# Infinite spaces that admit (fast) exhaustive search

**An application of topology to computation**

Martín Escardó

School of Computer Science, University of Birmingham, UK

LFCS Seminar, 5th September 2006

## Verbal reply from a friend

*“I work with* finite *sets that admit* slow *exhaustive search.”*

Reinhold Heckmann, August 2006, Dagstuhl seminar.

## Aims

Main Entry: exhaustive
Function: adjective
: testing all possibilities or considering all elements :

Merriam-Webster dictionary Online

1. To mechanically exhaust infinitely many possibilities in finite time.
2. Better, to know when this can be done. Fairly tight results.
3. And efficiently if possible. Tentative, but rather encouraging results.

## I’ll talk about certain algorithms . . .

. . . that require topology to come up with them, and understand them,

. . . but don’t require topology to profit from them.

I’ll assume that

> less than half of the audience knows the required topology,
>
> but the other more-than-half doesn’t,

and I’ll address both.

## The natural numbers don't admit exhaustive search

*Given a decidable property $p$ of natural numbers,*
*it is not possible to decide whether or not $p(n)$ holds*
*for every single natural number $n$.*

Well known corollary of the unsolvability of the Halting problem.

## But some *infinite* sets do admit exhaustive search

For example the Cantor space (Berger 1990).

Plan:

0. Berger's example.

1. Derivation of this as a corollary of a more general result.

2. Show that non-trivial instances of search can be surprisingly fast,

3. but that not all of them can be fast.

## Berger’s example in Haskell

```
type N = Integer
type Cantor = N -> Bool   -- ranged over by a

(#) :: Bool -> Cantor -> Cantor
x # a = \i -> if i == 0 then x else a(i-1)

epsilon :: (Cantor -> Bool) -> Cantor
epsilon p = if p(True #(epsilon(\a -> p(True #a))))
            then True #(epsilon(\a -> p(True #a)))
            else False#(epsilon(\a -> p(False#a)))

exists, forall :: (Cantor -> Bool) -> Bool
exists p = p(epsilon p)
forall p = not(exists(\a -> not(p a)))
```

## Sample application of Berger’s functional

Equality of certain functions is decidable:

```
equal :: (Cantor -> N) -> (Cantor -> N) -> Bool
equal f g = forall(\a -> f a == g a)
```

This contradicts the beliefs of functional programmers.

And of theoretical computer scientists too.

## Experiments

(1.73GHz Pentium M, using ghci under ubuntu/debian.)

```
> equal (\a -> a 11) (\a -> a 11)
True (0.08 secs, 3676992 bytes)

> equal (\a -> a 11) (\a -> a 12)
False (0.16 secs, 8008304 bytes)

> equal (\a -> a 11) (\a -> a 16)
False (3.01 secs, 141776964 bytes)

> equal (\a -> a 11) (\a -> a 20)
False (54.66 secs, 2547638808 bytes)

> equal (\a -> a 11) (\a -> a 30)
```

## Experiments

```
> exists (\a -> a 15)
True (0.86 secs, 45706760 bytes)

> exists (\a -> a 16)
True (1.81 secs, 96117200 bytes)

> exists (\a -> a 17)
True (3.82 secs, 202569248 bytes)

> exists (\a -> a 18)
True (8.06 secs, 426410336 bytes)

> exists (\a -> a 19)
True (16.95 secs, 895589112 bytes)
```

## Why is it slow?

One answer. It is in practice.

Exponential time in the modulus of uniform continuity.

Another answer. Has to be in theory.

Can polynomially encode an intractable problem.

## Encoding satisfiability

```
data Prop = P N
          | Not Prop
          | And Prop Prop
          | Or  Prop Prop

valid :: Prop -> Cantor -> Bool
valid (P i)     a = a i
valid (Not p)   a = not(valid p a)
valid (And p q) a = (valid p a) && (valid q a)
valid (Or  p q) a = (valid p a) || (valid q a)

satisfiable, tautology :: Prop -> Bool
satisfiable p = exists(\a -> valid p a)
tautology   p = forall(\a -> valid p a)
```

## But this is not the final answer

1. E.g. type-checking in Haskell/ML is intractable in theory but fast in practice.

2. We have found a modification of Berger’s algorithm that doesn’t depend on the modulus of continuity and is fast in several surprising instances.

## Plan

0. Berger's example. ✓

1. Derivation of this as a corollary of a more general result.

2. Show that non-trivial instances of search can be surprisingly fast,

3. but that not all of them can be fast. ✓

We proceed to (1) now.

## Machinery

Higher-type computation.

Gödel's system $T$ and some strongly normalizing extensions.

PCF (typed lambda-calculus with booleans, integers and fixed-point recursion).

Haskell (practical version of this).

Topology and domain theory

To discover higher-type programs and prove their correctness.

Semantics.

Chiefly computational adequacy of Scott's model of PCF.

But also some topological semantics of (extended) system $T$.

## Higher-type computation

I’ll talk about total functionals.

Topological semantics of system $T$ and extensions.

Partial functionals will arise in constructions and proofs.

Domain-theoretic semantics of PCF.

I’ll apply the known relationships between the two semantics.

## Some key (and trivial) topological spaces and domains

$2 = \{0, 1\} = \{\mathrm{False}, \mathrm{True}\}$, space of discrete booleans.

Classifies clopen (or decidable) subsets.

$\mathcal{B} = 2_\perp$, domain of lifted booleans.

Classifies pairs of disjoint open subsets.

$\mathbb{S} = \{\perp, \top\}$, Sierpisnki space/domain (aka unit type (sic)).

Classifies open (or semi-decidable) subsets.

## Exhaustively searchable sets

I'll introduce three notions for a subset $K$ of a type $X$:

1. $\mathbb{S}$-exhaustible.

2. $2$-exhaustible.

3. Exhaustively searchable, or just searchable for short.

Topologically, they turn out to be equivalent.

Computationally, their relationship is subtler and not fully known.

Here it is very important that we are working with total objects.

## Exhaustively searchable sets

1. $\mathbb{S}$-exhaustible.

   Can semi-decide that all possibilities hold.

2. $\mathbb{2}$-exhaustible.

   Can decide that all possibilities hold.

3. Exhaustively searchable.

   Can come up with an example if there is one.

## $\mathbb{S}$-exhaustible subset of a type

A set $K \subseteq X$ is $\mathbb{S}$-exhaustible if the functional $\forall_K : (X \to \mathbb{S}) \to \mathbb{S}$ defined by

$$\forall_K(p) = \top \iff p(x) = \top \text{ for every } x \in K$$

is continuous/computable.

One can semi-decide, for any given semi-decidable predicate $p$, whether it holds for *all* elements of $K$.

## $2$-exhaustible subset of a type

Replace $\mathbb{S}$ by $2$ in the previous definition.

A set $K \subseteq X$ is $2$-exhaustible if the functional $\forall_K \colon (X \to 2) \to 2$ defined by

$$\forall_K(p) = \mathrm{True} \iff p(x) = \mathrm{True} \text{ for every } x \in K$$

is continuous/computable.

One can decide, for any given decidable predicate $p$, whether or not it holds for *all* elements of $K$.

## Exhaustively searchable subset of a type

A set $K \subseteq X$ is searchable if there is a continuous/computable $\varepsilon_S \colon (X \to 2) \to X$ such that for every $p \in (X \to 2)$

1. $\varepsilon_K(p)$ always belongs to $K$,

2. if there is $x \in K$ such that $p(x) = \mathrm{True}$, then $\varepsilon_K(p)$ provides a particular example of such an $x$, that is, $p(\varepsilon_K(p)) = \mathrm{True}$.

Notice that $\varepsilon_K$ is not uniquely determined by $K$.

## Summary of the notions

$K \subseteq X$ is

1. $\mathbb{S}$-exhaustible: $\forall_K : (X \to \mathbb{S}) \to \mathbb{S}$.
2. $2$-exhaustible: $\forall_K : (X \to 2) \to 2$.
3. Searchable: $\varepsilon_K : (X \to 2) \to X$, with image contained in $K$.

Topologically searchable: Continuous $\varepsilon_K$.

Effectively searchable: Computable $\varepsilon_K$.

## Topological lemma

Apart from the empty set, which is exhaustible but not searchable:

1. *The three notions coincide for the total higher types.*

2. *And they are equivalent to the notion of topological compactness.*

Proof. Requires fairly sophisticated topological machinery.

Forthcoming application. Termination of certain algorithms.

## Added after the talk

It seems that a number of people in the audience thought that the three notions coincide for “all” spaces which admit computability.

This is not so. I am talking here about the higher types over the natural numbers.

For other spaces (e.g. the real numbers), the three notions don’t coincide.

But they do coincide for a large class of spaces: e.g. all Hausdorff spaces in which the compact sets are zero-dimensional in the relative topology.

I have a more general answer to this, but I prefered to omit it from the talk.

## Computational lemma

*Searchable $\implies$ $2$-exhaustible.*

Proof (Berger 1990). Given $\varepsilon_K$, first define

$$\exists_K(p) = p(\varepsilon_K(p)),$$

and then

$$\forall_K(p) = \neg\exists_K(\lambda x.\neg p(x)).$$

This simple observation will be crucial to not-so-simple constructions.

## Theorem

$\mathbb{S}$*-exhaustible sets are closed under images and products.*

Topologically, this is clear.

Computationally, we restrict to *countable* products.

Proof (Escardó (2004)). Write down a program and use the countable Tychonoff Theorem to establish termination.

## Proposition

*$2$-exhaustible sets are closed under images and finite products.*

Topogically, this is clear.

Computationally: see Escardó (2004). Not very difficult anyway.

## Theorem

*Searchable sets are closed under*

1. *images,*
2. *finite products, and*
3. *countable products,*

Topologically, this is clear.

Computationally, this requires coming up with some algorithms.

## 1. Images of searchable sets

If $\varepsilon_K$ realizes the searchability of $K \subseteq X$ and $f\colon X \to Y$, then $\varepsilon_{f(K)}$ defined by

$$\varepsilon_{f(K)}(q) = f(\varepsilon_K(q \circ f))$$

realizes the searchability of $f(K)$.

To understand this:

Write the rhs as $f(\varepsilon_K(\lambda x.q(fx)))$.

Read $\varepsilon_K(\lambda x.q(fx))$ as “find $x \in K$ such that $q(fx)$”.

All we have to do is apply $f$ to that $x$.

## 2. Finite products of searchable sets

Given searchable sets $K$ and $L$ in types $X$ and $Y$ with realizers $\varepsilon_K$ and $\varepsilon_L$, the functional $\varepsilon_{K\times L}$ defined by

1. find $x_0 \in K$ such that $p(x_0, y)$ for some $y \in L$,
2. find $y_0 \in L$ such that $p(x_0, y_0)$,
3. let $\varepsilon_{K\times L}(p) = (x_0, y_0)$

realizes the searchability of $K \times L$ relative to $X \times Y$.

## 2. Finite products of searchable sets

More formally,

$$\varepsilon_{K\times L}(p) = (x_0, y_0),$$

where

$$\begin{aligned} x_0 &= \varepsilon_K(\lambda x.p(x, \varepsilon_L(\lambda y.p(x,y)))), \\ y_0 &= \varepsilon_L(\lambda y.p(x_0, y)). \end{aligned}$$

Here the existential quantification has been reduced to search.

The countable case iterates this idea.

## 3. Countable products of searchable sets

Given search functionals $\varepsilon_{K_i} \colon (X \to 2) \to X$,

we wish to define a search functional $\varepsilon_{\prod_i K_i} \colon (X^\omega \to 2) \to X^\omega$.

We let $\varepsilon_{\prod_i S_i}(p) = \vec{x} = x_0 x_1 x_2 \ldots x_n \ldots$, where

$x_0 \in K_0$ is such that $p(x_0 \alpha)$ for some $\alpha \in \prod_i K_{i+1}$.
$x_1 \in K_1$ is such that $p(x_0 x_1 \alpha)$ for some $\alpha \in \prod_i K_{i+2}$.
$x_2 \in K_2$ is such that $p(x_0 x_1 x_2 \alpha)$ for some $\alpha \in \prod_i K_{i+3}$.
$\vdots$
$x_n \in K_n$ is such that $p(x_0 x_1 \ldots x_n \alpha)$ for some $\alpha \in \prod_i K_{i+n+1}$.
$\vdots$

Here $x_n$ is found using $\varepsilon_{K_n}$

Existential quantifications can be reduced to search, recursively.

## 3. Countable products of searchable sets

Let's change notation.

Given $\varepsilon \in ((X \to 2) \to X)^\omega$, we wish to find $\Pi(\varepsilon) \in (X^\omega \to 2) \to X^\omega$.

I.e., we want a functional $\Pi \colon ((X \to 2) \to X)^\omega \to ((X^\omega \to 2) \to X^\omega)$.

$\Pi(\varepsilon)(p)(0) = x_0$ such that $p(x_0\alpha)$ for some $\alpha \in \prod_i K_{i+1}$.
$\Pi(\varepsilon)(p)(1) = x_1$ such that $p(x_0x_1\alpha)$ for some $\alpha \in \prod_i K_{i+2}$.
$\Pi(\varepsilon)(p)(2) = x_2$ such that $p(x_0x_1x_2\alpha)$ for some $\alpha \in \prod_i K_{i+3}$.
$\vdots$
$\Pi(\varepsilon)(p)(n) = x_n$ such that $p(x_0x_1 \dots x_n\alpha)$ for some $\alpha \in \prod_i K_{i+n+1}$.
$\vdots$

Here $x_n$ is found using $\varepsilon_n$.

The quantification in $\Pi(\varepsilon)(p)(n)$ is derived from $\Pi(\lambda i.\varepsilon_{i+n+1})$.

## 3. Countable products of searchable sets

Now domain theory proves handy.

The topological space $X$ corresponds to a domain $D$.

Under this indentification, we can take $\Pi$ to be the least continuous functional such that

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

where

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

## 3. Countable products of searchable sets

To prove that $\Pi$ is total, crucially use that $D$ is algebraic.

(Continuity suffices.)

Use topological lemma and Tychonoff to show $p$ uniformly continuous.

Conclude by induction on the modulus of uniform continuity.

Because $\Pi$ is PCF-definable, it is computable.

In particular, if $\varepsilon$ is computable, so is $\Pi(\varepsilon)$.

## 3. Countable products of searchable sets

The definition of $\Pi$ can be reduced to one line:

$$\Pi(\varepsilon)(p) = f(\varepsilon_0(p \circ f)) \text{ where } f(x) = x \sharp \Pi(\varepsilon')(\lambda\alpha.p(x \sharp \alpha)).$$

$x \sharp \alpha$ is the sequence with head $x$ and tail $\alpha$.

$\varepsilon'$ is the tail of the sequence $\varepsilon$.

But in practice the previous can be made more efficient.

## Examples of exhaustively searchable spaces

$2$.

$$\varepsilon_2(p) = p(\mathrm{True}) \qquad (= \mathrm{if}\, p(\mathrm{True})\, \mathrm{then\, True\, else\, False}).$$

Cantor space $2^{\mathbb{N}}$.

$$\varepsilon_{2^{\mathbb{N}}} = \Pi(\lambda i.\varepsilon_2).$$

$\mathbb{Z}_{n+1} = \{0, \ldots, n\}$. Because it is finite and non-empty.

$\prod_n \mathbb{Z}_{n+1}$. Use $\Pi$ and the fact that $\mathbb{Z}_{n+1}$ is searchable uniformly in $n$.

## Counter-examples

Discrete $\mathbb{N}$.

Baire space $\mathbb{N}^{\mathbb{N}}$.

Because they are not topologically compact.

(Before we had, for $\mathbb{N}$: because this violates the Halting problem.)

## Plan

0. Berger's example. ✓

1. Derivation of this as a corollary of a more general result. ✓

2. Show that non-trivial instances of search can be surprisingly fast,

3. but that not all of them can be fast. ✓

It remains to do (2).

## Coding the above theorems and proofs in Haskell

Type of realizers of searchable subspaces.

```
type Subspace x = (x -> Bool) -> x
```

## Coding the above theorems and proofs in Haskell

Basic examples.

Name realizers after the subspaces they search over.

```
bool :: Subspace Bool
bit  :: Subspace N
z    :: N -> Subspace N

bool = \p -> p True
bit  = \p -> if p 0 then 0 else 1

z 0 = undefined
z 1 = \p -> 0
z n = \p -> if p(n-1) then n-1 else z(n-1) p
```

## Coding the above theorems and proofs in Haskell

Example factory 1: Closure under images.

```
image :: (x -> y) -> Subspace x -> Subspace y

image f k = \q -> f(k(q.f))
```

## Coding the above theorems and proofs in Haskell

Example factory 2: Closure under binary products.

```
times :: Subspace x -> Subspace y -> Subspace(x,y)

(k ‘times‘ l) p = (x0,y0)
            where x0 = k(\x -> p(x,l(\y -> p(x,y))))
                  y0 = l(\y -> p(x0,y))
```

## Coding the above theorems and proofs in Haskell

Example factory 3: Closure under countable products.

```
prod :: (N -> Subspace x) -> Subspace(N -> x)

prod e p n = e n (\x->q x n(prod(\i->e(i+n+1))(q x n)))
  where q x n a = p(\i -> if i  < n then prod e p i
                     else if i == n then x
                                    else a(i-n-1))
```

## Coding the above theorems and proofs in Haskell

Particular examples:

```
cantor :: Subspace(N -> Bool)
cantor = prod(\i -> bool)

bcantor :: Subspace(N -> N)
bcantor = prod(\i -> bit)

factorial :: Subspace(N -> N)
factorial = prod(\i -> z(i+1))
```

## Coding the above theorems and proofs in Haskell

Quantifiers: They are bounded now.

They take a (compact) subspace to quantify over.

```
exists, forall :: Subspace x -> (x -> Bool) -> Bool

exists k p = p(k p)

forall k p = not(exists k(\x -> not(p x)))
```

## Coding the above theorems and proofs in Haskell

Equality of functions on a subspace:

```
equal :: Eq y => Subspace x -> (x->y) -> (x->y) -> Bool

equal k f g = forall k(\x -> f x == g x)
```

## Experiments

Consider

```
f,g,h :: (N -> N) -> N

f a = 10*a(3^400)+100*a(4^400)+1000*a(5^400)

g a = 10*a(3^400)+100*a(4^400)+1000*a(6^400)

h a = 10 + if a(4^400) == 1 then j+100 else j
    where i = if even(a(5^400)) then 0 else 1000
          j = if  odd(a(3^400)) then i else i-10
```

## Experiments

We get:

```
> equal bcantor f g
False
(0.90 secs, 161158104 bytes)

> equal bcantor f h
True
(0.07 secs, 11935752 bytes)
```

Time $2^{6^{400}}$ using Berger's algorithm.

## But not good enough yet

Consider

```
f',g',h' :: (N -> N) -> N

f' a = a(10*a(3^400)+100*a(4^400)+1000*a(5^400))

g' a = a(10*a(3^400)+100*a(4^400)+1000*a(6^400))

h' a = a(10 + if a(4^400) == 1 then j+100 else j)
    where i = if even(a(5^400)) then 0 else 1000
          j = if  odd(a(3^400)) then i else i-10
```

## But not good enough yet

```
> equal bcantor f’ g’
[no answer for a long time]
```

## Second version

Memoize. But how? Apply suitably defined identity function.

```
prod :: (N -> Subspace x) -> Subspace (N -> x)

prod e p = b
where b=id’(\n->e n(\x->q x n(prod(\i->e(i+n+1))(q x n))))
      q x n a = p(\i -> if i  < n then b i
                   else if i == n then x
                                  else a(i-n-1))
```

## Convoluted implementation `id′` of the identity

Store a function into an infinite binary tree, exploiting call by need.

```
data T x = B x (T x) (T x)

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

decode :: T a -> (N -> a)
decode (B x l r) n | n == 0    = x
                   | odd n     = decode l ((n-1) ‘div‘ 2)
                   | otherwise = decode r ((n-2) ‘div‘ 2)

id’ :: (N -> x) -> (N -> x)
id’ = decode.code
```

## Experiments

```
> equal bcantor f’ g’
False
(3.18 secs, 379345012 bytes)

> equal bcantor f’ h’
True
(1.50 secs, 178959112 bytes)
```

## Experiments

The previous example gets even faster:

```
> equal bcantor f g
False
(0.15 secs, 21369096 bytes)

> equal bcantor f h
True
(0.11 secs, 9022888 bytes)
```

## Experiments

```
> exists tcantor(\a->a(5ˆ500)+a(6ˆ600)+a(7ˆ700)+
a(8ˆ800)+ a(9ˆ900)>=4ˆ400)

False (4.81 secs, 623124312 bytes)

> exists tcantor(\a->a(5ˆ500)+a(6ˆ600)+a(7ˆ700)+
a(8ˆ800)+a(9ˆ900)==11)

False (4.76 secs, 622509432 bytes)

> exists tcantor(\a->a(5ˆ500)+a(6ˆ600)+a(7ˆ700)+
a(8ˆ800)+a(9ˆ900)==10)

True (1.03 secs, 131308552 bytes)
```

## Theorem

*The expression*

```
> equal cantor (\a -> a m) (\a -> a n)
```

*runs in time* $\log(\max(m,n))$.

Rather than $2^{\max(m,n)}$.

Doubly exponential gain!

## Experiments

Testing when two finite lists have the same set of elements:

```
projset :: [N] -> ((N -> Bool) -> Bool)
projset [] = \a -> True
projset (n:l) = \a -> (a n) && (projset l a)

eqset :: [N] -> [N] -> Bool
eqset k l = equal cantor (projset k) (projset l)
```

Notice that

```
projset [n1,...,nk] = \a -> a n1 && ... && a nk
```

## Experiments

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

True
(0.24 secs, 14849372 bytes)

> eqset [789^34,345^55,45,5,1000,4,10^100,5000,23^45]
[5,789^34,45,1000,23^45,5000,345^55,4]

False
(4.02 secs, 226002644 bytes)
```

## Computing the modulus of uniform continuity

```
modulus :: Eq y => ((N -> Bool) -> y) -> N
modulus f =
  mu(\n->forallC(\a->forallC(\b->eq n a b-->(f a==f b))))

mu :: (N -> Bool) -> N
mu p = if p 0 then 0 else 1 + mu(p.(1+))

forallC = forall cantor
forallZ n = forall (z(n+1))

eq n a b = forallZ n(\i -> a i == b i)

False-->p=True
True -->p=p
```

## Experiments

```
> modulus (\a -> a 6 && a 2)
6
(2.24 secs, 98132376 bytes)
```

But the computation of the modulus is exponential on the modulus.

## Refined modulus

For $I \subseteq \mathbb{N}$, and $\alpha, \beta \in 2^{\mathbb{N}}$, define

$$\alpha =_I \beta \iff \forall i \in I . \alpha_i = \beta_i.$$

Modulus of $f \colon 2^{\mathbb{N}} \to \mathbb{N}$:

Smallest $I \subseteq \mathbb{N}$ such that $\forall \alpha, \beta \in 2^{\mathbb{N}} . \alpha =_I \beta \implies f\alpha = f\beta$.

## Run time more generally

The run time of

```
    forall cantor p
```

is roughly proportional to

$$2^{\mathrm{cardinality}(I)},$$

assuming unit cost to evaluate $p(\alpha)$ for each $\alpha$.

(More precise estimates are possible.)

## End

1. Maybe this is now usable in practice?

For e.g.

Real-number computation (cf. Simpson).

Program verification and model-checking.

2. This gives an interesting example to higher-type complexity theorists.

3. I hope you regard this as a non-trivial, potentially useful, application of topology to computer science.