Computational Foundations of Basic Recursive Function Theory*

Robert L. Constable
Scott Fraser Smith

88-904
March 1988

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported in part by NSF grants CCR8502243 and DCR8303327.

Computational Foundations of Basic Recursive Function Theory*

Robert L. Constable and Scott Fraser Smith
Cornell University

Abstract

The theory of computability, or *basic recursive function theory* as it is often called, is usually motivated and developed using Church's Thesis. Here we show that there is an alternative computability theory in which some of the basic results on unsolvability become more absolute, results on completeness become simpler, and many of the central concepts become more abstract. In this approach computations are viewed as mathematical objects, and the major theorems in recursion theory may be classified according to which axioms about computation are needed to prove them.

The theory is a typed theory of functions over the natural numbers, and there are unsolvable problems in this setting independent of the existence of indexings. The unsolvability results are interpreted to show that the partial function concept, so important in computer science, serves to distinguish between classical and constructive type theories (in a different way than does the decidability concept as expressed in the law of excluded middle). The implications of these ideas for the logical foundations of computer science are discussed, particularly in the context of recent interest in using constructive type theory in programming.

1 Introduction

It is widely believed that there is one absolute notion of computability, discovered in the 30's by Church, Kleene, Turing, Gödel and Post and characterized by proofs that various models of computation (e.g., Turing machines and random access machines) give rise to the same concept, as well as by a belief in Church's Thesis, which in turn leads to a well-developed theory of unsolvability[8]. This *standard theory* accepts Church's Thesis, and in Roger's book[18] it is explicitly used to develop the theory. We want to challenge the assumption that this is the only acceptable view.

We have discovered through our attempts to provide a formal foundational theory for computer science[4,5,19] that there is an interesting, perhaps compelling, alternative to the standard theory. The goal of this paper is to explain this alternative.

One of the requirements for a theory of the kind we imagine is that it be adequate to explain all of the basic notions of computation and, where appropriate, relate them to basic notions of mathematics. So it should explain algorithms and functions, data types and sets, computations, resource expenditure, unsolvability, etc. It should also provide the rules to settle what is true about these basic concepts. We call such theories *foundational.*

In attempting to design a foundational theory of computation, we found that specific computing models and their properties are not a suitable basis. Such properties depend on specific discrete data types, such as natural numbers or strings, and it is not clear how to generalize them to other data types while preserving their essential character. The operational models of computability, say RAM's (random access machines), specify too much irrelevant and *ad hoc* detail. Some abstract approaches[10,21,22] take partial functions to be indexable, which is not justified on *a priori* grounds; others are too abstract to be of much relevance to computation. So we had to look elsewhere for the basis of a computation theory. A natural place to look is at a foundational theory for mathematics which can be interpreted computationally, such as a type theory over the natural numbers. In this setting the notion of an *indexing* is an enumeration of the class of partial functions. It is consistent to affirm or deny such indexings. The surprising result is that an interesting notion of unsolvability may be conceived even if we deny the existence of indexings.

*This work was supported in part by NSF grants CCR8502243 and DCR8303327

2 A theory of computing

2.1 Nature of the theory

Although the concept of an *algorithm* is central, we treat it as a meta-notion in this theory. Two algorithms are equal only if they have the same "structure", but we do not formalize either algorithm or this equality in the object theory here, although we consider an extension to accomplish this in section 4.4. We represent in the theory those mathematical objects that algorithms compute, *numbers* and *functions*; if f and g compute functions then *as functions* they are equal precisely if $f(a) = g(a)$ for all a in the domain of f and g. For simplicity and for comparison to the standard theories, we have three base types, **N**, the nonnegative integers $0, 1, 2, \ldots$; **1**, the type with one element; and **2**, the type with two elements. The theory is higher order in that if S and T are types, then so is $S \rightarrow T$, the type of all *computable functions* from S into T. S is the *domain type* and T the *range type* of these functions.

The types defined above are basic; in addition, associated with each type T is it's "bar type", denoted $\overline{T}$. Intuitively $\overline{T}$ represents the *computations* of elements of type T treated as equal if they yield the same result. But it is not necessary to construe bar types as computations, as will be seen in the semantics section below.

It is significant that the bar types are defined *after* the basic types. We first understand the ordinary mathematical objects, then we come to understand computations of them. This means in the case of functions, for instance, that we understand *total* functions before we understand the *partial* functions.

2.2 Possible interpretations of the theory

A theory of the kind presented here can be understood at a foundational level, and it makes sense to regard the axioms as the final arbiter of meaning. This is the approach taken in ITT[15] and Nuprl[4]. It is also possible to provide a concrete *computational semantics* for the theory based on relations on the terms, in particular by defining a relation $s \leftarrow t$ to mean that t evaluates to s, and then defining type membership and equality using this notion of computation[1]. We give such an interpretation in section 2.5 below. Although the theory is consistent with respect to such models, we do not mean to suggest that a computation theory must be based on *concrete* notions. It is also sensible to interpret this theory over an intuitive and *abstract* constructive theory of functions and types (or sets). The basic concept is that of a *mental construction.* On such an account, the notion of algorithm, computable function, and type are open-ended. This theory is consistent for such a semantics as well.

2.3 The syntax

The syntactic categories are *variables*, *terms*, *types*, *formulas* and *rules.*

If r, s, t, f are *terms* and x is a *variable*,
we may construct the terms

$0, 1, 2, \ldots$	the *numerical constants*,
$\lambda x.t$	an *abstraction*,
$s(t)$	*application*,
$s; t$	*composition*,
$succ(r)$	*successor*,
$pred(r)$	*predecessor*,
$zero(r; s; t)$	a *decision term*, and
$fix(f)$	the *fixed point term.*

Associated with terms are the *base types*

N,	the *natural numbers*,
1,	the *unit type*,
2,	the *boolean type*,

and if S and T are types, then

$S \rightarrow T$	the *function space*, and
$\overline{S}$	the *bar type*,

are also types, provided that S itself is not a bar type in the second clause.
formulas will include these atomic formulas:

$t \in T$	type membership,
$s = t \in T$	equality over type T,
$t\downarrow$	convergence, and
$t\uparrow$	divergence.

We will also allow the ordinary connectives $A \,\&\, B$, $A \vee B$, $A \Rightarrow B, \neg A$, $\forall x{:}T.A$, $\exists x{:}T.A$ (*and, or, implies, not, for all* and *there exist* respectively). We allow the possibility of further formulas.

2.4 The rules

There is a collection of rules for typing, computing, and reasoning, but for space considerations we explicitly mention here only the fixed point rule:

$$\forall f{:}\overline{T} \rightarrow \overline{T}.\, fix(f) \in \overline{T}.$$

In this paper we will use the computational semantics defined next as the justification for the proofs that follow, although these proofs may also be carried out in an axiomatic setting[3,19].

$v \leftarrow t$ is defined as:

$n \leftarrow n$	where n is 0,1,2,...
$\lambda x.b \leftarrow \lambda x.b$	
$v \leftarrow succ(a)$	iff $n \leftarrow a$ and n plus one is v
$v \leftarrow pred(a)$	iff $n \leftarrow a$ and n minus one is v
$v \leftarrow zero(a; b; c)$	iff $n \leftarrow a$ and if n is 0 then $v \leftarrow b$ else $v \leftarrow c$
$v \leftarrow a(c)$	iff $\lambda x.b \leftarrow a$ and $v \leftarrow b[c/x]$
$v \leftarrow fix(f)$	iff $v \leftarrow f(fix(f))$
$v \leftarrow a; b$	iff $a\downarrow$ and $v \leftarrow b$

Figure 1: Evaluation

2.5 The computational semantics

A precise semantics can be given for this theory by defining a reduction relation and then inductively classifying terms into types based on their values. Let $v \leftarrow t$ mean that term t reduces to term v (a *value*) by a sequence of head-reductions. This relation is operationally defined in figure 1. Note that the only possible values in this computation system are numbers and lambda terms. For example, letting F be $\lambda f.\lambda x.zero(x; 0; f(pred(x)))$, $fix(F)(1)$ reduces as follows:

$$\lambda x.zero(x; 0; fix(F)(pred(x)))(1)$$
$$zero(1; 0; fix(F)(pred(1)))$$
$$fix(F)(pred(1))$$
$$\lambda x.zero(x; 0; fix(F)(pred(x)))(pred(1))$$
$$zero(pred(1); 0; fix(F)(pred(pred(1))))$$
$$zero(0; 0; fix(F)(pred(pred(1))))$$
$$0$$

Therefore, $0 \leftarrow fix(F)(1)$.

Define $t\downarrow$ as $\exists s.(s \leftarrow t)$ and $t\uparrow$ as $\neg(t\downarrow)$.

Definition 1 *Define $s = t \in T$ for s and t closed terms by induction on types as follows:*

$s = t \in T$ *iff*

if T is $\mathbf{1}$*, then*
 $0 \leftarrow s$ *and* $0 \leftarrow t$
if T is $\mathbf{2}$*, then*
 $b \leftarrow s$ *and* $b \leftarrow t$ *for b either* 0 *or* 1.
if T is N*, then*
 $n \leftarrow s$ *and* $n \leftarrow t$ *for some n one of* $0, 1, 2, \ldots$
if T is $A \rightarrow B$, then
 $\lambda x.b_1 \leftarrow s$ *and* $\lambda y.b_2 \leftarrow t$ *and*
  $\forall$ *terms a_1 and $a_2 \in A$, $a_1 = a_2 \in A$*
  implies $b_1[a_1/x] = b_2[a_2/y] \in B$.
if T is $\overline{A}$ and A is not a bar type, then
 $(s\downarrow$ *iff* $t\downarrow)$ *and* $s\downarrow$ *implies* $s = t \in A$.

We let $t \in T$ abbreviate $t = t \in T$.

The rules of the theory are provably sound under this semantics[19], but since we are not providing the rules here, we rely directly on the semantics in the proofs that follow. Because of this it is perhaps useful to notice a few properties of definition 1:

Theorem 1 *For all types T,*
 (i) $s = t \in T \Rightarrow t = s \in T$
 (ii) $s = t \in T$ & $t = u \in T \Rightarrow s = u \in T$.

Theorem 2 *For all types T,*
 $t \in T$ & $u \leftarrow t$ & $u \leftarrow s \Rightarrow t = s \in T$.

$a = b \in \mathbf{N}$ means a and b both evaluate to the same natural number, say n. $\lambda x.b \in A \rightarrow B$ means for all a and a' that are equal members of A, $b[a/x]$ and $b[a'/x]$ are equal members of B. These functions thus behave as mathematical functions should: they map equal arguments to equal values.

3 Basic results

3.1 Overview

Our plan for this section is to examine certain basic concepts and results from recursive function theory over the natural numbers, say as presented in [18,20], and to find analogues of them in the theory of section 2. We start with undecidability results, then look at analogues of recursively enumerable sets, and then of reduction and completeness.

The unsolvability results are particularly easy to understand in this theory. We can argue that it is not possible to solve the "halting problem" for functions $f \in \mathbf{N} \rightarrow \overline{\mathbf{N}}$, say for specificity the problem "does $f(0)$ halt?" One way to express this problem is to notice that for every such f, $f(0)$ belongs to $\overline{\mathbf{N}}$. So we are equivalently asking for any computation $t \in \overline{\mathbf{N}}$, whether we can tell if t halts, that is, if there is a function $h \in \overline{\mathbf{N}} \rightarrow \mathbf{2}$ such that $h(t) = 1$ iff t halts. The answer is no, because if we assume that h exists, then we can define the function

$$d = fix(\lambda x.zero(h(x); \uparrow; 1)) \in \overline{\mathbf{N}},$$

where $\uparrow$ is some element of $\overline{\mathbf{N}}$ known to diverge such as $fix(\lambda t.t)$. $d \in \overline{\mathbf{N}}$ follows by the fixpoint rule because the body is in the type $\overline{\mathbf{N}} \rightarrow \overline{\mathbf{N}}$. By computing the fix term, we have $d = zero(h(d); \uparrow; 1) \in \overline{\mathbf{N}}$. If $h(d) = 0$, then d should diverge, but in fact $d = 1$; so it converges, and we reach a similar contradiction if $h(d) = 1$. So the assumption that h exists leads to a contradiction.

There is nothing special about $\overline{\mathbf{N}}$ in the argument except that there is an element such as $1 \in \mathbf{N}$. So the argument in general applies to any type T with some $t_0 \in T$. If we assume there is $h \in \overline{T} \to \mathbf{2}$ such that $h(t) = 1$ iff t converges, then we may define

$$d = f(\lambda x.zero(h(x); \uparrow; t_0)) \in \overline{T}.$$

The argument depends essentially on the self-referential nature of $fix(f)$, which has the type $\overline{T}$ where f is of type $\overline{T} \to \overline{T}$. This simple unsolvability argument cannot be valid in a classical type theory which takes $A \to B$ to denote the type of *all* functions from A into B, because in that case there surely is a function $\overline{T} \to \mathbf{2}$ solving the halting problem. This argument thus also shows that the constructive type-theoretic notion of partial function differs in some fundamental way from the classical notion.

In this type-theoretic setting we can establish other unsolvability results by reduction, and a version of Rice's Theorem, which summarizes these results, can be proved.

We consider any subcollection of terms in a type T to be a *class* of terms of T. In formal language theory, the concept of an *acceptable set* is important; that idea is captured here by saying that a class C_T over a type T is acceptable iff C_T consists of those values on which a partial function with domain T converges. We can define a kind of complement of an acceptable class C_T as being those values on which a partial function with domain T *diverges.* A complete acceptable class may be defined, and it is surprising that in this context, any nontrivial acceptable set is complete. This is essentially a consequence of the extensional equality of bar types.

3.2 Classes

Many of the theorems in the paper are about classes of elements over a type. For example, we consider the class K of all convergent elements of $\overline{\mathbf{N}}$; this is written as $\{x{:}\overline{\mathbf{N}} \mid x\downarrow\}$. Although such classes can be defined formally, say in type theory[4,15] or in set theory, we prefer an informal treatment which is applicable to a variety of formalizations. The notation we use for a class C_T over a type T is $\{x{:}T \mid exp\}$ where exp is some expression in x. We say $t \in \{x{:}T \mid exp\}$ for $t \in T$ when exp holds of t. Classes over bar types such as K are extensional collections of computations in that two computations are considered equal exactly when one converges iff the other does and if they converge they converge to the same value.

3.3 Unsolvability

We say that a class is *decidable* when there is a (total computable) function to determine when an element of the underlying type belongs to the class. A simple way to define this follows:

Definition 2 *C_T is decidable iff*
$\exists f{:}T \to \mathbf{2}. \forall x{:}T. x \in C_T \Leftrightarrow f(x) = 1 \in \mathbf{2}$

In the world of standard recursive function theory, the decidable classes over $\mathbf{N}$ are a small subset of the set of all subsets of $\mathbf{N}$. They are at the bottom of the Kleene hierarchy and form the lowest degree in the classification of these sets by reducibility orderings. We shall see that in this theory they too form a "small" subset of the set of all classes over $\mathbf{N}$, and more generally over any type T.

Definition 3 *Let* $K_{\overline{T}} = \{x{:}\overline{T} \mid x\downarrow\}$.

Theorem 3 (Unsolvability) *For all types T which have members, say $t_0 \in T$, $K_{\overline{T}}$ is not decidable.*

proof. See section 3.1 above.
qed.

The class of diverging computations is also not decidable.

Definition 4 *Let* $divK_{\overline{T}} = \{x{:}\overline{T} \mid x\uparrow\}$.

Theorem 4 *For any type T with members, $divK_{\overline{T}}$ is not decidable.*

proof. This is just like theorem 3; assume h decides membership and look at $d = fix(\lambda x.zero(h(x); \uparrow; t_0))$ where $t_0 \in T$.
qed.

There are other kinds of unsolvable problems. For example, consider functions $f \in S \to \overline{T}$ where S and T have members. Then the class

$$W_{S\to\overline{T}} = \{f{:}S \to \overline{T} \mid \exists y{:}S.\, f(y)\downarrow\}$$

is not decidable. To see this, suppose it were decidable. Then we could decide $K_{\overline{T}}$ because for each $x \in \overline{T}$ we can build an $f \in S \to \overline{T}$ which is the constant function returning x, i.e. f is $\lambda y.x$, and we notice that $\exists y{:}S.\, f(y)\downarrow$ iff $x\downarrow$. So if $h \in (S \to \overline{T}) \to \mathbf{2}$ decides $W_{S\to\overline{T}}$ then $\lambda x.h(\lambda y.x) \in \overline{T} \to \mathbf{2}$ decides $K_{\overline{T}}$. We have proved:

Theorem 5 (Weak Halting) *For any types S and T with members, $\{f : S \to \overline{T} \mid \exists x : S.\, f(x)\downarrow\}$ is not decidable.*

The proof proceeded by *reducing* the class $K_{\overline{T}}$ to the class $W_{S\to\overline{T}}$. This is a general method of establishing unsolvability, characterized by this definition.

Definition 5 *Class C_S is* reducible *to class C_T, written $C_S \preceq C_T$, iff there is a function $f \in S \rightarrow T$ such that $\forall x{:}S.\ x \in C_S \Leftrightarrow f(x) \in C_T$.*

Fact 1 *$\preceq$ is reflexive and transitive.*

When reducing to a class over a bar type, say $C_{\overline{T}}$, the reduction function $f \in S \rightarrow \overline{T}$ might yield a nonterminating computation, so it is a partial function. It seems unnatural to use partial functions for reduction, but there is no harm in this because we can always replace them by total functions into the type $\mathbf{1} \rightarrow \overline{T}$. That is, given $f \in S \rightarrow \overline{T}$, replace it by $g \in S \rightarrow (\mathbf{1} \rightarrow \overline{T})$ where $g(x) = \lambda y.f(x)$, and y does not occur free in f. This gives an equivalent total reduction because $t \in C_{\overline{T}} \Leftrightarrow \lambda x.t \in C_{\mathbf{1}\rightarrow\overline{T}}$: the "dummy" lambda abstraction serves to stop computation.

Rice's theorem summarizes the unsolvability results by characterizing the decidable classes of computations over any bar type in a strong way. In this setting, Rice's theorem says that all decidable classes of computations are trivial.

Definition 6 *For any type T call a class C_T* trivial *iff*

$$(\forall x{:}T.\, x \in C_T) \vee (\forall x{:}T.\, \neg(x \in C_T)).$$

Theorem 6 (Rice) *For all types T, $C_{\overline{T}}$ is decidable iff $C_{\overline{T}}$ is trivial*

proof. ($\Leftarrow$) This follows directly, for $\lambda x.1$ characterizes the maximal class, and $\lambda x.0$ characterizes the minimal (empty) one.
($\Rightarrow$) suppose $f \in \overline{T} \rightarrow \mathbf{2}$ decides $C_{\overline{T}}$. Since f is total, we know that $f(\uparrow) = 0$ or $f(\uparrow) = 1$; we show for the case $f(\uparrow) = 0$ that the class must be minimal, and for $f(\uparrow) = 1$ that it must be maximal.
case $f(\uparrow) = 0$: We show C is trivial by showing it is minimal, i.e. $\forall t{:}\overline{T}.\ f(t) = 0$. Let $t \in \overline{T}$ be arbitrary. We may show $f(t) = 0$ arguing by contradiction because the equality is decidable. So, assume $f(t) \neq 0$. $div K_{\overline{T}}$ may then be shown to be decidable using the function

$$h = \lambda x.f(x;t) \in \overline{T} \rightarrow \mathbf{2}.$$

For h to characterize $div K_{\overline{T}}$ means $h(x) = 0 \Leftrightarrow x\uparrow$.
($\Rightarrow$) $h(x) = 0$ implies $f(x;t) = 0$. Supposing $x\downarrow$, $f(x;t) = f(t) = 0$, but this contradicts our assumption, so $x\uparrow$.
($\Leftarrow$) $x\uparrow$ means $h(x) = f(x;t) = f(\uparrow) = 0$. $div K_{\overline{T}}$ is not decidable by theorem 4, so we have a contradiction.
case $f(\uparrow) = 1$: We show C is maximal, i.e. $\forall t{:}\overline{T}.\, f(t) = 1$. This case is similar to the previous except that we switch the output of the reduction function h to make it

$$h = \lambda x.zero(f(x;t); 1; 0) \in \overline{T} \rightarrow \mathbf{2}.$$

qed.

3.4 Acceptable classes

One of the basic concepts in the study of formal languages is that of an *acceptable set.* For example, the regular sets are those accepted by a finite automaton, and the deterministic context free languages are those accepted by deterministic pushdown automata. It is a major result of standard recursive function theory that the recursively enumerable sets (r.e. sets) are exactly those accepted by Turing machines. In this setting, an acceptable class is one whose elements can be recognized by a partial function, precisely:

Definition 7 *A class C_T is* converge-acceptable *or just* acceptable *iff*
$$\exists f{:}T \rightarrow \overline{\mathbf{1}}.\ \forall x{:}T.\ x \in C_T \Leftrightarrow f(x)\downarrow$$

A class C_T is diverge-acceptable *iff*
$$\exists f{:}T \rightarrow \overline{\mathbf{1}}.\ \forall x{:}T.\ x \in C_T \Leftrightarrow f(x)\uparrow$$

The canonical acceptable class is $K_{\overline{T}}$, and we may show

Theorem 7 *For all types T, $K_{\overline{T}}$ is acceptable.*

proof. The accepting function f is $\lambda x.(x;0) \in \overline{T} \rightarrow \overline{\mathbf{1}}$, which converges exactly when its argument x converges.
qed.

The diverge-acceptable classes are needed to deal with the idea of the complement of an acceptable class. In a constructive setting, there is often no single concept to replace the classical idea of a complement. In classical recursion theory, complements have the property that any subset S of $\mathbf{N}$, any element of $\mathbf{N}$ either lies in S or in its complement, i.e. if $\sim S$ denotes the complement, then $\forall x{:}\mathbf{N}.\,(x \in S \vee x \in \sim S)$. But taken constructively this definition says that membership in S is decidable. In the case of acceptable but not decidable classes S, we cannot in general say that $\sim S$ is not acceptable. The diverge-acceptable classes serve as an analogue of a complement.

Theorem 8 *For any type T with members, $div K_{\overline{T}}$ is diverge-acceptable.*

proof. The diverge-acceptor function f is $\lambda x.(x;0) \in \overline{T} \rightarrow \overline{\mathbf{1}}$.
qed.

We also know that $divK_{\overline{T}}$ is not acceptable, so *div* acts like a complement. $K_{\overline{T}}$ is not diverge-acceptable either.

Theorem 9 *For any type T with members,*

(i) *$K_{\overline{T}}$ is not diverge-acceptable.*
(ii) *$divK_{\overline{T}}$ is not acceptable.*

proof. For (i), suppose f diverge-accepted $K_{\overline{T}}$ and $t_0 \in T$; define

$$d = fix(\lambda x.(f(x); t_0)) \in \overline{T}.$$

$d\downarrow$ iff $d\uparrow$ follows directly, which is a contradiction. The proof of (ii) is similar.
qed.

3.5 Unions and intersections

We may take unions, intersections, and negations of classes, defined as

Definition 8

$$\begin{array}{ll} c \in \sim A_T \text{ iff} & c \in T \,\&\, c \notin A_T \\ c \in A_T \cup B_T \text{ iff} & c \in A_T \vee c \in B_T \\ c \in A_T \overset{w}{\cup} B_T \text{ iff} & \neg(c \notin A_T \,\&\, c \notin B_T) \\ c \in A_T \cap B_T \text{ iff} & c \in A_T \,\&\, c \in B_T \end{array}$$

The weak union $c \in A_T \overset{w}{\cup} B_T$ is needed because it is not always possible to form a strong union constructively; that requires that we may decide which class each term falls in.

The decidable classes over any type T are closed under union, intersection and negation.

Theorem 10 (Decidable boolean operations)

For any type T and for any decidable classes A_T, B_T over T, the union, $A_T \cup B_T$, intersection $A_T \cap B_T$, and complement $\sim A_T$ are also decidable.

proof. Suppose that f_A accepts A_T and f_B accepts B_T; then

$$\lambda x.zero(f_A(x); 1; 0)$$

accepts $\sim A_T$,

$$\lambda x.zero(f_A(x); zero(f_B(x); 0; 1); 1)$$

accepts $A_T \cup B_T$, and

$$\lambda x.zero(f_A(x); 0; zero(f_B(x); 0; 1))$$

accepts $A_T \cap B_T$.
qed.

The acceptable classes over any type T are closed under intersection, namely if f_A accepts A_T and f_B accepts B_T, then $\lambda x.f_A(x); f_B(x)$ accepts $A_T \cap B_T$. If f_A and f_B accept by divergence, then this composite function accepts the weak union $A_T \overset{w}{\cup} B_T$. One might expect the acceptable classes to be closed under union as well, since in standard recursion theory the r.e. sets are closed under union. But the standard result requires that we *dovetail* the computation $f_A(x)$ with the computation $f_B(x)$. That is, we run f_A for a fixed number of steps, then f_B for some number, then f_A for a fixed number of steps, then f_B for some number, then f_A again, then f_B, etc., until one of them terminates. In the theory presented so far, this cannot be done because we do not have access to the structure of the computation. We will discuss this situation further in section 4.2 where we add a new operator to the theory which captures certain aspects of dovetailing. So the best we can claim now (as proved above) is:

Theorem 11 (Intersection of acceptable classes) *For any type T, the acceptable classes over T are closed under intersection, and the diverge-acceptable classes are closed under weak union.*

3.6 Complete classes

In standard recursive function theory, a class such as $K_{\overline{\mathbf{N}}}$ is complete in the sense that any acceptable class can be reduced to it. The idea of completeness has been very important and led to such notions as complete sets for various complexity classes, e.g., polynominal time complete sets. Here there is also an interesting notion of completeness.

Definition 9 *Call a class C_T* acceptably-complete *if C_T is acceptable and for all types S and acceptable classes D_S, D_S is reducible to C_T, i.e. $D_S \preceq C_T$. Likewise C_T is* diverge-acceptably complete *if C_T is diverge-acceptable and for all types S and diverge-acceptable classes D_S, $D_S \preceq C_T$.*

Theorem 12 (Complete classes)
For all nonempty types T,

(i) *$K_{\overline{T}}$ is acceptably-complete and*
(ii) *$divK_{\overline{T}}$ is diverge-acceptably complete.*

proof. (i) Let $f \in \overline{T} \rightarrow \overline{1}$ accept $K_{\overline{T}}$, and suppose $t_0 \in T$ and D_S is an arbitrary acceptable class with acceptor function g. Then, define the reduction function

$$m = \lambda s.(g(s); t_0) \in S \rightarrow T.$$

For arbitrary $s \in S$, it must be that $s \in D_S \Leftrightarrow m(s) \in K_{\overline{T}}$, i.e. $g(s)\downarrow \Leftrightarrow f(m(s))\downarrow$.
($\Rightarrow$) $g(s)\downarrow \Rightarrow m(s) = t_0$, so $f(m(s))\downarrow$ (we know $t_0 \in K_{\overline{T}}$).
($\Leftarrow$) $f(m(s))\downarrow$ means $m(s)\downarrow$ since f characterizes $K_{\overline{T}}$, so $g(s); t_0\downarrow$, meaning $g(s)\downarrow$.
(ii) This proof is similar to (i).
qed.

4 A Family of Computation Theories

We envision a family of computation theories, each with a different basis for what constitutes computation. The basic theory of the previous section can be extended in numerous ways; each extension gives rise to a different collection of theorems, all extensions of the basic results of the previous section. The computational facts on which particular theorems depend is an interesting issue in its own right, carving up the mass of theorems of standard recursion theory into smaller clusters. We will add some axioms about uniform behavior of computations, add the ability to dovetail and to count the steps of computations, and add non-mathematical intensional types which extend the scope of reasoning.

It is possible to consider an even more basic computation theory where there is a Kleene least number operator μ to define partial functions instead of *fix*. All Turing computable functions are definable in this theory, but it does not account for the self-referential nature of computation and there are potentially no unsolvable problems.

4.1 Uniformity Principles

There are two *uniformity principles* which allow functions applied to diverging computations to be more precisely characterized:

$$\forall f{:}\,\overline{A}\rightarrow\overline{B}.\, f(\uparrow)\downarrow \Rightarrow \forall a{:}\,\overline{A}.\, f(a)\downarrow \qquad \text{(I)}$$

$$\forall f{:}\,\overline{A}\rightarrow\overline{B}.\, f(\uparrow)\uparrow \Rightarrow \forall a{:}\,\overline{A}.\, (f(a)\downarrow \Rightarrow a\downarrow) \qquad \text{(II)}$$

There are two justifications for these principles. The first justification explicitly uses the computational semantics and the evaluation relation $\leftarrow$ defined therein.

Theorem 13 *Semantically, I and II are true.*

proof. (I) When $f(\uparrow)\downarrow$, the argument $\uparrow$ must not have been computed, for that would mean in an extensional setting that the computation would have to diverge. If the argument was not computed, it could be anything, so $\forall a{:}\,\overline{A}.\, f(a)\downarrow$.
(II) The argument to f could not have been ignored, because f is not a constant function. Therefore, the argument must have been computed, so if $f(a)\downarrow$, $a\downarrow$ as well.
qed.

The other justification follows if we accept Markov's principle, $\neg t\uparrow \Rightarrow t\downarrow$. These results are then directly provable, with no need to take a semantic viewpoint.

Theorem 14 *Markov's Principle $\Rightarrow$ I & II.*

proof. (I) Take an arbitrary $f \in \overline{A} \Rightarrow \overline{B}$, with $f(\uparrow)\downarrow$. Suppose $f(a)\uparrow$ for arbitrary $a \in \overline{A}$; we will show a contradiction. Note that $a \neq \uparrow$, because otherwise $f(a)\downarrow$; and by Markov, we may thus conclude $a\downarrow$. We now assert $K_{\overline{A}}$ is diverge-acceptable. Define its diverge-accepting function $h = \lambda x.f(x;a); 0 \in \overline{A}\rightarrow\overline{1}$. We only need to show

$$\forall t{:}\,\overline{A}.\, h(t)\uparrow \Leftrightarrow t\downarrow,$$

and this follows from the definition of h:

$$h(t)\uparrow \Leftrightarrow f(t;a)\uparrow \Leftrightarrow \neg t\uparrow \Leftrightarrow t\downarrow.$$

But this is a contradiction, for $K_{\overline{A}}$ is not diverge-acceptable (theorem 9). Therefore $\neg f(a)\uparrow$, which by Markov allows us to conclude $f(a)\downarrow$.
(II) Assume $f(\uparrow)\uparrow$ and $f(a)\downarrow$; we show $a\downarrow$ by showing $\neg a\uparrow$. Suppose $a\uparrow$; then $f(a)\uparrow$ because $a = \uparrow$, contradicting our assumption.
qed.

A strong characterization of the acceptable classes over bar types may now be given. Accepting functions $f \in \overline{T}\rightarrow\overline{1}$ are required to map equal computations to the same result, and we show below that this means all nontrivial classes must be complete.

Definition 10 *C_T is strongly nontrivial $\Leftrightarrow$*
$\exists t_0{:}\,T.\, t_0 \in C_T \;\&\; \neg\forall t{:}\,T.\, t \in C_T$

Theorem 15 (acceptability characterization)

$C_{\overline{T}}$ is acceptable &
$C_{\overline{T}}$ is strongly nontrivial $\Rightarrow$
$C_{\overline{T}}$ is acceptably-complete .

proof. $C_{\overline{T}}$ is acceptable means that for all t, the acceptor function $f_C(T)\downarrow \Leftrightarrow t \in C_{\overline{T}}$. Also, by the nontriviality assumption $t_0 \in C_{\overline{T}}$.

We may assert $f_C(\uparrow)\uparrow$: if $f_C(\uparrow)\downarrow$, then by I we have $\forall t{:}\,\overline{T}.\, f_C(t)\downarrow$, contradicting the nontriviality of $C_{\overline{T}}$. $t_0\downarrow$ then follows by II.

We next show $C_{\overline{T}}$ is acceptably-complete. Let D_S be an arbitrary acceptable class, with an accepting function $f_D \in S \to \overline{1}$. It must be that $D_S \preceq C_{\overline{T}}$. Let m be $\lambda t.(f_D(t); t_0) \in S \to \overline{T}$. For m to be the reduction function it must satisfy

$$\forall t{:}\, S.\, f_D(t)\downarrow \Leftrightarrow f_C(m(t))\downarrow .$$

($\Rightarrow$) Suppose $f_D(t)\downarrow$; then $(f_D(t); t_0) = t_0$, so $f_C(m(t)) = f_C(f_D(t); t_0) = f_C(t_0)$, which converges because $t_0 \in C_{\overline{T}}$.
($\Leftarrow$) suppose $f_C(f_D(t); t_0)\downarrow$; by uniformity II, that means $f_D(t); t_0\downarrow$, so $f_D(t)\downarrow$.
qed.

Using Rice's theorem (theorem 6), we may prove

Corollary 1 *For all types T, $C_{\overline{T}}$ is acceptable $\Rightarrow$ $C_{\overline{T}}$ is decidable $\vee_C$*[1] *$C_{\overline{T}}$ is acceptably-complete*

proof. For acceptable C, this is equivalent to proving

$$\neg C \textit{ is decidable } \Rightarrow \neg\neg C \textit{ is acceptably-complete }.$$

By theorem 15, we have

$$\neg\neg C \textit{ is strongly nontrivial } \Rightarrow \neg\neg C \textit{ is acceptably-complete },$$

and the corollary will thus follow from

$$\neg C \textit{ is decidable } \Rightarrow \neg\neg C \textit{ is strongly nontrivial }.$$

We prove this by proving

$$\neg C \textit{ is decidable } \Rightarrow \neg C \textit{ is trivial}$$

and

$$\neg C \textit{ is trivial } \Rightarrow \neg\neg C \textit{ is strongly nontrivial },$$

both of which follow by straightforward propositional reasoning.
qed.

4.2 Dovetailing Computations

In the basic theory, there is no possibility of dovetailing computations. In standard recursion theory, two computations may be dovetailed with the aid of a universal machine, but this theory is not endowed with such a machine, so we directly add dovetailing. We define the *dovetailing constructor* $a||b$ to simultaneously compute a and b.

[1] $A \vee_C B$ is a classical disjunction, $\neg(\neg A \,\&\, \neg B)$.

Definition 11 *Define a new computation relation $v \overset{\parallel}{\longleftarrow} t$ which has all of the clauses of figure 1, and with the additional clause*

$$v \overset{\parallel}{\longleftarrow} a||b \Leftrightarrow v \overset{\parallel}{\longleftarrow} a \vee v \overset{\parallel}{\longleftarrow} b.$$

The computation relation $\overset{\parallel}{\longleftarrow}$ is not a function: $1 \overset{\parallel}{\longleftarrow} 1||2$, and $2 \overset{\parallel}{\longleftarrow} 1||2$. Such multivalued terms make no sense inhabiting our existing types, so we redefine $v \leftarrow t$ as a deterministic restriction of the above relation:

Definition 12 *Redefine $v \leftarrow t$ as follows:*
$v \leftarrow t$ *iff* $v \overset{\parallel}{\longleftarrow} t \;\&\; \forall v'.\, v' \overset{\parallel}{\longleftarrow} t \Rightarrow v'$ *is* v.
Redefine $t\downarrow$ as:
$t\downarrow$ *iff* $\exists v.\, v \overset{\parallel}{\longleftarrow} t$.

The type system over this computation system is then defined as in definition 1. It is possible to dovetail computations where one or both may have no value at all; this is reflected in

Fact 2 $a||b \in \overline{T}$ *if*
$a \in \overline{T} \;\&\; b \in \overline{T} \;\&\; (a\downarrow \;\&\; b\downarrow \Rightarrow a$ *is* $b)$.

A more liberal use of parallelism would be allowed if there were types which could have multivalued terms as inhabitants.

With dovetailing, we may enlarge our collection of acceptable classes. Most importantly, acceptable classes are now provably closed under union.

Theorem 16 *C_T is acceptable & D_T is acceptable $\Rightarrow C_T \cup D_T$ is acceptable .*

proof. The accepting function for $C_T \cup D_T$ is $\lambda x.f_C(x)||f_D(x) \in T \to \overline{1}$.
qed.
By a similar argument, diverge-acceptable classes can be shown to be closed under intersection.

Using fixpoints, it is possible to dovetail infinitely many computations.

Theorem 17 $\{g{:}\,\mathbf{N}\to\overline{\mathbf{N}} \mid \exists n{:}\,\mathbf{N}.g(n)\downarrow\}$ *is acceptable.*
proof. The accepting function is

$$\lambda g.\mathit{fix}(\lambda h.\lambda x.g(x); 0||h(x+1))(0) \in (\mathbf{N} \to \overline{\mathbf{N}}) \to \overline{1}$$

which computes to

$$(g(0); 0)||(g(1); 0)||(g(2); 0)|| \ldots$$

This computation terminates just in case $g(n)$ terminates for some n.
qed.

In standard recursion theory, if a class is acceptable and diverge-acceptable, it is also decidable, because we compute both and know one or the other will halt for any element of the domain. This does not follow constructively because it is impossible to say that one or the other will halt. It is however provable using Markov's principle:

Theorem 18 *Markov's principle implies*
C_T is acceptable &
C_T is diverge-acceptable $\Rightarrow$ C_T is decidable .

proof. Suppose f accepts C , and g diverge-accepts C. Then define the following function to dovetail the two:

$$r = \lambda x.(f(x);1)||(g(x);0) \in T \rightarrow \overline{\mathbf{2}}$$

Before proceeding, we check to make sure r is of the indicated type. For arbitrary x, we wish to show $(f(x);1)||(g(x);0) \in \overline{\mathbf{2}}$. Using fact 2 above, we only need to show

$$(f(x);1)\downarrow \;\&\; (g(x);0)\downarrow \Rightarrow (f(x);1) = (g(x);0) \in \mathbf{2}$$

But, the antecedent will never be true, for then $f(x)$ and $g(x)$ would both converge by Markov, but that means $x \in C$ and $x \notin C$, a contradiction. If r is to decide C, r must be total. By Markov, we only need to show that for arbitrary x, r does not diverge. Suppose $r(x)\uparrow$; then, by the definition of $||$, $(f(x);1)\uparrow$ and $(g(x);0)\uparrow$, meaning $x \notin C$ and $\neg(x \notin C)$, a contradiction. Thus, $r \in T \rightarrow \mathbf{2}$. It is easy to see that r in fact decides C.
qed.

4.3 Measuring Computations

Terminating computations are generally accepted to be composed of a finite number of discrete steps. However, there is nothing in the basic theory which asserts this finiteness. Many results about computations hinge on their finite nature, and it is therefore worthwhile to extend the theory to explicitly assert finiteness. To constructively assert that each terminating computation is finite is to assert that it has some finite step count n. We must also assert that this step count is unique, which all but restricts the computation system to being deterministic. In the computational semantics this gives rise to a three-place evaluation relation.

Definition 13
Define the following evaluation relations:
$v \overset{n}{\leftarrow} t$ *iff* t *evaluates to* v *in* n *or fewer steps*
$t\downarrow^n$ *iff* $\exists v.\, v \overset{n}{\leftarrow} t$
$t\uparrow^n$ *iff* $\neg t\downarrow^n$.
$\leftarrow$ *is then redefined as*
$v \leftarrow t$ *iff* $\exists n.\, v \overset{n}{\leftarrow} t$.

The type system is defined as in definition 1, using this new notion of evaluation. The computation theory is now non-extensional, because computations have a property besides their value, their step count, and terms with equal values may have differing step counts. Such computation systems are said to be *intensional.* Since the step counts constructively exist, we may add a term to the computation system to count steps. Such a counter would diverge if the computation diverged, so instead we add a more powerful total term:

Definition 14 *Extend the definition of computation in figure 1 by adding the following clauses:*

$$0 \leftarrow comp(t)(n) \Leftrightarrow t\uparrow^n$$
$$1 \leftarrow comp(t)(n) \Leftrightarrow t\downarrow^n.$$

Fact 3 *We may characterize comp by*

$$(\exists n{:}\mathbf{N}.\, 1 \leftarrow comp(t)(n)) \Leftrightarrow t\downarrow.$$

With *comp*, we have enough power to *define* a deterministic dovetailing constructor, $||$:

Definition 15 $a||b =$
$$fix(\lambda d.\lambda n.zero(comp(a)(n);\; zero(comp(b)(n); d(n+1); b);\; a))(0)$$

This function returns whichever of a or b first terminates.

It is now possible to prove some classes which do not involve bar types are unsolvable:

Definition 16
$V = \{f{:}\mathbf{N} \rightarrow \mathbf{N} \mid \exists n{:}\mathbf{N}.\, f(n) = 1 \in \mathbf{N}\}$
$divV = \{f{:}\mathbf{N} \rightarrow \mathbf{N} \mid \forall n{:}\mathbf{N}.\, f(n) = 1 \in \mathbf{N}\}$.

It is easy to show

Fact 4
V is acceptable and $divV$ is diverge-acceptable .

More importantly, we may prove

Theorem 19 *V is not decidable.*

proof. Suppose V was decidable, with a decision function $s \in (\mathbf{N} \rightarrow \mathbf{N}) \rightarrow \mathbf{2}$. We may then construct $h \in \overline{\mathbf{N}} \rightarrow \mathbf{2}$ to solve the halting problem:

$$h = \lambda x.s(\lambda n.comp(x)(n)).$$

We assert

$$\forall x{:}\overline{N}.\, x\downarrow \Leftrightarrow h(x) = 1 \in \mathbf{2}.$$

($\Rightarrow$) Suppose $x\downarrow$. We wish to show $h(x) = 1$, which by definition means $s(\lambda n.comp(x)(n)) = 1$, which in turn means $\exists n{:}\mathbf{N}.\ comp(x)(n) = 1 \in \mathbf{N}$. This follows directly from fact 3.
($\Leftarrow$) Suppose $h(x) = 1$, meaning $\exists n{:}\mathbf{N}.\ comp(x)(n) = 1 \in \mathbf{N}$, so $x\downarrow$.
qed.

The intensional nature of *comp* opposes the extensional nature of functions in this theory, which greatly restricts the class of functions that may use *comp*. For example, we may not show $K \preceq V$, because the expected reduction $m = \lambda x.\lambda n.comp(x)(n)$ is not in the type $\overline{\mathbf{N}} \to (\mathbf{N} \to \mathbf{N})$. This is because $x = y \in \overline{\mathbf{N}}$ does not mean x and y have an equal number of computation steps, so $m(x)$ might be different from $m(y)$.

4.4 Intensional types

To get around the shortcomings of a strongly extensional type system, a new type constructor *i-T* is added to the theory of the previous section. This strips whatever mathematical equality there was off of T, so that only identical terms are equal in *i-T*. Equality for i-types is an algorithmic, and not mathematical, equality. Semantically,

Definition 17 *Add the following clause to definition 1, the inductive definition of $s = t \in T$:*
if T is i-A, then s is t and $s \in A$.

The members of $i\text{-}A \to B$ are intensional *operations* from A to B, for equal members of A do not have to be mapped to equal members of B.

Fact 5 *The type $i\text{-}A \to B$ may be characterized by*
$$\lambda x.b \in i\text{-}A \to B \text{ iff } \forall a{:}A.\ b[a/x] \in B.$$

proof. This follows directly from the semantic definitions.
qed.

Intensional analogues of decidability, acceptability, and diverge-acceptability may be defined, using operations instead of functions to characterize them.

Definition 18
C_T *is* i-decidable *iff*
$$\exists f{:}\, i\text{-}T \to \mathbf{2}.\ \forall x{:}T.\ x \in C_T \Leftrightarrow f(x) = 1 \in \mathbf{2}$$
C_T *is* i-acceptable *iff*
$$\exists f{:}\, i\text{-}T \to \overline{\mathbf{1}}.\ \forall x{:}T.\ x \in C_T \Leftrightarrow f(x)\downarrow$$
C_T *is* i-diverge-acceptable *iff*
$$\exists f{:}\, i\text{-}T \to \overline{\mathbf{1}}.\ \forall x{:}T.\ x \in C_T \Leftrightarrow f(x)\downarrow$$

$f \in A \to B$ implies $f \in i\text{-}A \to B$, so the i-classes include all standard classes.

An intensional notion of reducibility may also be defined, again by using operators instead of functions.

Definition 19 *$C_S \preceq_i C_T$ iff there is a function $f \in i\text{-}S \to T$ such that $\forall x{:}S.\ x \in C_S \Leftrightarrow f(x) \in C_T$.*

This relation is reflexive and transitive. i-completeness is defined as being complete i-acceptable (diverge-acceptable) with respect to $\preceq_i$. In this intensional setting, we may now show V is i-acceptably-complete, by showing $K_{\mathbf{N}} \preceq_i V$.

Theorem 20 *V is i-acceptably-complete.*
proof. To show $K_{\overline{\mathbf{N}}} \preceq_i V$, define the reduction function
$$m = \lambda x.(\lambda n.comp(x)(n)) \in i\text{-}\overline{\mathbf{N}} \to (\mathbf{N} \to \mathbf{N})$$
Operations need not respect the equality of elements in $\overline{\mathbf{N}}$, so m is of the indicated type. For arbitrary $x \in \mathbf{N}$, we need to show $x \in K_{\overline{\mathbf{N}}}$ iff $m(x) \in V$, i.e. $x\downarrow$ iff $\exists n{:}\mathbf{N}.\ comp(x)(n) = 1 \in \mathbf{N}$. This follows immediately from fact 3.
qed.

The class of total functions from $\mathbf{N}$ to $\overline{\mathbf{N}}$ can now be shown to lie strictly above K and *divK*.

Definition 20 *Tot, the class of total functions, is*
$$\{f{:}\mathbf{N} \to \overline{\mathbf{N}} \mid \forall n{:}\mathbf{N}.\ f(n)\downarrow\}$$

Theorem 21 *Tot lies above both $K_{\overline{T}}$ and $divK_{\overline{T}}$:*
(i) $K_{\overline{T}} \preceq_i Tot$
(ii) $divK_{\overline{T}} \preceq_i Tot$

proof. (i) The reduction function is $\lambda x.(\lambda n.x) \in \overline{T} \to (\mathbf{N} \to \overline{\mathbf{N}})$.
(ii) The reduction function is
$$m = \lambda x.(\lambda n.zero(comp(x)(n); 0; \uparrow)) \in i\text{-}\overline{T} \to (\mathbf{N} \to \overline{\mathbf{N}}).$$
For arbitrary $x \in \overline{T}$, $x \in divK_{\overline{T}}$ iff $x\uparrow$ iff $\forall n : \mathbf{N}.\ comp(x)(n) = 0$ iff $\forall n{:}\mathbf{N}.\ zero(comp(x)(n); 1; \uparrow)\downarrow$ iff $\forall n{:}\mathbf{N}.\ m(x)(n)\downarrow$ iff $m(x) \in Tot$.
qed.

Theorem 22
(i) $Tot \not\preceq_i K_{\overline{T}}$
(ii) $Tot \not\preceq_i divK_{\overline{T}}$

proof. (i) Suppose $Tot \preceq_i K_{\overline{T}}$. $divK_{\overline{T}} \preceq_i Tot$ by the above, but the transitivity of $\preceq_i$ gives $divK_{\overline{T}} \preceq_i K_{\overline{T}}$, contradicting theorem 9.
(ii) This case is similar to (i).
qed.

Without an intensional reducibility ordering, *Tot* cannot be placed above $divK_{\overline{T}}$. The reducibility ordering of standard recursion theory is also intensional, so is more like $\preceq_i$ than the mathematical reducibility $\preceq$.

5 Related Work

Abstract recursion theory is a rich area of research, with many varied approaches to be found, some of them related to our work (for a review, see [9]). However, all postulate the indexability of computations, which leads to the universal machine and S–m–n theorems. We mention here two different approaches.

Wagner[22] has developed an algebraic account of computability, the *Uniformly Reflexive Structure*, or URS. This theory was elaborated and extended by Strong [21], Friedman[10], and Barendregt[2]. This theory is essentially a theory of combinators with an if–then construct to compare terms and an explicit diverging element $*$. From this, the universal machine and S–m–n theorems may be proven.

Another account which has more resemblance to this work is Platek's inductive definability approach to recursion theory[17], further expounded by Feferman[11] and Moschovakis[16]. In this typed theory, types are interpreted as sets, and functions are taken to be partial maps from sets to sets which are monotone; monotonicity guarantees that the class of functions will be closed under fixpoints, which means that a rich class of computations much like the ones of this paper may be interpreted to lie in this structure.

Beyond this initial point, their approach completely diverges from that of this paper. Recursion theory cannot be carried out in a setting where functions are interpreted as sets, for there is no *structure* to the computations: all functions with the same input-output behavior are identified. They thus proceed by considering conditions under which enumerations will in fact exist, and under such conditions can prove the universal machine and S–m–n theorems. This approach is more *ad hoc* than we would like a foundational theory to be.

6 Conclusions

The constructive recursive function theory (CRFT) of this paper is quite different from classical basic recursive function theory (BRFT). Most importantly, BRFT assumes an indexing of all partial recursive functions, which allows the universal machine theorem to be proven. It also uses Church's thesis to confer absoluteness and relevance to the results. In CRFT it is the fixpoint principle, a more directly self-referential fact than the universal machine theorem, which leads to unsolvable problems. The fixpoint principle gives basic unsolvability results, and each additional assumption gives rise to another collection of theorems, as our results demonstrate.

Type theory is a natural setting for recursion theory; its generality gives an absoluteness and relevance to the results. The results generalize to types such as ordinals, trees, infinite lists, and real numbers, without the need to build new theories for each type.

CRFT also impacts type theory because the types cannot now be given purely classical interpretations: if $A \rightarrow \overline{B}$ were all set-theoretic functions from A to $B \cup \{\uparrow\}$, fixpoints could not exist for all functions. The concept of partial function in CRFT thus serves to distinguish classical from constructive type theory in a way different from the presence or absence of the law of excluded middle. In a type theory such as Nuprl[4] where mathematical propositions can be represented via types, the excluded middle law itself is inconsistent in the presence of partial types: if we had a method of determining for all propositions P whether P were true or false, we could use this to show some term t either halts or doesn't, which contradicts the unsolvability of the halting problem.

Acknowledgements

We would like to thank Elizabeth Maxwell for her cheerful patience in preparing this manuscript and learning LaTeX. We also appreciate the insightful comments of Stuart Allen and David Basin in discussions of this work.

References

[1] Stuart F. Allen, *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* Doctoral Dissertation, Computer Science Department, Cornell University, 1987. Also as *Computer Science Department Technical Report*, TR 87–866. Cornell University, Ithaca, NY, 1987.

[2] Henk Barendregt, Normed Uniformly Reflexive Structures. In *Lambda-calculus and Computer Science Theory, Lecture notes in Computer Science*, vol. 37, 1975, pages 272–286.

[3] David Basin, An Environment for Automated Reasoning about Partial Functions. To appear at *CADE-9*, Argonne, Illinois, May 1988.

[4] Robert L. Constable et. al., *Implementing Mathematics with the Nuprl Proof Development System.* Prentice–Hall, 1986.

[5] Robert L. Constable and Scott Fraser Smith, Partial Objects in Constructive Type Theory. In *Symposium on Logic in Computer Science*, 1987, pages 183–193.

[6] Robert L. Constable, A Constructive Theory of Recursive Functions. *Computer Science Department Technical Report*, TR 73–186. Cornell University, Ithaca, NY, 1973.

[7] Thierry Coquand and Gerard Huet, Constructions: A Higher Order Proof System for Mechanizing Mathematics. EUROCAL 85, Linz, Austria, April 1985.

[8] M. Davis (editor), *The Undecidable.* Raven Press, Hewlett, N.Y. 1965.

[9] A. P. Ershov, Abstract Computability on Abstract Structures. In *Algorithms in Modern Math and Computer Science, Lecture Notes in Computer Science,* Vol. 122. Springer–Verlag, New York, 1981, pages 397–420.

[10] Harvey Friedman, Axiomatic Recursive Function Theory. In *Logic Colloquium '69*, North-Holland, Amsterdam, 1974, pages 385–404.

[11] Solomon Feferman, Inductive schemata and recursively continuous functionals. In *Logic Colloquium '76*, North-Holland, Amsterdam, 1977, pages 373–392.

[12] J. Y. Girard, Une extension de l'interpretation de Godel a l'analyse, et son application a l'elimination des coupures dans l'analyse et la theorie des types. In *2nd Scandinavian Logic Symposium*, J. E. Fenstad, ed. North–Holland, Amsterdam, 1971, pages 63–92.

[13] Stephen C. Kleene. *Introduction to Metamathematics.* Van Nostrand, Princeton, NJ, 1952.

[14] Stephen C. Kleene, Recursive Functionals and Quantifiers of Finite Type I. *Trans. Amer. Math. Soc.* vol. 91 (1959), pages 1–52.

[15] Per Martin-Lõf, Constructive Mathematics and Computer Programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science.* North–Holland, Amsterdam, 1982, pages 153–175.

[16] Yiannis N. Moschovakis, Abstract recursion as a foundation for the theory of algorithms. *Computation and proof theory* (Aachen, 1983), *Lecture notes in Mathematics*, vol. 1104. Springer-Verlag, New York, 1984, pages 289–364.

[17] Richard A. Platek, *Foundations of Recursion Theory*, Doctoral Dissertation, Stanford University, 1966.

[18] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, New York, 1967.

[19] Scott F. Smith, *Partial objects in type theory*, Doctoral dissertation, Cornell University, May 1988 (expected).

[20] Robert I. Soare, *Recursively Enumerable Sets and Degrees.* Springer-Verlag, New York, 1987.

[21] H. Ray Strong, Algebraically Generalized Recursive Function Theory. *IBM J. Res. Devel.* vol. 12, 1968, pages 465–475.

[22] Eric G. Wagner, Uniformly Reflexive Structures: On the nature of Gõdelizations and Relative Computability. *Trans. Amer. Math. Soc.*, v. 144 (1969), pages 1–41.