1. Honesty is Needed

Computations manipulate representations of objects, most often strings of symbols taken from some finite alphabet, rather than the objects themselves. Numbers, for example, are usually denoted by sequences of decimal, or binary, or unary digits. In logic, one therefore distinguishes between numbers $n$, which reside in an ideal Platonic world, and numerals $\text{alt1}_n$, their symbolic representation as (first-order) terms. Similarly, graphs, which are set-theoretic objects, are either represented as lists of edges (pairs of nodes) or as a binary adjacency matrix.

Given that representation is a necessity, some natural questions arise immediately:

- How much of a difference can the choice of representation make to computability or complexity measurements. ($Answer$: It can make all the difference between computable and uncomputable, or between tractable and intractable.)
- Who gets to choose the representation: Abe who formulates the queries, or Cay who designs the program to answer them? ($Our answer$: Cay may reinterpret Abe’s formulation any way she sees fit, but the reinterpretation is part and parcel of the answering process.)
- What is wrong with a representation of graphs that lists nodes in the order of a Hamiltonian path, if there is such—in which case deciding the question takes linear time? ($Answer$: Cay will only be able to quickly answer the question about a Hamiltonian path, whereas she would have a hard time performing basic graph operations, such as adding an edge.)
Is it right to say that the parity of an integer (that is, whether it is even or odd) can be determined in constant time, when that is the case only for very specific representations of numbers (least-significant-first binary, as opposed to ternary, say)? (Short answer: No.)

Garey and Johnson [9, pp. 9–10] address the questions of representation and computational models, as they impact the measurement of computational complexity. They assert that it matters little, as long as one sticks to what is considered “reasonable”:

The intractability of a problem turns out to be essentially independent of the particular encoding scheme and computer model used for determining time complexity.

They go on to explain why at length:

Let us first consider encoding schemes. Suppose for example that we are dealing with a problem in which each instance is a graph. . . . Such an instance might be described by simply listing all the vertices and edges, or by listing the rows of the adjacency matrix for the graph, or by listing for each vertex all the other vertices sharing a common edge with it (a “neighbor” list). Each of these encodings can give a different input length for the same graph. However, it is easy to verify that the input lengths they determine differ at most polynomially from one another, so that any algorithm having polynomial time complexity under one of these encoding schemes also will have polynomial time complexity under all the others. In fact, the standard encoding schemes used in practice for any particular problem always seem to differ at most polynomially from one another. It would be difficult to imagine a “reasonable” encoding scheme for a problem that differs more than polynomially from the standard ones.

This discussion is followed by a caveat:

Although what we mean here by “reasonable” cannot be formalized, the following two conditions capture much of the notion:

1. the encoding of an instance $I$ should be concise and not “padded” with unnecessary information or symbols, and
2. numbers occurring in $I$ should be represented in binary (or decimal, or octal, or in any fixed base other than 1).

If we restrict ourselves to encoding schemes satisfying these conditions, then the particular encoding scheme used should not affect the determination of whether a given problem is intractable.
The main concern expressed in the above is that the input size should faithfully reflect the complexity of the input object. The choice of size can make a big difference, of course [3]:

The computational complexity of a problem should not be obscured by a particular representation scheme. Many problems are “fast” under the unary representation, as many computationally (probably) intractable problems in number theory are also “fast” under unary representation, such as factoring, discrete logarithm. But that is not honest complexity theory. The time is really exponential, compared to a more “reasonable” representation scheme of the information, such as in binary. (Italicics ours.)

There are other ways in which a choice of representation may be unreasonable, besides being unnecessarily large. It could give away the answer or harbor hints that make the task easier than it really is. That is the problem with a representation of graphs that lists nodes in Hamiltonian order. Our proposal for measuring complexity honestly will solve this problem by taking into consideration both the cost of computing a function as well as the cost of generating the function’s input.

This article looks at questions of “honesty” of representation in various contexts. We begin with what we feel is the underlying problem posed by representations (Section 2). After proposing a solution (Sections 3 and 4), we consider how it resolves the problem of honest computability (Section 5). Then we turn to see how this also solves the problem of honest complexity (Section 6). With the solution in place, we analyze why considering formal languages, rather than functions, does not work (Section 7) and relate honesty to Martin Davis’s definition of universality (Section 8).

2. Dishonest Representations

The complexity attributed to the computation of a function $f$ over some abstract domain $A$, say graphs, is normally measured in terms of the resources required by its best implementation on some particular model of computation, most commonly the random access machine (RAM). This implementation, however, computes a function $\widehat{f}$ over some other, concrete domain $C$, say binary strings. So, prior to considering the cost of running $\widehat{f}$, one should establish that $\widehat{f}$ does actually implement $f$.

However, there is little meaning to a claim that a single function $\widehat{f}$ over domain $C$ implements a function $f$ over domain $A$ honestly, without also stating how the two domains are related. So let’s say that a concrete (partial) function $\widehat{f} : C \to C$ implements an abstract (partial) function $f : A \to A$ with respect to a particular injective representation $\rho : A \xrightarrow{1:1} C$ if $\rho(f(x)) = \widehat{f}(\rho(x))$ for all $x \in A$. [In the case of partial functions, we also would demand that $\widehat{f}(\rho(x))$ be undefined whenever $f(x)$ is.] See Figure 2. This
definition can be naturally extended to functions with arity wider than 1 and multivalued representations. (See Definitions 1 and 2.)

The choice of representation can make all the difference in the world. If one is not honest, then a computable function can end up implementing an uncomputable one by getting the representation itself to do most of the work. For example, consider any standard enumeration $T_{M_n}, n = 0, 1, \ldots,$ of Turing machines, and define the following uncomputable functions:

- $h : \mathbb{N} \to \mathbb{N}$ enumerates (the numbers of) those machines that halt on the empty tape;
- $\bar{h} : \mathbb{N} \to \mathbb{N}$ enumerates those that do not.

So $h(\mathbb{N}) \cup \bar{h}(\mathbb{N}) = \mathbb{N}$, where $h(\mathbb{N})$ is the image $\{h(n) : n \in \mathbb{N}\}$ of $h$ and $\bar{h}(\mathbb{N})$ is the image of $\bar{h}$. Then the uncomputable function

$$H(n) := \begin{cases} \min h(\mathbb{N}) & \text{if } T_{M_n} \text{ halts} \\ \min \bar{h}(\mathbb{N}) & \text{if } T_{M_n} \text{ does not halt} \end{cases}$$

is implemented by the computable parity function $(n \mod 2)$ under the following bijective representation:

$$\rho(n) := \begin{cases} 2h^{-1}(n) + 1 & \text{if } T_{M_n} \text{ halts} \\ 2\bar{h}^{-1}(n) & \text{if } T_{M_n} \text{ does not halt} \end{cases}$$

The problem with the above “implementation” of the halting function obviously lies in the representation, which clearly gives the impression of itself doing the impossible.

3. Honest Representations

The cause of the problem we just saw with dishonest representation is not the mapping but the lack of context for it. Note that the integer successor function, for example, cannot be implemented by any computable function under the nefarious representation of Section 2, though it is part and parcel of our normal view of the naturals. As we will see, we can allow an honest representation to be any arbitrary injective (multivalued) function as long as we also pay attention to the internal structure of the abstract domain.

Imagine that Abe, the person posing instances of a problem, thinks in terms of an abstract domain $A$, such as integers, graphs, or pictures. Abe must have some means of describing for himself each of the elements of $A$, most commonly by means of a finite set $G$ of generators of $A$ (cf. [12, 13, 2]). These generators give structure to $A$ and meaning to its elements as described by terms $T$ over $G$. For generators to do their job, every element of $A$ must be equal to at least one (ground) term in $T$; so at least one generator must be a scalar constant (of arity 0). (Unlike the development in [2], we are not insisting that the generators form a free term algebra: more than one term can designate the same abstract element.)

Examples of generators for the natural numbers include:
Figure 1. An abstract undirected, unlabeled graph.

0 (nullary zero)
•' (postfix successor \( \lambda n. n + 1 \)),

(in unary “caveman” style), as well as
0 (nullary zero, \( \lambda.0 \), usually suppressed)
•0 (postfix doubling, \( \lambda n. 2n \))
•1 (postfix doubling plus one, \( \lambda n. 2n + 1 \))

for the commonplace binary representation, and
0 (nullary zero, \( \lambda.0 \), usually suppressed)
•0 (postfix tripling, \( \lambda n. 3n \))
•1 (postfix tripling plus one, \( \lambda n. 3n + 1 \))
•2 (postfix tripling plus two, \( \lambda n. 3n + 2 \))

for ternary. With the latter two, there are infinitely many representations of the number zero.

For undirected, unlabeled graphs, \( G \), with vertices \( V \) (\( G \) denotes the set of graphs whose vertices are taken from a set \( V \) of vertices), an example of a set of generators is

\[
\square : V \quad (\text{nullary first-vertex})
\]

\[
\bullet' : V \to V \quad (\text{postfix next-vertex})
\]

\[
\emptyset : G \quad (\text{nullary empty-graph})
\]

\[
\bullet ; \bullet : G \times V \to G \quad (\text{binary add-vertex to graph})
\]

\[
(\bullet) + \bullet ' : G \times V \times V \to G \quad (\text{ternary add-edge to graph})
\]

Over these generators, the graph depicted in Figure 1 is the value of the ground term

\[
(\emptyset ; \square ; \square ' ; \square '') + \square \cdot \square ',
\]

wherein there is an edge between the “first” and “second” vertices, as well as of the term

\[
(\emptyset ; \square '' ; \square ' ; \emptyset) + \square '' \cdot \square ',
\]

wherein there is an edge between the “third” and “second” vertices.

Accordingly, we formalize the notion of honest representation as an injective multivalued function from an abstract domain that is structured by generators. Recall that a multivalued function \( \rho : A \Rightarrow C \) (or set-valued function \( \rho : A \to \mathcal{P}(C) \)) is injective if \( \rho(x) \cap \rho(y) = \emptyset \) for all distinct \( x, y \in A \).

**Definition 1 (Honest Representation).**

- An abstract domain is a set \( A \) of elements equipped with a finite set \( G \) of generators for it, plus the binary equality relation = and Boolean
functions $\mathcal{B}$ and $\mathcal{R}$.

$f'(x) \subseteq f(\mathcal{R}(x))$.

\begin{figure}
\centering
\begin{tikzpicture}
\node at (0,0) (f) {$f(x)$};
\node at (0,-1) (r) {$\mathcal{R}(x)$};
\node at (1,-1) (f_r) {$f'(\mathcal{R}(x)) \subseteq \mathcal{R}(f(x))$};
\node at (-0.5,0) (x) {$x$};
\draw (x) -- (f);
\draw (f) -- (r);
\draw (r) -- (f_r);
\end{tikzpicture}
\caption{The function $f'$ implements the function $f$ via a representation $\mathcal{R}$ between their domains.}
\end{figure}

\begin{itemize}
\item The equality relation and Boolean constants are required for interpreting the output, as we will see.
\item An alternative to the proposed generator-based approach for describing abstract elements would be to define them by means of a set of relations. For graphs, this might be the relation telling whether an edge is present between two given vertices. We argue, in Section 7, that this alternative does not fit the bill. Intuitively, a set of functions allows one to also generate the representations, while a set of relations does not.
\end{itemize}

\section{Honest Implementation}

A function $\mathcal{F}$ over some concrete domain $\mathcal{C}$ honestly implements a function $f$ over an abstract domain $\mathcal{A}$ if it preserves the functionality of $f$ under the representation, while also preserving the meaning of the domain elements as given by the domain generators. Formally:

\begin{definition}[Honest Implementation]
Consider an abstract domain $\mathcal{A}$ with generators $\mathcal{G}$, a domain $\mathcal{C}$, and an honest representation $\mathcal{R}: \mathcal{A} \Rightarrow \mathcal{C}$.

\begin{itemize}
\item A function $\mathcal{F}$ over $\mathcal{C}$ implements a function $f$ of arity $k$ over $\mathcal{A}$ via the representation $\mathcal{R}$ if, for every $\bar{x} \in \mathcal{A}^k$, we have $\mathcal{F}(\mathcal{R}(\bar{x})) \subseteq \mathcal{R}(f(\bar{x}))$.
\item A function $\mathcal{F}$ over $\mathcal{C}$ honestly implements a function $f$ over $\mathcal{A}$ via the representation $\mathcal{R}$ if it implements $f$, while also providing an implementation of every generator $\mathcal{G} \in \mathcal{G}$ and of equality over $\mathcal{A}$.
\end{itemize}
\end{definition}

See the illustration in Figure 2.

Notice that $\mathcal{F}$ implements a function $f$ with respect to a specific set of generators. If Abe considers an abstract domain with generators that are natural, computable, and trackable for him, but completely useless for his
HONEST COMPUTABILITY AND COMPLEXITY

sister Sarah, then \( \widehat{f} \) is an honest implementation of \( f \) for Abe but not for Sarah.

5. Honest Computability

We first demonstrate the reasonableness of our demands on implementations by taking a look at effective computation.

Taking advantage of our definition of honest implementation, we are prompted to define honest computability over arbitrary abstract domains as follows:

**Definition 3 (Honest Computability).** A function \( f \) over an abstract domain \( A \) is honestly computable if it can be honestly implemented by a Turing machine or by a recursive function.

This is reasonable, considering that almost everyone deems Turing machine computations over strings and recursive functions over the natural numbers to be effective. See also [7].

Our definition guarantees that concrete representations for all the elements of the abstract domain can also be effectively generated. The key to everything is that an honest representation \( \rho \) always has a computable definition:

\[
\rho(g(x_1, \ldots, x_\ell)) := \widehat{g}(\rho(x_1), \ldots, \rho(x_\ell))
\]

for each generator \( g \in G \) of the abstract domain, where \( \widehat{g} \) is its concrete implementation.

For this to indeed constitute an effective definition, we need to be able to extract the arguments \( x_i \) of \( g(x_1, \ldots, x_\ell) \) given its value in \( A \). Assuming the ability to check equality over \( A \), one can effectively generate all terms \( T \) over \( G \) in some order, remembering how they have been constructed, so that when the abstract value \( g(x_1, \ldots, x_\ell) \) is encountered, one knows that \( g \in G \) was used to construct it and also what the values of its arguments were (or could be, when there is more than one term per value). Furthermore, the inverse function \( \rho^{-1} : \mathbb{N} \to A \) (and likewise \( \rho^{-1} : \Sigma^* \to A \)) can be computed for every number in the image \( \rho(\mathbb{N}) \) of the representation, by enumerating the elements of \( A \) and looking at their values under \( \rho \).

For example, consider the generators zero, \( 0 \), and successor, \( s \), of the naturals. Suppose these are mapped to \( \widehat{0} \) and \( \widehat{s} \), respectively, under an honest representation \( \rho : \mathbb{N} \xrightarrow{1-1} \mathbb{N} \). For simplicity, we assume that \( \rho \) is a single-valued function. Then \( \rho \) can be defined recursively as follows:

\[
\rho(0) = \widehat{0} \\
\rho(s(n)) = \widehat{s}(\rho(n))
\]

Therefore, any \( f : \mathbb{N} \to \mathbb{N} \) that is implemented by a recursive \( \widehat{f} : \mathbb{N} \to \mathbb{N} \) under this representation must itself be recursive, since

\[
f(n) = \rho^{-1}(\widehat{f}(\rho(n)))
\]
is the composition of computable functions.

Under the above (lax) assumptions, if the concrete function \( \hat{h} \) is computable, then the implemented function \( h \) can in fact be programmed effectively by an abstract state machine \[10\] over the combined domain \( A \sqcup \mathbb{N} \):

\[
h(g(x_1, \ldots, x_\ell)) := \rho^{-1}(\hat{h}(\rho(g(x_1, \ldots, x_\ell))))
\]

for each \( g \in G \). For this to work, we presume the availability of an effective equality test for \( A \).

It seems right, then, to consider a function over an arbitrary domain to be \textit{uncomputable} when there is an honest bijective representation under which it is implemented by an uncomputable function.

**Theorem 4.** A function \( h \) over a domain \( A \) is honestly computable iff every honest implementation of \( h \) via an honest bijective numerical representation \( \pi : A \leftrightarrow \mathbb{N} \) is recursive.

For simplicity, we deal here with single-valued representations, noting that the claim can be extended to multivalued representations. We also stick to unary functions.

**Proof.** First we show that if \( h \) is honestly implemented by a computable function, then any implementation \( \hat{h} \) via honest \( \pi \) must also be computable.

Let \( \hat{h} : \mathbb{N} \rightarrow \mathbb{N} \) implement \( h \) under \( \pi \), and let \( \hat{g} : \mathbb{N} \rightarrow \mathbb{N} \) denote the recursive implementation of each generator \( g \in G \) with respect to \( \pi \). If \( \hat{h} \), the function that implements \( h \) under some honest \( \rho \), is recursive, then \( \hat{h} \) must also be recursive. This is because

\[
\hat{h}(n) = \pi(h(\pi^{-1}(n))) = \pi(\rho^{-1}(\hat{h}(\rho(\pi^{-1}(n)))))) = \tau^{-1}(\hat{h}(\tau(n))),
\]

where \( \tau = \pi^{-1} \circ \rho \). By showing that \( \tau \) is honest, hence—by the previous discussion—effectively computable, it will follow that \( \hat{h} \) is recursive. Define

\[
\hat{g}(n) := \tau(\hat{g}(\tau^{-1}(n)))
\]

for each generator. To see that \( \tau \) is recursive, and hence \( \hat{g} \) is, we can work with any standard numerical encoding of terms over \( G \), and compute \( \pi^{-1} \) and \( \rho \) as indicated earlier.

Next we show that, if every honest bijective representation gives a computable implementation, then the function in question can be honestly implemented by a computable function regardless of which honest representation is chosen. Note that if there is an honest injective representation, then there also must be an honest bijective representation, obtained by enumerating the image of the injection.

Let \( \rho \) be an honest representation, and let \( h : A \rightarrow A \) be implemented by recursive \( \hat{h} \) under some effective bijection \( \pi \). The recursive function

\[
\hat{h}(n) := \rho(\pi^{-1}(\hat{h}(\pi(\rho^{-1}(n))))))
\]
implements \( h \), since
\[
\tilde{h}(\rho(x)) = \rho(\pi^{-1}(\tilde{h}(\pi(\rho^{-1}(\rho(x))))))
\]
\[
= \rho(\pi^{-1}(\tilde{h}(\pi(x))))
\]
\[
= \rho(\pi^{-1}(\pi(h(x))))
\]
\[
= \rho(h(x))
\]

\[\Box\]

6. Honest Complexity

Before one can measure complexity, one needs a measure of input size and a measure of the cost of a computation.

A size measure is associated with each (ground) term over the generators. This provides the flexibility of considering each of the various views of the same abstract element differently.

**Definition 5 (Size).**

- A size measure for an abstract domain is a function \( \cdot \) : \( T \to \mathbb{N} \), where \( T \) is the set of (ground) terms over the generators of the domain.
- The size of a vector of terms is the sum of the sizes of its components.

Examples of size measures for terms denoting graphs are tree height of the term, as well as the number of vertices or number of edges in a graph. Note that the two latter measures assign the same size to all terms of the same graph.

Complexity is measured with respect to this size, whatever it may be. That is, size is in “the eyes of the beholder”.

One might argue that a size measure should not be this arbitrary, but should enforce a compact representation of the abstract elements, as Garey and Johnson demanded of the representation of numbers in the paragraphs quoted at the outset, namely that the size of a natural number \( n \) should be \( O(\log n) \). In many cases, however, this is too demanding. For instance, a set of \( n \) elements taken from some unordered set may have \( n! \) reasonable representations. Checking equality between two such representations, in order to choose a single canonical representation for each set, might require a quadratic number of element comparisons. Even more involved is the case of graphs. If we are asked to decide the existence of a Hamiltonian path in an unlabeled graph, we should not demand that there be a unique or almost-unique way of constructing each graph, considering that graph isomorphism is a difficult problem. But there are exponentially many isomorphic graphs, so the standard representations of graphs are as wasteful as is the unary encoding of numbers.

The cost assigned to a computation over the concrete domain \( C \) depends on the relevant aspects of the computational model in question. For example, the cost can be the number of steps of a RAM model or the number of tape
cells used by a Turing machine. As with the size measure, cost is also in “the eyes of the beholder”. Given a cost measure for computation in the model, we define the cost of a term, as follows:

**Definition 6 (Cost).** The cost \( \kappa(\widehat{h}(t_1,\ldots,t_k)) \) of a concrete term \( \widehat{h}(t_1,\ldots,t_k) \) is the cost of a computation that first constructs the concrete values \( c_i \in C \) of each argument \( t_i \), and then computes \( \widehat{h}(c_1,\ldots,c_k) \), the value of \( \widehat{h} \) for the concrete values thus obtained.

Note that in some cases the cost of a computation might be the sum of the costs of its steps, as is natural for time complexity, while in other cases a different aggregation, such as maximum, is appropriate, as is done for space complexity.

Equipped with size and cost measures, we are ready to formalize our intuition of when a complexity measure is honest. The complexity of an implementation must take the specific means of representation into account. We have demanded that an honest implementation of an abstract function also provide implementations of the abstract domain’s equality and generators (Definition 2). We may assume that every generator has a unique implementation. (Different implementations should have different names, thus refer to different, and possibly equivalent, generators.)

Our definition of the complexity of a function resembles the standard one; it is just that our notion of the cost of computing a function includes the cost of generating the representation of the input.

**Definition 7 (Honest Complexity).** Consider an abstract domain \( A \) with ground generator terms \( T \) and an honest implementation \( \widehat{h} : C^k \to C \) over concrete domain \( C \), implementing a function \( h : A^k \to A \) over \( A \). Let \( m : \mathbb{N} \to \mathbb{N} \) be a complexity measure. We say that \( \widehat{h} \) has a worst-case complexity of at most \( m \) if

\[
\kappa(\widehat{h}(\bar{t})) \leq m(\bar{l}),
\]

for all tuples \( \bar{t} \in T^k \) of terms.

Average and probabilistic complexities can be defined analogously to worst-case complexity.

### 7. Dishonest Decisions

It is standard to classify the difficulty of a problem according to its membership in a set of functions or relations. For example, whether it is Turing-computable, in polynomial time, or in polynomial space. **Computational models**, such as Turing machines with arbitrary outputs, compute sets of (partial) functions, whereas **decision models**, such as finite automata or Turing machines with only “yes” or “no” outputs, compute sets of relations.

We argue that computational models, which implement functions, capture the essence of computational power more accurately than do decision models, which implement relations. For that reason, we based the notion of honest implementation and complexity, even for decision problems, on functions

---

1. RAMs are in fact optimal for time—up to a linear factor [6].
rather than on decision procedures. The underlying reason for the better adequacy of functions than relations is that the former allows one to generate the representations of objects.

We formally consider a computational model $M$ over a domain $A$ as its extensionality, namely as a set of functions $M \subseteq \{ f : A^n \to A : n \in \mathbb{N} \}$, and likewise a decision model $M$ over $A$ as a set of relations $M \subseteq \{ r \subseteq A^n : n \in \mathbb{N} \}$. We note that specific computational or decision models are defined via some internal mechanism, a point that will play a role in our arguments, while we only consider their extensionality for the sake of generality.

Our definition of honest implementation (Definition 2) considers the implementation of the required function together with a set of generators. Putting aside the purpose of the generators, it is about the implementation of a set of functions. The implementation notion is then basically about a set of functions, namely about implementing a computational model.

**Definition 8 (Model Implementation).** A model $\overline{M}$ implements a model $M$ via a representation $\rho$ if, for every function $f$ of $M$, there is a function $\overline{f}$ of $\overline{M}$ that implements $f$ via $\rho$.

We will see that decision models are “incomplete”, in the sense that one can easily “increase” their power via a representation that adds some information on top of the represented element [1]. For example, let $h$ be an uncomputable decision problem over $\{0, 1\}^*$, and consider the representation $\rho : \{0, 1\}^* \to \{0, 1\}^*$, where $\rho(w) = h(w)$. Then, Turing machines can decide, via the representation $\rho$, both $h$ and all of the standard Turing-decidable problems.

Furthermore, finite automata (DFAs) are already “strong enough” to decide, via a suitable representation, any countable set of relations [8]. The representation hides with each domain element a finite amount of data with respect to finitely many relations, such that each decision procedure gets all of the data it needs via the represented inputs.

**Proposition 9 ([8]).** For every countable set $S$ of relations over $\mathbb{N}$ there is an injection $\rho : \mathbb{N} \to \{0, 1\}^*$, such that $\rho(S) \subseteq \text{DFAs}$.

**Proof.** Let $r_1, r_2, \ldots$ be an enumeration of the relations in $S$. Define the representation $\rho : \mathbb{N} \to \{0, 1\}^*$ by $\rho(n) = r_1(n)r_2(n)\cdots r_n(n)$. Note that for every $n$, $|\rho(n)| = n$. Now, for every relation $r_i \in S$, consider the DFA $A_i$ depicted in Figure 3. One can easily verify that for every $n \in \mathbb{N}$, $A_i$ accepts $\rho(n)$ iff $n \in r_i$. Indeed, the first $i$ states in $A_i$ are fixed to accept or reject an input word of length $n$ according as to $n \in r_i$. For a word $w$ of length $n \geq i$, $w_i$ provides the answer whether $n \in r_i$.

One might assume that this disturbing sensitivity to representations is resolved once limiting representations to bijections, but this is unfortunately not the case.

**Proposition 10 ([8]).** For every countable set $S$ of relations over $\mathbb{N}$ there in a bijection $\pi : \mathbb{N} \to \{0, 1\}^*$, such that $\pi(S) \subseteq \text{DFAs}$. 
The above inherent incompleteness of decision models stems from their inability to generate the representation of the input. On the other hand, the set of recursive functions (REC) is complete, in the sense that it cannot honestly implement an uncomputable function, with respect to any representation.

**Theorem 11.** Consider a computational model $M$ over $\mathbb{N}$ and a representation $\rho: \mathbb{N} \Rightarrow \mathbb{N}$, such that $\text{REC} \subseteq M$ and REC implements $M$. Then $M = \text{REC}$.

**Proof.** We show that every function $h \in M$ is also in REC. Let $s$ be successor. Since REC implements $M$, there are functions $\hat{h}, \hat{s} \in \text{REC}$, such that for every $\bar{n} \in \mathbb{N}^*$, $\hat{h}(\rho(\bar{n})) \subseteq \rho(h(\bar{n}))$, and for every $n \in \mathbb{N}$, $\hat{s}(\rho(n)) \subseteq \rho(s(n))$.

Given a vector $\bar{n} := (n_1, \ldots, n_k) \in \mathbb{N}^*$, we can compute $h(\bar{n})$ by the following recursive procedure:

- Compute the vector $\bar{n} = (\hat{n}_1, \ldots, \hat{n}_k)$ that represents $\bar{n}$ by iterating, for every $1 \leq j \leq k$, over an index $i$, setting $x_0$ to some constant in $\rho(0)$ and $x_{i+1}$ to $\hat{s}(x_1)$. Then, we define $\hat{n}_j$ to be $\hat{s}^{n_j}(x_0)$.
- Compute $\hat{k} := \hat{h}(\bar{n})$. Note that, since $\hat{h}$ implements $h$, $\hat{h}$ diverges on $\bar{n}$ iff $h$ diverges on $\bar{n}$.
- Compute the number $\hat{k}$ that represents $k$ by iterating over an index $i$, setting $x_0$ to some constant in $\rho(0)$ and $x_{i+1}$ to $\hat{s}(x_1)$. After each iteration $i$, we check whether $\hat{s}^i(x_0) = \hat{k}$. We are guaranteed to stop after an iteration $k$, such that $\hat{k} \in \rho(k)$. Hence, $\hat{k}$ represents $k$ and we have that $h(\bar{n}) = k$.

Note that in order to check whether $\hat{s}^i(x_0) = \hat{k}$, we used the requirement that True and False each have a single representation, allowing the above recursive procedure to decide whether equality holds.

By the same token, Turing machines are complete whereas two-counter machines and the lambda calculus are not. See [1].

**Corollary 12.** Consider a model $M$ consisting of an arbitrary function $f$ and a finite set of computable generators. If TM implements $M$ then $f$ is computable.
Two questions naturally arise, following the completeness of recursive functions (Theorem 11 and Corollary 12) and the inherent incompleteness of decision models (Theorems 9 and 10):

(1) Where does the proof of Theorem 9 fail if aimed at showing that every set of functions can be computed, via a suitable representation, by finite transducers?

(2) A function is a special case of a relation, making computational models a special case of decision models. Hence, we can obviously say that there are complete decision models, can’t we?

We answer these two questions as follows:

(1) Aimed at computing a countable set of functions \{f_1, f_2, \ldots\}, the representation may be generalized to something like \(\rho(n) = f_1(n) \cdot f_2(n) \cdot \ldots \cdot f_n(n)\). Then, for every function \(f_i\), there is indeed a transducer \(T_i\), such that for every \(n\), \(T_i(\rho(n)) = f(n)\).

This is, however, not enough. For properly representing \(f_i\), we need a finite transducer \(T'_i\), such that \(T'_i(\rho(n)) = f_i(n)\). One might be tempted to suggest a representation \(\eta\) that already provides the represented values, meaning \(\eta(n) = \eta(f_1(n)) \cdot \eta(f_2(n)) \cdot \ldots \cdot \eta(f_n(n))\). This is, however, a circular definition: Let \(f_1\) be the successor function \(s\). Representing 1, we have \(\eta(1) = \eta(s(1)) = \eta(2) = \eta(s(1)) \cdot \eta(s(2)) = \ldots\).

(2) Indeed, a set of functions is also a set of relations, in which sense it can be viewed as a decision model. Yet, the model’s extensionality, namely the set of relations that it decides, is not arbitrary. It is derived from the internal mechanism of the model, which follows some natural definition. This mechanism will then also accept relations that are not necessarily functions.

For example, Turing decision machines accept general relations and they are not complete. One may define a more restricted decision model that accepts a relation only if it constitutes a function that is computed by a Turing machine. While this model is indeed complete, it is actually based on the notion of a Turing function machine rather than on a Turing decision machine.

8. **Honest Universality**

A function \(\psi\) is said to be “universal” if it computes a whole set of functions \(\Phi\) (such as all the recursive functions) by being supplied with the code \(\langle \varphi \rangle\) of a \(\varphi \in \Phi\) as an extra argument. If \(\psi\) works with a concrete domain, whereas the functions in \(\Phi\) operate on an abstract domain, then representations are needed.

Another potential problem with the notion of universal function is that some models of computation—like Turing machines—do not take their inputs separately, but, rather, all functions are unary (string-to-string for Turing machines). In such cases, one needs to be able to represent pairs
(and tuples) as single elements. One standard pairing function for the naturals is the injection \( (i, j) := 2^i3^j \). For strings, one usually uses an injection like \( (u, w) := u \cdot; w \), where “;” is some symbol not in the original string alphabet.

There are several ways to go. The pairing function could reside in the abstract domain \( A \), or in the concrete domain \( C \), or in the representation of \( A \) as \( C \). Regardless, this need raises a critical issue. Unless we demand that pairing be effective, there could be a machine that does too much, computing even non-effective functions. For example, a naïve definition might simply ask that pairing be injective and say that \( \psi \) is universal for some set \( \Phi \) of functions if \( \forall \varphi \in \Phi \text{ and } x \in C \), for some arbitrary encoding \( (\varphi', y) \mapsto [\varphi(y), \varphi', y] \), where the square brackets are some ordinary tupling function for the domain. Then a putative universal machine could effortlessly “compute” virtually anything, computable or otherwise, just by reading the encoded input pair.

Davis [4] and Rogers [11] proposed general definitions of universality for Turing machines and for partial recursive functions, respectively. Both insist that pairing be effectively computable. But we are talking about models in which no function takes two arguments, so we might not have an appropriate notion of computable binary function at our disposal. To capture effectiveness of pairing in such circumstances, we demand the existence of component-wise successor functions. Given a “successor” function \( s \) for domain \( C \) (that is, \( C = \{s^e(e)\} \) for some \( e \in C \)) and a pairing function \( \langle\cdot,\cdot\rangle: C \times C \rightarrow C \), the component-wise successor functions operate as follows: \( s_1 : (a, b) \mapsto (s(a), b) \) and \( s_2 : (a, b) \mapsto (a, s(b)) \). If \( s, s_1 \) and \( s_2 \) are computable, then we will say that pairing is effective. This is because one can program pairing so that \( (z, y) := s_1(s_2(e, e)) \), where \( z = s^e(e) \) and \( y = s^e(e) \).

And if pairing is effective, then its two projections (inverses), \( \pi_1 : (a, b) \mapsto a \) and \( \pi_2 : (a, b) \mapsto b \), are likewise effective. (Generate all representations of pairs in a zig-zag fashion, until the desired one is located. What the projections do with non-pairs is left up in the air.)

Another concern is that requiring that pairing be computable is too liberal for the purpose. One does not really want the pairing function to do all the hard real work itself. For example, the mapping could include \( \varphi(x) \) in the pair, even if it only can do that for \( \varphi \) that are known to be total (like, for the primitive recursive functions, of which there are infinitely many), or all functions that halt within some recursive bound. That would make it a trivial matter to be universal for those functions—just transcribe the answer from the input.
**Definition 13 (Honest Pairing).** A pairing function is honest if it is effective and bijective.

This way, there is no room for hiding information.

For bijective pairing with computable projections, there is an effective means of forming a pair \( \langle a, b \rangle \) by enumerating all of \( C \) until the two projections give \( a \) and \( b \), respectively. With bijectiveness alone, sans computability, one could still hide a fair amount of uncomputable information in a bijective mapping. For instance, imagine that 0 is the code of the totality predicate and that the rest of the naturals code the partial-recursive functions in a standard order. Map pairs \( (i + 1, z) \) to \( 3i \), \( \cdot \), \( \cdot \), where \( \cdot \) is a standard pairing; map \( (0, z) \) to \( 3j + 1 \) when \( z \) is the \( j \)th total (recursive) function; and map \( (0, z) \) to \( 3k + 2 \) when \( z \) is the \( k \)th non-total (partial recursive) function. Now, let \( U \) be some standard computable universal function. Then, for \( y \) divisible by 3, \( \psi(y) := U(y/3) \) would compute all the partial-recursive functions, whereas \( \psi(y) := y \equiv 1 \pmod{3} \) would compute the uncomputable totality predicate, when \( y = (0, z) \) is not divisible by 3.

**Definition 14 (Honest Universality).** Let \( \Phi \) be some set of unary functions over an abstract domain \( A \). A unary function \( \psi \) over a concrete domain \( C \) is universal for \( \Phi \), via pairing function \( \cdot \), \( \cdot \), \( \cdot \) over \( C \), if \( \Psi = \{ \lambda y.\psi(a,y) : a \in C \} \) implements \( \Phi \) via some representation. If, in addition, pairing is bijective, then we call the universal function honest.

That is, \( \psi \) is universal if \( \psi(\varphi^\prime, \rho(x)) = \sigma(\varphi(x)) \) for \( \varphi \in \Phi \) and \( x \in A \). Of course, we are interested in the case where both pairing and the universal function are computable.

**Theorem 15 ([5]).** Let \( \Phi \) be some set of unary functions over a domain \( A \), including generators and equality. Then, if there is a computable unary universal function (over any domain \( C \)) for \( \Phi \), via an effective pairing, then all the implemented functions \( \varphi \in \Phi \) are also computable.

Suppose \( \Phi = \{ \varphi_z \} \) is some standard enumeration of (the definitions of) the partial-recursive functions. Based on Davis’s (second) definition of a universal Turing machine, which relies on a notion of effective mappings between strings and numbers, namely, recursive in Gödel numberings, Rogers defines (in his third definition) what we may call Rogers-universality of a unary numerical function \( \psi \) to be the property that \( \varphi_z(x) = \gamma(\psi(z,x)) \) for some recursive bijection \( \gamma \) and effective (but perhaps dishonest) pairing \( \langle \cdot, \cdot \rangle \).

The following follows from the definitions:

**Theorem 16 ([5]).** If a function is Rogers-universal, then it is honestly universal. Furthermore, there must exist an honest computable universal function.

9. Discussion

Demanding that an implementation can also generate representations of the input precludes cheating on problems like Hamiltonian paths. It also
means that checking parity of a binary string should be considered linear-time (in input length), not constant-time. Put another way, presenting a number with least-significant digit first is just as dishonest as ordering the nodes of a graph by its Hamiltonian path.

In some cases, it is interesting to analyze a representation with respect to the complexity of a set of functions. Considering graphs, for example, it is common to compare between the adjacency list and adjacency matrix representations. While the former provides better efficiency for adding a vertex, it has a steeper cost for removing an edge. In these cases, the complexity of generating the input representation is yet another aspect of the complexity tradeoffs.

References