Chapter 6
Honest Computability and Complexity

Udi Boker and Nachum Dershowitz*

> **Goldstein:** And what causes you to say that?
> **Davis:** *Honesty*.
>
> —Martin Davis: An Interview Conducted by Andrew Goldstein
> (IEEE History Center, July 18, 1991)

For Martin—scientist, scholar, and thinker.

Abstract We raise some issues posed by the use of representations for data structures. A nefarious representation can turn the incomputable into computable, the non-recursively-enumerable into regular, and the intractable into trivial. To overcome such problems, we propose criteria for "honesty" of implementation. In particular, we demand that inputs to functions and queries to decision procedures be specified as constructor terms.

Udi Boker
School of Computer Science, Interdisciplinary Center, Herzliya, Israel, e-mail: udiboker@idc.ac.il

Nachum Dershowitz
School of Computer Science, Tel Aviv University, Ramat Aviv, Israel, e-mail: nachum@cs.tau.ac.il

* This author's research benefited from a fellowship at the Institut d'Études Avancées de Paris (France), with the financial support of the French state, managed by the French National Research Agency's "Investissements d'avenir" program (ANR-11-LABX-0027-01 Labex RFIEA+).

6.1 Honesty is Needed

Computations have no choice but to manipulate representations of objects rather than the objects themselves. Most often, strings of symbols taken from some finite alphabet are used for the purpose. Numbers, for example, are usually denoted by sequences of decimal symbols, or binary bits, or unary strokes (like the tally numbers of paleolithic times). In logic, one therefore distinguishes between numbers n, which reside in an ideal Platonic world, and numerals $\underline{n}$, their symbolic representation as (first-order) terms. Similarly, graphs, which are set-theoretic objects, are typically either represented as lists of edges (pairs of nodes) or as binary adjacency matrices.

Given that representation is an inescapable 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 incomputable, 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 process of answering.

- 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 specific question whether there is a Hamiltonian path, whereas she would have a much harder time performing basic graph operations, such as adding an edge.

- Is it legitimate 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 (namely, least-significant-first binary, as opposed to ternary, say)?

  Short answer: No.

Garey and Johnson [15, pp. 9–10] address the questions of representation and computational models as they impact the measurement of computational complexity. They assert upfront 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 [6]:

> 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. [*Italics 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 it may 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, for example; it puts the solution—when there is one—right in front of one's nose. Our proposal for measuring complexity honestly will solve this problem by taking into consideration both the cost of computing a given function as well as the cost of generating the function's inputs (the nodes and edges of a graph in the Hamiltonian case).

This chapter looks at questions of "honesty" of representation in various contexts. We begin with what we feel is the underlying problem posed by representations, namely, the camouflaging of extra information (Section 6.2). After proposing a solution (Sections 6.3 and 6.4), we consider how it resolves the problem of honest computability (Sections 6.5 and 6.6) and relate honesty to Martin Davis's definition of universality (Section 6.7). Then we turn to see how this proposal also solves the problem of honest complexity (Section 6.8). With a solution in place, we analyze why considering formal languages, rather than functions, does not work (Section 6.9).

6.2 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 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 concrete domain C, say binary strings, rather than A. So, prior to considering the cost of running $\widehat{f}$, one should first establish that $\widehat{f}$ does actually implement f.

However, there is little meaning to a claim that a single function $\widehat{f}$ over some domain C implements the intended function f over domain A without also specifying how the two domains are related. The following definition is common.[2]

Definition 1 (Simulation [single-valued representation]). A concrete (partial) function $\widehat{f}: C \to C$ *simulates* an abstract (partial) function $f: A \to A$ *with respect to* a particular injective representation $\rho: A \xrightarrow{\text{1-1}} C$ if $\rho(f(x)) = \widehat{f}(\rho(x))$ for all $x \in A$. [In the case of partial functions, we also demand that $\widehat{f}(\rho(x))$ be undefined whenever $f(x)$ is.]

One clearly needs to require, as we have, that a representation be injective. Otherwise, any and all functions could be simulated by the identity function with respect to a representation that maps the whole abstract domain to a single constant.

The above definition will be extended to functions with arity wider than 1 and multivalued representations in Definitions 3 and 5 below. Figure 6.1 depicts the multivalued case.

---

[2] Allowing different representations for input and output, as in "A more general notion of simulation is obtained if we let drop the requirement that $\mathfrak{R}^{(1)}$ and $\mathfrak{R}^{(2)}$ have the same input and output sets.... $\mathfrak{R}^{(1)}$ can be *weakly simulated* on $\mathfrak{R}^{(2)}$ if there exist such $\mathscr{E}$ and $\mathscr{D}$ with the property that for each program π_1 for $\mathfrak{R}^{(1)}$ there exists a program π_2 on $\mathfrak{R}^{(2)}$ such that $\mathscr{D}\mathfrak{R}^{(2)}_{\pi_2}\mathscr{E} = \mathfrak{R}^{(1)}_{\pi_1}$" [14, p. 21], can lead—if one is not careful—to the same kinds of problems we will encounter in Section 6.9.

$$
\begin{tikzcd}
x \arrow[r, "f"] \arrow[d, dashed, "\rho"'] & f(x) \arrow[d, dashed, "\rho"] \\
\rho(x) \arrow[r, Rightarrow, "\widehat{f}"] & \widehat{f}(\rho(x)) \subseteq \rho(f(x))
\end{tikzcd}
$$

Fig. 6.1 The function $\widehat{f}$ simulates the function f via a representation ρ between their domains.

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 incomputable one by getting the representation itself to do the bulk of the work.

Example 2. Consider any standard enumeration TM_m, $m = 0, 1, \ldots$, of Turing machines, and define the following incomputable functions over the natural numbers $\mathbb{N}$:

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

So $h(\mathbb{N}) \uplus \bar{h}(\mathbb{N}) = \mathbb{N}$, where $h(\mathbb{N})$ is the image $\{h(n) \mid n \in \mathbb{N}\}$ of h and $\bar{h}(\mathbb{N})$ is the image of $\bar{h}$. Then the *incomputable* function

$$H(m) := \begin{cases} \min h(\mathbb{N}) & \text{if } \mathsf{TM}_m \text{ halts} \\ \min \bar{h}(\mathbb{N}) & \text{if } \mathsf{TM}_m \text{ does not} \end{cases}$$

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

$$\rho(m) := \begin{cases} 2h^{-1}(m) + 1 & \text{if } \mathsf{TM}_m \text{ halts} \\ 2\bar{h}^{-1}(m) & \text{if } \mathsf{TM}_m \text{ does not}\,. \end{cases}$$

We have

$$\rho(m) \bmod 2 = \rho(H(m)) = \begin{cases} 1 & \text{if } \mathsf{TM}_m \text{ halts} \\ 0 & \text{if } \mathsf{TM}_m \text{ does not}\,, \end{cases}$$

as required. $\square$

The problem with the above "implementation" of the halting function obviously lies in the representation, which clearly gives the impression of itself doing the (computationally) impossible.

6.3 Honest Representations

The cause of the problem we just saw with the "dishonest" representation is not the (mathematically well-defined) mapping itself but rather the lack of suitable context for it. In particular, the integer successor function, for instance, cannot be implemented by any computable function under the nefarious representation ρ of the previous section, 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. [22, 23, 4, 9]). These generators give structure to A and meaning to its elements as described by ground terms H over G. For generators to do their job, every element of A must be equal to the value of at least one term in H; so at least one generator must be a scalar constant (of arity 0).[3]

Examples of generators for the natural numbers include:

$\mathsf{0}$ (nullary zero)
$\blacksquare'$ (postfix successor $\lambda n.\, n+1$),

(in unary "caveperson" style), as well as

$\mathsf{0}$ (nullary zero, $\lambda.0$, usually suppressed)
$\blacksquare\mathsf{0}$ (postfix doubling, $\lambda n.\, 2n$)
$\blacksquare\mathsf{1}$ (postfix doubling plus one, $\lambda n.\, 2n+1$)

for the commonplace binary representation, and

$\mathsf{0}$ (nullary zero, $\lambda.0$, usually suppressed)
$\blacksquare\mathsf{0}$ (postfix tripling, $\lambda n.\, 3n$)
$\blacksquare\mathsf{1}$ (postfix tripling plus one, $\lambda n.\, 3n+1$)
$\blacksquare\mathsf{2}$ (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

$\square : V$ (nullary first-vertex)
$\blacksquare' : V \to V$ (postfix next-vertex)
$\varnothing : G$ (nullary empty-graph)
$\bullet ; \blacksquare : G \times V \to G$ (binary add-vertex to graph)
$(\bullet) + \blacksquare \frown \blacksquare : G \times V \times V \to G$ (ternary add-edge to graph) .

[3] Unlike the development in [4], where effectiveness of an algorithm was at issue, here we are not insisting that the generators form a *free* term algebra (a Herbrand universe): more than one term may designate the same abstract element.

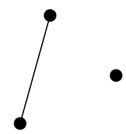

Fig. 6.2 An abstract undirected, unlabeled graph.

Over these generators, the graph depicted in Figure 6.2 is the value of the ground term

$$(\varnothing;\square;\square';\square'')+\square\frown\square' ,$$

wherein there is an edge between the "first" and "second" vertices. It is also the value of the term

$$(\varnothing;\square'';\square';\square)+\square''\frown\square' ,$$

wherein there is an edge between the "third" and "second" vertices. [In general, generators can be partial.]

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

Definition 3 (Representation).

- An *abstract domain* is a set A of elements, including (always) Boolean values TRUE and FALSE, equipped with a finite set G of generators for the whole domain, which also includes the binary equality relation $=$. Every element of A must be equal to at least one ground term over G.
- A *representation* of A in a "concrete" domain C is an *injective* multivalued function $\rho : A \overset{\text{1-1}}{\rightrightarrows} C$. We will insist that $\rho(\text{TRUE})$ and $\rho(\text{FALSE})$ are both finite.
- The representation $\rho(\langle a_1,\ldots,a_n\rangle)$ of a tuple of abstract elements a_i is the set $\rho(a_1)\times\rho(a_2)\times\ldots\times\rho(a_n)$, the set of all tuples $\langle c_1,\ldots,c_n\rangle$, such that $c_i\in\rho(a_i)$.

The equality relation and Boolean constants are required for interpreting the output, as we will see. Having only finitely many representations of TRUE and FALSE will allow Abe to understand and compare results of Cay's computations.

Having representations as multi-valued, rather than single-valued, functions gives the freedom to have many representations for the same abstract element, as is very commonly done in practice. For example, one may represent the rational number one-half by $1/2, 7/14$, etc., and the unordered set $\{7,2,3\}$ by the sequences $\langle 2,3,7\rangle, \langle 7,3,2\rangle$, etc.

The choice of what is "abstract" and what is "concrete" is in "the eyes of the beholder"; it is in the final analysis an arbitrary formal choice. One may view a number as an abstract entity, represented by a concrete string over the symbols 0 and 1, while another still views the symbol 1 as an abstract entity represented by some ink dots or electric pulse, etc. Likewise, the equality relation $=$ depends on the choice of what an abstract entity is. For example, if the abstract domain is graphs, then the graph of Figure 6.2 is a single entity and all of its different generating terms yield equal entities, while in the case that the abstract domain consists of graphs with numbered vertices, the different generating terms yield isomorphic, but unequal, entities.

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. It is well known that using such relations, rather than generating functions, increases the complexity of many procedures. (For example, exhaustively checking all vertex combinations for getting an adjacent vertex.) Furthermore, we argue in Section 6.9 that this alternative does not at all fit the bill. Intuitively, a set of functions allows one to also generate the representations, while a set of relations does not.

6.4 Honest Implementation

A function $\widehat{f}$ over some concrete domain C honestly implements a function f over an abstract domain 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.

We formally consider a *(computational) family* F over a domain A to be an algebra (in the universal-algebra sense), consisting of the domain (universe) A and operations F over A (of any arity), along with a matching vocabulary. Our definition of honest implementation will require the simulation of the desired function together with a set of generators. The implementation notion is then really about an algebra as a whole.

When we have cause to care about the intensionality (internal workings) of a computational mechanism, we will talk about a *(computational) model* comprising a set of *algorithms*, each of which involves a set of states, a subset of which are initial states, and a (partial and/or multivalued) transition function over states [18].

Definition 4 (Simulation [multivalued representation]).

- A function $\widehat{f}$ over a domain C *simulates* a function f of arity ℓ over a domain A via representation $\rho : A \overset{1\text{-}1}{\rightrightarrows} C$ if, for every $\bar{x} \in A^{\ell}$, we have $\widehat{f}(\rho(\bar{x})) \subseteq \rho(f(\bar{x}))$.
- Likewise, a family of functions $\widehat{F}$ over C *simulates* a family F over A (via representation ρ) if each $f \in F$ is simulated via the same ρ by some $\widehat{f} \in \widehat{F}$.

As usual, functions are extended to operate over sets by letting $f(S) := \{f(\bar{x}) \mid \bar{x} \in S\}$.

Definition 5 (Honest Implementation). Consider an abstract domain A with generators G.

- A family of functions $\widehat{F}$ over C provides an *implementation* of a family F over A if F is simulated by $\widehat{F}$.

- We will refer to the implementation as *close* if the simulation is via a bijection.
- An implementation $\widehat{F}$ over C is *honest* as long as F includes the generators G as well as equality.
- We will say that a function $\widehat{f}$ over C *honestly implements* a single function f over A if the implementation also supplies simulations $\widehat{\mathsf{g}}$ of each generator $\mathsf{g} \in \mathsf{G}$ including equality over A. In other words, we require that $\{\widehat{f}\} \cup \widehat{\mathsf{G}}$ implement $\{f\} \cup \mathsf{G}$, for some set $\widehat{\mathsf{G}}$ of concrete generators and implementation $\widehat{=}$ of abstract equality.

See the illustration in Figure 6.1.

The point is that $\widehat{f}$ 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 sister, Sal, then $\widehat{f}$ is an honest implementation of f for Abe but not for Sal.

We give next an example of an honest implementation of an abstract function over the rationals $\mathbb{Q}$ by means of a concrete function over strings.

Example 6. The task is to implement rational multiplication, $\mathsf{m} : \mathbb{Q} \times \mathbb{Q} \to \mathbb{Q}$, by means of a string-based model of computation.

- The abstract domain (A in the definition) is the set of rational numbers $\mathbb{Q}$ (with the subset of integers $\mathbb{Z}$ and its subsets the positive integers $\mathbb{Z}^+$ and negative integers $\mathbb{Z}^-$, plus the truth values), along with the following generators:

  $\mathsf{o} : \mathbb{Z}$ (nullary zero)
  $\mathsf{\imath} : \mathbb{Z}^+$ (nullary one)
  $\mathsf{s} : \mathbb{Z}^+ \to \mathbb{Z}^+$ (unary successor)
  $\mathsf{n} : \mathbb{Z}^+ \to \mathbb{Z}^-$ (unary negation)
  $\mathsf{q} : \mathbb{Z} \times \mathbb{Z}^+ \to \mathbb{Q}$ (quotient of an integer by a positive integer)

- The concrete domain (C) is the set Σ^* of finite strings over the symbols $\{0,1,2,3,4,5,6,7,8,9,-,\#\}$, plus TRUE and FALSE.
- Let $\underline{x} \in \Sigma^*$ denote the number $x \in \mathbb{Z}$ in decimal notation.
- The representation $\rho : \mathbb{Q} \overset{\text{1-1}}{\rightrightarrows} \Sigma^*$ is defined by $\rho(r) := \{\underline{x}\,\#\,\underline{y} \mid r = x/y,\ x \in \mathbb{Z},\ y \in \mathbb{Z}^+\}$.
- The implementations of the generators and equality are as follows:

  $\widehat{\mathsf{o}}$ returns the string `0`

  $\widehat{\mathsf{\imath}}$ returns the string `1`

  $\widehat{\mathsf{s}}(w)$: If w is the decimal representation $\underline{x}$ of $x \in \mathbb{Z}^+$, then $\widehat{\mathsf{s}}(w)$ returns the decimal representation $\underline{x+1}$ of $x+1$. Otherwise it is undefined.

  $\widehat{\mathsf{n}}(w)$ returns the string $-w$

  $\widehat{\mathsf{q}}(u,v)$ returns the string $u\,\#\,v$

  $\widehat{=}(u,v)$ returns TRUE iff $u = \underline{x_1}\,\#\,\underline{x_2}$, $v = \underline{y_1}\,\#\,\underline{y_2}$, and $x_1/x_2 = y_1/y_2$, for some $x_1, y_1 \in \mathbb{Z}$ and $x_2, y_2 \in \mathbb{Z}^+$,

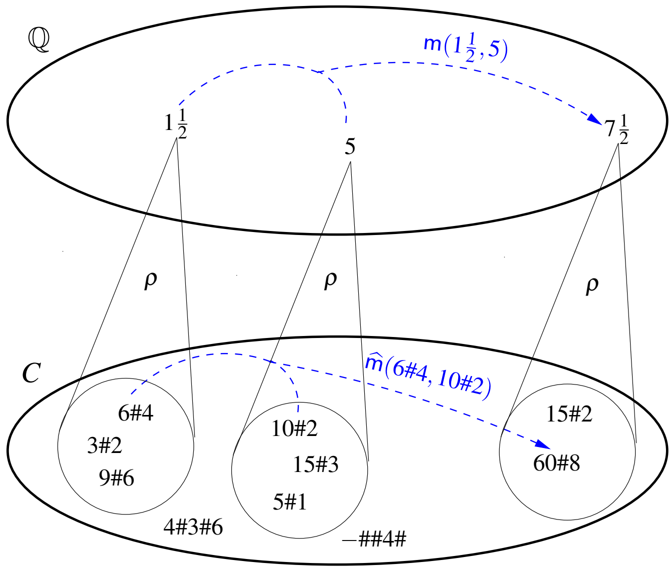

Fig. 6.3 Representing abstract rational numbers by concrete strings, and implementing the multiplication function.

- The implementation $\widehat{\mathsf{m}}(u,v)$ of multiplication is the following: If $u = \underline{x}_1 \# \underline{x}_2$ and $v = \underline{y}_1 \# \underline{y}_2$, where $x_1, y_1 \in \mathbb{Z}$ and $x_2, y_2 \in \mathbb{Z}^+$, then the implementation returns the string $\underline{x_1 \cdot y_1} \# \underline{x_2 \cdot y_2}$. Otherwise, multiplication is not defined.

  See Figure 6.3.

  An implementation of multiplication that reduces the resulting fraction is as honest an implementation as the above one. □

6.5 Honest Computability

We first demonstrate the reasonableness of our demands on implementations by taking a careful look at honest "effective" computations.

Since we believe the Church-Turing Thesis in light of the arguments and proofs in [4, 12], we shall use the term *effective computation* to stand for the functionality of a Turing machine over strings or of a recursive function over the natural numbers. Let TM denote the family of Turing-machine computable string functions and REC, the family of recursive numerical functions.

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

Definition 7 (Honest Computability). A function over an abstract domain is *honestly computable* if it, and generators of its domain, can be honestly implemented by the recursive functions REC over the natural numbers (or by the Turing-computable functions TM over strings).

This definition guarantees that concrete representations for all the elements of the abstract domain can also be effectively generated.[4] It follows that, if the Turing family TM implements a family consisting of an arbitrary function f over a domain A and a finite set of generators for A, then f is—by definition—honestly computable.

Lemma 8. *If the recursive functions simulate a set of generators via some representation, then that representation—restricted to be univalued—can be effectively defined by structural induction.*

For example, consider these generators G for the naturals $\mathbb{N}$: zero, o, and successor, s. Suppose they are mapped to the constant $\widehat{\mathsf{o}}$ and the recursive function $\widehat{\mathsf{s}}$, respectively, under a (multivalued) representation $\eta : \mathbb{N} \rightrightarrows \mathbb{N}$. Define ρ, a single-valued restriction of η, by (structural) induction (over H, the ground terms of G) as follows:

$$\begin{aligned} \rho(\mathsf{o}) &:= \widehat{\mathsf{o}} \\ \rho(\mathsf{s}(n)) &:= \widehat{\mathsf{s}}(\rho(n)) \end{aligned}$$

We have by induction that $\rho(n) \in \eta(n)$ for all n.

We get also that every function $f : \mathbb{N} \to \mathbb{N}$ that is implemented by a recursive $\widehat{f} : \mathbb{N} \to \mathbb{N}$ under η must also be recursive, since

$$f(n) = \rho^{-1}(\widehat{f}(\rho(n))) = \eta^{-1}(\widehat{f}(\eta(n)))$$

is the composition of computable functions. (The inverse ρ^{-1} of a single-valued computable representation is computable by search.)

A similar argument applies to other sets of generators for other abstract domains.

Specifically, under the above (lax) assumptions, if a concrete function $\widehat{f}$ is computable, then the implemented function f can in fact be programmed effectively as an abstract state machine (ASM) [16, 8]. ASMs are a framework providing a most general programming paradigm in which one can precisely express (ineffective as well as effective) algorithms over arbitrary domains. In our case, f may be effectively computed over the combined domain $A \uplus \mathbb{N}$ by programming:

$$f(\mathsf{g}(x_1,\ldots,x_\ell)) := \rho^{-1}(\widehat{f}(\widehat{\mathsf{g}}(\rho(x_1),\ldots,\rho(x_\ell))))$$

for each $\mathsf{g} \in \mathsf{G}$. For this to work, we presume the availability of an effective equality test for A.

To summarize the development so far, we would say that a function f over Abe's abstract domain A is honestly computed by Cay's implementation iff Cay can evaluate terms of the form $f(t_1,\ldots,t_\ell)$—or more generally terms over f and generators G of A—where the $t_i \in \mathsf{H}$ are terms over G, and Abe, the querier, can check the

[4] The developments in [4, 12, 5] do not directly address the issue of honesty.

results (using the equality predicate). Furthermore, f is deemed effective if Cay's implementation uses effective means (such as those provided by a Turing machine).

6.6 Honest Comparisons

The fact that honest implementations effectively generate representations for all abstract domain elements guarantees the "completeness" of the recursive functions and of Turing machines in the sense that no representation can enlarge its computational power.

Definition 9 (Completeness [3]). A family F is *complete* if it cannot simulate any strict superset of itself.

Theorem 10. *Consider a computational family F over the natural numbers $\mathbb{N}$, and suppose that the recursive functions REC simulate F and, furthermore, that $\mathsf{REC} \subseteq F$. Then $F = \mathsf{REC}$.*

Proof. We show that every function $h \in F$ is also in REC. Since $\mathsf{REC} \subseteq F$, we know the successor function s over $\mathbb{N}$ in also in F. Because REC implements F, there must be functions $\widehat{h}, \widehat{\mathsf{s}} \in \mathsf{REC}$, such that for every $\bar{n} \in \mathbb{N}^*$, $\widehat{h}(\rho(\bar{n})) \subseteq \rho(h(\bar{n}))$, and for every $n \in \mathbb{N}$, $\widehat{\mathsf{s}}(\rho(n)) \subseteq \rho(\mathsf{s}(n))$. (The notation S^* is used here and later for all finite tuples of elements of S.)

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

- Construct the vector
$$\widehat{n} = \langle \widehat{n}_1, \ldots, \widehat{n}_\ell \rangle = \langle \widehat{\mathsf{s}}^{\,n_1}(x_0), \ldots, \widehat{\mathsf{s}}^{\,n_\ell}(x_0) \rangle,$$
which represents $\bar{n}$, by choosing any $x_0 \in \rho(0)$ and applying $\widehat{s}$ to that value n_j times for the jth component.
- Compute
$$\widehat{k} := \widehat{h}(\widehat{n}) \; .$$
[If h is partial and diverges on $\bar{n}$, then the simulating function $\widehat{h}$ will likewise diverge on $\widehat{n}$.]
- Search for the number k that is represented by $\widehat{k}$, by computing
$$\min_{i \in \mathbb{N}} [\widehat{\mathsf{s}}^{\,i}(x_0) \mathrel{\widehat{=}} \widehat{k}].$$
This search is guaranteed to terminate after an iteration k, such that $\widehat{k} \in \rho(k)$, since there is a fixed finite set of truth values $\widehat{\text{TRUE}}$ and $\widehat{\text{FALSE}}$ to test for.
- At this moment, $\widehat{k}$ represents $k = h(\bar{n})$. $\square$

It follows that the recursive functions REC are complete in the defined sense. By the same token, Turing machines are complete (see [3]), whereas two-counter machines and the lambda calculus are not. Two-counter machines can neither square nor exponentiate [21], but famously implement all recursive functions via the (expansive) representation $\rho : n \mapsto 2^n$ (as shown by the late Marvin Minsky [19]).

We now know what it means for families to have the same computational power:

Definition 11 (Equipotence). Families $\widehat{F}$ and F are *equipotent* if they simulate each other.

The representations by means of which $\widehat{F}$ implements F and vice-versa need not be the same, even they both operate over the same domain.

For example, REC and TM are equipotent via representations like Gödel numbers and tally numbers.

By this definition, 2-counter machines are equipotent with 3-counter machines (though the former model includes strictly fewer functions), but not with 1-counter ones.[5]

If follows that the honest (and self-consistent) way to compare computational power (when representations are allowed), is to say that a family F' is *strictly more powerful* than another family F if F' simulates F but not vice-versa. This is in fact true for F', the recursive functions and F, the primitive recursive ones, but this popular claim requires showing that there is no (injective) representation whatsoever via which the primitive recursive functions can simulate all the recursive ones. See [3].

If an abstract function is computable whenever it has an honest recursive implementation, how can one show that an abstract function h is incomputable, short of trying all possible representations? The answer is that h is *incomputable* whenever there is at least one *bijective* representation that provides recursive implementations of the generators but a non-recursive implementation of h.

Theorem 12. *Every function h over an abstract domain with generators G that is honestly computable is also recursively implemented by each and every close numerical implementation of h (that is, an implementation in $\mathbb{N}$ via a bijective representation) that implements the operations in G by means of recursive functions.*

The reason we need a *bijective* counterexample to establish incomputability is that one can always have the part of the implementation that works with numbers outside the image $\rho(A)$ of a non-surjective representation ρ (like $\rho(n) = 2n$) do something outlandish (to odd numbers).

Proof. Let $\pi : A \leftrightarrow \mathbb{N}$ be a bijection from the abstract domain A of h, and let $\widetilde{h} : \mathbb{N} \to \mathbb{N}$ implement h under π. If $\widehat{h}$, the function that simulates h under some ρ, is recursive, then $\widetilde{h}$ must also be recursive. This is because

$$\widetilde{h}(n) = \pi(h(\pi^{-1}(n))) = \pi(\rho^{-1}(\widehat{h}(\rho(\pi^{-1}(n)))))$$

and, by Lemma 8, both ρ and π are computable from any standard numerical encoding of generator terms H over G. Hence, $\widetilde{h}(n)$ is recursive. $\square$

[5] "Combining these simulations, we see that two-counter machines are as powerful as arbitrary Turing machines (one-counter machines are strictly less powerful)" [17, p. 33]. But who says that one-counter machines cannot also simulate more than they can compute? They cannot [2, Thm. 40].

6.7 Honest Universality

A (partial) function ω is said to be "universal" for a whole family F of (partial) functions (such as all the recursive functions, for instance) by being supplied with the code $\ulcorner f \urcorner$ of the desired $f \in F$ as an extra argument. If ω works with a concrete domain C, whereas the functions in F operate on an abstract domain A, then representations $\rho : A \to C$ are in order once again. Then we would say that varyadic ω is universal for F (with respect to encoding $\ulcorner\cdot\urcorner : F \to C$ and representation $\rho : A \overset{1\text{-}1}{\rightrightarrows} C$) if

$$\omega(\ulcorner f \urcorner, \rho\bar{x}) \subseteq \rho(f(\bar{x}))$$

for all $f \in F$ and $\bar{x} \in A^*$ of the right length for the arity of f.

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 $\langle i, j\rangle := 2^i 3^j$. For strings, one usually uses an injection like $\langle u, w\rangle := u\,;w$, 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 an implementation of the universal function that does too much, computing even non-effective functions. For example, a naïve definition might simply ask that pairing be injective and say that ω is universal for some set F of functions if $f(x) = \omega(\langle \ulcorner f \urcorner, x\rangle)$ for all $f \in F$ and $x \in C$, for some arbitrary encoding $\ulcorner\cdot\urcorner : F \to C$ of functions. The problem is that an injective pairing could cheat and include the answer in the "pair". For Turing machines, say, the pair $\langle u, w\rangle$ might be represented as $u\,;w$ when machine u halts on input w and as $u\!:\!w$ when it doesn't. Better yet, one could map $\langle \ulcorner f \urcorner, y\rangle \mapsto [f(y), \ulcorner f \urcorner, y]$, 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 [7] and, later, Rogers [20] 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, $C = \{s^n(x_0)\}$ for some $x_0 \in C$) and a pairing function $\langle\cdot,\cdot\rangle : C \times C \xrightarrow{1\text{-}1} C$, the component-wise successor functions operate as follows: $s_1 : \langle a, b\rangle \mapsto \langle s(a), b\rangle$ and $s_2 : \langle a, b\rangle \mapsto \langle a, s(b)\rangle$. If s, s_1, and s_2 are all computable, then we will say that *pairing is effective*. This is because one can program pairing so that $\langle z, y\rangle := s_1^i(s_2^j\langle x_0, x_0\rangle)$, where $z = s^i(x_0)$ and $y = s^j(x_0)$. And if pairing is effective, then its two projections (inverses), $1\textsc{st} : \langle a, b\rangle \mapsto a$ and

2ND : $\langle a,b\rangle \mapsto b$, are likewise effective. (Generate all representations of pairs in a dovetailed, 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 $f(x)$ in the pair, even if it only can do that for f 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 bijectivity alone, sans computability, one could still hide a fair amount of incomputable 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 $3\langle i,z\rangle$, where $\langle\cdot,\cdot\rangle$ is a standard pairing; map $(0,z)$ to $3j+1$ when z is the (code of the) jth total (recursive) function; and map $(0,z)$ to $3k+2$ when z is the kth non-total (partial recursive) function. Now, let U be some standard computable universal function. Then, for y divisible by 3, $\omega(y) := U(y/3)$ would compute all the partial-recursive functions, whereas $\omega(y) := y \equiv 1 \pmod 3$ would compute the incomputable totality predicate when $y=(0,z)$ is not divisible by 3.

Definition 14 (Honest Universality). Let F be some family of unary functions over an abstract domain A. Unary function ω over concrete domain C is *universal* for F, via pairing function $\langle\cdot,\cdot\rangle$ over C, if $\{\lambda y.\,\omega\langle a,y\rangle \mid a \in C\}$ implements F. If, in addition, pairing is bijective, then we call the universal function *honest*.

That is, ω is universal if, for fixed representation $\rho : A \to C$ and encoding $\ulcorner\cdot\urcorner : F \to C$, we have $\omega\langle\ulcorner f\urcorner,\rho(x)\rangle = \rho(f(x))$, for $f \in F$ and $x \in A$. Of course, we are interested in the case where both pairing and the universal function are effectively computable.

Theorem 15 ([10]). *Let F be some family 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 F, via an effective pairing, then all the implemented functions in F are also computable.*

Suppose $F=\{f_z\}_z$ 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 refer to as the *universal property* of a unary numerical function

ω, namely, that $f_z(x) = \pi(\omega\langle z,x\rangle)$ for some recursive bijection π and effective (but perhaps dishonest) pairing $\langle\cdot,\cdot\rangle$.

The following follows from the definitions:

Theorem 16 ([10]). *If a function has the universal property, then it is honestly universal. Furthermore, there must exist an honest computable universal function.*

6.8 Honest Complexity

We turn now to the question of complexity of problems over abstract domains. But before one can measure complexity, one needs a measure of input size and a measure of the cost of a computation as a function of that size.

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. Since problems often involve more than one input value, we need to measure the size of tuples of terms.

Definition 17 (Size). A *size* measure for an abstract domain is a function $|\cdot| : \mathsf{H}^* \to \mathbb{N}$, where H^* is the set of tuples of (ground) terms over the generators of the domain.

Complexity is measured with respect to this size, whatever it may be.

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. Usually the size of a tuple is the sum of the sizes of its individual components.

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 order $\log n$. 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. It is also standard practice to store data in compressed form, and it can easily take exponential time and space to reconstruct before manipulating.

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. (RAMs are in fact nearly optimal for time and space [11].) 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 terms, as follows:

Definition 18 (Cost). The *cost* $\kappa(\widehat{h}(t_1,\ldots,t_\ell))$ of a concrete term $\widehat{h}(t_1,\ldots,t_\ell)$ is the cost of a computation that constructs the concrete values $c_i \in C$ arguments t_i and then computes $\widehat{h}(c_1,\ldots,c_\ell)$, the value of $\widehat{h}$ for the concrete values thus obtained.

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. Often, the declared size of the input is approximately the cost of constructing it, so the impact on complexity of including the cost of construction is negligible.

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 5). We may assume that every generator has a unique implementation. (Different implementations should have different names, thus refer to different, but 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 19 (Honest Complexity). Consider an abstract domain A with ground generator terms H and an honest implementation $\widehat{h} : C^\ell \to C$ over concrete domain C, implementing a function $h : A^\ell \to A$ over A. Let $m : \mathbb{N} \to \mathbb{N}$ be a complexity measure. Then we say that $\widehat{h}$ has *honest (worst-case) complexity* of at most m if $\kappa(\widehat{h}(\bar{t})) \leq m(|\bar{t}|)$, for all tuples $\bar{t} \in \mathsf{H}^\ell$ of terms.

Average and probabilistic complexities can be defined analogously.

While the complexity of implementing generators influences the complexity of implementing f, the complexity of implementing the equality relation need not affect it. For example with abstract graphs, equality checks are very involved, yet many graph operations need not check for graph equality (isomorphism). We do insist, however, that every implementation also implements the equality check in order to enforce a correct interpretation of the abstract domain—having the ability to use the equality implementation, Abe, the person posing instances of the function f, can verify whether the result of f's implementation is indeed proper. (Cf. Section 6.5 and the proof of Theorem 10.)

To sum up, to preclude dishonest measures of complexity, we require that the implementor Cay charge not only for calculating the answer to Abe's query, but also for building its native representation of the query from Abe's language of generators. That way, any new information hidden in the representation is put there by Cay and the costs incurred are charged for.

6.9 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 families, which implement functions, capture the essence of computational power more accurately than do decision families, which implement relations. For that reason, we based the notion of honest implementation and complexity, even for decision problems, on functions rather than on decision procedures. The underlying reason for the better adequacy of functions than relations is that the former can also comprise the means to generate the representations of objects.

As defined in Section 6.5, a computational family over a domain A is a set of functions $F \subseteq \{f : A^* \to A\}$; likewise a *decision family* over A is a set of relations $R \subseteq \{r \subseteq A^n \mid n \in \mathbb{N}\}$. Specific computational or decision families are defined via some internal mechanism, a *model of computation*, a point that will play a rôle in our arguments later.

Decision families are inherently incomplete, in that one can readily "increase" their power via a representation that adds some information on top of the represented element [3]. For example, let h be an incomputable decision problem over Σ^*, and consider the representation $\rho : \Sigma^* \to \Sigma^*$, where $\rho(w) = h(w)w$. (The representation just adds the incomputable bit $h(w)$ before the word w.) Then, Turing machines can "decide", via the representation ρ, both h and all of the ordinary Turing-decidable problems.

Surprisingly, the weak computational model of finite automata (FSAs) is already powerful enough to decide, via a suitable representation, *any* countable set of (decidable or undecidable) relations [13]. The representation hides with each domain element a finite amount of data of relevance to finitely-many relations, such that each decision procedure gets all the data it needs from the represented inputs.

Let Σ be the binary alphabet $\{0,1\}$ throughout the remainder of this section.

Lemma 20 ([13]). *For every countable set R of relations over the natural numbers $\mathbb{N}$, there is an injection $\rho : \mathbb{N} \to \Sigma^*$, such that the set FSA of finite automata simulates R via ρ, viewing a relation $r \subseteq \mathbb{N}$ as a Boolean function $r : \mathbb{N} \to \Sigma$.*

Accordingly, a FSA a computes the function $a : \Sigma^* \to \{\rho(0), \rho(1)\}$, returning $\rho(0)$ when the input word is rejected and $\rho(1)$ when accepted.

Proof. Let $r_1, r_2, \ldots$ be any enumeration of the relations in R. Define the representation $\rho : n \mapsto r_1(n)\, r_2(n) \cdots r_n(n)$. For every n, the length of $\rho(n)$ is n, and it gives explicit answers to the first n relational questions r_i. Now, for every relation $r_i \in R$, consider the FSA a_i depicted in Figure 6.4. One can easily verify that for each $m \in \mathbb{N}$, the automaton a_i accepts $\rho(m)$ iff $m \in r_i$. For an input word w of length

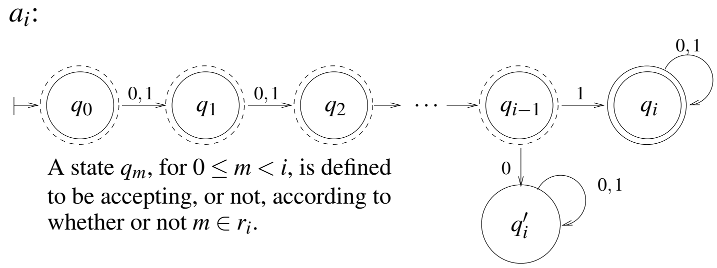

Fig. 6.4 The finite automaton a_i, which implements an arbitrary relation r_i via the representation ρ of the proof of Proposition 20.

$m \geq i$, a_i finds the answer whether $m \in r_i$ at the ith digit of w. For the finitely-many inputs of length $m < i$, representing numbers up to (but not including) i, the first i states of a_i are fixed to accept (and "return" $\rho(1)$) or reject (returning $\rho(0)$) the input word $\rho(m)$ according as to whether $m \in r_i$. $\square$

One might have presumed that this disturbing sensitivity to representations would be resolved by limiting representations to bijections, but this is unfortunately not the case, as shown in [13].

Theorem 21 ([13]). *For every countable set R of relations over $\mathbb{N}$ there is a bijection $\pi : \mathbb{N} \leftrightarrow \Sigma^*$, such that the set FSA of finite automata closely implements R via π.*

The above-decribed inherent incompleteness of decision families, that they can easily be enlarged by representing input differently, stems from their inability to generate the representation of the input. On the other hand, as shown in Theorem 10, the set of recursive functions is complete, in the sense that it cannot honestly implement an incomputable function, regardless of the choice of representation.

A question naturally arises considering the completeness of recursive functions (Theorem 10) and the inherent incompleteness of decision families (Propositions 20 and 21): Where does the proof of Proposition 20 break down if we try to modify it to demonstrate that every set of functions can be computed, via a suitable representation, by finite-state transducers (input-output automata)?

The answer is that, with the aim of computing a countable set of functions $\{f_1, f_2, \ldots\}$, the representation that is used in the proof of Proposition 20 may be generalized to something like $\rho(n) = f_1(n)\,\$\,f_2(n)\,\$\cdots\$\,f_n(n)$. Then, for every function f_i, there is indeed a transducer a_i, such that, for every n, we have $a_i(\rho(n)) = f_i(n)$. This, however, doesn't fit the bill. To properly represent f_i, we need for a_i to return $\rho(f_i(n))$, not $f_i(n)$. One might be tempted to suggest instead a representation η that already provides the represented values, as in $\eta(n) = \eta(f_1(n))\,\$\,\eta(f_2(n))\,\$\cdots\$\,\eta(f_n(n))$. This is, however, a circular definition: Let f_1 be the successor function s. Representing 1, we have $\eta(1) = \eta(\mathsf{s}(1)) = \eta(2) = \eta(\mathsf{s}(1))\,\$\,\eta(\mathsf{s}(2)) = \cdots$.

Finally, it may be worthwhile noting that Turing's halting problem is immune to the particular representation of programs [1], as are similar problems, though—as we have seen—decision procedures are quite sensitive to the representation of input data. Here the problem is to decide whether machine TM_m halts on input string w. Problem instances are pairs $\langle \ulcorner m \urcorner, w \rangle$ consisting of an encoding $\ulcorner m \urcorner$ of the machine along with the input w or an encoding $\ulcorner w \urcorner$ thereof. However, the pairing function itself must be honest, as explained in Section 6.7. In that situation, the encoding of any given machine (or computer program) can only hide a finite amount of information, not enough to answer the halting problem for all inputs to the machine, though the representation of those inputs themselves could hide the answers.

6.10 Discussion

We have proposed to regard an abstract function as honestly and effectively implemented if it can be effectively computed given its arguments as constructor terms. Analogously, we suggest that the cost of generating concrete representations of queries be included in the honestly considered cost of deciding problems regarding abstract objects.

Demanding of an implementation that it also generate its internal representations of the input from an abstract term description of that input precludes the hiding of incomputability in the representation used for concrete implementations and, likewise, obviates cheating on complexity problems by giving away the answer in the representation. 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 general, the sublinearity of various deterministic algorithms, ignoring the cost of constructing the input, strongly depends on how the input is presented.)

Often, one analyzes alternative representations with respect to the complexity of a set of basic functions. Considering graphs, for example, it is common to compare the adjacency-list representation with adjacency matrices. While the former provides greater efficiency for adding a vertex, it has a steeper edge removal cost. In these cases, the complexity of generating the input representation might be considered another aspect of the complexity tradeoffs.

> Many persons who are not conversant with mathematical studies, imagine that because the business of the [Analytical] engine is to give its results in *numerical notation*, the *nature of its processes* must consequently be *arithmetical* and *numerical*, rather than algebraical and analytical. This is an error. The engine can arrange and combine its numerical quantities exactly as if they were *letters* or any other *general* symbols; and in fact it might bring out its results in algebraical *notation*, were provisions made accordingly. It might develope three sets of results simultaneously, viz. *symbolic* results; *numerical* results (its chief and primary object); and *algebraical* results in *literal* notation. This latter however has not been deemed a necessary or desirable addition to its powers, partly because the necessary arrangements for effecting it would increase the complexity and extent of the mechanism to a degree that would not be commensurate with the advantages, where the main object of the invention is to translate into *numerical* language general formulae of analysis already known to us, or whose laws of formation are known to us. But it would be a mistake to

> suppose that because its *results* are given in the *notation* of a more restricted science, its *processes* are therefore restricted to those of that science. The object of the engine is in fact to give the *utmost practical efficiency* to the resources of *numerical interpretations* of the higher science of analysis, while it uses the processes and combinations of this latter.
>
> —Augusta Ada Lovelace, Notes to "On Babbage's Analytical Engine" (1843)
> [emphasis in the original]

References

1. Abramsky, S.: Undecidability of the halting problem: A self-contained pedagogical presentation (2011). Unpublished note
2. Boker, U.: The Influence of Domain Interpretations on Computational Models. PhD thesis, Tel Aviv University, School of Computer Science (2008)
3. Boker, U., Dershowitz, N.: Comparing computational power. Logic Journal of the IGPL **14** (2006) 633–648. Available at `http://nachum.org/papers/ComparingComputationalPower.pdf` (viewed February 9, 2016)
4. Boker, U., Dershowitz, N.: The Church-Turing thesis over arbitrary domains. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.): Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday. Lecture Notes in Computer Science, Vol. 4800. Springer (2008) 199–229. Available at `http://nachum.org/papers/ArbitraryDomains.pdf` (viewed February 9, 2016)
5. Boker, U., Dershowitz, N.: Three paths to effectiveness. In: Blass, A., Dershowitz, N., Reisig, W. (eds.): Fields of Logic and Computation: Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday. Lecture Notes in Computer Science, Vol. 6300, Berlin, Germany, Springer (2010) 36–47. Available at `http://nachum.org/papers/ThreePathsToEffectiveness.pdf` (viewed February 9, 2016)
6. Cai, J.y.: Computations over infinite groups. In: Budach, L. (ed.): Proceedings of the 8th International Symposium on Fundamentals of Computation Theory (FCT), Gosen, Germany. Lecture Notes in Computer Science, Vol. 529. Springer (1991) 22–32
7. Davis, M.: The definition of universal Turing machine. Proceedings Amer. Math. Soc. **8** (1957) 1125–1126
8. Dershowitz, N.: The generic model of computation. In: Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011, July 2012, Zürich, Switzerland). Electronic Proceedings in Theoretical Computer Science (2012) 59–71. Available at `http://nachum.org/papers/Generic.pdf` (viewed February 9, 2016)
9. Dershowitz, N., Falkovich, E.: A formalization and proof of the Extended Church-Turing Thesis. In: Proceedings of the Seventh International Workshop on Developments in Computational Models (DCM 2011). Electronic Proceedings in Theoretical Computer Science, Vol. 88, Zürich, Switzerland (2011) 72–78. Available at `http://nachum.org/papers/ECTT_EPTCS.pdf` (viewed February 9, 2016)
10. Dershowitz, N., Falkovich, E.: Honest universality. Special issue of the Philosophical Transactions of the Royal Society A **370** (2012) 3340–3348. Available at `http://nachum.org/papers/HonestUniversality.pdf` (viewed February 9, 2016)
11. Dershowitz, N., Falkovich, E.: The invariance thesis (2015). Submitted. Available at `http://nachum.org/papers/InvarianceThesis.pdf` (viewed February 9, 2016)
12. Dershowitz, N., Gurevich, Y.: A natural axiomatization of computability and proof of Church's Thesis. Bulletin of Symbolic Logic **14** (2008) 299–350. Available at `http://nachum.org/papers/Church.pdf` (viewed February 9, 2016)
13. Endrullis, J., Grabmayer, C., Hendriks, D.: Regularity preserving but not reflecting encodings. In: Proceedings of the 30th Annual ACM/IEEE Symposium on Logic in Computer

Science (LICS), Kyoto, Japan. IEEE (2015) 535–546. Available at `http://arxiv.org/pdf/1501.04835v1.pdf` (viewed February 9, 2016)
14. Engeler, E.: Formal Languages: Automata and Structures. Lectures in Advanced Mathematics. Markham Publishing Company, Chicago, IL (1968)
15. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman, New York, NY (1979)
16. Gurevich, Y.: Sequential abstract state machines capture sequential algorithms. ACM Transactions on Computational Logic **1** (2000) 77–111
17. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press, Cambridge, MA (2000)
18. Knuth, D.E.: Algorithm and program; information and data. Communications of the ACM **9** (1968) 654
19. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs, N.J. (1967)
20. Rogers, Jr., H.: On universal functions. Proceedings of the American Mathematical Society **16** (1965) 39–44. Available at `http://www.jstor.org/stable/2033997` (viewed February 9, 2016)
21. Schroeppel, R.: A two counter machine cannot calculate 2^{N}. Artificial Intelligence Memo #257, Massachusetts Institute of Technology, A. I. Laboratory (1972). Available at `ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-257.pdf` (viewed February 9, 2016)
22. Shapiro, S.: Acceptable notation. Notre Dame Journal of Formal Logic **23** (1982) 14–20
23. Weihrauch, K.: Computability. EATCS Monographs on Theoretical Computer Science, Vol. 9. Springer-Verlag, Berlin (1987)