Theoretical Computer Science 121 (1993) 89–112
Elsevier

Computational foundations of basic recursive function theory*

Robert L. Constable

Computer Science Department, Cornell University, Ithaca, NY 14853-7901, USA

Scott F. Smith

Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218, USA

Abstract

Constable, R.L. and S.F. Smith, Computational foundations of basic recursive function theory, Theoretical Computer Science 121 (1993) 89–112.

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 theorems in recursion theory may be classified according to which axioms of computation are needed to prove them.

The theory is about typed functions over the natural numbers, and it includes theorems showing that 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 1930s 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 of computability, as well as by a belief in

Correspondence to: R.L. Constable, Computer Science Department, Cornell University, Ithaca, NY 14853–7901, USA. Email: rc@cs.cornell.edu.

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

Church's thesis, which in turn leads to a well-developed theory of unsolvability [8]. This *standard theory* accepts Church's thesis, and it is explicitly used by Roger [26] to develop the theory. We want to present an alternative view.

We have discovered through our attempts to provide a formal foundational theory for computer science [5, 6, 29, 30] that there is an interesting 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, random access machines (RAMs), specify too much irrelevant and ad hoc detail. Some abstract approaches [10, 32, 33] 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 at is the theories developed in computer science to reason about functional programs. One such theory is LCF [15], essentially a theory of typed functional programs (much like those in the language PCF [25]) based on Scott's domain theory [28]. Others are the type theories over the natural numbers such as [21, 14] and Nuprl [6], foundational theories for mathematics which can be interpreted computationally. 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, but the surprising result is that there is an interesting notion of unsolvability even if we deny them.

LCF is based on the notion of continuous functions over domains and builds in a partial order relation as a primitive. This means that one can appeal to continuity arguments to establish undecidability. Such a theory departs radically from basic recursive function theory. Among the constructive type theories, only Nuprl thus far offers a theory of partial functions that can serve as a basis for recursion theory [5]. Here we present a simplified version of part of that theory, omitting the notion of computational induction.

In some sense indexing-free approach to unsolvability was known to Church and Kleene at the dawn of the subject because they developed computability theory first in the context of the untyped λ-calculus, as a theory of λ-definability. There was no need for indexings in this theory in order to achieve self-reference nor in order to contemplate the existence of functions to decide the halting problem. In fact Kleene has said [20] that his development of the recursion theorem arose from translating the result from the λ-calculus, where it is almost immediate once the Y-combinator is known,

into the μ-recursion formalism. Note that Kleene even used the same notation $\{e\}(a)$ for both theories, meaning the application of function e to argument a in the λ-calculus and meaning the application of the eth partial recursive function to argument a in his indexed theory of recursive functions.

The unsolvability argument in the untyped λ-calculus serves as an introduction to our theory. First, a quick review of the λ-calculus is in order. $\lambda x.b$ is the representation of a function in the λ-calculus: x is a variable which is the parameter to the function, and b is the body, which can be an arbitrary expression. $f(a)$ denotes application of function f to argument a. For instance, $\lambda y.(\lambda x.y(x(x)))(\lambda x.y(x(x)))$ is a λ-term. This is in fact a special term, the Y-combinator; it is a fixed point combinator, i.e. $Y(f)=f(Y(f))$ for any function f.

Theorem 1.1. *There is no λ-definable function to decide halting in the untyped λ-calculus.*

Proof. Suppose there existed a function h such that $h(x)=1$ if x halted, and $h(x)=0$, otherwise. Define $d=Y(\lambda x.$ if $h(x)=1$ then diverge else 0). Using the fact that $Y(d)=d(Y(d))$, $d=$ if $h(d)=1$ then diverge else 0. Consider how d executes: if $h(d)=1$ then the if-test succeeds, so d will diverge, but this contradicts the definition of h! Likewise, if $h(d)=0$ then $d=0$, again contradicting the definition of h. Therefore, h cannot exist. □

Consider three approaches to unsolvability. The most basic is the version presented directly above, involving a notion of function that permits direct self-application. Functions in classical set theory cannot take themselves as arguments, and the usual typing of functions (as in Russell's type theory for instance) precludes self-reference as well. In order to present the λ-calculus arguments in a conventional mathematical setting, Kleene introduced a second approach based on the concept of an *indexing* (or Gödelization, as he saw it) on which the standard recursion theory is based.

In this paper we offer a third approach to these ideas based on the concept that computations can be treated as objects and typed. We modify the above argument by directly adding a fixed-point operator, *fix*, which avoids the use of self-reference necessary to define Y in the untyped λ-calculus. This allows the above computation d to be typed, and the unsolvability of the halting problem may then be proven (see Section 3.1).

2. A theory of computing

2.1. Nature of the theory

Although the concept of an *algorithm* is central, we treat it as a metanotion in this theory, i.e. algorithms are those terms which denote functions. 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 it is ultimately desirable to capture

such concepts in this framework. 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. This is thus an *extensional* notion of equality. For simplicity and for comparison to the standard theories, we have three base types, **N**, the nonnegative integers 0, 1, 2, ...; **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. Thus far, this theory is closely related to the typed λ-calculus [16] and to PCF [25].

The types defined above are basic; in addition, associated with each type T is it's "bar type", denoted $\bar{T}$. Intuitively, $\bar{T}$ 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 of elements, 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 [21], Nuprl [6] and in Section 2.4 below. It is also possible to provide a concrete *computational semantics* for the theory by defining an operational relations $s \leftarrow t$ to mean that s is the result of evaluating or computing t, and then defining type membership and equality using this notion of computation [1]. Such an interpretation is given 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 such *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 could be that of a *mental construction*. In 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*, and *types*.

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*,
$s; t$	*sequentialization*,

$succ(r)$ *successor*,

$pred(r)$ *predecessor*,

$zero(r; s; t)$ a *decision term*, and

$fix(f)$ the *fixed point term*.

$\lambda x.t$ *binds* free occurrences of x in t. x occurs free in the body of the function, t, if it appears, yet is not bound by yet another λ therein. We use notation $b[a/x]$ to express the act of taking the term b and replacing all free occurrences of variable x by the term a, being careful to rename bound variables in b to avoid free variables in a becoming bound (capture). A term is closed if it has no free variables. For this paper, small letters except w–z denote terms, and w–z denote variables. Sequentialization $s;t$ denotes the execution of s followed by the execution of t. *zero* is a test of r for 0 value, returning s if r is 0 and t if r is some other number. The precise meanings of the terms will be made clear below. Note, this language is essentially call-by-name PCF [27, 25] with the addition of a sequencing operator, $s;t$.[1]

Associated with terms are the *base types*

N, the *natural numbers*,

1, the *unit type*,

2, the *boolean type*,

and, inductively, if S and T are types, then

$S \to T$ the *function space*, and

$\bar{S}$ the *bar type*,

and also types, provided that S itself is not a bar type in the second clause. In this paper, capital letters denote types.

2.4. The theory

Meaning is given to the types and terms via assertions. A collection of axiomatic principles is then given which defines the precise meaning of the assertions. The syntax, assertions, and principles together define the theory of computability that is the center of this paper.

[1] Sequencing is needed because we adopt a call-by-name semantics. In PCF which uses call-by-value, the effect of $s;t$ is accomplished by $(\lambda x.t)(s)$ where x is not free in t.

We may assert the following properties of types and terms.

- $s=t\in T$, meaning terms s and t are equal members of type T,
- $t\in T$, meaning t is a member of T and in fact defined, in terms of the previous assertion, as $t=t\in T$,
- $t\downarrow$, meaning a converges, and
- $t\uparrow$, meaning a diverges, in fact defined as meaning a does not converge.

The axiomatic principles for deriving truths in the theory are as follows.

Function introduction: If $b[t/x]=b'[t'/x']\in B$ for arbitrary $t=t'\in A$, then $\lambda x.b=\lambda x'.b'\in A\to B$.

Function elimination: If $f=f'\in A\to B$ and $a=a'\in A$, then $f(a)=f'(a')\in B$.

Bar introduction: If $a\downarrow$ iff $a'\downarrow$, and $a\downarrow$ implies $a=a'\in A$, then $a=a'\in\bar{A}$.

Bar elimination: If $a=a'\in\bar{A}$ and $a\downarrow$, then $a=a'\in A$.

Fixed point: If $f=f'\in\bar{A}\to\bar{A}$, then $fix(f)=fix(f')\in\bar{A}$.

Equality: $-=-\in-$ is a partial equivalence relation, i.e. it is transitive and symmetric.

(N, 1, 2): **N** is a type of natural numbers 0, 1, 2, ..., **2** is the subtype of **N** with members 0 and 1, and **1** is the subtype of **N** with member 0. Principles on these objects are taken as givens, e.g. *pred* and *succ* compute predecessors and successors (the predecessor of zero is zero), and induction on numbers is taken to be sound.

Logic: Constructive principles of logical reasoning may be used. Shorthand notation for logical expressions include "$\forall t:T\ldots$" meaning "for all t in type $T\ldots$", "$\exists t:T\ldots$" meaning "there exists a t in type $T\ldots$", & meaning and, $\vee$ meaning or, $\Rightarrow$ meaning implies, and $\Leftrightarrow$ meaning if and only if.

Beta: $(\lambda x.b)(a)=b[a/x]\in A$.

Fix: $fix(f)=f(fix(f))\in A$.

Sequence: $a;b=b\in A$, provided $a\downarrow$.

Strictness: If any of $succ(a)$, $pred(a)$, $a(b)$, $zero(a;b;c)$, $zero(0;a;b)$, or $zero(n;b;a)$ (where n is 1, 2, ...) terminates, then a also must terminate.

Value: $a\downarrow$ if $a\in\mathbf{N}$ or if $a\in A\to B$.

Divergence: $\neg fix(\lambda x.x)\downarrow$.

For instance, $fix(\lambda x.succ(x))\in\bar{\mathbf{N}}$ may be shown as follows: first, recall that $t\in T$ is defined as $t=t\in T$. By the fixed point principle, it then suffices to show $\lambda x.succ(x)\in\bar{\mathbf{N}}\to\bar{\mathbf{N}}$, which follows from the function introduction principle if under the assumption $n\in\bar{\mathbf{N}}$ we may show $succ(n)\in\bar{\mathbf{N}}$. From bar introduction, suppose $succ(n)\downarrow$ and show $succ(n)\in\mathbf{N}$: by strictness $n\downarrow$, so by bar elimination, $n\in\mathbf{N}$, so by a basic property of numbers, $succ(n)\in\mathbf{N}$. It is also useful to observe that $fix(\lambda x.succ(x))\uparrow$, because if it converged, by bar elimination and the above derivation it would be a number, but it corresponds to no natural number, as can easily be verified by induction.

Consider also the example of $\perp;3$ where $\perp$ is the term known to diverge, $fix(\lambda x.x)$. This element belongs to *any bar type* since assuming that it converges implies, by the sequence rule, that $\perp$ converges; this contradicts the divergence rule. The above rules

are not complete for the computational semantics we give below, but they are enough to develop the computability theory we want.

The most important rule for our purposes is the fixed point rule. A wide collection of partial functions may be typed with this rule, including all partial recursive functions.[2] The rule is also critical in the proof of the existence of unsolvable problems: in a theory with this rule removed, it would be possible to interpret the function spaces to be classical set-theoretic functions. The fixed point principle is powerful, and it can in fact be too powerful in some settings: in a full type theory such as Nuprl, it is inconsistent to allow all functions to have fixed points. There, the principle must be restricted to take fixed points over a collection of *admissible* types only [29]. In this theory the type structure is simple enough that all types are admissible.

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. This semantics shows the principles given in the previous section to be sound.

To evaluate computations, some notion of a *machine* is necessary; a relation is defined for this purpose. Let $v \leftarrow t$ mean that term v is the *value* of executing or reducing t using a sequence of head-reductions.[3] This relation is defined in Fig. 1. Note that the only possible values in this computation system are numbers and lambda abstractions. Also note that if the conditions for reduction are not met, then the computation *aborts*; it does not diverge. For instance, $a(c)$ will abort if a does not

$v \leftarrow t$ is the least-defined relation having the following properties	
$n \leftarrow n$	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; 0 minus 1 is 0
$v \leftarrow zero(a; b; c)$	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 $v \leftarrow b$

Fig. 1. Evaluation.

[2] It is easy to see that all partial recursive functions from **N** to **N** are typeable, for example all the μ-recursive definitions can be immediately translated into these terms.

[3] Head-reduction corresponds to call-by-name semantics for function calls.

evaluate to a λ-term. Thus 2(1) does not compute to anything, nor does $succ(\lambda x.x)$. For this presentation, as in [26] we treat aborting computation as divergent. We could distinguish abortion as a separate case without changing the results, but this would be an unnecessary complication.

For example, letting f be $\lambda y.\lambda x.zero(x; 0; y(pred(x)))$, we know $fix(f)(1)$ computes as follows:

$0 \leftarrow fix(f)(1)$ iff

$0 \leftarrow \lambda x.zero(x; 0; fix(f)(pred(x)))(1)$ iff

$0 \leftarrow zero(1; 0; fix(f)(pred(1)))$ iff

$0 \leftarrow fix(f)(pred(1))$ iff

$0 \leftarrow \lambda x.zero(x; 0; fix(f)(pred(x)))(pred(1))$ iff

$0 \leftarrow zero(pred(1); 0; fix(f)(pred(pred(1))))$ iff

$0 \leftarrow zero(0; 0; fix(f)(pred(pred(1))))$ iff

$0 \leftarrow 0$, which is obvious.

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

Define termination $t\downarrow$ as $(s \leftarrow t)$ for some s.

Definition 2.1. 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 **1**, then $0 \leftarrow s$ and $0 \leftarrow t$,

if T is **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 $\lambda x.b \leftarrow s$ and $\lambda y.b' \leftarrow t$ and for all a and $a' \in A$, $a = a' \in A$ implies $b[a/x] = b'[a'/y] \in B$,

if T is $\bar{A}$, then $(s\downarrow$ iff $t\downarrow)$ and $s\downarrow$ implies $s = t \in A$.

Some simple observations about this definition are now made. $a = b \in \mathbf{N}$ means a and b both evaluate to the same natural number n. If $f \in \mathbf{N} \rightarrow \mathbf{2}$, $\lambda x.b \leftarrow f$ and if $n \in \mathbf{N}$, $b[n/x] \in \mathbf{2}$. Since $f(n)$ and $b[n/x]$ both have the same values when computed, $f(n) \in \mathbf{2}$ as well. f is thus a total function mapping natural numbers to either 0 or 1, as expected. $f \in \mathbf{N} \rightarrow \bar{\mathbf{2}}$, on the other hand, means $f(n) \in \bar{\mathbf{2}}$ for some number n, so by the clause above defining bar types, $f(n)$ could diverge, so the function might not be total. Thus, there are distinct types for partial and total functions.

Theorem 2.2. *For all types T,*

(i) *if $t \in T$ and $u \leftarrow t$ and $u \leftarrow s$, then $t = s \in T$,*

(ii) *if $s = t \in T$ then $t = s \in T$,*

(iii) *if $s = t \in T$ and $t = u \in T$ then $s = u \in T$.*

Theorem 2.3. *The theory is provably sound under the interpretation of the assertions given by the computational semantics.*

This is easy to prove, because each of the principles enumerated in Section 2.4 is valid in the computational semantics (see [29] for the details, applying to an even richer theory). The computational semantics thus gives one sound interpretation of the theory, but this does not preclude other interpretations.

3. Basic results

3.1. Overview

Our plan for this section is to examine certain basic concepts and results from recursive function theory over natural numbers, say, as presented in [26, 31], and to show they have analoges in the theory just defined. We start with undecidability results, then look at analogs 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 there is no term in the theory to solve the "halting problem" for functions $f \in \mathbf{N} \to \bar{\mathbf{N}}$, 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 $\bar{\mathbf{N}}$. So we are equivalently asking for any computation $t \in \bar{\mathbf{N}}$, whether we can tell if t halts, i.e. whether there is a function $h \in \bar{\mathbf{N}} \to \mathbf{2}$ such that $h(t) = 1$ iff t halts. The answer is no, because if we assume that h exists, then we define the function

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

where, as before, $\bot$ is some element of $\bar{\mathbf{N}}$ known to diverge such as $\mathit{fix}(\lambda t.t)$. $d \in \bar{\mathbf{N}}$ follows by the fixed point principle because the body is in the type $\bar{\mathbf{N}} \to \bar{\mathbf{N}}$. By computing the *fix* term, we have $d = \mathit{zero}(h(d); 1; \bot) \in \bar{\mathbf{N}}$. 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 $\mathbf{N}$ 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 element $t_0 \in T$. If we assume there is $h \in \bar{T} \to \mathbf{2}$ such that $h(t) = 1$ iff t converges, then we may define

$$d = f(\lambda x.\mathit{zero}(h(x); t_0; \bot)) \in \bar{T}.$$

The argument makes essential use of the self-referential nature of $\mathit{fix}(f)$, which has the type $\bar{T}$ where f is of type $\bar{T} \to \bar{T}$. This simple unsolvability argument cannot be expressed 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 $\bar{T} \to \mathbf{2}$ 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. In general, the theory unfolds along the lines of basic recursive function theory. In a computation theory based on domains such as LCF, there is an axiom stating that all functions are monotone with respect to the domain partial order. From this axiom it is easy to show that no term of LCF can compute the halting function h above because it is not monotone. Computing theory done this way does not bear such a strong resemblance to recursive function theory.

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; details follow.

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 $\bar{\mathbf{N}}$; this is written as $\{x{:}\bar{\mathbf{N}} \mid x\downarrow\}$. Although such classes can be defined formally, say in type theory [4, 21] 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 $\{x{:}T \mid P(x)\}$ where $P(x)$ is a predicate in x. We say $t \in \{x{:}T \mid P(x)\}$ for $t \in T$ when $P(t)$ holds.

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 3.1. 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 3.2. Let $K_{\bar{T}} = \{x{:}\bar{T} \mid x\downarrow\}$.

Theorem 3.3 (Unsolvability). *For all types T which have members, meaning some $t_0 \in T$, $K_{\bar{T}}$ is not decidable.*

Proof. See Section 3.1. □

The class of diverging computations is also not decidable.

Definition 3.4. Let $divK_{\bar{T}} = \{x{:}\bar{T} \mid x\uparrow\}$.

Theorem 3.5. *For any type T with members, $divK_{\bar{T}}$ is not decidable.*

Proof. This is just like Theorem 3.3; assume h decides membership and look at $d = fix(\lambda x.zero(h(x); \bot; t_0))$ where $t_0 \in T$. □

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

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

of the functions that halt on at least one of their inputs, is not decidable. To see this, suppose it were decidable. Then we could decide $K_{\bar{T}}$ because for each $x \in \bar{T}$ we can build an $f \in S \to \bar{T}$ 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 $h \in (S \to \bar{T}) \to \mathbf{2}$ decides $W_{S\to\bar{T}}$ then $\lambda x.h(\lambda y.x) \in \bar{T} \to \mathbf{2}$ decides $K_{\bar{T}}$. We have proved:

Theorem 3.6 (Weak halting). *For any types S and T with members, $\{f{:}S\to\bar{T} \mid \exists x{:}S.\, f(x)\downarrow\}$ is not decidable.*

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

Definition 3.7. Class C_S is *reducible* to class C_T, written $C_S \preceq C_T$, iff there is a function $f \in S \to T$ such that $\forall x{:}S.\ x \in C_S \Leftrightarrow f(x) \in C_T$.

Fact 3.8. $\preceq$ *is reflexive and transitive.*

For Theorem 3.6, the mapping function is $f = \lambda x.\lambda y.x$. When reducing to a class over a bar type, say $C_{\bar{T}}$, the reduction function $f \in S \to \bar{T}$ 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 $\mathbf{1} \to \bar{T}$. That is, given $f \in S \to \bar{T}$, replace it by $g \in S \to (\mathbf{1} \to \bar{T})$ where $g(x) = \lambda y.f(x)$, and y does not occur free in f. This gives an equivalent total reduction

because $t \in C_{\bar{T}} \Leftrightarrow \lambda x.t \in C_{1 \to \bar{T}}$: 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 3.9. 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 3.10 (Rice). *For all types T, $C_{\bar{T}}$ is decidable iff $C_{\bar{T}}$ 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 $f \in \bar{T} \to \mathbf{2}$ decides $C_{\bar{T}}$. Since f is total, $f(\perp)=0$ or $f(\perp)=1$; show for the case $f(\perp)=0$ that the class must be minimal, and for $f(\perp)=1$ that it must be maximal.

Case $f(\perp)=0$: Show C is trivial by showing it is minimal, i.e. $\forall t{:}\bar{T}.\, f(t)=0$. Let $t \in \bar{T}$ be arbitrary. We may show $f(t)=0$ arguing by contradiction because the equality is decidable. So, assume $f(t) \neq 0$. $divK_{\bar{T}}$ may then be shown to be decidable using the function

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

For h to characterize $divK_{\bar{T}}$ 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 $h(x)=f(x;t)=f(\perp)=0$. $divK_{\bar{T}}$ is not decidable by Theorem 3.5, so we have a contradiction.

Case $f(\perp)=1$: Show C is maximal, i.e. $\forall t{:}\bar{T}.\, f(t)=1$. This case is similar to the previous except that the output of the reduction function h is switched to make it

$$h = \lambda x.zero(f(x;t);\, 1;\, 0) \in \bar{T} \to \mathbf{2}. \qquad \square$$

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. The following definition sets forth the idea in precise terms.

Definition 3.11. A class C_T is *converge-acceptable* or just *acceptable* iff

$$\exists f{:}T\to\bar{\mathbf{1}}.\ \forall x{:}T.\ x\in C_T \Leftrightarrow f(x)\downarrow.$$

A class C_T is *diverge-acceptable* iff

$$\exists f{:}T\to\bar{\mathbf{1}}.\ \forall x{:}T.\ x\in C_T \Leftrightarrow f(x)\uparrow.$$

The canonical acceptable class is $K_{\bar{T}}$, and we may now prove the following theorem.

Theorem 3.12. *For all types T, $K_{\bar{T}}$ is acceptable.*

Proof. The accepting function f is $\lambda x.(x;0)\in\bar{T}\to\bar{\mathbf{1}}$, which converges exactly when its argument x converges. □

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 for 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 analog of a complement.

Theorem 3.13. *For any type T with members, $divK_{\bar{T}}$ is diverge-acceptable.*

Proof. The diverge-acceptor function f is $\lambda x.(x;0)\in\bar{T}\to\bar{\mathbf{1}}$. □

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

Theorem 3.14. *For any type T with members,*
(i) *$K_{\bar{T}}$ is not diverge-acceptable.*
(ii) *$divK_{\bar{T}}$ is not acceptable.*

Proof. For (i), suppose f diverge-accepted K_T and $t_0\in T$; define

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

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

3.5. Unions and intersections

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

Definition 3.15.

$$c \in \sim A_T \quad \text{iff } c \in T \;\&\; c \notin A_T,$$

$$c \in A_T \cup B_T \text{ iff } c \in A_T \vee c \in B_T,$$

$$c \in A_T \overset{w}{\cup} B_T \text{ iff } \neg(c \notin A_T \;\&\; c \notin B_T),$$

$$c \in A_T \cap B_T \text{ iff } c \in A_T \;\&\; c \in B_T.$$

The weak union $c \in A_T \overset{w}{\cup} B_T$ is useful because it is not always possible to form a strong union constructively; this 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 3.16 (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 $A_T \cap B_T$. □

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 also 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 is the following theorem (proved above).

Theorem 3.17 (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 $K_{\mathbf{N}}$ 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. polynomial time complete sets. Here, there is also an interesting notion of completeness.

Definition 3.18. 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 3.19 (Complete classes). *For all nonempty types T,*

(i) $K_{\bar{T}}$ *is acceptably-complete and*

(ii) $divK_{\bar{T}}$ *is diverge-acceptably complete.*

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

$$m = \lambda s.(g(s); t_0) \in S \to \bar{T}.$$

For arbitrary $s \in S$, it must be that $s \in D_S \Leftrightarrow m(s) \in K_{\bar{T}}$, 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 $t_0 \in K_{\bar{T}}$).

($\Leftarrow$) $f(m(s))\downarrow$ means $m(s)\downarrow$ since f characterizes $K_{\bar{T}}$, so $g(s); t_0\downarrow$, meaning $g(s)\downarrow$.

(ii) This proof is similar to that of (i). □

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. These extensions are separate, because it may be desirable not to accept certain of them. 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 nonmathematical 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 defnable in this theory, but it does not account for the self-referential nature of computation, and there are no inherently unsolvable problems like those found here.

4.1. Uniformity principles

There are two *uniformity principles* which allow functions applied to diverging computations to be more precisely characterized. Let $\uparrow = \bot$, then:

$$\forall f{:}\bar{A}\to\bar{B}.\ f(\uparrow)\downarrow \ \Rightarrow\ \forall a{:}\bar{A}.f(a)\downarrow, \tag{I}$$

$$\forall f{:}\bar{A}\to\bar{B}.\ f(\uparrow)\uparrow \ \Rightarrow\ \forall a{:}\bar{A}.(f(a)\downarrow \Rightarrow a\downarrow). \tag{II}$$

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

Theorem 4.1. *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 $\forall a{:}\bar{A}.f(a)\downarrow$.

(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. □

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. Markov's principle is not constructively valid, but those readers who accept classical principles of reasoning can take Theorem 4.2 as an unconditional proof of the uniformity principles.

Theorem 4.2. *Markov's principle* $\Rightarrow$ I & II.

Proof. (I) Take an arbitrary $f\in\bar{A}\to\bar{B}$, with $f(\uparrow)\downarrow$. Suppose $f(a)\uparrow$ for arbitrary $a\in\bar{A}$; 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 $K_{\bar{A}}$ is diverge-acceptable. Define its diverge-accepting function $h=\lambda x.f(x;a)$; $0\in\bar{A}\to\bar{\mathbf{1}}$. We only need to show

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

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 $K_{\bar{A}}$ is not diverge-acceptable (Theorem 3.14). 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. □

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

Definition 4.3. C_T is strongly nontrivial $\Leftrightarrow \exists t_0{:}T.\ t_0\in C_T\ \&\ \neg\forall t{:}T.\ t\in C_T$.

Theorem 4.4 (Acceptability characterization). *$C_{\bar{T}}$ is acceptable and $C_{\bar{T}}$ is strongly nontrivial $\Rightarrow$ $C_{\bar{T}}$ is acceptably-complete.*

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

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

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

($\Leftarrow$) Suppose $f_C(f_D(t);t_0)\downarrow$; by uniformity II, that means $f_D(t);t_0\downarrow$, so $f_D(t)\downarrow$. □

Using Rice's theorem (Theorem 3.10), we may prove

Corollary 4.5. *For all types T, $C_{\bar{T}}$ is acceptable $\Rightarrow$ $C_{\bar{T}}$ is decidable $\vee_C$ $C_{\bar{T}}$ is acceptably complete.*[4]

Proof. For acceptable C, this is equivalent to proving

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

By Theorem 4.4, we have

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

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

and the corollary will thus follow from

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

We prove this by showing

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

and

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

both of which follow by straightforward propositional reasoning. □

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. Here, we only give the computational semantics and prove facts at the semantic level, but an axiomatization is also possible.

Definition 4.6. Define a new computation relation $v \overset{\|}{\leftarrow} t$ which has all of the clauses of Fig. 1, and with the additional clause

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

The computation relation $\overset{\|}{\leftarrow}$ is not a function: $1 \overset{\|}{\leftarrow} 1 \| 2$, and $2 \overset{\|}{\leftarrow} 1 \| 2$. 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 4.7. Redefine $v \leftarrow t$ as follows:

$$v \leftarrow t \text{ iff } v \overset{\|}{\leftarrow} t \;\&\; \forall v'.\, v' \overset{\|}{\leftarrow} t \Rightarrow v' \text{ is } v.$$

Redefine $t\downarrow$ as

$$t\downarrow \text{ iff } \exists v.\, v \overset{\|}{\leftarrow} t.$$

The type system over this computation system is then defined as in Definition 2.1. It is possible to dovetail computations where one or both may have no value at all; this is reflected in

Fact 4.8. $a \| b \in \bar{T}$ *if* $a \in \bar{T}$ & $b \in \bar{T}$ & $(a\downarrow$ & $b\downarrow \Rightarrow a$ *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 4.9. 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* $\lambda x. f_C(x) \| f_D(x) \in T \to \bar{\mathbf{1}}$. □

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 4.10. $\{g : \mathbf{N} \leftarrow \bar{\mathbf{N}} \mid \exists n : \mathbf{N}.\ g(n)\downarrow\}$ *is acceptable.*

Proof. The accepting function is

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

which computes to

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

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

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, as the following theorem demonstrates.

Theorem 4.11. *Markov's principle implies*

$$C_T \text{ is acceptable and } C_T \text{ is diverge-acceptable} \Rightarrow C_T \text{ is decidable.}$$

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

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

Before proceeding, we check to make sure r is of the indicated type. For arbitrary x, we wish to show $(f(x);1) \| (g(x);0) \in \bar{\mathbf{2}}$. Using Fact 4.8, we only need to show

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

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 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, $r \in T \to \mathbf{2}$. It is easy to see that r in fact decides C. □

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 4.12. Define the following evaluation relations:

$$v \overset{n}{\leftarrow} t \text{ iff } t \text{ evaluates to } v \text{ in } n \text{ or fewer steps}$$

$$t\downarrow^n \text{ iff } \exists v.v \overset{n}{\leftarrow} t$$

$$t\uparrow^n \text{ iff } \neg t\downarrow^n.$$

$\leftarrow$ is then redefined as

$$v \leftarrow t \text{ iff } \exists n.v \overset{n}{\leftarrow} t.$$

The type system is defined as in Definition 2.1, using this new notion of evaluation. The equality relation and the definition of types is just as in Definition 2.1 (thus equality is extensional). But the computation theory is now nonextensional, because computations have a property besides their value, their step count. 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 an *untyped* term to the computation system to count steps; but we cannot type it. Such a counter would diverge if the computation diverged, so instead we add a more powerful total term:

Definition 4.13. Extend the definition of computation in Fig. 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 4.14. *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 4.15. $a||b = fix(\lambda d.\lambda n.zero(comp(a)(n);\ zero(comp(b)(n);\ d(n+1);\ b);\ a))(0)$

This function returns whichever of a or b first terminates, and is typed as in Section 4.2.

It is now possible to prove that some classes which do not involve bar types are unsolvable because certain uses of *comp* are typeable:

Definition 4.16.

$$V = \{f{:}\mathbf{N}\to\mathbf{N} \mid \exists n{:}\mathbf{N}.\ f(n) = 1 \in \mathbf{N}\},$$

$$divV = \{f{:}\mathbf{N}\to\mathbf{N} \mid \forall n{:}\mathbf{N}.\ f(n) = 1 \in \mathbf{N}\}.$$

It is easy to show

Fact 4.17. *V is acceptable and $divV$ is diverge-acceptable.*

More importantly, we may prove

Theorem 4.18. *V is not decidable.*

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

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

We assert

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

($\Rightarrow$) 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 4.14.

($\Leftarrow$) Suppose $h(x) = 1$, meaning $\exists n{:}\mathbf{N}.\ comp(x)(n) = 1 \in \mathbf{N}$, so $x\downarrow$. □

The intensional nature of *comp* opposes the extensional nature of functions in this theory, which restricts the uses of *comp* which can be typed, and hence 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 $\bar{\mathbf{N}}\to(\mathbf{N}\to\mathbf{N})$. This is because $x = y \in \bar{\mathbf{N}}$ does not mean x and y have an equal number of computation steps, so $m(x)$

might be different from $m(y)$. To fully incorporate *comp* and other possible principles for keeping track of computational resources, the type system must then be extended to allow nonextensional functions, but that task is beyond the scope of this paper.

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, absent in our approach. We mention here two different approaches.

Wagner [33] has developed an algebraic account of computability, the *Uniformly Reflexive Structure*, or URS. This theory was elaborated and extended by Strong [32], Friedman [12], 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 can be proved.

Another account which has more resemblance to this work is Platek's inductive definability approach to recursion theory [24], further expounded by Feferman [10] and Moschovakis [22]. 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 fixed points, 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 they prove the universal machine and S–m–n theorems. This approach is more *ad hoc* than a foundational theory should be.

6. Conclusions

The constructive recursive function theory (CRFT) of this paper is quite different from the standard theory. More importantly, the standard theory assumes an indexing of all partial recursive functions, which allows the universal machine theorem to be proved. It also uses Church's thesis to confer absoluteness and relevance to the results. In CRFT it is the fixed point principle, a more directly self-referential fact than the universal machine theorem, which leads to unsolvable problems. The fixed point principle gives basic unsolvability results, and each additional assumption gives rise to another collection of theorems, as our results demonstrate. As suggested in the introduction, it is also possible to study unsolvability in the untyped λ-calculus, where there can be self-reference without indexings and the results are abstract.

A development of recursion theory similar to that of this paper could also be undertaken in such a setting.

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 accounts for each type.

CRFT also impacts type theory because the types cannot now be given purely classical interpretations: if $A \rightarrow \bar{B}$ were all set-theoretic functions from A to $B \cup \{\uparrow\}$, fixed points 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 the 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 does not, which contradicts the unsolvability of the halting problem.

Acknowledgment

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

References

[1] S.F. Allen, A non-type-theoretic semantics for type-theoretic language, Ph.D. Thesis, Computer Science Department, Cornell Univ., 1987. Also as Computer Science Department Tech. Report, TR 87–866, Cornell Univ., Ithaca, NY, 1987.

[2] H. Barendregt, Normed uniformly reflexive structures, in: *Lambda-calculus and Computer Science Theory*, Lecture Notes in Computer Science, Vol. 37 (Springer, Berlin, 1975) 272–286.

[3] D.A. Basin, An environment for automated reasoning about partial functions, in: *9th Internat. Conf. On Automated Deduction*, Lecture Notes in Computer Science, Vol. 310 (Springer, Berlin, 1988) 101–110.

[4] R.L. Constable, A constructive theory of recursive functions, Computer Science Department Technical Report, TR 73–186, Cornell University, Ithaca, NY, 1973.

[5] R.L. Constable and S.F. Smith, Partial objects in constructive type theory, in: *Symp. on Logic in Computer Science* (1987) 183–193.

[6] R.L. Constable et al., *Implementing Mathematics with the Nuprl Proof Development System* (Prentice-Hall, Englewood Cliffs, NJ, 1986).

[7] T. Coquand and G. Huet, Constructions: A higher order proof system for mechanizing mathematics, EUROCAL 85, Linz, Austria, April 1985.

[8] M. Davis, ed., *The Undecidable* (Raven Press, Hewlett, NY, 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, New York, 1981) 397–420.

[10] S. Feferman, Inductive schemata and recursively continuous functionals, in: *Logic Colloquium '76* (North-Holland, Amsterdam, 1977) 373–392.

[11] J.E. Fenstad, *Recursion Theory: An Axiomatic Approach* (Springer, Berlin, 1980).
[12] H. Friedman, Axiomatic recursive function theory, in: *Logic Colloquium '69* (North-Holland, Amsterdam, 1974) 385–404.
[13] J.Y. Girard, Une extension de l'interprétation de Gödel a l'analyse, et son application a l'élimination des coupures dans l'analyse et la théorie des types, in: J.E. Fenstad, ed., *Proc. 2nd Scandinavian Logic Symposium* (North-Holland, Amsterdam, 1971) 63–92.
[14] J.-Y. Girard, Y. Lafont and P. Taylor, *Proofs and Types* (Cambridge University Press, Cambridge, 1988).
[15] M. Gordon, R. Milner and C. Wadsworth, Edinburgh LCF: a mechanized logic of computation. Lecture Notes in Computer Science, Vol. 78 (Springer, New York, 1979).
[16] J.R. Hindley and J.P. Seldin, *Introduction to Combinators and λ-Calculus* (Cambridge University Press, Cambridge, 1986).
[17] C.A.R. Hoare and D.C.S. Allison, Incomputability, *Computing Surveys*, Vol. 4, 1972, 169–178.
[18] S.C. Kleene, *Introduction to Metamathematics* (Van Nostrand, Princeton, NJ, 1952).
[19] S.C. Kleene, Recursive functionals and quantifiers of finite type I, *Trans. Amer. Math. Soc.* **91** (1959) 1–52.
[20] S.C. Kleene, Origins of recursive function theory, in: *Proc. 20th Annual Symp. on the Foundations of Computer Science* (IEEE, New York, 1979) 371–382.
[21] P. Martin-Löf, Constructive mathematics and computer programming, in: *Proc. 6th Internat. Cong. for Logic, Methodology, and Philosophy of Science* (North-Holland, Amsterdam, 1982) 153–175.
[22] Y.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, New York, 1984) 289–364.
[23] E. Palmgren, Domain interpretations of Martin-Löf's partial type theory, *Ann. Pure Appl. Logic* **48** (1990) 135–196.
[24] R.A. Platek, Foundations of recursion theory, Ph.D. Thesis, Stanford University, 1966.
[25] G.D. Plotkin, LCF considered as a programming language, *Theoret. Comput. Sci.* **5** (1977) 223–257.
[26] H. Rogers, Jr., *Theory of Recursive Functions and Effective Computability* (McGraw-Hill, New York, 1967).
[27] D.S. Scott, A type-theoretical alternative to ISWIM, CUCH, OWHY, *Theoret. Comput. Sci.* **121** (1993) 411–440, this volume.
[28] D. Scott, Data types as lattices, *SIAM J. Comput.* **5** (1976) 522–587.
[29] S.F. Smith, Partial objects in type theory, Tech. Report 88-938, Ph.D. Thesis, Department of Computer Science, Cornell University, 1988.
[30] S.F. Smith, Partial objects in constructive type theory, submitted.
[31] R.I. Soare, *Recursively Enumerable Sets and Degrees* (Springer, New York, 1987).
[32] H. Ray Strong, Algebraically generalized recursive function theory, *IBM J. Res. Develop.* **12** (1968) 465–475.
[33] E.G. Wagner, Uniformly reflexive structures: On the nature of Gödelizations and relative computability, *Trans. Amer. Math. Soc.* **144** (1969) 1–41.