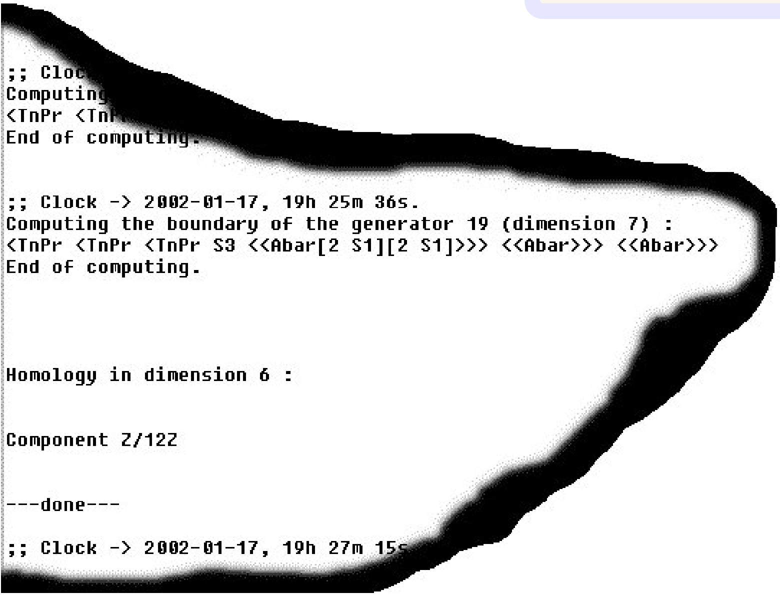
Lexical Closures

and

Complexity



Francis Sergeraert, Institut Fourier, Grenoble

9th European Lisp Symposium
Krakow – 9-10 May 2016

Semantics of colours:

Blue = **“Standard” Mathematics**

Red = **Constructive, effective,**

algorithm, machine object, . . .

Violet = **Problem, difficulty,**

obstacle, disadvantage, . . .

Green = **Solution, essential point,**

mathematicians, . . .

Plan.

- **1. Introduction.**
- **2. Effective Homology and Functional Objects.**
- **3. Functional Programming is Free!**
- **4. Polynomial complexity with respect to Functional Programming.**
- **5. Typical example of Kenzo computation.**

1. Introduction.

Personal programming language history:

- **1972 = Fortran.**
- **1973 = PL/I.**
- **1974 = Iris Assembly.** (1975 = first keyboard terminal!)
- **1977 = Pascal.** (1978 = end of punched cards!)
- **1984 = Maclisp.**
- **1986 = Common Lisp.**

- **1984 = Beginning of Effective Homology** $\Rightarrow$
- **1984 = Maclisp.**
- **1986 = Common Lisp.**
- **1990 = First program of Effective Homology.**
- **1998 = Kenzo program.**
- **2015 = Effective Homology is Polynomial.**

2. Effective Homology and Functional Objects.

Functional trick to code infinite objects.

Typical examples: $K(\mathbb{Z}, 1)$:

$K(\mathbb{Z}, 1)$ is a simplicial set (∼ simplicial complex).

Set of n-simplices:

$$K(\mathbb{Z}, 1)_n := \mathbb{Z}^n = \{(a_1, \ldots, a_n), a_i \in \mathbb{Z}\}$$

Examples of face operator in $K(\mathbb{Z}, 1)$:

$$\partial_2(3,4,5,6,7) := (3,4+5,6,7) = (3,9,6,7)$$

$$\partial_4(3,9,6,5,2) := (3,4,6,5+2) = (3,9,6,7)$$

Remark:

$$\partial_2(3,4,5,6,7) = \partial_4(3,9,6,5,2) = (3,9,6,7)$$

so that $(3,4,5,6,7)$ and $(3,9,6,5,2)$ share a common face.

$\Rightarrow$ **Terrible geometry!!**

$\Rightarrow$ **Terrible topology!!**

“Functional trick” =

the art of coding infinite objects by functional objects.

Many “regular” infinite objects

can be coded thanks to the functional trick.

Problem: How to obtain “global” properties of such objects?

Typical example:

Pocket computer $\supset$ functional coding of $\mathbb{Z}$.

Is the ring $\mathbb{Z}$ principal?

Example: How to compute the homology groups

of an infinite simplicial set functionnally coded?

Definitive obstacle: standard logical negative theorems:

Gödel, Turing, Church, Post.

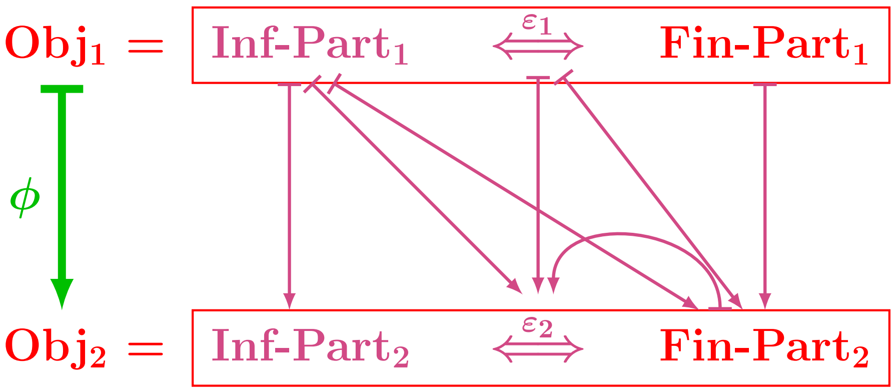
Effective Homology = Solution for Algebraic Topology:

Combining infinite objects and finite objects.

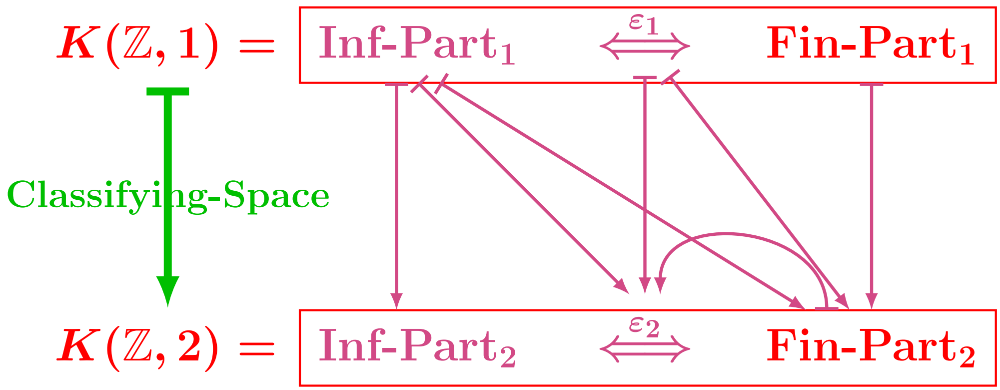
$$\text{Obj} = \boxed{\text{Inf-Part} \quad \overset{\varepsilon}{\Longleftrightarrow} \quad \text{Fin-Part}}$$

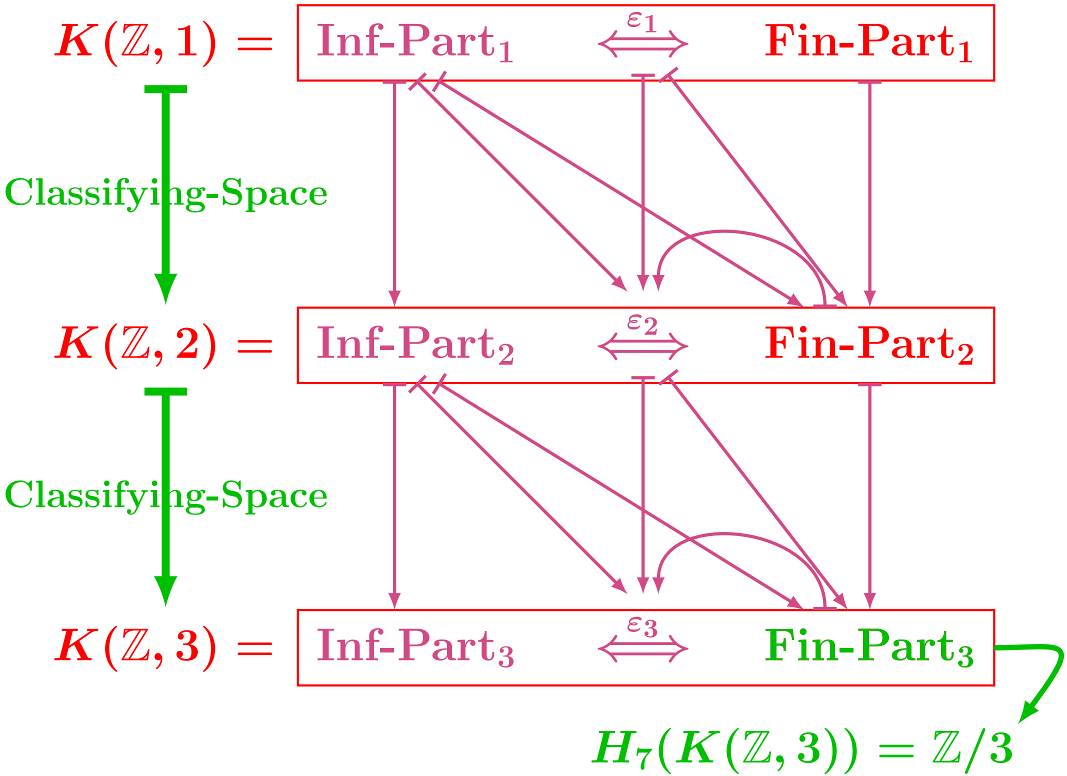
General scheme of Effective Homology:

ϕ = some mathematical constructor.



Example: the Classifying-Space constructor:




$K(\mathbb{Z},1) =$
Inf-Part$_1$
ε_1
Fin-Part$_1$
Classifying-Space
$K(\mathbb{Z},2) =$
Inf-Part$_2$
ε_2
Fin-Part$_2$
Classifying-Space
$K(\mathbb{Z},3) =$
Inf-Part$_3$
ε_3
Fin-Part$_3$
$H_7(K(\mathbb{Z},3)) = \mathbb{Z}/3$

3. Functional Programming is Free!

Most Functional Programming is done through Closures.

Two quite different points of view:

- **The definition of one closure in the source code**
  = Source Closure.

- **The closures generated at run-time = Closure Objects.**

Each source closure will generate
several closure objects at run-time.

Closure = Code + Environment

Closure = code to be executed

with respect to an environment.

In the source code:

- **Closure code = ordinary code.**
- **Environment = Implicit environment**

  defined by the lexical variables

  defined outside the closure, but visible from the closure.

Standard Toy Example:

```
(DEFUN MAKE-MULTIPLIER (FACTOR)
  #'(LAMBDA (ARG)
      (* FACTOR ARG)))
```

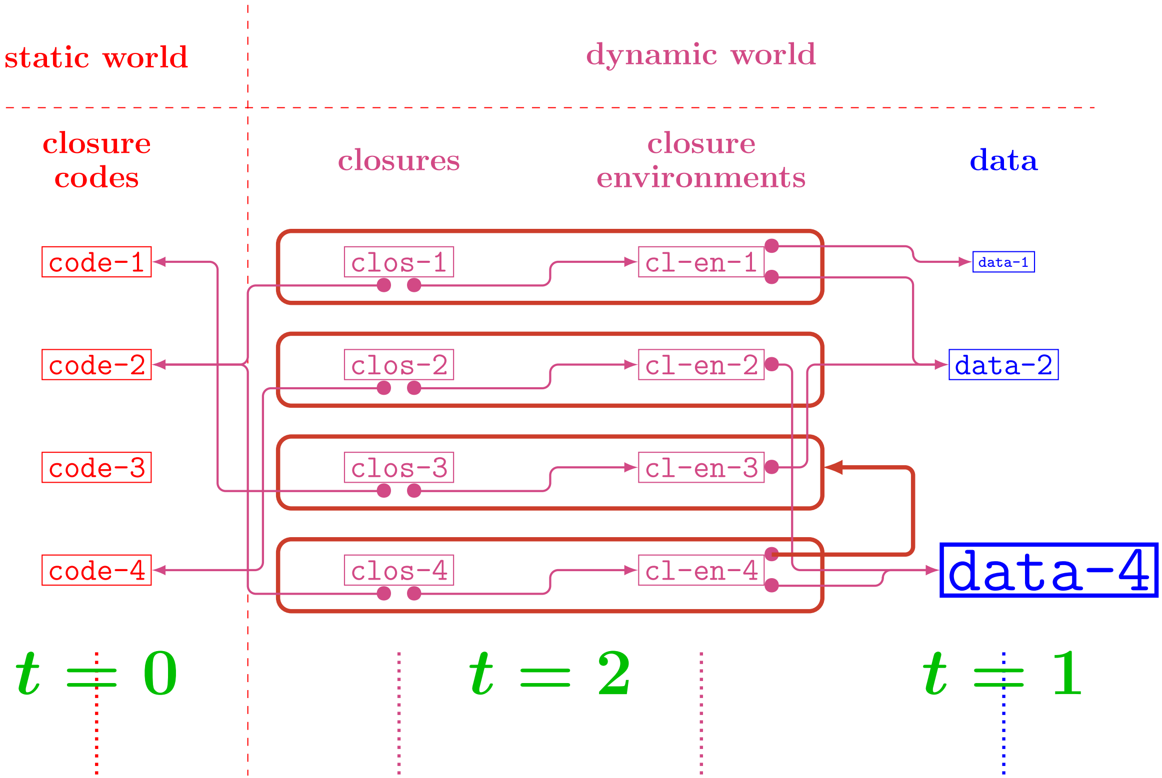
Source closure = `#'(LAMBDA (ARG) (* FACTOR ARG))`

Code = `(* FACTOR ARG)`

Environment = {`FACTOR`}

Closure Objects at Run-time:

- A source closure can generate
  an arbitrary number of closure objects at run-time.
- A closure object is a collection of machine addresses,
  one for the corresponding code,
  a fixed number for the environment variables.
- The generation cost of a closure object
  is constant for the same source closure,
  in particular independent of
  the values of the environment variables.


static world
dynamic world
closure
codes
closures
closure
environments
data
code-1
code-2
code-3
code-4
clos-1
clos-2
clos-3
clos-4
cl-en-1
cl-en-2
cl-en-3
cl-en-4
data-1
data-2
data-4
t = 0
t = 2
t = 1

Theorem: $\mathcal{P}$ = a program using the closure technology.

Every code segment of $\mathcal{P}$,

in part. every closure code of $\mathcal{P}$ is polynomial.

A fixed number of closures are generated.

The generation cost of a closure object

is constant for the same source closure,

in particular independent of

the values of the environment variables

$\Rightarrow$ $\mathcal{P}$ is polynomial.

<u>Corollary:</u>

The main programs of Effective Homology

have a polynomial complexity.

4. Polynomial complexity with respect to Functional Programming.

Type reminder:

Atomic types: Numbers, booleans, characters, symbols, ...

$\boldsymbol{a}$ atomic object $\Rightarrow$

Obvious notion of size $\boldsymbol{\sigma}(a)$.

Decidable types: Atomic objects, lists, arrays, records, ... made of decidable objects.

$\boldsymbol{a}$ decidable object $\Rightarrow$

Obvious notion of size $\boldsymbol{\sigma}(a)$.

Functional types: $\mathcal{T}_1$ **and** $\mathcal{T}_2$ = **types already defined.**

$\mathcal{T}_1 \to \mathcal{T}_2$:= **types of functional objects** α **satisfying**

$$a \in \mathcal{T}_1 \quad \Rightarrow \quad \alpha(a) \text{ terminates and } \alpha(a) \in \mathcal{T}_2.$$

The functional types can in turn be used to compose other arbitrary complex types.

Example: $A, \ldots, H$ **decidable types. Then the type:**

$$[(A \to B) \to (C \to D)] \to [(E \to F) \to (G \to H)]$$

is defined.

What about a size function for the objects of this type?

In ordinary programming:

$$\alpha : A \to B \quad \text{with } A \text{ and } B \text{ decidable.}$$

Then:

$$\sigma(\alpha(a)) \leq \tau(\alpha, a)$$

Proof: Turing machine model.

In functional programming:

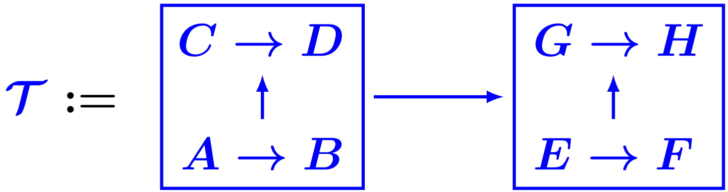
$$\alpha : \mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

can be a very small program α producing very quickly a very small functional object $\alpha(a) : \mathbb{N} \to \mathbb{N}$ being a terrible Ackermann function.

Solution ???

Standard type equivalences via currying and uncurrying:

$$A \to (B \to C) \;\underset{\text{uncurrying}}{\overset{\text{currying}}{\leftrightarrows}}\; (A \times B) \to C$$



Uncurrying $\Rightarrow$ $\mathcal{T}$ is equivalent to:

$$\{[[(A \to B) \times C] \to D] \times [(E \to F) \times G]\} \longrightarrow H$$

with all targets decidable if $A, \ldots, H$ are.

Key point = Notion of Polynomial Size.

Definition: A and B = Decidable types.

$\alpha \in A \to B$, d = natural integer.

Then: $$\sigma_d(\alpha) = \sup_{a \in A} \frac{\tau(\alpha, a)}{1 + \sigma(a)^d}$$

with $\tau(\alpha, a)$ = computing time

for α working on the input a.

$\Rightarrow$ Polynomial size with respect to a given degree d:

$$\tau(\alpha, a) \le \sigma_d(\alpha)(1 + \sigma(a)^d)$$

In general: $\sigma_d(\alpha) = +\infty$ if d small.

Case of $\alpha \in [A \to B] \to [C \to D]$ **with** $A, B, C, D,$ **decidable.**

Uncurrying $\Rightarrow \alpha \in [[A \to B] \times C] \to D$.

Notion of $\sigma_{d/d_{A\to B}}$**:**

$$\sigma_{d/d_{A\to B}}(\alpha) := \sup_{\phi \in A\to B,\, c\in C} \left[\frac{\tau(\alpha, (\phi, c))}{1 + (\sigma_{d_{A\to B}}(\phi) + \sigma(c))^d} \right]$$

$\Rightarrow$

$$\tau(\alpha, (\phi, c)) \le \sigma_{d/d_{A\to B}}(\alpha) \left[1 + (\sigma_{d_{A\to B}}(\phi) + \sigma(c))^d\right]$$

Definition: $\alpha \in [[A \to B] \times C] \to D$

$$= [A \xrightarrow{\phi} B] \to [C \xrightarrow{\alpha(\phi)} D]$$

α is polynomial if,

for every $d_{A\to B}$, there exists d $(= \pi_\alpha(d_{A\to B}))$

such that $\sigma_{d/d_{A\to B}}(\alpha) < +\infty$.

Corollary: For every d, there exists d' such that:

$$\sigma_{d'}(\alpha(\phi)) \leq 2^{2d'} \sigma_{d'/d}(\alpha)(1 + \sigma_d(\phi)^{d'})$$

$\Rightarrow$ α polynomial with respect to $\sigma_d(\phi)$ and $\sigma_{d'}(\alpha(\phi))$.

Systematic Uncurrying $\Rightarrow$

Can easily be generalized to arbitrarily complex situations.

$$\mathcal{T} := \boxed{\begin{array}{ccc} C & \to & D \\ & \uparrow & \\ A & \to & B \end{array}} \longrightarrow \boxed{\begin{array}{ccc} G & \to & H \\ & \uparrow & \\ E & \to & F \end{array}}$$

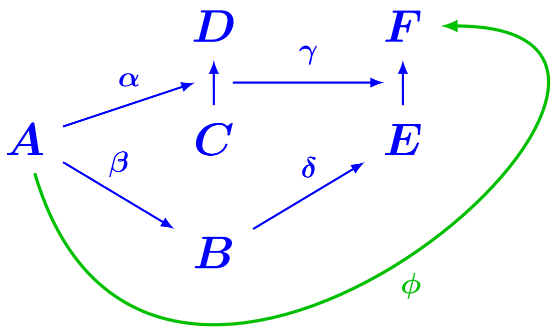
Uncurrying $\Rightarrow$ $\mathcal{T}$ is equivalent to:

$$\left\{\left[[(A \xrightarrow{d_1} B) \times C] \xrightarrow{d_2} D\right] \times [(E \xrightarrow{d_3} F) \times G]\right\} \xrightarrow{\boxed{d}} H$$

$\alpha \in \mathcal{T}$ is $\boxed{\text{polynomial}}$ if

for every (d_1, d_2, d_3), there exists d such that ...

Theorem: Arbitrary complex compositions of polynomial functions is a polynomial function.



$$\alpha, \beta, \gamma, \delta \text{ polynomial} \Rightarrow \phi \text{ polynomial}.$$

Remark: α and γ must generate closures, the cost of which is independent of the size of the initial input $a \in A$.

Corollary:

The main programs of Effective Homology

have a polynomial complexity.

5. Typical example of Kenzo computation.

Example: $\pi_5(\Omega(S^3) \cup_2 D^3) = (\mathbb{Z}/2)^4$

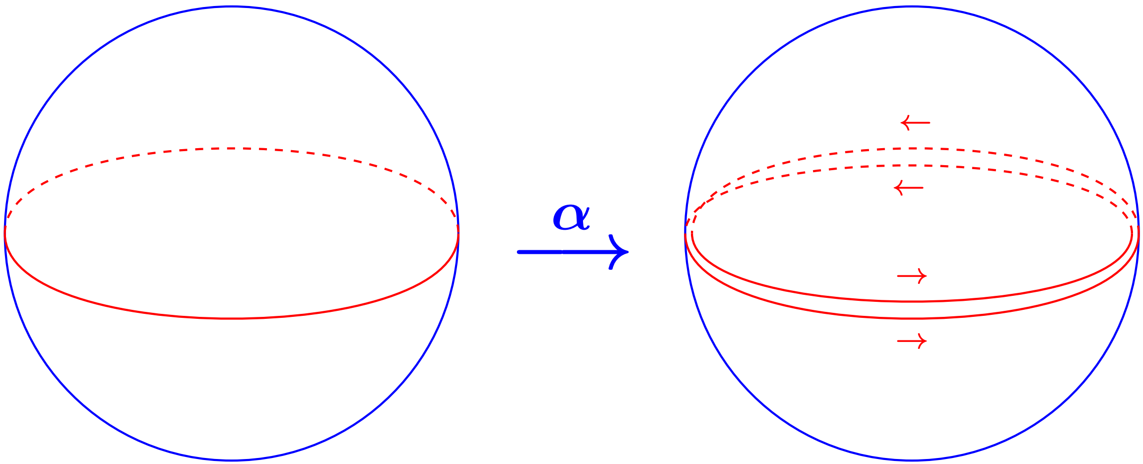
$\Omega(S^3) = \mathcal{C}(S^1, S^3)$

Natural subspace $S^2 \subset \Omega(S^3)$.

$\cup_2 D^3$ = Glue a 3-Disk by a map:

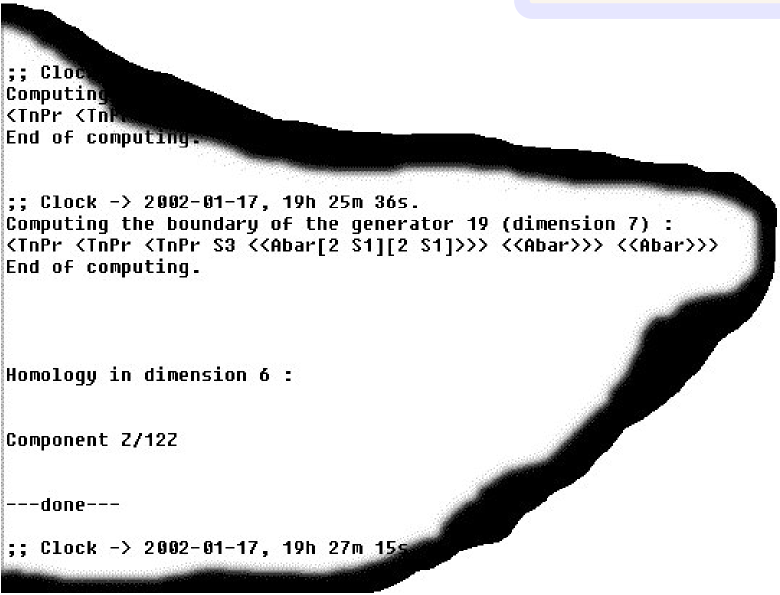
$\alpha : S^2 \to S^2 \subset \Omega(S^3)$ of degree 2.

$\mathcal{C}(S^5, (\Omega(S^3) \cup_2 D^3))$ has 16 connected components.



$D^3 \supset S^2$ $\qquad$ $S^2 \subset \Omega(S^3)$

The END


End of computing.
;; Clock -> 2002-01-17, 19h 25m 36s.
Computing the boundary of the generator 19 (dimension 7) :
<TnPr <TnPr <TnPr S3 <<Abar[2 S1][2 S1]>>> <<Abar>>> <<Abar>>>
End of computing.
Homology in dimension 6 :
Component Z/12Z
---done---