Infinite sets that admit fast exhaustive search

Martín Escardó
School of Computer Science, University of Birmingham, UK

Abstract. Perhaps surprisingly, there are infinite sets that admit mechanical exhaustive search in finite time. We investigate three related questions: What kinds of infinite sets admit mechanical exhaustive search in finite time? How do we systematically build such sets? How fast can exhaustive search over infinite sets be performed?

Keywords. Higher-type computability and complexity, Kleene–Kreisel functionals, PCF, Haskell, topology.

1. Introduction

A wealth of problems of interest have the following form:

given a set K and a property p, check whether or not all *elements of K satisfy p.*

We say that K is *exhaustible* if this problem can be algorithmically solved in finite time, for any decidable property p, uniformly in p. Thus, the input of the algorithm is p and the output is the truth value of the statement that all elements of K satisfy p. In the realm of higher-type computability theory, the algorithm has type $(C \to \mathbb{B}) \to \mathbb{B}$, where C is a type, $K \subseteq C$, and $\mathbb{B}$ is the type of booleans, so that $(C \to \mathbb{B})$ is the type of decidable predicates on C.

Clearly, finite sets of computable elements are exhaustible. What may be rather unclear is whether there are *infinite* examples. Intuitively, there can be none: how could one possibly check infinitely many cases in finite time? This intuition is correct when K is a set of natural numbers: it is a theorem that, in this case, K is exhaustible if and only if it is finite. But a proof is non-trivial, usually by reduction to the halting problem, goes beyond cardinality considerations, and relies on particular properties of the set of natural numbers that don't necessarily hold for other infinite sets.

Thus, the question remains, which kinds of infinite sets, if any, are exhaustible? It turns out that there is a rich supply. A first example, the Cantor set of infinite sequences of binary digits, goes back to the 1950's, as discussed in the related-work paragraph below.

Our primary contribution is a comprehensive investigation of such sets from the point of view of higher-type computability theory [18]. We develop tools for systematically building them and a characterization: they are closed under intersections with decidable sets, under the formation of computable images and of finite and countably infinite products, and in the non-empty case they are precisely the computable images of the Cantor set.

If a problem of the above form has a negative solution, one would like to be able to algorithmically find a counterexample. If this is possible, we say that the set K is *searchable*. It turns out that exhaustibility coincides with searchability, which supports the intuitive understanding of exhaustive search, but involves an elaborate construction.

The specifications of all of our algorithms can be understood without much background, but an understanding of the working of some of the algorithms requires a fair amount of topology, in addition to computability theory. The closure properties and characterization of exhaustibility resemble those of compactness in topology. This is no accident: exhaustible sets are to compact sets as computable functions are to continuous maps. This plays a crucial role in the correctness proofs of some of the algorithms, and, indeed, in their very construction.

Our secondary contribution is a preliminary investigation of efficiency and complexity. We have promising experimental results, implemented in the language Haskell [11], and tentative theoretical explanations. Here is our running example, with a gap to be filled, where we assume a previously defined type `N` of unbounded size natural numbers and a type `Bit` of binary digits:

```
type Cantor = N -> Bit
foreveryC :: (Cantor -> Bool) -> Bool
foreveryC p = ...
equalC :: (Cantor -> N) -> (Cantor -> N) -> Bool
equalC f g = foreveryC(\a -> f a == g a)
```

Now consider the following three functions:

```
f,g,h :: Cantor -> N
f a = a(10*a(3^80)+100*a(4^80)+1000*a(5^80))
g a = a(10*a(3^80)+100*a(4^80)+1000*a(6^80))
h a = if a(4^80) == 0 then a j else a(100+j)
  where i = if a(5^80) == 0 then 0 else 1000
        j = if a(3^80) == 1 then 10+i else i
```

The queries "`equalC f g`" and "`equalC f h`" answer `False` and `True` respectively, in less than $3s$ together in a 1.7GHz machine under the Glasgow Haskell compiler. This is in stark contrast with the algorithms we have been able to find in the literature [4], which take at least $2^{6^{80}}$ computation steps, as they are exponential in the modulus of uniform continuity.

We emphasize that the above algorithms, and all higher-type algorithms developed in this paper, use their functional inputs as black boxes, with no knowledge of their source code.

22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)
0-7695-2908-9/07 $25.00 © 2007 IEEE

Related work. According to personal communication by Normann, computability of Brouwer's Fan functional was known in the late 1950's. This immediately gives rise to the exhaustibility of the Cantor space. A number of authors have considered definability of the Fan functional in various formal systems. Normann [18] cites Tait (1958, unpublished), Gandy (around 1982, unpublished) and Berger [4] (1990). Tait showed that the Fan functional is not definable from Kleene's schemes S1–S9 interpreted over *total* functionals. Berger showed that it is PCF definable, and, in order to do that, he first explicitly defined a search functional for the Cantor space. Here PCF is an applied simply-typed lambda-calculus with arithmetic and fixed-point recursion [23, 19]. Berger observed that, for *partial functionals*, PCF definability coincides with S1–S9 definability. Then Hyland informed the community that Gandy was aware of the S1–S9 definability of the Fan functional for the partial interpretation of Kleene's schemes, although Gandy's definition seems to be lost.

Totality assumptions. Some of the above results crucially rely on a notion of totality. For example, to show that exhaustible sets are searchable, we need to assume that they consist of total elements. But there are two contenders for a notion of totality in higher-type computation, namely Kleene–Kreisel totality and hereditarily effective totality. Our results hold for the former but fail for the latter. This failure is to be expected: it is well known that, for the hereditarily effective notion, there is no total Fan functional [3], and hence the set of total elements of the Cantor type cannot be exhaustible. Put another way, the above algorithms for the Fan functional are total in the Kleene–Kreisel sense, but not in the hereditarily effective sense.

Organization. 2. Higher-type computability (background). 3. Exhaustible and searchable sets (definitions and basic properties). 4. Building new searchable sets from old (image and product). 5. Topological aspects of exhaustibility (compactness of exhaustible sets of total elements, used to derive the algorithms of Section 6). 6. Characterization of searchability. 7. Experiments and tentative explanations. 8. Concluding remarks.

2. Higher-type computability

As discussed in e.g. [18, 13, 12], there are many equivalent approaches to higher-type computation. Kleene defined the total functionals directly, but it has been found more convenient to work with the larger collection of partial functionals and isolate the total ones within them, as done by Kreisel. The approaches are equivalent, and such total functionals are often referred to as *Kleene–Kreisel functionals*. It turns out that, as discussed by Normann [18], this coincides with another approach known in the computer-science community: equivalence classes of total functionals on Scott domains.

Simple types. The *simple types* are defined by induction as

$$\sigma, \tau ::= o \mid \iota \mid \sigma \times \tau \mid \sigma \to \tau,$$

with usual rules for bracketing, where o and ι are ground types for booleans and natural numbers respectively.

Partial functionals. For each type σ, define a Scott domain D_σ of *partial functionals* of type σ by induction as follows:

$$D_o = \mathcal{B} = \mathbb{B}_\perp, \quad D_\iota = \mathcal{N} = \mathbb{N}_\perp,$$
$$D_{\sigma\times\tau} = D_\sigma \times D_\tau,$$
$$D_{\sigma\to\tau} = (D_\sigma \to D_\tau) = {D_\tau}^{D_\sigma}.$$

Here

$$\mathbb{B} = \{\mathrm{ff}, \mathrm{tt}\}$$

is the set of booleans and the products and exponentials are calculated in the cartesian closed category of continuous maps of Scott domains, where a *Scott domain* is an algebraic, bounded complete, and directed complete poset [1].

Total functionals. For each type σ, define, again by induction, a set $T_\sigma \subseteq D_\sigma$ of *total functionals* and a relation $\sim_\sigma$ on D_σ as follows, where γ ranges over ground types:

$$T_o = \mathbb{B}, \quad T_\iota = \mathbb{N},$$
$$x \sim_\gamma y \iff x, y \in T_\gamma \text{ and } x = y.$$
$$T_{\sigma\times\tau} = T_\sigma \times T_\tau,$$
$$(x, x') \sim_{\sigma\times\tau} (y, y') \iff x \sim_\sigma y \wedge x' \sim_\tau y',$$
$$T_{\sigma\to\tau} = \{f \in D_{\sigma\to\tau} \mid f(T_\sigma) \subseteq T_\tau\},$$
$$f \sim_{\sigma\to\tau} g \iff \forall x \sim_\sigma y. f(x) \sim_\tau g(y).$$

Then the set T_σ can be recovered from the relation $\sim_\sigma$ as $x \in T_\sigma \iff x \sim_\sigma x$, and the relation can be recovered from the set as $x \sim_\sigma y \iff x \sqcap y \in T_\sigma \iff x, y \in T_\sigma$ and x and y are bounded above (see e.g. [5] and [21]). In particular, $\sim_\sigma$ is an equivalence relation on T_σ.

Computability. A partial functional is computable iff it is PCF-definable from parallel-or and parallel-exists [19]. This is a theorem, but we take it as our definition. All computable functionals we construct are defined in PCF without parallel extensions. This definition includes, in particular, total functionals. An interesting fact, which we don't use, is that every *total* functional definable in PCF with parallel extensions is equivalent to one definable in PCF without parallel extensions [16]. Although the notion of totality plays a crucial role in the present work, most functionals we consider are not total by nature. But they produce total outputs for certain total inputs. For example, given a sequence of search operators, the countable-product functional developed in Section 4 produces a search operator for the product, but if the total input functionals are not search operators, then the result is not total in general.

Kleene–Kreisel functionals. The remainder of this section, which discusses material needed to construct the algorithms of Section 6, can be postponed until Section 5.

For each type σ, define a set C_σ of *Kleene–Kreisel functionals* of type σ and a surjection $\rho_\sigma : T_\sigma \to C_\sigma$ as follows, so that

$$C_\sigma \cong T_\sigma / \sim_\sigma .$$

For ground types and product types, define

$$C_o = T_o, \quad C_\iota = T_\iota, \quad \rho_\gamma(x) = x.$$
$$C_{\sigma\times\tau} = C_\sigma \times C_\tau, \quad \rho_{\sigma\times\tau} = \rho_\sigma \times \rho_\tau.$$

For function types, consider the diagram

$$\begin{tikzcd}
& D_\sigma \arrow[d, "f"'] \arrow[dr, phantom, "(1)"] & T_\sigma \arrow[l, hook'] \arrow[r, two heads, "\rho_\sigma"'] \arrow[d, dashed] \arrow[dr, phantom, "(2)"] & C_\sigma \arrow[d, dashed, "\phi"] \\
(\dagger) & D_\tau & T_\tau \arrow[l, hook'] \arrow[r, two heads, "\rho_\tau"] & C_\tau.
\end{tikzcd}$$

The square (1) commutes for some map $T_\sigma \to T_\tau$ if and only if $f \in T_{\sigma\to\tau}$, and in this case the map is uniquely determined as the (co)restriction of f. Moreover, in this case, there is a unique map ϕ making the square (2) commute, because ρ_σ is a surjection. We define

$$C_{\sigma\to\tau} = \{\phi\colon C_\sigma \to C_\tau \mid \exists f \in T_{\sigma\to\tau}.\text{(2) commutes}\},$$
$$\rho_{\sigma\to\tau}(f) = \text{the unique } \phi \text{ such that (2) commutes.}$$

Then for any σ and all $x, y \in D_\sigma$, we have that $x \sim_\sigma y$ iff $x, y \in T_\sigma$ and $\rho_\sigma(x) = \rho_\sigma(y)$

If ($\dagger$) commutes, we say that f is a *realizer* of ϕ. A Kleene–Kreisel functional is computable iff it has a computable realizer.

Lemma 2.1. *Every C_σ is a computable retract of $C_{\tau\to\iota}$ for some τ.*

Topological aspects of the Kleene–Kreisel functionals. A proof of the following inductive topological characterization, attributed to Hyland, can be found in [15].

Lemma 2.2. *Endow T_σ with the relative Scott topology and C_σ with the quotient topology of the surjection ρ_σ.*

1. *C_γ has the discrete topology for γ ground,*
2. *$C_{\sigma\times\tau} = C_\sigma \times C_\tau$ and*
3. *$C_{\sigma\to\tau} = {C_\tau}^{C_\sigma}$,*

where the product and exponential are calculated in the cartesian closed category of Hausdorff k-spaces.

For a brief treatment of k-spaces, also known as compactly generated spaces, see e.g. [14], and, for a more detailed one, see e.g. [9] or the references contained therein. A set is called *clopen* if it is both closed and open.

Lemma 2.3. *For every clopen $U \subseteq C_\sigma$ there is a total predicate $p \in (D_\sigma \to \mathcal{B})$ such that $\rho_\sigma^{-1}(U) \subseteq p^{-1}(\mathrm{tt})$ and $\rho_\sigma^{-1}(C_\sigma \setminus U) \subseteq p^{-1}(\mathrm{ff})$.*

Proof. Because U is clopen, its characteristic function $\chi_U\colon C \to \mathbb{B}$ is continuous, and hence so is the composite $i \circ \chi_U \circ \rho_\sigma\colon T_\sigma \to \mathcal{B}$, where $i\colon \mathbb{B} \to \mathcal{B}$ in the inclusion. Because T is dense in D (see e.g. [5]) and because Scott domains, and hence $\mathcal{B}$, are densely injective (see e.g. [10]), by definition of injectivity this extends to a continuous function $p\colon D_\sigma \to \mathcal{B}$. Then p is total by construction, and the extension property amounts to the above set inclusions. □

A space is *zero-dimensional* iff it has a base of clopen sets. It is an open problem whether the spaces C_σ are zero-dimensional [2, 17]. If they are, the following lemma becomes superfluous. The *zero-dimensional reflection $\mathcal{Z}C$* of a space C is obtained by taking the same set of points and the clopen sets as a base.

Lemma 2.4. *$\mathcal{Z}C_\sigma$ and C_σ have the same compact subsets.*

Proof. We first show that $\mathcal{KZ}C_\sigma = C_\sigma$ for any type σ, where $\mathcal{K}$ is the coreflector into the category of k-spaces. The property $\mathcal{KZ}C = C$ is easily seen to be inherited by retracts, and hence, by Lemma 2.1, it is enough to consider $\sigma = \tau \to \iota$, and hence $C = \mathbb{N}^Y$ for some k-space Y. Exponentials in k-spaces are given by the k-coreflection of the compact-open topology on the set of continuous maps. When the target is $\mathbb{N}$, the compact-open topology is clearly zero-dimensional and Hausdorff. Now, it is easy to see that $\mathcal{KZ}C = C$ iff there is some zero-dimensional topology whose k-reflection is C, and hence we are done. The result then follows from the well-known fact that a Hausdorff space has the same compact sets as its k-coreflection. □

3. Exhaustible and searchable sets

We now formulate the central notions investigated in this work. Through this paper, $D = D_\sigma$ and $D' = D_{\sigma'}$ for arbitrary simple types σ and σ'.

Definition 3.1. If K is a subset of D, we say that a predicate $p \in (D \to \mathcal{B})$ is *defined on* K if $p(x) \neq \bot$ for every $x \in K$.

Definition 3.2. We say that a set $K \subseteq D$ is *exhaustible* if there is a computable functional $\forall_K\colon (D \to \mathcal{B}) \to \mathcal{B}$ such that for any $p \in (D \to \mathcal{B})$ defined on K,

$$\forall_K(p) = \begin{cases} \mathrm{tt} & \text{if } p(x) = \mathrm{tt} \text{ for all } x \in K, \\ \mathrm{ff} & \text{if } p(x) = \mathrm{ff} \text{ for some } x \in K. \end{cases}$$

Such a functional is not uniquely determined, because its behaviour is not specified for predicates p that are not defined on K. For the sake of clarity, we shall often write "$\forall_K(\lambda x. \dots)$" as "$\forall x \in K. \dots$".

Clearly, it is equivalent to instead require the existence of a computable functional $\exists_K\colon (D \to \mathcal{B}) \to \mathcal{B}$ such that for any $p \in (D \to \mathcal{B})$ defined on K,

$$\exists_K(p) = \begin{cases} \mathrm{tt} & \text{if } p(x) = \mathrm{tt} \text{ for some } x \in K, \\ \mathrm{ff} & \text{if } p(x) = \mathrm{ff} \text{ for all } x \in K, \end{cases}$$

because such functionals are inter-definable as $\exists_K(p) = \neg\forall_K(\lambda x.\neg p(x))$ and $\forall_K(p) = \neg\exists_K(\lambda x.\neg p(x))$, and hence we'll freely switch between them.

Definition 3.3. We say that a set $K \subseteq D$ is *searchable* if there is a computable functional $\varepsilon_K : (D \to \mathcal{B}) \to D$ such that, for every predicate $p \in (D \to \mathcal{B})$ defined on K,

1. $\varepsilon_K(p) \in K$, and
2. $p(\varepsilon_K(p)) = \mathrm{tt}$ if $p(x) = \mathrm{tt}$ for some $x \in K$.

Again, notice that ε_K is not uniquely determined by K.

Thus, $\varepsilon_K(p)$ is an example of an element of K for which p holds, if such an element exists. But notice that we require that $\varepsilon_K(p) \in K$ even if there is no such example. With $1 = \{\star\}$, an equivalent definition, which will not be used, is that

1. K has a computable element e_K, and
2. there is $\varepsilon'_K : (D \to \mathcal{B}) \to 1 + D$ computable such that $\varepsilon'_K(p) = \star$ if there is no example, and otherwise $\varepsilon'_K(p) \in K$ and $p(\varepsilon'_K(p)) = \mathrm{tt}$.

In fact, given ε_K one can define $e_K = \varepsilon_K(\lambda x.\,\mathrm{tt})$ and

$$\varepsilon'_K(p) = \text{if } p(\varepsilon_K(p)) \text{ then } \varepsilon_K(p) \text{ else } \star.$$

Conversely, given ε'_K and e_K as specified, one can define

$$\varepsilon_K(p) = \text{if } \varepsilon'_K(p) = \star \text{ then } e_K \text{ else } \varepsilon'_K(p).$$

Lemma 3.4. *Searchable sets are exhaustible.*

Proof. Define $\exists_K(p) = p(\varepsilon_K(p))$. □

The empty set is exhaustible with realizer $\forall_\emptyset(p) = \mathrm{tt}$, but it is not searchable because the condition $\varepsilon_\emptyset(p) \in \emptyset$ cannot hold. But we shall see in Section 6 that, for non-empty *entire* sets, defined below, the two notions turn out to agree, although with a non-trivial construction and proof.

Definition 3.5. We say that a set K is *entire* if it consists of total elements and is closed under total equivalence.

Notice that if p is total then it is defined on every entire set. Even if p is not total and K is not entire, $p(x) = p(x')$ for all $x \sim x'$ in K, because if $x \sim x'$ then x and x' are bounded above and hence so are $p(x)$ and $p(x')$, which then must be equal as they are non-bottom by definition. But if $x \in K$ and $x' \sim x$ for x' outside K, it doesn't follow that $p(x') \neq \bot$ (consider e.g. $K = \{\lambda i.\,\mathrm{tt}\}$ for $\sigma = \iota \to o$ and $p(\alpha) = \alpha(\bot)$).

Let $D^\omega = (\mathcal{N} \to D)$ and, for any sequence K_i of subsets of D, let $\prod_i K_i$ be the set of functions $\alpha \in D^\omega$ with $\alpha_i = \alpha(i) \in K_i$ for all $i \in \mathbb{N} \subseteq \mathcal{N}$. The following closure properties of entire sets are easily verified:

Lemma 3.6.

1. *If $p \in (D \to \mathcal{B})$ is defined on $K \subseteq D$ and K is entire, then so is the set $K \cap p^{-1}(\mathrm{tt}) = \{x \in K \mid p(s) = \mathrm{tt}\}$.*
2. *If $K \subseteq D$ and $K' \subseteq D'$ are entire, so is $K \times K' \subseteq D \times D'$*
3. *If K_i is a sequence of entire subsets of D, then $\prod_i K_i$ is an entire subset of D^ω.*

Definition 3.7. The image of an entire set by a total function doesn't need to be entire, but it consists of total elements, and hence its closure under total equivalence is entire. We refer to this as its *entire image*. (Thus, entire images are defined for total functions and entire sets only.)

We implement part of the above in the language Haskell as follows, where the letter `d` is a type variable corresponding to the domain D, and the definition of `forsome` corresponds to the proof of Lemma 3.4.

```
type Searcher d = (d -> Bool) -> d
type Quantifier d = (d -> Bool) -> Bool

forsome, forevery :: Searcher d -> Quantifier d
forsome k p = p(k p)
forevery k p = not(forsome k(\x -> not(p x)))
```

4. Building new searchable sets from old

We develop algorithms that show that exhaustible and searchable sets are closed under various constructions. Starting from the finite sets, this allows one to build plenty of infinite searchable sets, and in particular to fill the gap in the introduction (to be revisited in Section 7).

Proposition 4.1. *For any $p \in (D \to \mathcal{B})$ defined on $K \subseteq D$, if K is exhaustible then so is the set*

$$K_p = K \cap p^{-1}(\mathrm{tt}) = \{x \in K \mid p(x) = \mathrm{tt}\}.$$

Moreover, if K is searchable and K_p is non-empty, then K_p is searchable.

Proof. Define

$$\exists_{K_p}(q) = \exists_K(p \wedge q),$$
$$\varepsilon_{K_p}(q) = \text{if } \exists_K(p \wedge q) \text{ then } \varepsilon_K(p \wedge q) \text{ else } \varepsilon_K(p),$$

where $u \wedge v = \text{if } u \text{ then } v \text{ else } \mathrm{ff}$, and $p \wedge q$ is defined pointwise. □

Proposition 4.2. *Exhaustible and searchable sets are closed under the formation of computable images, and also under the formation of computable entire images.*

Proof. Given $f : D \to D'$ and $K \subseteq D$ exhaustible, define

$$\forall_{f(K)}(q) = \forall x \in K.q(f(x)).$$

This proves closure of exhaustible sets under images. Regarding entire images of entire exhaustible sets, if f is total and K is entire with entire image L, then we can take $\forall_L = \forall_{f(K)}$. To verify this, let q be defined on L. Then q is defined on $f(K) \subseteq L$, and hence if $q(l) = \mathrm{tt}$ for all $l \in L$, then $\forall_L(q) = \mathrm{tt}$. If, on the other hand, $q(l) = \mathrm{ff}$ for some $l \in L$, then $l \sim f(x)$ for some $x \in K$. But then $q(f(x)) = \mathrm{ff}$, and so $\forall_T(q) = \mathrm{ff}$, which concludes the verification.

For $K \subseteq D$ searchable, define

$$\varepsilon_{f(K)}(q) = f(\varepsilon_K(\lambda x.q(f(x))).$$

That is, first find x such that $q(f(x))$ holds, using ε_K, and then apply f to it. This proves closure under images, and the argument for entire images is similar to the previous. □

Proposition 4.3. *Exhaustible and searchable sets are closed under the formation of finite products.*

Proof. For $K \subseteq D$ and $K' \subseteq D'$ exhaustible, define

$$\forall_{K\times K'}(p) = \forall x \in K.\forall x' \in K'.p(x, x').$$

For $K \subseteq D$ and $K' \subseteq D'$ searchable, to compute $\varepsilon_{K\times K'}(p)$ we first find $x \in K$ such that there is $x' \in K'$ with $p(x, x')$, and then find $x' \in K'$ such that $p(x, x')$, i.e.

$$x = \varepsilon_K(\lambda x.\exists x' \in K'.p(x, x')),$$
$$x' = \varepsilon_{K'}(\lambda x'.p(x, x')),$$

using the fact that searchable sets are exhaustible, and let $\varepsilon_{K\times K'}(p) = (x, x')$. □

We now consider countable products of searchable sets, assuming that the components K_i of the product $\prod_i K_i$ are all subsets of the same type D. Given search functionals

$$\varepsilon_{K_i} \in ((D \to \mathcal{B}) \to D),$$

we wish to find a search functional

$$\varepsilon_{\prod_i K_i} \in ((D^\omega \to \mathcal{B}) \to D^\omega).$$

We begin with an informal derivation and explanation of our algorithm (Definition 4.4), which iterates the idea of proof of Proposition 4.3. We let

$$\varepsilon_{\prod_i K_i}(p) = x_0x_1x_2 \ldots x_n \ldots,$$

where

$x_0 \in K_0$ is such that $\exists\alpha \in \prod_i K_{i+1}.p(x_0\alpha)$,
$x_1 \in K_1$ is such that $\exists\alpha \in \prod_i K_{i+2}.p(x_0x_1\alpha)$,
...
$x_n \in K_n$ is such that $\exists\alpha \in \prod_i K_{i+n+1}.p(x_0x_1 \ldots x_n\alpha)$,
...

The component x_n will be found using ε_{K_n}, and existential quantifications will be recursively reduced to search. To make this precise, we change notation. Given

$$\varepsilon \in ((D \to \mathcal{B}) \to D)^\omega,$$

such that ε_i searches over K_i, we wish to find

$$\Pi(\varepsilon) \in (D^\omega \to \mathcal{B}) \to D^\omega$$

that searches over $\prod_i K_i$. That is, we want a functional

$$\Pi\colon ((D \to \mathcal{B}) \to D)^\omega \to ((D^\omega \to \mathcal{B}) \to D^\omega)$$

that transforms a sequence of search operators over D into a search operator over D^ω:

$\Pi(\varepsilon)(p)(0) = x_0$ s.t. $\exists\alpha \in \prod_i K_{i+1}.p(x_0\alpha)$,
$\Pi(\varepsilon)(p)(1) = x_1$ s.t. $\exists\alpha \in \prod_i K_{i+2}.p(x_0x_1\alpha)$,
...
$\Pi(\varepsilon)(p)(n) = x_n$ s.t. $\exists\alpha \in \prod_i K_{i+n+1}.p(x_0x_1 \ldots x_n\alpha)$,
...

To complete the derivation of the functional Π, we reduce the existential quantification to a suitable recursive call to Π. If the functional Π is to meet its specification, $\Pi(\lambda i.\varepsilon_{i+n+1})$ should search over $\prod_i K_{i+n+1}$. But a searchable set is exhaustible by Lemma 3.4. To implement the proof of this lemma in our situation, for any given p, n, x_n, define

$$p_{n,x_n}(\alpha) = p(x_0x_1 \ldots x_{n-1}x_n\alpha)$$

Then

$$\exists\alpha \in \textstyle\prod_i K_{i+n+1}.p(x_0x_1 \ldots x_n\alpha)$$

is equivalent to

$$p_{n,x_n}(\Pi(\lambda i.\varepsilon_{n+i+1})(p_{n,x_n})).$$

To find x_n such that this holds, we use ε_n:

$$\Pi(\varepsilon)(p)(n) = \varepsilon_n(\lambda x_n.p_{n,x_n}(\Pi(\lambda i.\varepsilon_{n+i+1}))(p_{n,x_n})).$$

Because we don't want a different variable x_n for each n, we rename the variable to simply x:

Definition 4.4. The product functional Π, with type as specified above, is recursively defined by

$$\Pi(\varepsilon)(p)(n) = \varepsilon_n(\lambda x.p_{n,x}(\Pi(\lambda i.\varepsilon_{n+i+1}))(p_{n,x}))$$

where

$$p_{n,x}(\alpha) = p\left(\lambda i.\begin{cases}\Pi(\varepsilon)(p)(i) & \text{if } i < n,\\ x & \text{if } i = n,\\ \alpha_{i-n-1} & \text{if } i > n.\end{cases}\right)$$

Theorem 4.5. *If each ε_i searches over a set $K_i \subseteq D$ then $\Pi(\varepsilon)$ searches over $\prod_i K_i$.*

Proof. By construction, it is clear that $p(\Pi(\varepsilon)(p)) = \mathtt{tt}$ iff there is $\alpha \in \prod_i K_i$ such that $p(\alpha) = \mathtt{tt}$, provided the recursion converges in the sense that $\Pi(\varepsilon)(p) \in \prod_i K_i$.

To establish this, we first show that $p(\Pi(\varepsilon)(p)) \neq \bot$ for any p defined on $\prod_i K_i$. For ε and p fixed, and for each finite sequence β over D, define

$$W(\beta) = \Pi(\varepsilon)(\lambda\alpha.p(\beta\alpha)),$$

where $\beta\alpha$ denotes the concatenation of β and α. It is easy to see that $W(\beta)$ satisfies the equation

$$W(\beta) = xW(\beta x) \text{ where } x = \varepsilon_{|\beta|}(\lambda y.p(\beta yW(\beta y))).$$

Here βx denotes the sequence β extended by the element x, and $xW(\beta x)$ is the sequence with first element x followed by the sequence $W(\beta x)$, and $|\beta|$ denotes the length of β.

Claim: For any $\beta \in \prod_{i<|\beta|} K_i$, if $p(\beta W(\beta)) = \bot$ then then there is $x \in K_{|\beta|}$ such that $p(\beta xW(\beta x)) = \bot$, and hence such that also $p(\beta x\bot) = \bot$.

We establish the contrapositive, i.e. if $\lambda y.p(\beta yW(\beta y))$ is defined on $K_{|\beta|}$ then $p(\beta W(\beta)) \neq \bot$. By specification of $\varepsilon_{|\beta|}$, we have that $x := \varepsilon_{|\beta|}(\lambda y.p(\beta yW(\beta y)) \in K_{|\beta|}$. By the above equation for $W(\beta)$ and by the assumption, we have that $p(\beta W(\beta)) = p(\beta xW(\beta x)) \neq \bot$, which concludes the proof of the claim.

For the sake of contradiction, assume $p(\Pi(\varepsilon)(p)) = \bot$. Then $p(\beta W(\beta)) = \bot$ for β empty. Hence, repeatedly applying the above claim starting with β empty, we get $\alpha \in \prod_i K_i$ such that $p(\alpha_0\alpha_1 \cdots \alpha_n\bot) = \bot$ for every n, and hence $p(\alpha) = \bot$ by continuity of p, which contradicts the hypothesis that p is defined on $\prod_i K_i$ and concludes the proof that $p(\Pi(\varepsilon)(p)) \neq \bot$ for all ε such that ε_i searches over K_i and all p defined on $\prod_i K_i$.

Finally using this, an easy argument by course-of-values induction on n shows that, for all ε such that ε_i searches over K_i and all p defined on $\prod_i K_i$, we have that $\Pi(\varepsilon)(p)(n) \in K_n$. Therefore $\Pi(\varepsilon)(p) \in \prod_i K_i$. □

We don't have a corresponding result for exhaustible sets (but see Theorem 6.1 and [6, 8]). The following consequence gives a uniform continuity principle, where $\alpha =_n \beta$ means that $\alpha_i = \beta_i$ for all $i < n$, and $\alpha_{|n}$ is defined by $\alpha_{|n}(i) = \alpha_i$ for $i < n$ and $\alpha_{|n}(i) = \bot$ for $i \geq n$.

Corollary 4.6. *If $p \in (D^\omega \to \mathcal{B})$ is defined on a product $\prod_i K_i$ of searchable sets, then there is a number n such that for all $\alpha, \beta \in \prod_i K_i$,*

$$\alpha =_n \beta \implies p(\alpha) = p(\beta).$$

Proof. Let $(==) \in (\mathcal{B} \times \mathcal{B} \to \mathcal{B})$ denote the unique total function such that $(x == y) = \mathtt{tt}$ iff $x \sim y$. Then $\forall_{\prod_i K_i}(\lambda\alpha.p(\alpha) == p(\alpha)) = \mathtt{tt}$. If we define $p_{|n}(\alpha) = p(\alpha_{|n})$, then $p = \bigsqcup_n p_{|n}$ and hence $(\lambda\alpha.p(\alpha) == p(\alpha)) = \bigsqcup_n(\lambda\alpha.p_{|n}(\alpha) == p(\alpha))$. So, by continuity of $\forall_{\prod_i K_i}$, there is n such that $\forall_{\prod_i K_i}(\lambda\alpha.p_{|n}(\alpha) == p(\alpha)) = \mathtt{tt}$. (We cannot conclude that $p_{|n}(\alpha) == p(\alpha)$ for all $\alpha \in \prod_i K_i$ because there is no reason why $\lambda\alpha.p_{|n}(\alpha) == p(\alpha)$ should be defined on $\prod_i K_i$.) Choose $\gamma \in \prod_i K_{i+n}$, and define $q_n(\alpha) = q(\alpha_0\alpha_1 \ldots \alpha_{n-1}\gamma)$. Then $p_{|n}(\alpha) \sqsubseteq q_n(\alpha)$, and $\forall_{\prod_i K_i}(\lambda\alpha.q_n(\alpha) == p(\alpha)) = \mathtt{tt}$ by monotonicity. Now $\lambda\alpha.q_n(\alpha) == p(\alpha)$ is defined on $\prod_i K_i$ and hence $q_n(\alpha) = p(\alpha)$ for all $\alpha \in \prod_i K_i$. But if $\alpha =_n \beta$ then $q_n(\alpha) = q_n(\beta)$, and so $p(\alpha) = p(\beta)$ if $\beta \in \prod_i K_i$. □

The proof of Theorem 4.5 doesn't give information about the number of steps needed to compute the existential quantification $p(\Pi(\varepsilon)(p))$. Denote by $\mathrm{fan}(p)$ the smallest n satisfying the condition of Corollary 4.6. For example, if $p(x) = (f(x) = g(x))$ for f and g as defined in the introduction, then $\mathrm{fan}(p) = 6^{80}$. A related notion is used in Section 7 to formulate a conjecture regarding the time complexity of such existential quantifications.

The above functionals can be coded in the language Haskell as follows, where `r` plays the role of the subscripted p. Notice that, oddly, the notation `(d,d')` indicates the product, rather than pairing, of the types `d` and `d'`, but `(x,y)` indicates the pairing of the elements `x` and `y`. Similarly, $\lambda x. \ldots$ is written `\x->...`. Quotes indicate that a binary curried function is used as an infix operator. Apart from these idiosyncrasies, the notation is the same as that of PCF, with usual metalinguistic modes of expression incorporated in the language, such as `where` clauses.

```
image :: (d -> e) -> Searcher d -> Searcher e
image f k = \q -> f(k(\x -> q(f x)))

times :: Searcher d -> Searcher d' -> Searcher(d,d')
(k `times` k') p = (x,x')
     where x = k (\x -> forsome k'(\x' -> p(x,x')))
           x'= k'(\x'-> p(x,x'))

prod :: (N -> Searcher d) -> Searcher(N -> d)
prod e p n=e n(\x->r n x(prod(\i->e(i+n+1))(r n x)))
  where r n x a = p(\i -> if i  < n then prod e p i
                          else if i == n then x
                                          else a(i-n-1))
```

We now apply this to fill the gap in the introduction, by defining the search operator for the Cantor set (total elements of the type of binary sequences) as the product of countably many copies of the search operator for bits. We name search operators after the sets they search over.

```
bit :: Searcher Bit
bit  = \q -> if q 1 then 1 else 0
cantor :: Searcher Cantor
cantor = prod(\i -> bit)
foreveryC = forevery cantor
```

The above product functional is not as fast as required for the experiment reported in the introduction. But it seems to have good information-theoretic complexity and a small modification will be enough to make it fast in our running example and others (Section 7).

The Fan functional can be defined as follows, where `eq n` realizes the decidable relation $(=_n)$ on Cantor, `mu` is the minimization operator, and `==>` is boolean implication:

```
  fan :: (Cantor -> Bool) -> Bool
  fan p = mu(\n->foreveryC(\a->foreveryC(\b->
                 eq n a b ==> (p a == p b))))
```

Thus, once an algorithm for quantification over the Cantor set has been obtained, this algorithmic definition is the same as the mathematical definition of the Fan functional.

As another example, for any computable $f : \mathbb{N} \to \mathbb{N}$, the set of sequences α with $\alpha_i \leq f(i)$ is searchable: a search operator is $\Pi(\lambda i.z(f(i) + 1))$, where z is a functional such that $z(n)$ implements search over the set $\{0, 1, \ldots, n-1\}$.

5. Topological aspects of exhaustible sets

The results of this section, which are interesting in their own right, are applied in Section 6 in order to derive algorithms that lead to a characterization of searchable sets.

In [6], a notion analogous to exhaustibility, with the Sierpinski domain $\mathcal{S} = \{\bot, \top\}$ playing the role of the boolean domain $\mathcal{B}$, is considered. A crucial lemma in [6] is that the (now unique) exhaustion functional $\forall_K : (D \to \mathcal{S}) \to \mathcal{S}$ is continuous iff the set K is compact in the Scott topology of D. Hence, because computable functionals are continuous, Sierpinski-exhaustible sets are compact, and so Sierpinski exhaustibility is seen as articulating an algorithmic version of the topological notion of compactness. The computational idea is that, given any *semi-decidable* property of D, one can *semi-decide* whether it holds for all elements of K. Closure properties analogous to the above are established for Sierpinski exhaustibility in [6] (and redeveloped from a purely operational point of view in [8]).

The present investigation can be seen as a natural follow-up of that work that arises by asking what changes if one moves from semi-decision problems to decision problems.

One significant change is that continuity of a boolean exhaustion operator $\forall_K : (D \to \mathcal{B}) \to \mathcal{B}$ doesn't entail the compactness of K any longer (Examples 5.2 below). However, a similar, but not quite the same, conclusion holds for sets of *total* elements (with variation also justified in Examples 5.2). We invoke the material on Kleene–Kreisel functionals of Section 2 to formulate and prove this. As in the previous sections, $D = D_\sigma$ for some unspecified σ, and, additionally $T = T_\sigma, C = C_\sigma$ and $\rho = \rho_\sigma : T_\sigma \to C_\sigma$. By the *shadow* of a set $K \subseteq T$ we mean its ρ-image in C.

Lemma 5.1. *The shadow of any exhaustible set of total elements is compact in the Kleene–Kreisel topology.*

Proof. Let $K \subseteq T$ be exhaustible. By Lemma 2.4 and the fact that clopen sets are closed under finite unions, to establish compactness of $\rho(K)$, it is enough to consider a directed clopen cover $\mathcal{U}$. By Lemma 2.3, for every $U \in \mathcal{U}$ there is a total $p_U \in (D \to \mathcal{B})$ with

$$(\dagger) \quad \rho^{-1}(U) \subseteq p_U^{-1}(\mathrm{tt}) \text{ and } \rho^{-1}(C \setminus U) \subseteq p_U^{-1}(\mathrm{ff}).$$

Define predicates $q_U, r \in (D \to \mathcal{B})$ by $q_U^{-1}(\mathrm{tt}) = p_U^{-1}(\mathrm{tt})$, $r^{-1}(\mathrm{tt}) = \bigcup_{U \in \mathcal{U}} p_U^{-1}(\mathrm{tt})$ and $q_U^{-1}(\mathrm{ff}) = r^{-1}(\mathrm{ff}) = \emptyset$. Then $q_U \sqsubseteq p_U$, the set $\{q_U \mid U \in \mathcal{U}\}$ is directed, and $r = \bigsqcup_{U \in \mathcal{U}} q_U$. Because $\rho(K) \subseteq \bigcup \mathcal{U}$, we have that $K \subseteq r^{-1}(\mathrm{tt})$ and hence $\forall_K(r) = \mathrm{tt}$. So, by continuity of $\forall_K$, there is $U \in \mathcal{U}$ with $\forall_K(q_U) = \mathrm{tt}$, and hence with $\forall_K(p_U) = \mathrm{tt}$ by monotonicity. Let $x \in K$. Then $p_U(x) = \mathrm{tt}$ by specification of $\forall_K$ and the fact that p_U is total and hence defined on K. But then $\rho(x) \in U$, for otherwise ($\dagger$) would entail $p_U(x) = \mathrm{ff}$. This shows that $\rho(K) \subseteq U$, and so $\rho(K)$ is compact. ☐

This gives a topological view of the computational fact stated in the introduction that exhaustible sets of natural numbers must be finite: all compact sets are finite in a discrete space. We now justify the strengthened and weakened forms of the hypothesis and conclusion of Lemma 5.1 with respect to the cited lemma on compactness of Sierpinski-exhaustible sets [6].

Examples 5.2. (1) *There are non-compact, exhaustible sets.* (So one needs to assume something such as totality.) By [22, 20], any second-countable space, e.g. the real line $\mathbb{R}$, can be embedded into the domain $D = \mathcal{B}^\omega$. But $\mathbb{R}$ is a connected space, which is equivalent to saying that every continuous boolean-valued map defined on it is constant. Hence $p \in (D \to \mathcal{B})$ is defined on $\mathbb{R}$ iff it is constant on $\mathbb{R}$. Therefore $\mathbb{R}$ is trivially exhaustible: $\forall_{\mathbb{R}}(p) = p(0)$. But it is not compact.

(2) *Although the shadow of an exhaustible set of total elements is compact, the set itself doesn't need to be compact.* In fact, there is a trivial and pervasive counter-example. Let $D = ((\mathcal{N} \to \mathcal{N}) \to \mathcal{N})$ and $f \in D$ be total. Then the total equivalence class K of f, as is well known and easy to verify, doesn't have a minimal element, and hence cannot be compact. But K is exhaustible with $\forall_K(p) = p(f)$.

(However, there is a natural sense in which *all* exhaustible sets *are* compact. Part of the argument of Lemma 5.1 shows that any exhaustible set K is compact in the *weak topology*, namely the coarsest topology on D such that all predicates $p \in (D \in \mathcal{B})$ defined on K are continuous. This is generated by directed unions of basic open sets of the form $p^{-1}(\mathrm{tt})$ with p as above, because such sets are closed under finite unions and intersections.)

For a partial converse of Lemma 5.1, say that $K \subseteq D$ is *continuously exhaustible* if there is a continuous, not necessarily computable, map $\forall_K \in ((D \to \mathcal{B}) \to \mathcal{B})$ satisfying the conditions of Definition 3.2.

Proposition 5.3. *Any non-empty entire set with compact shadow is an entire continuous image of the Cantor set and hence is continuously exhaustible.*

Proof. By e.g. [9], any compact subset of C is countably based (even though C is not). But any non-empty compact Hausdorff countably based space is a continuous image of the Cantor space. Hence there is a continuous map $\mathbb{B}^{\mathbb{N}} \to C$ with image $\rho(K)$ for any entire set $K \subseteq D$. Then any realizer $\mathcal{B}^\omega \to D$ has entire image K. ☐

6. Characterization of searchability

By the *Cantor set* we mean the set of total elements of $\mathcal{B}^\omega$.

Theorem 6.1. *The following are equivalent for any non-empty entire set $K \subseteq D$:*

1. *K is exhaustible.*
2. *K is searchable.*
3. *K is a computable entire image of the Cantor set.*

Proof. The bottom-up implications are Lemma 3.4 and Proposition 4.2 applied to the Cantor set. The other directions are proved below. ☐

We prove a uniform version of the implication (1) ⇒ (2):

Lemma 6.2. *There is a computable functional*

$$\Phi : ((D \to \mathcal{B}) \to \mathcal{B}) \to ((D \to \mathcal{B}) \to D)$$

such that $\Phi(\exists_K)$ is a search realizer for K for any exhaustible non-empty entire set K with realizer $\exists_K$.

We reduce this to the following:

Lemma 6.3. *There is a computable functional*

$$\Gamma : ((D \to \mathcal{B}) \to \mathcal{B}) \to D$$

such that $\Gamma(\exists_K) \in K$ for any realizer $\exists_K$ of the exhaustibility of a non-empty entire set K.

The reduction works as follows. By Proposition 4.1, for any $p \in (D \to \mathcal{B})$ decidable on K, the set $K_p = K \cap p^{-1}(\mathrm{tt})$ is exhaustible. Hence the algorithm

$$\varepsilon_K(p) = \text{if } \exists_K(p) \text{ then } \Gamma(\exists_{K_p}) \text{ else } \Gamma(\exists_K)$$
realizes the searchability of K. That is, if K_p is non-empty, find and element in K_p, otherwise find an element in K. We can take $\exists_{K_p}(q) = \exists x \in K.p(x) \wedge q(x) = \exists_K(p \wedge q)$ by the proof of Proposition 4.1, and hence we can define
$$\Phi(\phi)(p) = \text{if } \phi(p) \text{ then } \Gamma(\lambda q.\phi(p \wedge q)) \text{ else } \Gamma(\phi),$$
which completes the reduction of Lemma 6.2 to Lemma 6.3.

By Lemma 2.1, it is enough to establish Lemma 6.3 for $D = (E \to \mathcal{N})$ with $E = D_\tau$ arbitrary, because the types D that satisfy the condition of Lemma 6.3 are closed under retracts, as a short routine argument shows.

Let $K \subseteq D$ be a non-empty, entire exhaustible set. Our task is to find, uniformly in the realizer $\exists_K$ of the exhaustibility of K, some $g \in K$. Recall that we are also allowed to use $\forall_K$.

By the Kleene–Kreisel density theorem (see e.g. [5]), there is an effective enumeration e_n of a set of total elements of $E = D_\tau$ such that $\boldsymbol{e}_n = \rho(e_n)$ is dense in C_τ. To construct g, first define a total function $\gamma\colon \mathbb{N} \to \mathbb{N}$ by
$$\gamma(i) = \mu n.\exists f \in K.f(e_i) = n \wedge \forall j < i.f(e_j) = \gamma(j).$$
Our function g will be such that $g(e_i) = \gamma(i)$. The set
$$K_i = K \cap \{f \in D \mid \forall j < i.f(e_j) = \gamma(j)\}$$
is clearly non-empty for every i and $K_i \supseteq K_{i+1}$. Because this is the intersection of K with a decidable set, it is exhaustible by Proposition 4.1, and hence its shadow $\boldsymbol{K}_i$ is compact by Lemma 5.1. Because $C_{\tau\to\iota}$ is Hausdorff,
$$\boldsymbol{K}_\infty = \bigcap_i \boldsymbol{K}_i$$
is non-empty. But $\boldsymbol{f} \in \boldsymbol{K}_\infty$ iff $\boldsymbol{f}(\boldsymbol{e}_i) = \gamma(i)$ for all i, and so if $\boldsymbol{f}, \boldsymbol{f}' \in \boldsymbol{K}_\infty$ then $\boldsymbol{f}(\boldsymbol{e}_i) = \boldsymbol{f}'(\boldsymbol{e}_i)$ for all i and hence $\boldsymbol{f} = \boldsymbol{f}'$ by density. Thus, $\boldsymbol{K}_\infty$ is a singleton, say $\{\boldsymbol{g}\}$. This use of γ was proposed by Matthias Schröder, but the algorithm for computing $\boldsymbol{g}$, developed below, is ours.

Lemma 6.4. *For every total $x \in E$ there is n such that $f(x) = f'(x)$ for all $f, f' \in K_n$.*

Proof. Let $B_{\boldsymbol{x}} = \{\boldsymbol{f} \in C_{\tau\to\iota} \mid \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x})\}$ where $\boldsymbol{x} = \rho(x)$. Then $\bigcap_i \boldsymbol{K}_i \subseteq B_{\boldsymbol{x}}$ as this amounts to $\boldsymbol{g} \in B_{\boldsymbol{x}}$, and, because each $\boldsymbol{K}_i$ is compact and $C_{\tau\to\iota}$ is Hausdorff, there is n such that already $\boldsymbol{K}_n \subseteq B_{\boldsymbol{x}}$ as $B_{\boldsymbol{x}}$ is clearly open. So for all $\boldsymbol{f} \in \boldsymbol{K}_n$ one has $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{x})$, and hence $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{f}'(\boldsymbol{x})$ for all $\boldsymbol{f}, \boldsymbol{f}' \in \boldsymbol{K}_n$. □

Now K_i is exhaustible uniformly in $\exists_K$ and i with
$$\exists_{K_i}(p) = \exists h \in K.p(h) \wedge \forall j < i.h(e_j) = \gamma(j).$$
Hence, to define $g(x)$, we can first compute
$$\mu n.\forall f, f' \in K_n.f(x) = f'(x),$$
and then let
$$g(x) = \mu m.\exists f \in K_n.f(x) = m.$$
Then g realizes $\boldsymbol{g}$ by construction, and hence $g \in K$ because K is entire. The dependency of g from $\exists_K$ is clearly uniform, and defines the required functional Γ. This concludes the proof of Lemma 6.3 and hence that of Lemma 6.2.

We also have a uniform version of $(1) \Rightarrow (3)$:

Lemma 6.5. *There is a computable functional*
$$\Psi : ((D \to \mathcal{B}) \to \mathcal{B}) \to (\mathcal{B}^\omega \to D)$$
such that K is the entire image of $\Psi(\exists_K)$ for any exhaustible non-empty entire set K with realizer $\exists_K$.

The proof is similar to that of Lemma 6.2 with a modification of Lemma 6.3. By e.g. [9], any compact subset of a Kleene–Kreisel space is countably based. We adapt the standard topological proof of the fact that any non-empty countably based compact Hausdorff space is a continuous image of the Cantor space. Rather than constructing a single function g as above, we construct a binarily branching infinite tree of approximations to functions like g. A point α of the Cantor space gives a path in this tree, and we use a modification of the above algorithm to follow that path and evaluate $g_\alpha(x)$ in such a way that $g_\alpha \in K$ and any $g \in K$ is equivalent to some g_α.

7. Experiments and tentative explanations

Experiments. We shall argue below that if we used the Haskell implementation of the product functional given in Section 4 to derive an implementation of the functional `foreveryC`, we would need at least $2^{2^{12}}$ recursion unfoldings for the experiments reported in the introduction, which cannot possibly be performed in the claimed $3s$. Recall that

```
f,g :: Cantor -> N
f a = a(10*a(3^80)+100*a(4^80)+1000*a(5^80))
g a = a(10*a(3^80)+100*a(4^80)+1000*a(6^80))
```

The number 12 in the above calculation is informally obtained as follows. The computation of `f a == g a`, for any given `a`, "uses" 12 arguments of `a`, namely, 3^{80}, 4^{80}, 5^{80}, 6^{80}, plus the 8 arguments that arise by summing subsets of $\{10, 100, 1000\}$. The comparison algorithm derived from the product functional is so good that it generates queries `a` for the black box `\a -> f a == g a` so that only these 12 arguments will get evaluated, but, at the same time, so bad that, disappointingly, it generates $2^{2^{12}}$ queries, with an exponential amount of repetition, rather than only 2^{12}, as should be enough.

In order to reduce the time from $2^{2^{12}}$ steps to $3s$ in the given experiment, we re-implement the product functional as follows:

```
prod' :: (N -> Searcher d) -> Sercher(N -> d)
prod' e p = b
 where b = id'(\n->e n(\x->r x n(prod'(\i->e(i+n+1))
                                        (r x n))))
       r x n a = p(\i -> if i  < n then b i
                         else if i == n then x
                                        else a(i-n-1))
data T d = B d (T d) (T d)

code :: (N -> d) -> T d
code f = B (f 0) (code(\n -> f(2*n+1)))
                 (code(\n -> f(2*n+2)))
```

```
decode :: T d -> (N -> d)
decode (B x l r) 0 = x
decode (B x l r) n = if odd n
                     then decode l ((n-1) `div` 2)
                     else decode r ((n-2) `div` 2)

id' :: (N -> d) -> (N -> d)
id' = decode.code
```

Here `id'` is the identity function (on non-strict arguments). Hence this definition of `prod` is semantically equivalent to the previous. But it is expected to be exponentially faster. If the type d is interpreted as the domain D, the type `T d` is that of infinite, binarily branching trees with nodes labelled by elements of the domain D. The idea is to store a function `N -> d` in a tree in a breadth-first manner, and then retrieve it back. This certainly imposes some overhead in the computation, but it is logarithmic. The point is that Haskell is call-by-need. Once a node of such a tree is evaluated, the result is stored, and if it is queried again, it will be immediately available. Therefore, the entries of the sequence returned by `prod e p` are never recalculated, avoiding the doubly exponential behaviour of the original definition.

Now consider the following implementation of the product functional:

```
(x # a)(i) = if i == 0 then x else a(i-1)
tl a = \i -> a(i+1)
prod'' e p =
  let x = e 0(\x->p(x#(prod''(tl e)(\a->p(x#a)))))
  in x#(prod''(tl e)(\a->p(x#a)))
```

It is easy to see by induction on indices that this agrees with the previous. This doesn't need an explicit memoization as the previous, as duplicate evaluation is avoided by the `let` clause. However, because the result is generated from left to right, this introduces a linear-time overhead. Notice that this implementation can be easily modified to work with lazy lists.

Consider also an implementation of Berger's search operator [4]:

```
berger :: (Cantor -> Bool) -> Cantor
berger p = if p(0 # berger(\a -> p(0 # a)))
           then 0 # berger(\a -> p(0 # a))
           else 1 # berger(\a -> p(1 # a))
```

This also generates the result from left to right, but it has the disadvantage that evaluation of any entry forces evaluation of the previous entries, which is not the case for `prod''`. We mention in passing that Simpson [24] applied Berger's search operator to show that there is a sequential algorithm for Riemann integration when real numbers are implemented as infinite sequences of digits.

Now, only the implementation `prod'` can cope with the experiment reported in the introduction — see Conjecture 7.1 below.

Here is another experiment. Given a finite set S, define a predicate $p_S \colon \mathcal{B}^\omega \to \mathcal{B}$ by $p_S(\alpha) = \bigwedge_{i \in S} \alpha_i$. Then two finite sets S and T are equal iff $p_S = p_T$. Using the same idea to check whether two finite lists have the same set of elements, the query

```
> eqset
[2^700,789^34,6^600,345^55,45,5,1000,4,10^100,5000,23^45]
[5,789^34,45,1000,23^45,5000,345^55,4,10^100,6^600,2^700]
```

answers `True` in $0.89s$ using the Glasgow Haskell interpreter this time. A file with the programs discussed in this paper is available at [7].

Tentative explanations. For simplicity, we confine our attention to predicates p defined on the Cantor set, as in the above examples. There are two notions of modulus of uniform continuity on the Cantor set that arise often. The first, as in Corollary 4.6, says that there is a smallest number $n = \operatorname{fan}(p)$ such that for all total $\alpha, \beta \in \mathcal{B}^\omega$, if $\alpha =_n \beta$ then $p(\alpha) = p(\beta)$. The second says that there is a smallest number $m = m(p)$ such that $p(\alpha) = p(\alpha_{|m})$ for all total $\alpha \in \mathcal{B}^\omega$. Then $\operatorname{fan}(p) \leq m(p)$ holds.

We consider refined versions of these two notions. Given a set $I \subseteq \omega$, define $\alpha =_I \beta$ iff $\alpha_i = \beta_i$ for all $i \in I$, and define $\alpha_{|I}(i) = \alpha_i$ if $i \in I$ and $\alpha_{|I}(i) = \bot$ otherwise. Then there is a smallest set $I = \operatorname{FAN}(p)$ such that $\alpha =_I \beta$ implies $p(\alpha) = p(\beta)$ for all total $\alpha, \beta \in \mathcal{B}^\omega$, and there is a smallest set $I = M(p)$ such that $p(\alpha) = p(\alpha_{|I})$ for all total $\alpha \in \mathcal{B}^\omega$. These sets exist and are finite by uniform continuity, and $\operatorname{FAN}(p) \subseteq M(p) \subseteq \{i \mid i < m(p)\}$. The set $\operatorname{FAN}(p)$ is uniformly decidable in p, but the set $M(p)$ is not. However, it is the set $M(p)$ that arises in our considerations. We may formalize the above partial explanations by saying that p *uses* its argument at i iff $i \in M(p)$.

Conjecture 7.1. *Let $p \in (\mathcal{B}^\omega \to \mathcal{B})$ be total and assume unit cost to evaluate $p(\alpha)$ at any total argument α. We conjecture that, under call-by-need, the evaluation of $\exists(p)$ takes time proportional to:*

$2^{2^{\text{cardinality of } M(p)}}$	*using* `prod`
$2^{\sum_{n \in M(p)} 1+\log(n+1)}$	*using* `prod'`
$2^{\sum M(p)}$	*using* `prod''`
$2^{m(p)}$	*using* `berger`.

This is confirmed for relatively small examples, where increasing $M(p)$ by a few elements increases the time as predicted by the above formulas. Thus, on information-theoretic grounds, it seems that `prod'` is asymptotically optimal. Notice that $\exists(p) = p(\varepsilon(p))$ and hence the computation of $\varepsilon(p)$ is, to some extent, driven by p itself. The extent to which this is so depends on the particular algorithm ε. For Berger's algorithm, the computation of $\varepsilon(p)(n)$ forces the evaluation of $\varepsilon(p)(i)$ for all $i < n$. On the other hand, using `prod`, only $\varepsilon(p)(i)$ with $i \in M(p)$ are ever evaluated, but unfortunately more often than necessary. The algorithms `prod'` and `prod''` fix this in different ways. Is there an elegant way of achieving the same performance as `prod'` but without using the infinite-tree trick, in the style of `prod''`?

8. Concluding remarks

The algorithms developed in this work have purely computational specifications, which allow them to be used, for example by typical functional programmers, without knowledge of specialized mathematical techniques in the theory of computation. However, the correctness proofs of the deeper algorithms crucially rely on topological techniques. In this sense, this work is a genuine application of topology to computation: theorems formulated in the language of computation, proofs developed in the language of topology.

But there is another sense in which topology proves to play a crucial role. Compact sets in topology are advertised as sets that behave, in many important respects, as if they were finite. Then exhaustively searchable sets *ought* to be compact. And compact sets are known to be closed under continuous images and under finite and infinite products. Moreover, for countably based Hausdorff spaces, they are the continuous images of the Cantor space. Hence searchable sets *ought* to have corresponding closure properties and characterization, which is what this work establishes, *motivated* by these considerations. Thus, in a more abstract level, topology is applied as a paradigm for discovering unforeseen notions, algorithms and theorems in computability theory.

In this enterprise, the technically challenging aspects of this investigation include Lemma 5.1, not only in finding a proof, but fundamentally in discovering a formulation that matches computational reality (see Counter-examples 5.2).

Notice that the correctness proofs of Section 4 can be directly interpreted in the operational setting [6, 8]. But a development of operational counter-parts for those of Section 6 is left as an open problem. This requires an operational reworking of Section 5, which seems challenging.

Finally, in this work, topology also plays a role in higher-type complexity: we have applied the notion of uniform continuity to measure the size of functional inputs in the formulation of run times for higher-type algorithms.

Acknowledgments. I have benefited from discussions with Andrej Bauer, Ulrich Berger, Dan Ghica, Achim Jung, John Longley, Matthias Schröder, and Alex Simpson. I also thank Dag Normann for having answered questions regarding the history and technical ramifications of the subject, and for sending me a copy of Tait's unpublished manuscript — but the reader should consult his paper [18] for a more accurate and detailed account.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3 of *Oxford science publications*, pages 1–168. 1994.

[2] A. Bauer, M.H. Escardó, and A.K. Simpson. Comparing functional paradigms for exact real-number computation. volume 2380 of *Lect. Not. Comp. Sci.*, pages 488–500, 2002.

[3] M.J. Beeson. *Foundations of Constructive Mathematics*. Springer, 1985.

[4] U. Berger. *Totale Objekte und Mengen in der Bereichstheorie*. PhD thesis, Mathematisches Institut der Universität München, 1990.

[5] U. Berger. Total sets and objects in domain theory. *Ann. Pure Appl. Logic*, 60(2):91–117, 1993.

[6] M.H. Escardó. Synthetic topology of data types and classical spaces. *Electron. Notes Theor. Comput. Sci.*, 87:21–156, 2004.

[7] M.H. Escardó. Haskell program for exhaustive search over infinite sets. `http://www.cs.bham.ac.uk/~mhe/papers/exhaustive.hs`, School of Computer Science, University of Birmingham, Summer 2006.

[8] M.H. Escardó and W.K. Ho. Operational domain theory and topology of a sequential programming language. In *Proceedings of the 20th Annual IEEE Symposium on Logic In Computer Science*, pages 427–436, 2005.

[9] M.H. Escardó, J. Lawson, and A. Simpson. Comparing Cartesian closed categories of (core) compactly generated spaces. *Topology Appl.*, 143(1-3):105–145, 2004.

[10] G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M. Mislove, and D.S. Scott. *Continuous Lattices and Domains*. CUP, 2003.

[11] G. Hutton. *Programming in Haskell*. CUP, 2007.

[12] J.R. Longley. On the ubiquity of certain type structures. *Mathematical Structures in Computer Science*. To appear.

[13] J.R. Longley. Notions of computability at higher types. I. In *Logic Colloquium 2000*, volume 19 of *Lect. Notes Log.*, pages 32–142. Assoc. Symbol. Logic, Urbana, IL, 2005.

[14] S. Mac Lane. *Categories for the Working Mathematician*. Springer, 1971.

[15] D. Normann. *Recursion on the countable functionals*, volume 811 of *Lec. Not. Math.* Springer, 1980.

[16] D. Normann. Computability over the partial continuous functionals. *J. Symbolic Logic*, 65(3):1133–1142, 2000.

[17] D. Normann. Comparing hierarchies of total functionals. *Logical Methods in Computer Science*, 1(2):1–28, 2005.

[18] D. Normann. Computing with functionals—computability theory or computer science? *Bull. Symbolic Logic*, 12(1):43–59, 2006.

[19] G.D. Plotkin. LCF considered as a programming language. *Theoret. Comput. Sci.*, 5(1):223–255, 1977.

[20] G.D. Plotkin. $\mathbb{T}^\omega$ as a universal domain. *J. Comput. System Sci.*, 17:209–236, 1978.

[21] G.D. Plotkin. Full abstraction, totality and PCF. *Math. Structures Comput. Sci.*, 9(1):1–20, 1999.

[22] D.S. Scott. Data types as lattices. *SIAM J. Comput.*, 5:522–587, 1976.

[23] D.S. Scott. A type-theoretical alternative to CUCH, ISWIM and OWHY. *Theoret. Comput. Sci.*, 121:411–440, 1993. Reprint of a 1969 manuscript.

[24] A. Simpson. Lazy functional algorithms for exact real functionals. *Lec. Not. Comput. Sci.*, 1450:323–342, 1998.

[25] M.B. Smyth. Topology. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 1 of *Oxford science publications*, pages 641–761. 1992.