phi^\circ_k = n \Longrightarrow \Phi^1_i(n) = \Psi^1_j(\Psi^\circ_k)$$

(b) Given any $\Psi_j: \Sigma\mathbb{D}^\circ \to \Sigma\mathbb{D}^\circ$ we can find a $\Phi_i: \mathbb{D} \to \mathbb{D}$ such that * holds.

Proof: This theorem follows as a special case of the main theorem in § 9, therefore its proof sketch need only be examined by those wanting to see a simpler proof unencumbered by the full notational generality required for arbitrary types.

(a) Given Φ_i and x do the following simultaneously:

(i) evaluate x

(ii) compute $\Phi_i(\bot)$

If $\Phi_i(\bot)$ has a value, then output that value. If x evaluates to a decimal value n, then compute $\Phi_i(n)$ and output that value.

This procedure succeeds in defining an operation because Φ_i

is a continuous function. It defines an effective operation because the entire procedure described is computable.

(b) Given $\Psi_j: \Sigma \to \Sigma$ and given n, first compute an index, x, for n (say $\psi_x(y) = n$ for all n). Then simultaneously

(i) compute $\psi_j(x)$

(ii) compute $\psi_j(x_0)$ where x_0 is an index for $\perp$, i.e. $\Psi^0_{x_0} = \perp$.

If $\psi_j(x)$ evaluates to m_1, then $\Phi(n) = m_1$

If $\psi_j(x_0)$ evaluates to m_2, then put $\Phi(n) = m_2$.

Now how do we know that the resulting function Φ is actually a Φ_j? It is conceivable that by using information about the indices, ϕ_j might compute some non-continuous function.

To show that this is impossible, consider the two possible cases:

(a) $\psi_j(x_0) = m_2 \neq \perp$ and $\psi_j(x) = m_1 \neq \perp$ for $\Psi^0_x \neq \Psi_{x_0}$ and $m_1 \neq m_2$

(b) $\psi_j(x_0) = m_2 \neq \perp$ and $\psi_j(x) = m_2$, $m_2 \neq \perp$.

Consider case (a). In order to decide whether $\psi_i(i)$ halts, we produce a function $\psi_{t(i)}$ such that t(i) is an index for $\perp$ iff $\psi_i(i)$ diverges. Otherwise we make $t(i) \equiv_e x$. Now to decide the halting problem in index we simply compute $\psi_j(t(i))$. If the value is m_1, then $\psi_i(i)$ converges. If the value is m_2, then $\psi_i(i)$ diverges.

In case (b), to decide whether $\psi_i(i)$ halts we again produce a function $\psi_{t(i)}$ such that t(i) is an index for $\perp$ if $\psi_i(i)$

diverges and otherwise is an index for x. We can again decide the halting problem.

Q.E.D.

3.5 non-deterministic computable functions

In classical recursivo function theory, non-deterministic algorithms, are rarely mentioned. In fact Kleene [7] does not permit them in his treatment of type 1 or type 2 [illegible]. But in computing theory these algorithms are vital. Not only are they important in the study of parallelism and attendant subjects in the study of operating systems, but they are also important in computational complexity and the theory of algorithms.

As in the case of the $\bar{\Phi}_i$, we can define extended non-deterministic computable functions $\hat{\Phi}_i : \hat{D} \to \hat{D}$ in at least two ways; we can provide a new formalism with non-deterministic instructions (such as the non-deterministic goto, GOTO $L_1 : L_2$ or parallel assignments (as in [2] etc.) or we can define them in terms of the W_i. Again we adopt the second alternative, yet again for comparison the reader might want to examine the language which would result from adding a non-deterministic goto. Say that GOTO $L_1 : L_2$ is allowed as a statement.

Notice that in a non-deterministic programming language, a program, say $\hat{\phi}_i$, may calculate several different values, say $\perp, y_1, y_2$, on input x depending on what branch of the goto is taken. When this happens we say that the program is <u>indeterminate on x</u> (which amounts to being overdefined in our notations).

To define the non-deterministic partial computable extended functions we proceed as in the case of $\mathcal{F}_i$ by arithmetizing a basis and using pairing functions to interpret each W_i as an enumeration of basis pairs which define the graph of $\hat{\mathcal{F}}_i$.

<u>Definition</u> 5: Given pairing functions $p:\mathbb{N}^2 \to \mathbb{N}$ and a basis for $\hat{\mathbb{D}}$, say $B_0 = \bot$, $B_1 = \hat{}$, $B_2 = 0, \ldots, B_{n+2} = n, \ldots$ and the ordering $\sqsubseteq$ on $\hat{\mathbb{D}}$, then the r.e. set W_i defines a non-deterministic partial computable extended function $\hat{\mathcal{F}}_i : \hat{\mathbb{D}} \to \hat{\mathbb{D}}$ by the condition $\hat{\mathcal{F}}_i(x) = \bigsqcup\{B_y \mid \langle z,y\rangle \in W_i \text{ and } B_z \sqsubseteq x\}$. The <u>weak</u> <u>domain</u> of $\hat{\mathcal{F}}_i$ is $\{x \mid \hat{\mathcal{F}}_i(x) \neq \hat{}\}$ and the <u>domain</u> of $\hat{\mathcal{F}}_i$ is $\{x \mid \hat{\mathcal{F}}_i(x) \in \mathbb{N}\}$.

Remarks: (1) To compute $\hat{\mathcal{F}}_i$ on input x, find the least upper bound of the elements B_y such that $\langle z,y\rangle \in W_i$ and $B_z \sqsubseteq x$. Notice, if $\langle z,y\rangle$ and $\langle z,y'\rangle$ are in W_i and $y \neq y'$, then $\hat{\mathcal{F}}_i(x) = \hat{}$. Also notice if $\langle z_1,y_1\rangle$, $\langle z_2,y_2\rangle \in W_i$ and $y_1 \neq y_2$, then $\hat{\mathcal{F}}_i(\hat{}) = \hat{}$.

(2) These conditions clearly define a unique continuous function on $\hat{\mathbb{D}}$. In the case when we know that the output is numerical, the value is computable because we can search W_i looking for a pair $\langle z,y\rangle$, $\bot \neq B_y \neq \hat{}$.

(3) Notice also that this definition allows functions of the sort $\mathcal{F}(\bot) = 0$, $\mathcal{F}(x) = \hat{}$, for all $x \neq \bot$. This may seem strange, but we can compute such a function by the following program:

```
TYPE(0 → 0)
PRINT 0
READ X
PRINT X+1
END
```

(4) It is interesting to notice that non-deterministic functions $\hat{\mathcal{F}}_i : \hat{\mathbb{D}} \to \hat{\mathbb{D}}$ do not compute "any more than" the deterministic functions $\mathcal{F}_i : \mathbb{D} \to \mathbb{D}$ in the sense that given any graph $\{\langle x,y \rangle \mid x,y \in \mathbb{N}\}$ of a non-deterministic function $\hat{\mathcal{F}}_i$, that graph is a subgraph of a deterministically computed function. To find the deterministic function we need only use the single-valuing of W_i.

It is, however, the case that the $\hat{\mathcal{F}}_i$ are more complex than $\mathcal{F}_i$ in the sense that their domains are more complex. The domains of $\mathcal{F}_i$ are r.e. sets, but the domains of $\hat{\mathcal{F}}_i$ cannot be recursively enumerated. To see this consider the function defined by the program :

```
TYPE(0 → 0)
READ X
PRINT 0
Y ← φX(X)
PRINT Y+1
END
```

For $x \in \mathbb{N}$, whenever $\phi_x(x)$ diverges the output of the program will be 0, but when $\phi_x(x)$ converges, the output will be ^. Thus domain $\hat{\mathcal{F}} = \{x \mid \phi_x(x) \text{ diverges}\}$.

3.6 non-deterministic effective operations

The reasons that led us to examine the effective operations $\Psi_j : \Sigma\mathbb{D} \to \Sigma\mathbb{D}$ lead us also to consider non-deterministic effective operations, $\hat{\Psi}_j : \hat{\Sigma\mathbb{D}} \to \hat{\Sigma\mathbb{D}}$. We can define them via a programming language or via the ψ_i. We choose the latter approach. The following definitions are simplicity itself.

First to define the non-deterministic recursive partial functions $\hat{\psi}_i : \mathbb{N} \to \mathbb{N}$, we use the W_i as follows.

Definition 6: Given pairing functions $\mathbb{N}^2 \to \mathbb{N}$, an r.e. set W_i defines a non-deterministic recursive partial function $\hat{\psi}_i : \mathbb{N} \to \hat{\mathbb{D}}$ by the condition

$$\hat{\psi}_i(x) := \begin{cases} \hat{} & \text{if } \exists <x,y>, <x,z> \in W_i,\ z \neq y. \\ y & \text{if } \exists <x,y> \in W_i \text{ and } \psi_i(x) \neq \hat{}. \\ \bot & \text{if } \neg\exists <x,y> \in W_i. \end{cases}$$

Now using the definition of Ψ_j as a model we say

Definition 7: a non-deterministic effective operation of type 1, $\hat{\Psi}_j : \hat{E\mathbb{D}} \to \hat{E\mathbb{D}}$, is defined by the non-deterministic partial recursive function $\hat{\psi}_j : \mathbb{N} \to \hat{\mathbb{D}}$ iff for all $x,y \in E°$, $x \equiv_\circ y$ implies $\hat{\psi}_j = \hat{\psi}_j(y)$ in $\hat{\mathbb{D}}$, in which case $\hat{\Psi}_j(\hat{\varphi}_x) := \hat{\psi}_j(x)$.

Let $\hat{E}^1$ denote the indices of these operations.

4. TYPE TWO

4.1 type 1 inputs

We now want to examine what happens when we lift the results for type 1 up to type 2. We identify functionals, i.e. type $(0 \to 0) \times 0 \to 0$, with operators of type 2, i.e. type $(0 \to 0) \to (0 \to 0)$.

Clearly operators and functionals are important objects in programming languages. They appear in even a primitive language like FORTRAN. Advancing from type 1 to type 2 requires new concepts because the input objects are so different. Here are some of the important differences which affect this paper.

(1) Input objects x^1: $\mathbb{D}^\circ \to \mathbb{D}^\circ$ cannot be completely represented by a unique finite notation system as can the type 0 objects in $\mathbb{N}$ and $\mathbb{D}$, and as can the type 1 inputs to effective operations of type 2.

(2) Non-determinism is an essential feature of type 2 computability, as we show in Theorem 3, §4.5.

(3) There is no literature which treats all the objects of interest to us, so in particular there is no simple theoretical programming language model for non-determinism (such as those cited above in § 2).

The problem of representing inputs determines our approach to type 2 computability. The idea is to represent type 1 input objects by finite approximations to them. This is also the approach taken in treating functions $\phi:\mathbb{N} \to \mathbb{N}$ as inputs to type 2 operators over $\mathbb{N}$. A finite approximation to $\phi(\ )$ is a finite partial function, say $F:\mathbb{N} \to \mathbb{N}$ such that $F(\ ) \subseteq \phi(\ )$.

We can represent every function $\phi:\mathbb{N} \to \mathbb{N}$ as a limit of finite functions F_i such that F_{i+1} is more defined than F_{i+1}, thus $\phi = \bigsqcup F_i$. (For example, see Kleene's proof of the Recursion Theorem, [7] p349)

To represent functions Φ: $\mathbb{D}^\circ \to \mathbb{D}^\circ$ we might also try using finite functions $\mathbb{D}^\circ \to \mathbb{D}^\circ$. But now we have a problem. Should we represent arbitrary functions $\mathbb{D}^\circ \to \mathbb{D}^\circ$ or only the continuous functions? If we seek guidance from the $\mathbb{N} \to \mathbb{N}$ case we see that each $\phi:\mathbb{N} \to \mathbb{N}$ satisfies the condition $\phi(\perp) = \perp$ and is thus continuous. This suggests we approximate only continuous functions.

Further, and more decisive, reasons for using continuous functions appear at type 3. Looking ahead for a moment, we can see that one cannot hope to represent arbitrary type 2 objects by monotone chains of finite approximations because there are too many arbitrary type 2 functions. On the other hand, one might expect our approach to continuous objects to carry over to type 2 inputs.

For these reasons (which will be elaborated in the type 3 discussion) we consider only continuous functions from $\mathbb{D}^\circ \to \mathbb{D}^\circ$.

<u>Definition</u> 1: $\mathbb{D}^1 := \{\Phi : \mathbb{D}^\circ \to \mathbb{D}^\circ \mid \Phi \text{ is continuous under } \sqsubseteq\}$.

We will choose continuous finite functions as our <u>basis</u> <u>for</u> $\underline{\mathbb{D}^1}$. We denote these by $B_0^1, B_1^1, B_2^1, \ldots$. Because these B_i^1 are continuous, the basis $\{\langle\bot,1\rangle\ldots\}$ represents the constant function $\Phi(x) = 1$ for all $x \in \mathbb{N}$. Likewise any function $\{\quad \langle\bot,y\rangle\}$ represents $\Phi(x) = y$ for all $x \in \mathbb{D}$.

Given these definitions we observe that

Proposition 1: Every $\Phi \in \mathbb{D}^1$ is the least upper bound of a chain of basis elements, i.e. $\Phi := \bigsqcup B_{i_j}$ where $B_{i_j} \sqsubseteq B_{i_{j+1}}$.

We observe also that the computable type 1 objects, $\mathbb{C}\mathbb{D}^1$, are simply those continuous elements of $\mathbb{D}^1$ with r.e. approximating chains. This situation suggests our approach to type 2 computability.

4.2 discussion of type 2 computability over $\mathbb{D}$

Given the representation of type 1 inputs and thinking ahead to defining type 3 objects, it is natural to try to define computable type 2 objects as r.e. sequences of basis elements. Before we can do this we must define type 2 basis elements.

A "finite piece" of a type 2 object will be a finite function paired with a finite function. These correspond to the argument value pairs, $\langle x,y\rangle$, of the type 1 case. Basis elements are then finite unions of these elements(just as functions are unions of sets of argument value pairs).

For example, consider the operator $\Phi^2: \mathbb{D}^1 \to \mathbb{D}^1$ which maps each x^1 to the function $\lambda y^\circ[x^1(2y)]$, thus $\lambda y[2y]$ is mapped to $\lambda y[4y]$. Some typical argument value pairs are:

$B^2_{i_1} := \langle\{\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle\}, \{\langle 0,0\rangle, \langle 1,2\rangle\}\rangle$

$B^2_{i_2} := \langle\{\langle 5,5\rangle, \langle 7,3\rangle, \langle 9,1\rangle\}, \{\langle \bot,\bot\rangle\}\rangle$

$B^2_{i_3} := \langle\{\langle 5,5\rangle, \langle 7,3\rangle, \langle 9,1\rangle, \langle 10,2\rangle\}, \{\langle 5,2\rangle\}\rangle$

We can now take as basis elements unions of compatible argument value pairs. Perhaps computable type 2 objects will be r.e. sequences of these bases. But this is not quite correct. Consider the following type 2 object:

<u>Example</u>: <u>if</u> x^1 is defined on an even integer <u>then</u> 2
<u>if</u> x^1 is defined only on odd integers <u>then</u> 1
if x^1 is undefined <u>then</u> $\bot$.

We can enumerate a set of basis elements for this functional (type$(0 \to 0) \to 0$). The enumeration is quite simple, e.g.
$\langle\{\langle 0,0\rangle\}, 2\rangle$, $\langle\{\langle 1,1\rangle\}, 1\rangle$, $\langle\{\langle 0,0\rangle, \langle 1,1\rangle\}, 2\rangle$, ...
Clearly this functional is not computable.

If we define the computable type 2 objects by a formalism, it will be clear what is missing. Likewise if we compare to our definition of type 1 objects it will be clear what is missing. We need to impose continuity.†

4.3 definitions of type 2 computable objects over $\mathbb{D}$

We can now define type 2 computable objects over $\mathbb{D}$. The idea is to single-value and make continuous with respect to type 2 the subsets of basis elements, and then say that $\mathcal{F}^2_i(x^1) :=$ $\bigsqcup\{B^1_j \mid \langle k,j\rangle \in W_{s_2(i)}$ and $B^1_k \sqsubseteq x^1\}$. The difficulty is that we must define an appropriate single-valuing procedure for sets of basis elements (note in the non-deterministic case this subtlety disappears).

Single-valuing can be understood from a few simple examples. Clearly given the argument value pairs (encoded as integers of course) in a set W, $\{\langle\langle 0,0\rangle, \langle 1,1\rangle\rangle, \langle 0,2\rangle\}$ and $\{\langle\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle\rangle, \langle 0,3\rangle\}$ we have an instance of a non-single-valued set W. This type of incompatibility is easy to recognize. Namely, there are basis elements $\langle B_a,B_b\rangle$, $\langle B_c,B_d\rangle$ such that $B_a \sqsubseteq B_c$ but not $B_b \sqsubseteq B_d$.

We might also have the situation that W contains $\{\langle\langle 0,0\rangle, \langle 3,3\rangle\rangle, \langle 0,2\rangle\}$ and $\{\langle\langle 1,1\rangle, \langle 3,3\rangle\rangle, \langle 0,3\rangle\}$.. In this case the set is also not single-valued, but now we have $\langle a,b\rangle$, $\langle c,d\rangle$ where B_a and B_b are not comparable under $\sqsubseteq$. However B_a and B_b are

---

† Even in the case of type 2 objects $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$ continuity is a crucial property; whereas in the type 1 case, if we consider functions $\mathbb{N} \to \mathbb{N}$ then continuity disappears. At type 1 it is necessary only for functions $\mathbb{D} \to \mathbb{D}$.

compatible in the sense that their union, $B_a \cup B_b$ is a basis element. Whenever that happens, the pair B_b and B_d must also be compatible if the definition suggested above is to work. Intuitively we can see this if we imagine how a computation using W might proceed. Suppose the input is x^1. Then the program examines the values $x^1(0)$, $x^1(1)$, $x^1(2)$. If the result is [illegible] independent of the time taken for the oracle to respond, then it must give the same value when $x^1(0)$, $x^1(2)$ converge first or when $x^1(1)$, $x^1(2)$ converge first. But it does not if W represents the program.

We must also consider the case where W contains $\{\langle\langle 1,1\rangle, \langle 0,1\rangle\rangle, \langle 0,2\rangle\}$ and $\{\langle\langle 0,1\rangle, \langle 1,1\rangle\rangle, \langle 0,3\rangle\}$. Such a set is also not single-valued because the input pair $\langle \underline{1},1\rangle$ must mean that the basis element has value 1 on all inputs.

To single-value a set W_i with respect to type 2 means to produce a new r.e. set $W_{s_2(i)}$ which can represent an enumeration of finite pieces of a type 2 object. To do this we cancel from W_i those argument value pairs which are not compatible with what is already in $W_{s_2(i)}$. Incompatibility among value-argument pairs means that they can not all be part of the same function, e.g. $\langle 0,1\rangle$ and $\langle 0,0\rangle$ are incompatible argument value pairs for a type 1 function.[†] In the type 2 case, two argument value pairs, say $\langle B_a^1, B_b^1\rangle$ and $\langle B_c^1, B_d^1\rangle$ are incompatible iff B_a^1 and B_c^1 are compatible but B_b^1 and B_d^1 are not.

These examples lead us to the following definition of single-valuing with respect to type 2.

---

† We can view type 1 single-valuing in the same way: we cancel an incompatible pair, e.g. $\langle 0,1\rangle$ and $\langle 0,2\rangle$ are incompatible. Conversely, finite functions are compatible iff their union is a finite function.

<u>Definition</u> 2: Given pairing functions $p : \mathbb{N}^2 \to \mathbb{N}$ and a basis for $\mathbb{D}^1$, say $B^1_0, B^1_1, \ldots$ and an r.e. set W_i, the single-valued with respect to type 2 set, $W_{s_2(i)}$, is obtained by canceling an argument value pair $<c,d>$ iff there is a pair $<a,b>$ which was previously enumerated such that B^1_a, B^1_c are compatible but B^1_b and B^1_d are not. This procedure is recursive because it is possible to decide the compatibility condition for a finite set of basis elements.

<u>Definition</u> 3: Define $\mathcal{F}^2_i : \mathbb{D}^1 \to \mathbb{D}^1$ by $W_{s_2(i)}$ as $\mathcal{F}^2_i(x^1) := \bigsqcup\{B_b \mid <a,b> \in W_{s_2(i)} \text{ and } B_a \sqsubseteq x^1\}$. Let $\mathbb{CD}^2$ be the class of <u>computable type 2 objects</u>.

Remark: Computing $\mathcal{F}^2_i$ on input $x^1 \in \mathbb{D}^1$ results in another continuous type 1 object, say y^1, i.e. $\mathcal{F}^2_i(x^1) = y^1$. To obtain a numerical value we must evaluate y^1 on some type 0 input x^0. Intuitively to obtain the numerical value $y^1(x^0)$ given x^0 we compute as follows. Compute finite segments of x^1 by calling for the value of x^1 on various inputs (doing this assumes some type of parallelism in the evaluation of x^1 so that a call to x^1 on points where it is undefined does not stop the computation).†

Simultaneously with the calls to x^1, search $W_{s(i)}$ for pairs $<a,b>$ where B^1_a is a finite part of x^1 obtained by evaluating x^1. Once such a pair is found, search B^1_b looking for a pair $<x^0,y^0>$. If such a pair is found, then output y^0.

From the definition of these objects it is easy to see

Proposition 2: The type 2 computable operators $\mathcal{F}^2_i : \mathbb{D}^1 \to \mathbb{D}^1$ are continuous, i.e. $\mathcal{F}^2_i(\lim x^1_j) = \lim \mathcal{F}^2_i(x^1_j)$.

---

†We obtain another class of operators if we assume that x^1 is evaluated in the order called for by the enumeration of $W_{s_2(i)}$.

Remark: We can also state the definition of single-valuing and type 2 computable operator in terms of type 2 basis elements rather than in terms of argument value pairs. In this case the definition of $\mathcal{F}_i$ has the form

$$\mathcal{F}_i^2 := \bigcup\{B_k^2 \mid k \in U_i^2\}$$

where U_i^2 is a single-valued set of indices of type 2 basis elements. This is the approach taken in §7.

4.4 type 2 effective operations

It is reasonably clear how type 2 effective operations should be defined. This is even done in Rogers [16]. In this case the inputs are concretely represented by indices.

Definition 4: The type 2 effective operations $\Psi_j^2 : \mathbb{ED}^1 \to \mathbb{ED}^1$ are defined by those functions $\psi_j : \mathbb{N} \to \mathbb{N}$ such that $x \equiv_1 y$ implies $\psi_j(x) \equiv_1 \psi_j(y)$. Let $\mathbb{ED}^2$ denote the set of type 2 effective operations.

It will be important in what follows to know that these effective operations are continuous. We prove this now for this special case. We also prove this in much more generality as Lemmas in 9.2. So this proof is mainly expository and is necessary only for those readers who do not want to read the general proof or for those who want a simple introduction to the general proof.

Lemma 1: The effective type 2 operators, Ψ_j^2, are monotone, i.e. $x^1 \sqsubseteq y^1 \Rightarrow \Psi_j^2(x^1) \sqsubseteq \Psi_j^2(y^1)$.

<u>Proof</u>: Suppose that some Ψ_j is not monotone. Then there are type 1 objects x^1 and y^1 with $x^1 \sqsubseteq y^1$ but not $\Psi_j(x^1) \sqsubseteq \Psi_j(y^1)$.

If $x^1 \sqsubseteq y^1$, then there are basis elements $B^1_{i_1}$, $B^1_{i_2}$ such that $B_{i_1} \sqsubseteq B_{i_2}$ and $B^1_{i_1} \sqsubseteq x^1$, $B_{i_2} \sqsubseteq y^1$, but not $\Psi_j(B^1_{i_1}) \sqsubseteq \Psi_j(B^1_{i_2})$. This is because if not $\Psi_j(x^1) \sqsubseteq \Psi_j(y^1)$, then there is some finite part of these values for which this is true. Those finite parts are generated by basis elements. Suppose we write

$A_1 = \Psi_j(B^1_{i_1})$, $A_2 = \Psi_j(B^1_{i_2})$. Then not $A_1 \sqsubseteq A_2$.

To complete the proof we will show that we can use the fact that Ψ_j is an effective operation and the fact that $A_1 \not\sqsubseteq A_2$ to decide the halting problem To this end, consider the following special indices for B_{i_1} and B_{i_2}. Arrange a description of the basis elements so that b(i) is an index for B_{i_2} if $\phi_i(i)$ converges and is an index for B_{i_1} if $\phi_i(i)$ diverges (somewhere in the basis B_{i_2} there is a pair <x,y> which is not in B_{i_1}, otherwise not $B_{i_1} \sqsubseteq B_{i_2}$, so have <x,y> included in B_{i_1} iff $\phi_i(i)$ halts).

Now consider the ways in which $A_1 \not\sqsubseteq A_2$ might hold. There are two ways.

(1) $\exists x \quad A_1(x) = y_1 \neq y_2 = A_2(x) \qquad y_1, y_2 \in \mathbb{N}$

(2) $\exists x \quad A_1(x) = y \quad A_2(x) = \perp \qquad y \in \mathbb{N}$

(Of course $A_1 \not\sqsubseteq A_2$ may happen for both reasons.)

In case (1) it is clear how we could solve the halting problem. We compute $\Psi_j(b(i))$ and ask whether $A_1(x) = y_1$, or $A_2(x) = y_2$.

In case (2) we must be more careful. To decide whether $\phi_i(i)$ halts, we start computing $\phi_i(i)$ and $\Psi_j(b(i))$. If $\phi_i(i)$ halts we know what happens. On the other hand $\Psi_j(b(i))$ returns an index for A_1 or A_2. We compute with this index on x and if a value $y \in \mathbb{N}$ results, then we know that $\phi_i(i)$ diverges. Thus we can solve the halting problem.

Theorem 1: The type 2 effective operators are continuous over the cpo D^1, i.e. for every monotone chain x_i^1, $\Psi_j(\lim x_i) = \lim \Psi_j(x_i)$.

Proof: This theorem follows from §9.2 and the proof is given for purely expository purposes. We consider two cases.

(1) $\Psi_j(\lim x_i) \sqsubseteq \lim \Psi_j(x_i)$. To prove this we notice that $\Psi(\lim x_i)$ is a function, so we can evaluate it on $y \in D^0$. We consider the subcases

(a) $\Psi_j(\lim x_i)(y) = \bot$

(b) $\Psi_j(\lim x_i)(y) = z \in \mathbb{N}$.

In case (a) we are done because clearly $(\lim \Psi_j(x_i))(y) \sqsupseteq \bot$. For convenience of expression let us write the limit function $\lim \Psi_j(x_i)$ as Ψ_ω.

In case (b), some finite basis B is used to compute z, say $\Psi_j(B)(y) = z$. But then by Lemma 1 , since $B \subseteq x_i$ for some i, $\Psi_j(B) \sqsubseteq \Psi_j(x_i) \sqsubseteq \lim \Psi(x_i) = \Psi_\omega$. So $\Psi_\omega(y) = z$ also.

(2) $\lim \Psi_j(x_i) \sqsubseteq \Psi_j(\lim x_i)$

Again we consider two cases

(a) $\Psi_\omega(y) = \bot$

(b) $\Psi_\omega(y) = z \in \mathbb{N}$

As before in case (a) we are done. In case (b), we know that $\Psi_j(y) = z$ can be computed using some $\Psi_j(x_i)$. But then by the monotonicity lemma, since $x_i \sqsubseteq \lim x_i$, $\Psi_\omega(y) = \Psi_j(x_i)(y) \sqsubseteq \Psi_j(\lim x_i)$.

Q.E.D.

We are now in a position to state and sketch a proof of the main theorem for the special case of type 2 objects.

Theorem 2:

(a) Given $\Phi_i^2 : \mathbb{D}^1 \to \mathbb{D}^1$ we can find a $\Psi_j^2 : \mathbb{ED} \to \mathbb{ED}^1$ such that * holds.

* $\Psi_x^1 = y^1 \in \mathbb{CD}^1 \Rightarrow \Phi_i^2(y^1) = \Psi_j^2(\Psi_x^1)$

(b) Given $\Psi_j^2 : \mathbb{ED}^1 \to \mathbb{ED}^1$ we can find a $\Phi_i^2 : \mathbb{D}^1 \to \mathbb{D}^1$ such that * holds.

Proof: Again the reader is reminded that this proof is given for expository purposes only and should be ignored by the reader who will examine the proof of our main theorem (9.2).

(a) To define Ψ_j^2 on input Ψ_x^1, start evaluating x and forming (finite) basis elements while simultaneously searching for these elements in the $W_{s_2(i)}$ defining Φ_i^2. Output the index of a procedure for treating the basis elements as functions of type 1. This works because Φ_i is continuous.

(b) Given Ψ_j^2 and an oracle for $n^1 \in \mathbb{D}$, calculate finite function segments of n^1. These are basis elements. Find indices for them and evaluate them with Ψ_j^2. Find finite segments of the output and put the resulting value-argument pairs into an r.e. W set. This r.e. set will define Φ_i because Ψ_j^2 is continuous.

Q.E.D.

4.5 type 2 non-deterministic partial computable operators

Some form of non-determinism or parallelism is necessary to compute the most general class of computable type 2 objects on arbitrary (continuous) inputs.† For example, the functional defined in §2.3

$$y := (1 \dot{-} 0 \cdot \phi(x)) \mid OR \mid (1 \dot{-} 0 \cdot \phi(x+1))$$

can be seen to require non-determinism because a deterministic procedure would decide whether to evaluate $\phi(x)$ or $\phi(x+1)$ first. If it chooses $\phi(x)$ and only $\phi(x+1)$ is defined, then the procedure will never finish evaluating $\phi(x)$ so it will never discover that $\phi(x+1)$ is defined.

The important question is how to treat operators which may be indeterminate on certain inputs. On one hand we can follow Kleene and eliminate such operators, considering them to be non-computable. This leaves us with the subclass of determinate non-deterministic partial computable operators.

On the other hand, we can attempt to compute with operators which may have indeterminate values on some inputs. This leads to the full class of non-deterministic partial computable operators.

The domain $\hat{\mathfrak{D}}$ not only indicates that we are considering this new class of operators, but it allows us to define an ordering in terms of which the new operators will be continuous. To see what this ordering should be, consider the following simple example of

† This most general class is called by Rogers the partial recursive operators. He does not distinguish the deterministic and non-deterministic varieties. Kleene [7,9,10] does not allow the most general class. He allows only the recursive operators in Rogers' terminology or in our terminology he allows only the deterministic partial computable operators.

an operator $\tilde{\mathcal{F}}$ on the inputs Φ_0, Φ_1, Φ_3. These are defined by:

$$\Phi_0(x) := \begin{cases} 0 \text{ if } x = 0 \\ \perp \text{ otherwise} \end{cases} \qquad \Phi_1(x) := \begin{cases} 1 \text{ if } x = 1 \\ \perp \text{ otherwise} \end{cases}$$

$$\Phi_3(x) := \begin{cases} 0 \text{ if } x = 0 \\ 1 \text{ if } x = 1 \\ \perp \text{ otherwise} \end{cases}$$

$\tilde{\mathcal{F}}(\Phi) := \Phi(0) \mid \text{OR} \mid \Phi(1)$. Thus $\tilde{\mathcal{F}}(\Phi_0) = 0$, $\tilde{\mathcal{F}}(\Phi_1) = 1$, but $\tilde{\mathcal{F}}(\Phi_3)$ is undefined. Since $\Phi_0 \sqsubseteq \Phi_3$ and $\Phi_1 \sqsubseteq \Phi_3$ if $\tilde{\mathcal{F}}$ is to be monotone, then we need

$$\tilde{\mathcal{F}}(\Phi_0) \sqsubseteq \tilde{\mathcal{F}}(\Phi_3) \text{ and } \tilde{\mathcal{F}}(\Phi_1) \sqsubseteq \tilde{\mathcal{F}}(\Phi_3).$$

This can happen only if $\tilde{\mathcal{F}}(\Phi_3)$ has a value larger than all integers. This leads us to define the ordering

$$\perp \sqsubseteq x \sqsubseteq \hat{} \quad \text{for all } x \in \mathbb{N}$$

In this ordering the operators are continuous.

It is fairly easy to see that allowing non-deterministic operators over $\hat{\mathbb{D}}$ provides a larger class of operators than those over $\mathbb{D}$. For example the operator $\mid$ OR $\mid$ cannot be made computable over $\mathbb{D}$. Moreover, this class is larger in a stronger sense than in the type 1 case. Namely, one can prove the following theorem due to Myhill and Shephordson (see Rogers [17] pp. 281-282).

<u>Theorem</u> 3: There is a total function $f:\mathbb{N} \to \mathbb{N}$ and a partial function $\phi:\mathbb{N} \to \mathbb{N}$ such that for some $\hat{\mathcal{F}}_i$, $f = \hat{\mathcal{F}}^2_{i_0}(\phi)$, but for no $\mathcal{F}^2_i$, $f = \mathcal{F}^2_i(\phi)$.

We will not state the definition of non-deterministic effective operations over $\hat{\mathbb{D}}$ in this section, nor will we state the main theorem for the domain $\hat{\mathbb{D}}$ since both are given in §8,9, and no new ideas can be illustrated neatly by this special case.

5. TYPE THREE

5.1 comparison with other notions of computability

In the cases of type 1 and type 2 computability, our concept of a continuous partial computable function on $\mathbb{D}$ agrees with the usual notions of computability on $\mathbb{N}$ as much as possible. That is, in the case of type 1 objects, the $\bar{\mathcal{F}}_i$ and $\hat{\mathcal{F}}_i$ are defined by the same algorithms which compute ψ_j and $\hat{\psi}_j$.

In the case of type 2 objects on $\mathbb{N}$, the inputs are partial functions from $\mathbb{N}$ to $\mathbb{N}$, denoted $\mathcal{P}(\mathbb{N};\mathbb{N})$. Given a $\phi \in \mathcal{P}(\mathbb{N};\mathbb{N})$, it has a canonical embedding in $\mathbb{D}^1$; namely it is mapped to ϕ^+ defined by

$$\phi^+(x) := \begin{cases} \bot & x = \bot \\ \phi(x) & x \in \mathbb{N} \end{cases}$$

These ϕ^+ objects are continuous. Furthermore, the computable operators themselves, $F\colon \mathcal{P}(\mathbb{N};\mathbb{N}) \to \mathcal{P}(\mathbb{N};\mathbb{N})$, are continuous on the domain $\mathcal{P}(\mathbb{N};\mathbb{N})$ with the ordering $\subseteq$ defined as $\phi_1 \subseteq \phi_2$ iff $\phi_1^+ \sqsubseteq \phi_2^+$.

So the natural generalization of type 2 computability theory to the domain $\mathbb{D}$ calls for continuous objects operating on continuous inputs. Therefore the type 1 and 2 theory on $\mathbb{N}$ agrees with the type 1 and 2 theory on $\mathbb{D}$. This is not true for type 3 and higher types.

In the first place, the concept of a computable object of type $n \geq 3$ on arbitrary partial functions is not developed in the literature. Kleene [9,10] has considered higher type computability on total functions, and Platek [16] has considered a special case of higher type computability on partial functions. These approaches allow discontinuous inputs. For example, consider the object f^2 defined by

$$f^2(x^1) := \begin{cases} \lambda y[x^1(2y)] & \text{if } x^1 \text{ is partial recursive} \\ \lambda y[x^1(3y)] & \text{if } x^1 \text{ is not partial recursive.} \end{cases}$$

The object f^2 cannot be represented in $\mathbb{D}^2$ because it is not the limit of basis elements of type 2, i.e. if one examines f^2 on finite inputs, the output will always be a piece of $\lambda y[x^1(2y)]$; but in the limit the basis elements may define a non-recursive function, so the output may discontinuously change in the limit.

In general at type $n \geq 3$, the number of functions which can be input is larger than $2^{\aleph_0}$. which is the number which can be approximated by basis elements.

Thus at type 3 and beyond our definition of computability over $\mathbb{D}$ is much more restricted than the notion of general computability over $\mathbb{N}$ with function oracles for arbitrary (non-discontinuous) inputs over $\mathbb{N}$ as in Kleene [8, 9 ,10]. However, we justify this approach, via our main theorem, as an appropriate definition for a recursion theory applicable to programming language semantics.

5.2 role of continuity

Why do we consider only continuous inputs to our operators? Historically speaking the answer is that in our original approach to constructing computable Scott models we first used effective operations. We tried to extend the result to arbitrary computable operators, and found we could succeed if we limited the inputs to continuous objects. We justified this condition by the observation that in "real" programming languages the inputs are always computable (hence effective operations model the real situation) and these objects are continuous. Moreover, we saw the approach via effective operations as modeling operational semantics and the approach via computable objects as similar to Scott's work on mathematical semantics in which the notion of continuity is crucial.

At type 3 one justification for continuous inputs is that we could find no other approach that worked.

At type 2 we offered less arbitrary reasons in §4.1 for our restriction to continuous type 1 inputs, but even at type 2 discontinuous inputs (such as $\phi(\perp) = 1$ and $\phi(x) = 0$ for all $x \in \mathbb{N}$) are meaningful (and can be approximated by finite functions) and therefore our definition of computability is clearly restrictive.†

6. PRELIMINARIES FOR THE GENERAL THEORY

6.1 program

In the remainder of this paper we develop a general theory which generalizes to all finite types the ideas that we have discuss for lower types (computable operators, effective operators and their equivalence). We then cover briefly (typeless) computable operators which constitute a λ-calculus model. Their study was our original motivation for developing a theory of computable operators of all (finite) types. Once this theory is on hand, the typeless operators that we were looking for can be obtained easily.

†Another justification for continuity of inputs is that in a constructive mathematics, even if Church's thesis is not accepted, it is reasonable and interesting to assume the weaker condition that arbitrary functions on $\mathbb{N}$ are continuous because an arbitrary functio in constructive mathematics is effectively computable; hence continu

In this section we recall some basic notions that we are going to use (types, cpo's, continuous functions, etc.). A more detailed presentation can be found for instance in [3] or [13].

6.2 types

The types that we are going to use are defined by:

-- 0 is a type (it is the only basic type)

-- if τ, σ are types then $(\tau \to \sigma)$ is a type

-- there are no other types.

Among those types, we have selected the integer types to be defined inductively by $n+1 := (n \to n)$. In the literature, integers are sometimes used to denote the "functional types" defined inductively by $n+1 := (n \to 0)$ [8] . Our definition is the one which is usually used in connection with λ-calculus models. Let us emphasize that the integer types are simply a subset of all types, but logically the other types are as good as well. Integer types are introduced only because they often allow simplification of the notation. Another notational convenience for types is the convention that $\tau \to \sigma \to \rho$ denotes the type $(\tau \to (\sigma \to \rho))$.

6.3 complete partially ordered sets

Complete partially ordered sets and their use in programming language semantics (fixed point semantics) have been discussed in [12], [3]. We review only the basic definitions and properties that we need in this paper.

-- a cpo (complete partially ordered set) is a partially ordered set with the properties that:

(i) Each ascending chain has a least upper bound.

(ii) There is a least element, denoted by $\perp$ (bottom).

-- the appropriate functions between cpo's D and D' are the functions that respect least upper bounds of chains. They are called <u>continuous functions</u>. The set of all continuous functions from D to D' is itself a cpo under the pointwise induced partial ordering and is denoted by [D,D']. Given a cpo D^o, we can define for each type $\tau \to \sigma$ a cpo $D^{\tau \to \sigma}$ inductively by $D^{\tau \to \sigma} := [D^\tau, D^\sigma]$.

-- The product of any number of cpo's is obtained by taking the cartesian product of the underlying sets with the componentwise induced partial ordering.

-- Continuous functions with several arguments can be viewed as functions on iterated function spaces since there is a continuous natural isomorphism $[D \times D', D''] \cong [D,[D',D'']]$ for any cpo's D,D' and D". We will therefore allow writing $\alpha(x_1,\ldots,x_n)$ instead of $\alpha(x_1)(x_2)\ldots(x_n)$, but we do not introduce explicitly a "many-argument type" since it is easier for the formal treatment to have only one-argument functions to consider.

<u>6.4 two remarks</u>

(I) In the previous sections we have encountered two cpo's, namely

$$\mathbb{D}^\circ = \underset{\perp}{\mathbb{N}} \qquad \text{and} \qquad \hat{\mathbb{D}}^\circ = \underset{\perp}{\hat{\mathbb{N}}}$$

$\mathbb{D}^\circ$ was used for deterministic operators, $\hat{\mathbb{D}}^\circ$ for non-deterministic operators. For the general theory, we work abstractly with a cpo D°, thinking of it to be either $\mathbb{D}^\circ$ or $\hat{\mathbb{D}}^\circ$. All restrictions that we impose on D° are properties that $\mathbb{D}^\circ$ and $\hat{\mathbb{D}}^\circ$ have in common, so that our theory applies to both deterministic and non-deterministic operators, depending on the actual choice of D°.

(II) All our notions of computable objects will be based on an indexing of r.e. sets, $\{W_j\}$. We do not specify how this indexing is obtained. However, we require that a canonical procedure for enumerating each W_j be given.

7. COMPUTABLE OPERATORS OF TYPE τ

In this section we introduce the notion of computable operators of arbitrary type τ (with continuous inputs). We first generalize the idea of an r.e. basis (as discussed for low types) to arbitrary type τ. A computable operator of type τ will then be the limit of an r.e. chain of basis elements of type τ.

7.1 bases

The six properties (axioms) that we require below for an r.e. basis B of a cpo D are not at all independent. But we want to present them in a form in which they are actually needed to guarantee that (1) B is r.e. and generates D, (2) we can compute with elements represented by r.e. sequences of basis elements and (3) the continuous function space of two cpo's with r.e. bases has itself an r.e. basis.

Given a cpo D we define

-- A subset $X \subseteq D$ is called compatible (what we actually mean is that the elements of X are compatible) if X has an upper bound in D.

-- A subset $B \subseteq D$ is called an r.e. basis of D if

1) B is an r.e. subset of D $(\beta_0 = \bot, \beta_1, \beta_2, \ldots)$

2) Each element $x \in D$ is the lub of a chain of basis elements

3) For all chains $\{x_i\} \subseteq D$ and all $j \in \mathbb{N}$:

$\bigsqcup x_i \sqsupseteq \beta_j \Longrightarrow$ there is an i s.t. $x_i \sqsupseteq \beta_j$

4) It is decidable whether for $j,k \in \mathbb{N}$:

(a) $\beta_j \sqsubseteq \beta_k$

(b) $\{\beta_j, \beta_k\}$ is compatible

5) For all $b_1, \ldots, b_n \in B$:

$\{b_1, \ldots, b_n\}$ is compatible $\Longleftrightarrow$ b_i's are pairwise compatible

6) If $\{\beta_i, \beta_j\}$ is compatible then we can find a $k \in \mathbb{N}$ s.t. $\beta_k = \bigsqcup\{\beta_i, \beta_j\}$.†

If we list all elements of $\mathbb{D}^\circ$ (or $\hat{\mathbb{D}}^\circ$ respectively), we obviously obtain an r.e. basis for $\mathbb{D}^\circ$ (or $\hat{\mathbb{D}}^\circ$ respectively). What we want to show is that $\mathbb{D}^\tau$ ($\hat{\mathbb{D}}^\tau$resp.) has an r.e. basis for all types τ. This follows immediately from the theorem below. Remember that $D^{\tau \to \sigma}$ is defined to be $[D^\tau, D^\sigma]$, i.e. the space of all <u>continuous</u> functions from D^τ to D^σ.

<u>Theorem</u> 1: Let D and D' be cpo's with r.e. bases B and B'. Then we can construct an r.e. basis $\bar{B}$ for $\bar{D} := [D,D']$.

<u>Proof</u>: We think of basis elements as generalized finite functions. They can be described by a finite collection of functions which "only have a value on one argument", let us say on $b \in B$ we want the value $b' \in B'$. However, in order to make such a simple function continuous we have to require at least that for all $x \sqsupseteq b$ we get also the value b'.

Formally now, we define

$(b,b') := \lambda x \in D.$ <u>If</u> $x \sqsupseteq b$ <u>then</u> b' <u>else</u> $\perp'$.

and prove some properties about this definition.

† In our general treatment here we use B to denote a bssis and β_i to denote basis elements. This should not be confused with our earlier notation where we used β_i^τ and $\hat{\beta}_i^\tau$ to denote basis elements of $\mathbb{D}^\tau$ and $\hat{\mathbb{D}}^\tau$ respectively.

Lemma 1: For all $b \in B$, $b' \in B'$: (b,b') is an element of $\bar{D}$.

Proof: Axiom 3) for B.

Lemma 2: Each monotonic function $m: B \to D'$ extends uniquely to a continuous function $\mathfrak{F}: D \to D'$.

Proof: If $x = \bigsqcup b_i$, then $\mathfrak{F}(x)$ must necessarily be equal to $\bigsqcup_i m(b_i)$.

We have to show:

(i) $\mathfrak{F}$ is well defined

(ii) $\mathfrak{F}$ is continuous.

Let $x_i = \bigsqcup_\ell b_i^\ell$ and $x = \bigsqcup_k b^k$ such that $\{x_i\}$ is a chain and x is its l.u.b. For all i,ℓ we have $b_i^\ell \sqsubseteq x_i \sqsubseteq x = \bigsqcup b^k$. Axiom 3 then implies that there is a k such that $b_i^\ell \sqsubseteq b^k$. Therefore $\bigsqcup_{i,\ell} m(b_i^\ell) \sqsubseteq \bigsqcup_k m(b^k)$. On the other hand, $b^k \sqsubseteq x = \bigsqcup x_i$ for all k implies that there is an i such that $b^k \sqsubseteq x_i = \bigsqcup_\ell b_i^\ell$ (Axiom 3). Using axiom 3 once more tells us that there is an ℓ such that $b^k \sqsubseteq b_i^\ell$. Therefore $\bigsqcup_k m(b^k) = \bigsqcup_{i,\ell} m(b_i^\ell)$ and (i) and (ii) follow.

Let $\bar{b}_i := (b_i, b_i')$ for $i = 1,\ldots,n$.

Lemma 3: The following statements are equivalent:

(a) $\{\bar{b}_1,\ldots,\bar{b}_n\}$ is compatible.

(b) $\bar{b}_i$'s are pairwise compatible

(c) $\bigsqcup\{\bar{b}_1,\ldots,\bar{b}_n\}$ exists.

Proof: (a) $\Rightarrow$ (b) and (c) $\Rightarrow$ (a) are trivial. We show : (b) $\Rightarrow$ (c). Assume that the $\bar{b}_i$'s are pairwise compatible. Then for all $b \in B$, the $\bar{b}_i(b)$'s are pairwise compatible, hence compatible (axiom 5) and therefore their lub $\{\bar{b}_i(b)\}$ exists (using axiom 6 repeatedly). The function $m: B \to D'$ defined by $m(b) := \bigsqcup\{\bar{b}_i(b)\}$ is monotonic and therefore extends uniquely to a function $\mathfrak{F}: D \to D'$. $\mathfrak{F}$ is the lub of $\{\bar{b}_1,\ldots,\bar{b}_n\}$.

Now consider $\bar{b}_i = (b_i, b_i')$ for $i = 1,2$.

<u>Lemma 4</u>: $\{\bar{b}_1, \bar{b}_2\}$ is compatible iff($\{b_1, b_2\}$ is not compatible or $\{b_1', b_2'\}$ is compatible).

<u>Proof</u>:

$\Longrightarrow$: Suppose $\{\bar{b}_1, \bar{b}_2\}$ compatible and $\{b_1, b_2\}$compatible. Then $(\bar{b}_1 \sqcup \bar{b}_2)(b_1 \sqcup b_2) \sqsupseteq b_1', b_2'$, i.e. $\{b_1', b_2'\}$ is compatible.

$\Longleftarrow$: (i) Suppose $\{b_1, b_2\}$ not compatible. Then the function $\lambda x.$ <u>If</u> $x \sqsupseteq b_1$ <u>then</u> b_1' <u>else</u> (<u>If</u> $x \sqsupseteq b_2$ <u>then</u> b_2' <u>else</u> $\bot'$) is continuous and $\sqsupseteq \bar{b}_1, \bar{b}_2$.

(ii) Suppose $\{b_1', b_2'\}$ is compatible. Then $(\lambda x.\ b_1' \sqcup b_2') \sqsupseteq \bar{b}_1, \bar{b}_2$.

With these preliminaries it is easy to define a basis $\bar{B}$ for $\bar{D}$. We simply enumerate all $\sqcup\{(\beta_{i_1}, \beta_{j_1}'), \ldots, (\beta_{i_n}, \beta_{j_n}')\}$ that exist (whether these lub's exist or not is decidable by lemmas 3 and 4). We have then to check the axioms for an r.e. basis. From the construction and the lemmas, the axioms 1), 4b), 5) and 6) are immediately verified. We only give the verifications for the remaining 3 axioms.

<u>axiom 2)</u> Given $\Phi \in \bar{D}$. By continuity, $\Phi(\beta_k) = \sqcup \beta_{k_i}'$ shows that Φ is the least upper bound of the set $\{(\beta_k, \beta_{k_i}') \mid k,i\} \subseteq \bar{B}$. Since $\bar{B}$ contains all lub's of finite sets of basis elements as well, we can construct a chain $\{\bar{b}_\ell\}$ which approximates Φ.

<u>axiom 3)</u> Let $\bar{b}_k = (b_k, b_k')$ for $i = 1,\ldots,n$. And let $\{\bar{x}_i\}$ be a chain in $\bar{D}$.

$$\sqcup \bar{x}_i \sqsupseteq \sqcup\{\bar{b}_1,\ldots,\bar{b}_n\}$$
$$\Longleftrightarrow \sqcup \bar{x}_i \sqsupseteq \bar{b}_k \text{ for } k = 1,\ldots,n$$
$$\Longleftrightarrow \sqcup \bar{x}_i(b_k) \sqsupseteq b_k' \text{ for } k = 1,\ldots,n \text{ (definition of } \bar{b}_k)$$
$$\Longleftrightarrow \exists i \quad \bar{x}_i(b_k) \sqsupseteq b_k' \text{ for } k = 1,\ldots,n \text{ (axiom 3 for D')}$$
$$\Longleftrightarrow \exists i \quad \bar{x}_i \sqsupseteq \sqcup\{\bar{b}_1,\ldots,\bar{b}_n\}.$$

<u>axiom 4a)</u> Let again $\bar{b}_i = (b_i,b_i')$ for $i = 1,\ldots,n+m$.

$$\sqcup\{\bar{b}_1,\ldots,\bar{b}_n\} \sqsubseteq \sqcup\{\bar{b}_{n+1},\ldots,\bar{b}_{n+m}\}$$
$$\Longleftrightarrow \bar{b}_i \sqsubseteq \{\bar{b}_{n+1},\ldots,\bar{b}_{n+m}\} \text{ for } i = 1,\ldots,n$$
$$\Longleftrightarrow b_i' \sqsubseteq \sqcup\{\bar{b}_{n+1},\ldots,\bar{b}_{n+m}\}(b_i) \text{ for } i = 1,\ldots,n.$$
$$\Longleftrightarrow b_i' \sqsubseteq \sqcup\{\bar{b}_{n+1}(b_i),\ldots,\bar{b}_{n+m}(b_i)\} \text{ for } i = 1,\ldots,n.$$

The last statement is decidable since we have to compare only basis elements of B'.

<u>7.2 definition of the computable operators</u>

Starting with a cpo D° which has an r.e. basis B°, our theorem guarantees that each D^τ has also an r.e. basis B^τ. The computable elements of D^τ are going to be those which are represented by an r.e. set of basis elements, i.e. those which are the lub of an r.e. set of basis elements. If we think of D^τ as being $\hat{\mathbb{D}}^\tau$ then each such set has a lub. However, if D^τ is $\mathbb{D}^\tau$, this is not true. So for the general theory we have to single-value the W_j's with respect to the enumeration of the basis B^τ. The single-valuing procedure gives us for each W_j an r.e. set $W_{s_\tau(j)}$ which we will write as U_j^τ . It is defined by:

-- Enumerate W_j: $x_1,x_2,\ldots$

-- U_j^τ : $y_1,y_2,\ldots$ is obtained by

(a) $y_1 = x_1$

(b) $$y_{n+1} = \begin{cases} x_{n+1} & \text{if } \{\beta^\tau_{y_1},\ldots,\beta^\tau_{y_n},\beta^\tau_{x_{n+1}}\} \text{ is compatible} \\ y_n & \text{otherwise} \end{cases}$$

U_j^τ is clearly r.e. and $\{\beta_k^\tau \mid k \in U_j^\tau\}$ is compatible (because each finite subset is compatible). If $\{\beta_k \mid k \in W_j\}$ happens to be compatible then $U_j^\tau = W_j$.

We now can define the operator with gödel number j of type τ, Φ_j^τ, by:

$$\Phi_j^\tau := \bigsqcup\{\beta_k^\tau \mid k \in U_j^\tau\}$$

We call those the <u>computable</u> <u>operators of</u> <u>type</u> <u>τ</u>, $CD^\tau :=$ $\{\Phi_j^\tau \mid j \in \mathbb{N}\}$. For a concrete choice of D^τ ($\mathbb{D}^\tau$ or $\hat{\mathbb{S}}^\tau$), we obtain respectively the deterministic or the non-deterministic computable operators of type τ.

We have discussed these objects on lower types. Let us now indicate how we can compute with those objects of arbitrary type.

We first have to decide on how we want to represent an input of type τ. If it happens to be a Φ_i^τ then we know that the index i defines the single-valued r.e. set U_i^τ which defines Φ_i^τ by $\Phi_i^\tau :=$ $\bigsqcup\{\beta_k^\tau \mid k \in U_i^\tau\}$. But now remember that the basis B^τ generates D^τ, thus all elements of D^τ are of the form $\Sigma^\tau = \bigsqcup\{\beta_k^\tau \mid k \in S\}$, where $S \subseteq \mathbb{N}$ is some single-valued subset of $\mathbb{N}$. Equally, for each j, the operator $\Omega^\sigma := \Phi_j^{\tau \to \sigma}(\Sigma^\tau)$ is of the form $\Omega^\sigma = \bigsqcup\{\beta_k^\sigma \mid k \in O\}$ for some $O \subseteq \mathbb{N}$. The question now is: how do we "compute O from S"? We use the following model (which is used also by Rogers to define enumeration operators). We think of S as being presented element by element. Then from each finite piece of S we obtain a finite piece of the "output set" O. So suppose that at time n we know $\{s_1,\ldots,s_m\} \subseteq S$. We then add to the output set an index ℓ with the property $\beta_\ell^\sigma = (\bigsqcup\{\beta_{k_1}^{\tau \to \sigma},\ldots, \beta_{k_n}^{\tau \to \sigma}\})(\bigsqcup\{\beta_{s_1}^\tau,\ldots,\beta_{s_m}^\tau\})$, where k_r is the r-th element in the enumeration of $U_i^{\tau \to \sigma}$.

Remark: The output set O obviously depends on the order in which S is presented. However, the operator described by O depends only on S.

It is now easy to see that if S is an r.e. set, then we can find effectively from its index i an index k of the output set (which is thus r.e. also). This can be done for all types. In addition, for type 0 we can do more than give the index of an element. We can partially compute the element of D° that is denoted by an index. We have discussed this in detail earlier in this paper.

8. EFFECTIVE OPERATORS OF TYPE τ

8.1 generalized partial recursive functions

The idea of effective operators is to let higher type objects be induced by partial functions (deterministic or non-deterministic) on the indices (programs) of the arguments. Consequently, an effective operator is not defined on all continuous functions of appropriate type as this was the case for computable operators. However, we will show in the next section that the two notions of operators of arbitrary type are equivalent. In particular, this implies that each effective operator has a unique continuous extension to arbitrary continuous inputs. Again, we develop the theory abstractly so that it applies to both the deterministic and the non-deterministic case, depending on the actual choice of D°. Formally, we require for D° that

1) D° is a cpo

2) $\mathbb{N}$ is embedded into D°, i.e.

$$\mathbb{N} \hookrightarrow D^\circ$$
$$k \mapsto \underline{k}$$

3) It is decidable whether $\{\underline{k},\underline{\ell}\}$ is compatible

4) $\underline{k},\underline{\ell}$ compatible implies that $\underline{k}\cup\underline{\ell}$ exists.

We first define n-argument partial recursive functions with values in D°. This allows us to treat deterministic and non-deterministic partial recursive functions simultaneously (it all depends on the actual choice of D°). We define these functions using the indexing of r.e. sets $\{W_j\}$ that we used before. In addition we assume that a pairing function p^2 on the integers is given. We write $\langle x_1,x_2\rangle$ for $p^2(x_1,x_2)$ and define inductively $\langle x_1,\ldots,x_n,x_{n+1}\rangle := \langle\langle x_1,\ldots,x_n\rangle, x_{n+1}\rangle$. For computable operators of type τ we had to single-value the W_j's with regard to the basis of type τ; here we have to single-value the W_j's with respect to D° and (n+1)-tuples given by the pairing function, just as we did in §3.1 for n=1 to define ψ_j. The r.e. set obtained from single-valuing W_j in this way is denoted by V_j^n. It is defined by:

V_j^n: $y_1,y_2,\ldots$ is obtained from

W_j: $x_1,x_2,\ldots$ by:

(a) $y_1 = x_1$

(b)
$$y_{k+1} = \begin{cases} y_k \text{ if } x_{k+1} = \langle p_1,\ldots,p_n,q\rangle \text{ and} \\ \exists \ell \leq k: (y_\ell = \langle p_1,\ldots,p_n,q'\rangle \text{ and } \{\underline{q},\underline{q}'\} \\ \text{not compatible.} \\ x_{k+1} \text{ otherwise.} \end{cases}$$

This single-valuing has the effect that for all $x_1,\ldots,x_n \in \mathbb{N}$, $\sqcup\{\underline{k} \mid \langle x_1,\ldots,x_n,k\rangle \in V_j^n\}$ exists. This allows the definition of

$$\psi_j^n : \underbrace{\mathbb{N}\times \ldots \times\mathbb{N}}_{n \text{ times}} \to D^\circ$$

to be

$$\psi_j^n(x_1,\ldots,x_n) := \sqcup\{\underline{k} \mid \langle x_1,\ldots,x_n,k\rangle \in V_j^n\}.$$

That these generalized partial recursive functions have the s-m-n property can be seen immediately. All we really need are the S_n^1-functions. Remember that these are recursive functions with the property that for all j, $x_0,\ldots,x_n$

$$\psi_j^{n+1}(x_0,\ldots,x_n) = \psi^n_{S_n^1(j,x_0)}(x_1,\ldots,x_n).$$

8.2 definition of the effective operators

For the inductive definition of the effective operators of type τ we have to know what kind of arguments this type expects. Notice that each type τ has a unique decomposition

$$\tau = \tau_1 \to \tau_2 \to \ldots \to \tau_p \to 0.$$

We call p the length of τ, $|\tau| := p$; it is the number of arguments that τ expects. In the following, τ_i for $i \le |\tau|$ will refer always to the decomposition of τ.

We can now define $\underline{E^\tau \subset \mathbb{N}}$, the indices of effective operators of type τ, and $\underline{e \equiv_\tau e'}$, the equivalence relation between indices of effective operators of type τ, inductively by:

-- (i) $E^\circ = \mathbb{N}$; (ii) $e \equiv_\circ e' \leftrightarrow \psi_e^\circ = \psi_{e'}^\circ$.

-- (i) $e \in E^{\tau \to \sigma} \subseteq \mathbb{N}$ iff

(a) $\forall x \in E^\tau$: $S^1_{|\sigma|}(e,x) \in E^\sigma$, and

(b) $\forall x,x' \in E^\tau$: $x \equiv_\tau x' \Longrightarrow S^1_{|\sigma|}(e,x) \equiv_\sigma S^1_{|\sigma|}(e,x')$

(ii) For $e,e' \in E^{\tau \to \sigma}$:

$$e \equiv_{\tau \to \sigma} e' \iff \forall\, x \in E^{\tau}:\ S^1_{|\sigma|}(e,x) \equiv_{\sigma} S^1_{|\sigma|}(e',x)$$

The <u>effective</u> <u>operators</u> <u>of</u> <u>type</u> <u>τ</u>, <u>ED^{τ}</u>, are obtained as follows:

-- $ED^{\circ} = \{\Psi^{\circ}_e \in D^{\circ} \mid e \in E^{\circ}\}$ where $\Psi^{\circ}_e = \psi^{\circ}_e$

-- $ED^{\tau \to \sigma} = \{\Psi^{\tau \to \sigma}_e : ED^{\tau} \to ED^{\sigma} \mid e \in E^{\tau \to \sigma}\}$

where $\Psi^{\tau \to \sigma}_e(\Psi^{\tau}_x) := \Psi^{\sigma}_{S^1_{|\sigma|}(e,x)}$

Remark: (a) For $e \in E^{\tau}$, $x_i \in E^{\tau_i}$ $(\tau = \tau_1 \to \dots \to \tau_n \to 0)$:

$$\Psi^{\tau}_e(\Psi^{\tau_1}_{x_1}) \dots (\Psi^{\tau_n}_{x_n}) = \psi^n_e(x_1,\dots,x_n).$$

(b) For $e,e' \in E^{\tau}$:

$$e \equiv_{\tau} e' \iff \forall\, x_i \in E^{\tau_i}:\ \psi^n_e(x_1,\dots,x_n) = \psi^n_{e'}(x_1,\dots,x_n)$$

The effective operators provide a notion of higher type computabili[illegible] which is probably the one that we expect when we think of a programming language. If a function arises as an input for an operator, it is given by some kind of a program, or equivalently by a gödel number. If we compare this notion of computability with the computable operators introduced earlier, there are two immediate observations. On one hand, effective operators seem to be more restricted because they can take as arguments only other effective operators. But on the other hand, computable operators seem to be more restricted because they are required to be continuous. That the two notions of computability are in fact equivalent will be shown in the next section.

9. EQUIVALENCE BETWEEN COMPUTABLE AND EFFECTIVE OPERATORS

9.1 summary of notations

This short summary is intended to ease the reference to the notations that we have already introduced. The set of computable operators of type τ is defined to be $CD^{\tau} = \{ \Phi_j^{\tau} \mid j \in \mathbb{N}\}$ with extensional equality relation (with respect to arbitrary continuous arguments). Φ_j^{τ} is obtained by $\Phi_j^{\tau} := \bigsqcup\{\beta_k^{\tau} \mid k \in U_j^{\tau}\}$. The β_k^{τ}'s are basis elements of type τ and U_j^{τ} is an r.e. set obtained from W_j by "single-valuing". This has the effect that the lub in the definition of Φ_j^{τ} exists. The set of effective operators of type τ $ED^{\tau} = \{\Psi_e^{\tau} \mid e \in E^{\tau}\}$ has been defined inductively. Each τ has a unique decomposition $\tau = \tau_1 \to \dots \to \tau_n \to 0$. If $x_i \in E^{\tau_i}$ (i.e. x_i is an index of an effective operator of type τ_i) and $e \in E^{\tau}$, then

$$\Psi_e^{\tau}(\Psi_{x_1}^{\tau_1})\dots(\Psi_{x_n}^{\tau_n}) = \psi_e^n(x_1,\dots,x_n).$$

ψ_e^n is a (generalized) partial recursive function with values in D°. It is defined by $\psi_e^n(y_1,\dots,y_n) = \bigsqcup\{\underline{k} \mid \langle y_1,\dots,y_n,k\rangle \in V_j^n\}$. V_j^n is the r.e. set obtained from W_j by single-valuing for the purpose of making the lub exist.

9.2 the main theorem

We are now ready to show that the two notions of higher type computability are equivalent for both deterministic and non-deterministic operators respectively. For D° we clearly require all properties that we required when we introduced computable operators on one hand (basis) and effective operators on the other hand (embedding of $\mathbb{N}$ into D°). When we use common properties of D° and $\hat{D}^\circ$ that we have not formalized for D°, we will say so.

The theorem below states that the systems $\{CD^\tau\}$ and $\{ED^\tau\}$ are isomorphic (with respect to application).

Main Theorem

Assume that recursive functions $E^\circ = \mathbb{N} \underset{g_\circ}{\overset{h_\circ}{\Longleftrightarrow}} \mathbb{N}$

are given with the property that $\Psi^\circ_e = \Phi^\circ_{g_\circ(e)}$ and $\Phi^\circ_j = \Psi^\circ_{h_\circ(j)}$.

Then there are for each type τ recursive functions g_τ and h_τ :

$$\begin{tikzcd} \mathbb{N} \arrow[r, "g_\tau"] & \mathbb{N} \arrow[dl, "h_\tau"] \\ E^\tau \arrow[u, phantom, "\cup"] & \end{tikzcd}$$

with the properties

(I) For all $e \in E^{\tau\to\sigma}$, $x \in E^\tau$, $y \in E^\sigma$:

$$\Psi^{\tau\to\sigma}_e(\Psi^\tau_x) = \Psi^\sigma_y \Longrightarrow \Phi^{\tau\to\sigma}_{g_{\tau\to\sigma}(e)}(\Phi^\tau_{g_\tau(x)}) = \Phi^\sigma_{g_\sigma(y)}$$

(II) For all $j,i,k \in \mathbb{N}$:

$$\Phi^{\tau\to\sigma}_j(\Phi^\tau_i) = \Phi^\sigma_k \Longrightarrow \Psi^{\tau\to\sigma}_{h_{\tau\to\sigma}(j)}(\Psi^\tau_{h_\tau(i)}) = \Psi^\sigma_{h_\sigma(k)}$$

(III) g_τ and h_τ induce an equivalence between ED^τ and CD^τ.

Remark: The functions $g_\circ$ and $h_\circ$ (which must induce the identity $CD^\circ = ED^\circ$) can be given easily for both deterministic and non-deterministic operators, so the theorem can be applied in either case.

Proof: The proof is by induction on the structure of the types. We assume that the theorem is true for τ,σ,σ_i, where $\sigma = \sigma_1 \to \dots \to \sigma_n \to 0$, and show that it is true for type $\tau \to \sigma$

The easy part is to show that each computable operator gives rise to an effective operator. We do this first:

(a) definition of $h_{\tau\to\sigma}$

Set $h_{\tau\to\sigma}(j)$:= index of ψ^{n+1} which satisfies

$$\psi^{n+1}(x,x_1,\ldots,x_n) := \Phi_j^{\tau\to\sigma}(\Phi^{\tau}_{g_\tau}(x),\Phi^{\sigma_1}_{g_{\sigma_1}}(x_1),\ldots,\Phi^{\sigma_n}_{g_{\sigma_n}}(x_n)).$$

and show:

-- $h_{\tau\to\sigma}(j) \in E^{\tau\to\sigma}$

-- (II) of the theorem.

These are straightforward verifications.

(b) definition of $g_{\tau\to\sigma}$

For $e \in E^{\tau\to\sigma}$, $\Phi^{\tau\to\sigma}_{g_{\tau\to\sigma}(e)}$ should be equivalent to $\Psi_e^{\tau\to\sigma}$. If we apply $\Psi_e^{\tau\to\sigma}$ to effective operators corresponding to basis elements of type τ, we can determine exactly what $\Phi^{\tau\to\sigma}_{g_{\tau\to\sigma}(e)}$ should be. What is not obvious however is that $\Psi_e^{\tau\to\sigma}$ induces a continuous element. We first prove two lemmas which express exactly this fact.

Let $c(k)$ be a canonical index for $\{k\}$, i.e. $W_{c(k)} = \{k\}$, therefore $\Phi^{\tau}_{c(k)} = \beta^{\tau}_k$.

Lemma 1: (monotonicity) For all $k,j \in \mathbb{N}$, $e \in E^{\tau\to\sigma}$, $x_i \in E^{\sigma_i}$

$$\beta_k^\tau \subseteq \Phi_j^\tau \Longrightarrow \psi_e^{n+1}(h_\tau(c(k)), x_1,\ldots,x_n) \subseteq \psi_e^{n+1}(h_\tau(j),x_1,\ldots,x_n)$$

Lemma 2: (compactness) For all $j \in \mathbb{N}$, $e \in E^{\tau\to\sigma}$, $x_i \in E^{\sigma_i}$ there exists a $k \in \mathbb{N}$ such that $\beta_k^\tau \subseteq \Phi_j^\tau$ and

$$\psi_e^{n+1}(h_\tau(c(k)),x_1,\ldots,x_n) = \psi_e^{n+1}(h_\tau(j),x_1,\ldots,x_n).$$

Proof of lemmas: We show that the negation of each of the lemmas allows us to solve the halting problem ($\psi_z(z)\downarrow$). Therefore the proofs are not constructively valid.

Proof of Lemma 1: Assume that there exist $k, j \in \mathbb{N}$, $e \in E^{\tau \to \sigma}$ and $x_i \in E^{\sigma_i}$ such that $\beta_k^\tau \sqsubseteq \Phi_j^\tau$ and

$\psi_e^{n+1}(h_\tau(c(k)), x_1, \ldots, x_n) \not\sqsubseteq \psi_e^{n+1}(h_\tau(j), x_1, \ldots, x_n)$.

We define an r.e. set $W_{v(z)}$ for each $z \in \mathbb{N}$ by:

$W_{v(z)}$: output k.

<u>If</u> $\phi_z(z)\downarrow$ <u>then</u> output U_j^τ.

<u>Properties</u>:

-- $U_{v(z)}^\tau = W_{v(z)}$

-- $\Phi_{v(z)}^\tau = \Phi_j^\tau \Longleftrightarrow \phi_z(z)\downarrow$

-- $\Phi_{v(z)}^\tau = \beta_k^\tau \Longleftrightarrow \phi_z(z)\uparrow$

For deciding whether $\phi_z(z)\downarrow$ we can ask simultaneously:

(a) $\phi_z(z)\downarrow$?

(b)* $\psi_e^{n+1}(h_\tau(v(z)), x_1, \ldots, x_n) \not\sqsubseteq \psi_e^{n+1}(h_\tau(j), x_1, \ldots, x_n)$?

*Remark: If we use $\mathbb{D}^\circ$ or $\hat{\mathbb{D}}^\circ$ for D° then (b) is semidecidable, so we have what we want. For an abstract theory, a sufficient condition for D° to ensure semidecidability of (b) would be for instance the requirement $B^\circ = D^\circ$.

Proof of Lemma 2: Assume that there exist $j \in \mathbb{N}$, $e \in E^{\tau \to \sigma}$, $x_i \in E^{\sigma_i}$ such that for all $k \in \mathbb{N}$:

$\beta_k^\tau \sqsubseteq \Phi_j^\tau \Longrightarrow \psi_e^{n+1}(h_\tau(c(k)), x_1, \ldots, x_n) \sqsubset \psi_e^{n+1}(h_\tau(j), x_1, \ldots, x_n)$.

This means that no "finite segment" of Φ_j^τ is sufficient for ψ_e^{n+1}, therefore U_j^τ must be infinite; U_j^τ: $k_1, k_2, k_3, \ldots$

For each $z \in \mathbb{N}$ we define an r.e. set $W_{v(z)}$:

For $n \in \mathbb{N}$ do

<u>If</u> $\phi_z(z)\not\downarrow^{\leq n}$ <u>then</u> output k_n.

Remark: $\phi_z(z)\not\downarrow^{\leq n}$ means: $\phi_z(z)$ does not converge in at most n steps and $\phi_z(z)\downarrow^{=n+1}$ means $\phi_z(z)$ converges in exactly n+1 steps.

Properties:

-- $U^{\tau}_{v(z)} = W_{v(z)}$

-- $\Phi^{\tau}_{v(z)} = \Phi^{\tau}_{j} \Leftrightarrow \phi_z(z)\downarrow$

-- $\Phi^{\tau}_{v(z)} = \sqcup\{\beta^{\tau}_{k_1},\ldots,\beta^{\tau}_{k_n}\} \sqsubset \Phi^{\tau}_{j} \Longleftrightarrow \phi_z(z)\downarrow^{=n+1}$

For deciding whether $\phi_z(z)\downarrow$ we ask simultaneously :

(a) $\phi_z(z)\downarrow$?

(b)* $\psi^{n+1}_e(h_\tau(v(z)),x_1,\ldots,x_n) \sqsupseteq \psi^{n+1}_e(h_\tau(j),x_1,\ldots,x_n)$?

*Remark: As for Lemma 1, question (b) is semidecidable for both $\mathbb{D}^{\bullet}$ and $\hat{\mathbb{D}}^{\bullet}$.

We now define $g_{\tau\to\sigma}(e)$ in the way we have already outlined briefly. For $\Phi^{\tau\to\sigma}_{g_{\tau\to\sigma}(e)}$, we enumerate all basis elements that we obtain from considering $\Psi^{\tau\to\sigma}_e$ applied to effective operators corresponding to basis elements of type τ. The lemmas that we have just proved will then be used to show that $g_{\tau\to\sigma}(e)$ has the desired properties.

Formally now, we define $W_{g_{\tau\to\sigma}(q)}$ for all $q \in \mathbb{N}$ by:

$W_{g_{\tau\to\sigma}(q)}$: enumerate all $i \in \mathbb{N}$ such that

$\beta^{\tau\to\sigma}_i = (\beta^{\tau}_k,\beta^{\sigma}_r)$ with $r \in U^{\sigma}_{p(k)}$ where $p(k) = g_\sigma(s^1_n(q,h_\tau(c(k))))$

Proposition: For all $e \in E^{\tau\to\sigma}$:

$$U^{\tau\to\sigma}_{g_{\tau\to\sigma}(e)} = W_{g_{\tau\to\sigma}(e)}$$

Remark: This proposition tells us that all basis elements that we have enumerated are indeed available for $\Phi^{\tau\to\sigma}_{g_{\tau\to\sigma}(e)}$.

Proof: We show that $\{\beta^{\tau\to\sigma}_{i_1},\beta^{\tau\to\sigma}_{i_2}\}$ is compatible for all $i_1,i_2 \in W_{g_{\tau\to\sigma}(e)}$.

Let $\beta_{i_j}^{\tau\to\sigma} = (\beta_{k_j}^{\tau}, \beta_{r_j}^{\sigma})$ and assume that $\{\beta_{k_1}^{\tau}, \beta_{k_2}^{\tau}\}$ is compatible. Let k be such that $\beta_k^{\tau} = \sqcup\{\beta_{k_1}^{\tau}, \beta_{k_2}^{\tau}\}$. We have to show that $\{\beta_{r_1}^{\sigma}, \beta_{r_2}^{\sigma}\}$ is compatible (§7, Lemma 4).

Let $p(k_j) = g_\sigma(s_n^1(e, h_\tau(c(k_j))))$. Since $i_j \in W_{g_{\tau\to\sigma}}(e)$, $\beta_{r_j}^{\sigma} \subseteq \Phi_{p(k_j)}^{\sigma}$. Using Lemma 1 we can show that $\Phi_{p(k_j)}^{\sigma} \sqsubseteq \Phi_{p(k)}^{\sigma}$ since $\beta_{k_j}^{\tau} \sqsubseteq \beta_k^{\tau}$. Therefore $\beta_{r_j}^{\sigma} \sqsubseteq \Phi_{p(k)}^{\sigma}$ for $j = 1,2$, i.e. $\{\beta_{r_1}^{\sigma}, \beta_{r_2}^{\sigma}\}$ is compatible.

So now we know that for $e \in E^{\tau\to\sigma}$,

$$\Phi_{g_{\tau\to\sigma}(e)}^{\tau\to\sigma} = \sqcup\{\beta_k^{\tau} \mid k \in W_{g_{\tau\to\sigma}}(e)\}.$$

We have to show that $g_{\tau\to\sigma}$ satisfies (I) of the theorem which is equivalent to saying:

* For all $e \in E^{\tau\to\sigma}$, $x \in E^{\tau}$, $x_i \in E^{\sigma_i}$

$$\Phi_{g_{\tau\to\sigma}(e)}^{\tau\to\sigma}(\Phi_{g_\tau(x)}^{\tau})(\vec{\Phi}) = \psi_e^{n+1}(x, x_1, \ldots, x_n)$$

where $\vec{\Phi} := (\Phi_{g_{\sigma_1}(x_1)}^{\sigma_1}, \ldots, \Phi_{g_{\sigma_n}(x_n)}^{\sigma_n})$.

Since $\Phi_{g_{\tau\to\sigma}(e)}^{\tau\to\sigma}(\Phi_{g_\tau(x)}^{\tau}) = \sqcup\{\Phi_{g_{\tau\to\sigma}(e)}^{\tau\to\sigma}(\beta_k^{\tau}) \mid \beta_k^{\tau} \sqsubseteq \Phi_{g_\tau(x)}^{\tau}\}$

and $\Phi_{g_{\tau\to\sigma}(e)}^{\tau\to\sigma}(\beta_k^{\tau}) = \sqcup\{\Phi_{p(\ell)}^{\sigma} \mid \beta_\ell^{\tau} \sqsubseteq \beta_k^{\tau}\}$,

we deduce

$$\begin{aligned}\Phi_{g_{\tau\to\sigma}(e)}^{\tau\to\sigma}(\Phi_{g_\tau(x)}^{\tau})(\vec{\Phi}) &= \sqcup\{\Phi_{p(\ell)}^{\sigma}(\vec{\Phi}) \mid \beta_\ell^{\tau} \sqsubseteq \Phi_{g_\tau(x)}^{\tau}\} \\ &= \sqcup\{\psi_e^{n+1}(h_\tau(c(\ell)), x_1, \ldots, x_n) \mid \beta_\ell^{\tau} \sqsubseteq \Phi_{g_\tau(x)}^{\tau}\end{aligned}$$

Our two lemmas finally tell us that this is equal to $\psi_e^{n+1}(x, x_1, \ldots, x_n)$ which concludes the proof of *.

The only thing left in the proof of the theorem is (III), i.e. that $h_{\tau \to \sigma}$ and $g_{\tau \to \sigma}$ induce an equivalence between $CD^{\tau \to \sigma}$ and $ED^{\tau \to \sigma}$. This is immediate since both sets are defined with extensional equality and furthermore, inequality of two elements of $CD^{\tau \to \sigma}$ shows up on computable arguments (even on basis elements) of type τ.

10. TYPELESS OPERATORS

We mentioned earlier that we were led to investigate computable operators of arbitrary type in the process of constructing a λ-calculus model which (1) contains only oomputable objects and (2) contains "all" computable objects of finite types. If we now use the notion of computable operators that we have defined, the construction of a computable model containing all those computable operators of finite type becomes very easy. Roughly speaking, we simply take all elements which are limits of r.e. sequences of finite type basis elements in the λ-calculus model D. To be somewhat more specific without going too much into details, let us mention that the λ-calculus model D over a cpo $D°$ -- as discussed in [3] -- is constructed essentially the same way as the model originally given by Scott [20], except that everything is done over cpo's rather than continuous lattices. Thus, the λ-calculus model D is a cpo and the finite type domains D^{τ} are (homomorphically) embedded into D by the continuous functions $\iota_{\tau}: D^{\tau} \to D$. If we define $B := \{\iota_{\tau}\beta_j^{\tau} \mid \tau \in \underline{\text{Types}}, j \in \mathbb{N}\}$ (actually it is sufficient to have only images of integer type, i.e. $\{\iota_n \beta_j^n\}$), it can be shown that B is an r.e. basis for D.

Once we know this, we can single-value the W_j with respect to this (enumeration of the) basis B. Let us denote the resulting r.e. set by U_j. The j'th typeless operator is then defined by $\mathcal{F}_j := \{\beta_k | \ k \in U_j\}$ and $CD := \{\mathcal{F}_j \ | \ j \in \mathbb{N}\}$. Obviously, if we restrict the $\mathcal{F}_j$'s to computable arguments, the extensional equality on CD remains the same. This allows us to verify that CD is indeed a λ-calculus model. That it contains all finite type computable operators is immediate from the definition. We can call it computable for several reasons. Not only can application be described by an effective function on the indices of the elements of CD, but the elements themselves can be subject to computations, for instance their 0-component in D° can be computed. More generally, the projections $\Pi_\tau : CD \to CD^\tau$ are all computable, so we can get a pretty good insight into the nature of ~~these~~ objects.

Acknowledgments

We would like to thank Helene Jacobowitz for her careful job of typing and improving the manuscript.

REFERENCES

[1] Church, A. The Calculi of Lambda-Conversion. Ann. of Math. Studies 6, Princeton, 1941.

[2] Constable, Robert L. and David Gries. On Classes of Program Schemata . SIAM J. Computing, 1, 1, March 1972, 66-118.

[3] Egli, Herbert. An Analysis of Scott's λ-Calculus Models. TR 73-191, Cornell University. Dec. 1973.

[4] Egli, Herbert. Programming Language Semantics Using Extensional λ-Calculus Models. TR 74-206, Cornell University.

[5] Gandy, R.O. Computable functionals of finite type I. Sets, Models and Recursion Theory, ed. by J.N. Crossley, North-Holland, 1967, 202-242.

[6] Harrison, J. Equivalence of the effective operations and the hereditarily recursively continuous functionals , Foundations of Classical Analysis, report of Stanford summer seminar, 1963, Appendix vc.

[7] Kleene, S.C. Introduction to Metamathematics, D. Van Nostrand, Princeton, 1952.

[8] Kleene, S.C. Countable Functionals , Constructivity in Mathematics, ed. A. Heyting, North-Holland, 1959. 81-100..

[9] Kleene, S.C. Herbrand-Gödel-style Recursive Functionals of Finite Types. Recursive Function Theory, 1962, Providence, 49-75.

[10] Kleene, S.C. Turing machine computable functionals of finite type I, Logic,Methodology and Philosophy of Science, Proc. 1960 International Congress, Stanford University Press, Stanford, California, 1962, 38-45.

[11] Kreisel, G., D. Lacombe, and J.R. Schoenfield. Partial

recursive functionals and effective operations , Constructivity in Mathematics, ed. A. Heyting, North-Holland, 1959, 290-297.

[12] Milner, Robin. Implementation and applications of Scott's logic for computable functions , Proc. ACM Conf. on Proving Assertions About Programs, Los Cruces, New Mexico, 1972, 1-6.

[13] Milner, Robin. Models of LCF , AIM-186/CS-332, Computer Science Department, Stanford University, 1973.

[14] Minsky, Marvin.Computation, Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

[15] Park, David. Fixed Point Induction and Proofs of Program Properties. Machine Intelligence 5, American Elsevier, New York, 1959, 59-78.

[16] Platek, Richard Alan. Foundations of Recursion Theory. Ph.D. thesis, Stanford University, Jan. 1966.

[17] Rogers, Hartley,Jr. Theory of Recursive Functions and Effective Computability. McGraw-Hill, New York, 1967.

[18] Scott, Dana. The Lattice of Flow Diagrams, in Lecture Notes in Mathematics, 188, Symposium on Semantics of Algorithmic Languages, ed. E. Engeler, Springer-Verlag, 1970, 311-366.

[19] Scott, Dana. Outline of a Mathematical Theory of Computation. Proc. 4th Annual Princeton Conference on Information Sciences & Systems. Princeton, 1970, 169-176.

[20] Scott, Dana. Continuous Lattices , Proc. Dalhouise Conference on Toposes, Algebraic Geometry and Logic, Lecture Notes in Mathematics, 274, Springer-Verlag, Berlin, 1972.

[21] Scott, Dana. Data Types as Lattices , lecture notes in Amsterdam, 1972.

[22] Scott, Dana and Christopher Strachey. Toward a Mathematical Semantics for Computer Languages. Proc. of the Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, New York, 1971, 19-46.

[23] Stenlund, S. Combinators, λ-terms, and Proof Theory. D. Reidel Publishing Co., Dordrecht-Holland, 1972.

[24] Strachey, C. and C. Wadsworth. Continuations, A Mathematical Semantics for Handling Full Jumps , Oxford University Computing Laboratory, Technical Monograph PRG-11, 1974.

[25] Strong, H.R. High Level Languages of Maximum Power. Conference Record 1971, 12th Annual Symposium on Switching and Automata Theory, East Lansing, Michigan, Oct. 1971, 1-4.