Astérisque

FRANCIS SERGERAERT

Functional coding and effective homology

Astérisque, tome 192 (1990), p. 57-67

<http://www.numdam.org/item?id=AST_1990__192__57_0>

FUNCTIONAL CODING AND EFFECTIVE HOMOLOGY*

by *Francis SERGERAERT*

Summary.

The functional coding technique, which is the essential basis of the effective homology theory, is explained. A very elementary example, the functions with effective growing, is used in order to describe the nature of this technique. Finally, the effective homology theory is quickly defined and its results are stated; see [SRG].

1. Coding

The situation encountered by a mathematician working with a computer can be roughly described as follows.

Let $\overline{\mathcal{C}}$ be the "computer world" (some set) and $\mathcal{M}$ the "mathematical world" (another set). In this text, all the computer things are marked by overlining. If this mathematician has to work with the elements of a set $A \subset \mathcal{M}$, he must define a *coding* for A ; such a coding is a set $\overline{A} \subset \overline{\mathcal{C}}$ and a coding map $\chi_A : \overline{A} \to A$ establishing a correspondence between $\overline{A}$ (the computer version of A) and A itself.

If $\overline{x} \in \overline{A}$, then $x = \chi_A(\overline{x})$ is *coded* by $\overline{x}$, or $\overline{x}$ *codes* x .

The following example is perhaps the first learned in the computer courses : $A = \mathbb{N}$ (the integers), $\overline{A} = \{\text{bit strings}\}$, $\chi_A =$ the well-known map .

---

* Talk given at the *Congress "Computational Geometry and Topology and Computation in Teaching Mathematics" SEVILLA, 1987*

S.M.F.
Astérisque 192 (1990)

The important point is that the coding map χ_A must be considered as an element of $\mathcal{M}$ so that any mathematical trick can be used for the definition of such a coding map. This talk is devoted to the *algorithmic trick.*

In other respects, the example of $\mathbb{N}$ and the bit strings could be trying to suggest that a coding map χ_A should be bijective. But this need not be the case. At first, in many situations, many different codings $\overline{x}_1, \overline{x}_2, \ldots$ can naturally exist for some (or any) element x in A and there does not necessarily exist a good method of choosing a particular coding. Next, it happens that a very natural coding $\chi_A : \overline{A} \to A$ can be defined, which is not surjective; then the image of χ_A is an interesting subset of A , which cannot really be otherwise defined : it is the subset of the recursive (or effective) elements of A with respect to the coding χ_A .

Note that $\overline{\mathcal{C}}$ is a countable set so that if A is not, the coding map χ_A cannot be surjective; this is a frequent situation.

2. Functional coding

Suppose you have to work with a computer on the finite subsets of $\mathbb{N}$:

$$A = \mathcal{P}_F(\mathbb{N}) = \{X \subset \mathbb{N} \text{ s.t. } \sharp X < \infty\} .$$

The usual coding method is the use of integer lists; in many programming languages the set $\overline{A}$ of the integer lists can be considered as a subset of the computer world $\overline{\mathcal{C}}$, and the coding map is the obvious one :

$$\overline{\mathcal{C}} \supset \overline{A} \xrightarrow{\chi_A} A \subset \mathcal{M}$$

$$\texttt{(1 3 14 16)} \mapsto \{1,3,14,16\}$$

$$\texttt{(1 3 5 7 ... 99999)} \mapsto \{1,3,5,7,\ldots,99999\}$$

$$\texttt{()} \mapsto \emptyset$$

But there is a quite different method, the functional method, which consists in using the algorithmic trick.

So let A be the set $\mathcal{P}_F(\mathbb{N})$ and $\overline{A}$ the set of the algorithms $\overline{\alpha}$ which can work on any integer n and satisfy :

a) the answer $\overline{\alpha(n)} \in \{\overline{\texttt{false}}, \overline{\texttt{true}}\}$

b) $\{n \in \mathbb{N} \text{ s.t. } \overline{\alpha(n)} = \overline{\texttt{true}}\} \in \mathcal{P}_F(\mathbb{N})$.

In the good programming languages, such algorithms can be considered as elements of $\mathcal{C}$. From this point of view, the lambda calculus at a theoretical level, and the Lisp language at a practical level, are the best ones.

The coding map is obvious

$$\overline{A} \ni \overline{\alpha} \xmapsto{\chi_A} \{n \in \mathbb{N} \text{ s.t. } \overline{\alpha(n)} = \overline{\texttt{true}}\}$$

Examples (Lisp-written) :

a)

```
(lambda(n) (member n '(1 3 14 16)))
```

$$\chi_A \downarrow$$

$$\{1,3,14,16\}$$

b)

```
(lambda(n) (member n '(1 3 5 ... 99999)))
```

$$\chi_A \downarrow$$

$$\{1,3,5,...,99999\}$$

This is a very expensive coding : it needs 294 469 characters !

c) Better coding of the same subset :

```
(lambda(n) (and (<  n 100000)
                (oddp n)))
```

Now a string of 37 characters is sufficient.

Of course we see that χ_A is not injective.

d)

```
(lambda(n) 'false)
```

or

```
(lambda(n) (= (+ n 1) (+ n 2)))
```

$$\chi_A \downarrow$$

$$\emptyset$$

and so on ...

But there is now a very interesting remark : the same coding can be used without change with the finiteness hypothesis omitted.

We set :

$$A = \mathcal{P}(\mathbb{N}) = \{X \subset \mathbb{N}\}$$

$$\overline{A} = \{\text{algorithms } \overline{\alpha} \text{ which can work on any integer } \overline{n} \text{ and answers } \overline{\texttt{false}} \text{ or } \overline{\texttt{true}}\} .$$

$$\chi_A : \overline{A} \longrightarrow A$$
$$\alpha \mapsto \{n \in \mathbb{N} \text{ s.t. } \overline{\alpha(n)} = \overline{\texttt{true}}\}$$

Examples :

a)

```
(lambda (n) 'true)
```

$$\chi_A \downarrow$$
$$\mathbb{N}$$

So a string of 16 characters is sufficient to code the biggest subset.

b)

```
(lambda (n) (oddp n))
```

$$\chi_A \downarrow$$
$$\{1,3,5,7,...\}$$

c)

```
(lambda (n) ... code that examines if
                n is a counter-example of
                Goldbach's conjecture ...)
```

$$\chi_A \downarrow$$
$$G$$

Today, nobody knows if G is empty or not.

Note that $\overline{A}$ is countable again ($\overline{C}$ is countable) when $\mathcal{P}(\mathbb{N})$ is not; the image of χ_A is the (countable) set of the recursive (or effective) subsets of $\mathbb{N}$.

As in the other ordinary coding situations, computer functions can be written in order to realize some operations on such codings ; see the **compose** function in [STL], pp. 37-38. If on your lisp machine, you execute :

```
(defun union (s1 s2)
    #'(lambda (n) (or (funcall s1 n)
                      (funcall s2 n))))
```

then a lisp function "**union**" is defined which can compute the functional coding for the union of two so coded subsets of $\mathbb{N}$; here is an example of use :