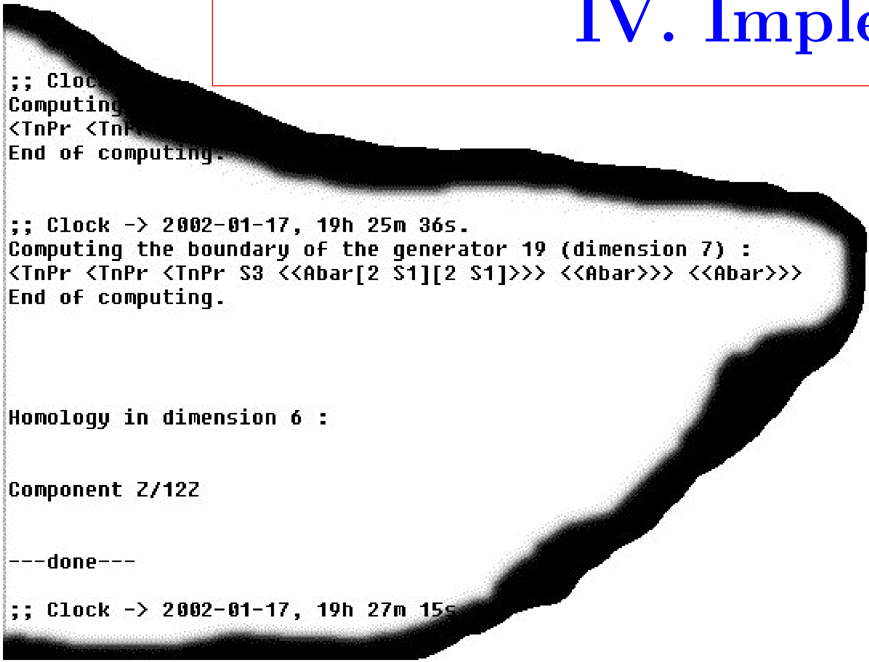
Algebraic Topology

(Castro-Urdiales tutorial)

IV. Implementation



Francis Sergeraert, Institut Fourier, Grenoble, France
Castro-Urdiales, January 9-13, 2006

Various levels of Mathematics:

1. **Philosophy: Interesting but what about "Truth"?**

2. **"Ordinary" Mathematics (= ZF or BvN):**

   ⇒ Truth but with respect to

   a somewhat arbitrary axiomatic system.

3. **Constructive Mathematics:**

   ⇒ Truth with respect to

   which can be handled by a theoretical computer.

4. **Implemented Mathematics:**

   ⇒ "Concrete" truth + Important feedback.

Main interesting problems

in Implemented Algebraic Topology:

1. **Role of Functional Programming.**
2. **Sophisticated Data Types**

   $\Rightarrow$ Sophisticated Object Oriented Programming.
3. **Role of Zermelo's Theorem.**
4. **Functional programming and memory management.**
5. **Macro-generation.**

Functional Programming.

Very important role when implementing

Eilenberg-MacLane-Steenrod Categories.

Ordinary work	Categorical work
Ordinary objects $3,\ 1 + X,\ \ldots$	Categorical objects $\Omega^2(S^4), P^\infty\mathbb{R}, \ldots$
Functions $* \mapsto 3 + 3X$	Functors $\times \mapsto \Omega^2(S^4) \times P^\infty(\mathbb{R})$
$\emptyset$ $\emptyset$	Functions inside Objects $\partial_i(\sigma) = ??$

An object in a category $\mathcal{C}$ is a set of coherent functions.

Ring $= (\in, =, +, -, *)$ **Simplicial Set** $= (\in, =, \{\partial_i^m\}, \{s_i^m\})$

Ring functor:

$$[X] : \overbrace{(\in_1, =_1, +_1, -_1, *_1)}^{A} \longmapsto \overbrace{(\in_2, =_2, +_2, -_2, *_2)}^{A[X]}$$

Topological functor:

$$\Omega : \overbrace{(\in_1, =_1, \{\partial_i^m\}_1, \{s_i^m\}_1)}^{X} \longmapsto \overbrace{(\in_2, =_2, \{\partial_i^m\}_2, \{s_i^m\}_2)}^{\Omega X}$$

Functor: $(\phi_1, \phi_2, \phi_3, \ldots) \longmapsto (\psi_1, \psi_2, \psi_3, \ldots)$

Important notion of lexical closure.

Local environments inside procedures

are classical and necessary.

Consider a functional procedure $F : \phi \mapsto \psi$

with ϕ and ψ procedures.

In general F, ϕ and ψ must work

with their own environments.

What about their respective status?

Lexical closure rule for dynamically created procedures:

A dynamically created procedure works in the environment which was the current environment at creation time.

This created procedure may, as a side effect, modify this environment.

Other procedures (dynamic or not) seeing (totally or partially) such an environment will know the possible modifications of this environment.

Lexical closure rule $\Rightarrow$ easy categorical programming.

Standard organization of categorical programming.

Example of loop-space functor.

$$\Omega : \overbrace{(\in_1, =_1, \{\partial_i^m\}_1, \{s_i^m\}_1)}^{X} \longmapsto \overbrace{(\in_2, =_2, \{\partial_i^m\}_2, \{s_i^m\}_2)}^{\Omega X}$$

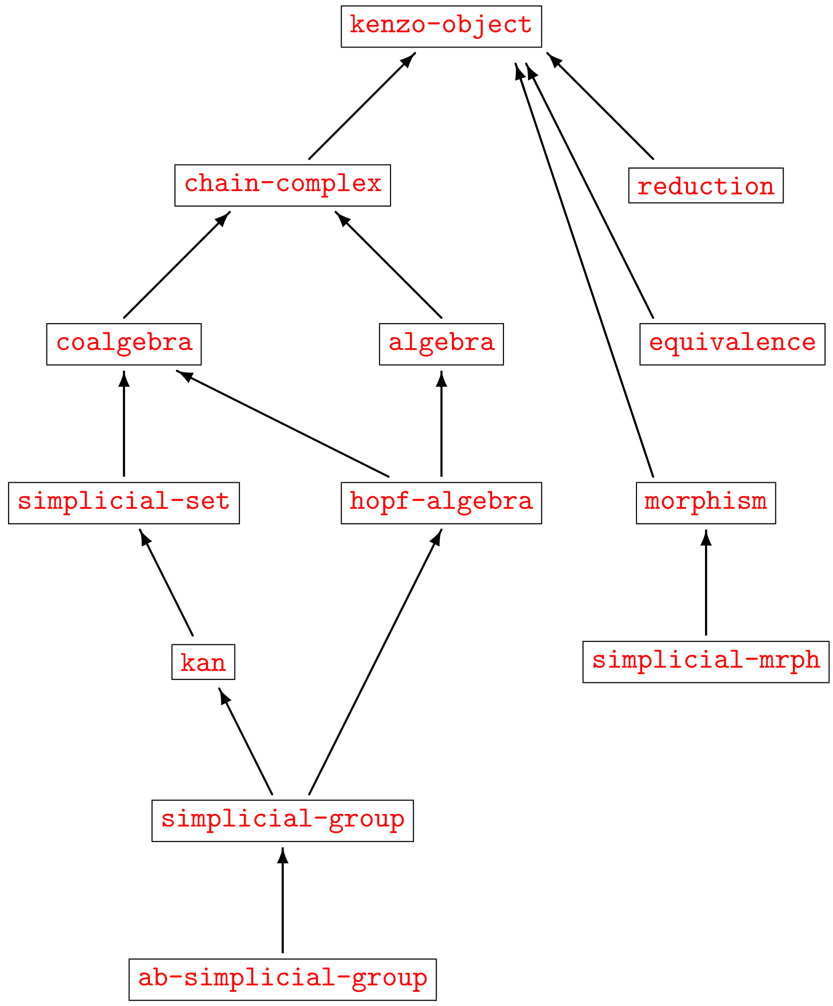
Every functional component of the image is obtained from the functional components of the source object by standard functional programming.

$$\ldots \quad \Omega_\partial : (=_1, \{\partial_i^m\}_1) \mapsto \{\partial_i^m\}_2 \quad \ldots$$

+ Put together these "partial" functors.

$$\Omega : \overbrace{(\in_1, =_1, \{\partial_i^m\}_1, \{s_i^m\}_1)}^{X} \mapsto \overbrace{(\Omega_\in(\in_1), \Omega_=(=_1), \Omega_\partial(=_1, \{\partial_i^m\}_1), \ldots)}^{\Omega X}$$

Data Types in Categorical Programming



A sophisticated OOP system is required.

Experience shows CLOS = Common Lisp Object System is quite good.

Main ingredients:

1. **Common Lisp itself defined with respect to CLOS (= specific instance of MOP = Meta-Object-Protocol).**
2. **OOP = Classes + Instances + Generic Functions + Methods + Method Combinations.**
3. **Powerful initialization system for instances.**
4. **Powerful method combination system.**

CLOS very powerful and flexible $\Rightarrow$

not so easy to design the right strategy!

General principles according to experience:

1. **Never destructively modify an instance component:**

   much too dangerous!

2. **Extending an instance or an instance component**

   is frequently necessary.

3. **Changing an instance class to a weaker one is forbidden,**

   to a stronger one is frequently necessary.

4. **Lazy management of induced components**

   frequently necessary.

Example of infinite loop

generated by a naive OOP management.

Let A be an algebra.

In particular $A = [\ldots, \mu : A \times A \to A, \ldots]$.

It is frequent to use A algebra $\Rightarrow A \times A =$ algebra.

If this naively implemented:

$A=[\ldots, \mu : A \times A \to A, \ldots]$
$\Rightarrow A \times A = [\ldots, \mu_2 : A^4 \to A^2, \ldots]$
$\Rightarrow A^4 = [\ldots, \mu_4 : A^8 \to A^4, \ldots] \Rightarrow$ **infinite loop!!**

Solution = lazy management of the $A \times A$ component.

Functional Programming and Memory Management.

Every run is split in two successive steps:

1. **"Automatic" writing of**
   a large oriented graph of functional objects.
   Short runtime.
2. **Use of these functional objects for a specific computation.**
   Possible very long runtime.

Many functional objects called and called again
for the same arguments.

What about memory management?

Extreme possible strategies:

1. **Lazy strategy.**

   Needs minimal memory and maximal time.

2. **Remember strategy.**

   Needs maximal memory

   and not necessarily minimal time.

Where is the happy medium ?

Only simple heuristics are used in the Kenzo program:

1. **If a computation is trivial, do not save it in memory!**

2. **If a computation has needed much time,**

   it is probably a good idea to save it!

Consider a situation where $f(a) := g(h(b(a)), k(c(a)))$.

Main ingredients:

1. **Computing times of $b(a)$ and $c(a)$.**
2. **Computing times of $h(b(a))$ and $k(c(a))$ ($b(a)$ and $c(a)$ given).**
3. **Computing time of $g(h(b(a)), k(c(a)))$ ($h(b(a))$ and $k(c(a))$ given).**

Problem: **What results do you save ?**

Role of Zermelo theorem.

Zermelo theorem: For every set E,

a well-ordering can be defined over E.

Role of this non-constructive theorem

in constructive mathematics?

"Zermelo remark" in implemented mathematics:

Any implemented set can be provided

with a constructive well-ordering.

Proof obvious. **Use not at all obvious!!**

What is an implemented set?

Definition: An implemented set is an algorithm $\mathcal{U} \to \mathbb{B}$.

Cantor-Russell theorem:

The "collection" of implemented sets

cannot be organized as an implemented set.

Is Zermelo remark really a remark?

Obvious proof: machine address.

Non-compatible with garbage collector!

Other proof ???

Two very different practical uses of Zermelo remark.

1. Hash coding =

= hashing function + sequential collision management.

Concrete efficiency ???

2. For every sensible implemented set,

a simple efficient well-ordering can be defined.

Combined with dichotomic retrieving:

Allows significantly efficient store-and-retrieve process

for remember strategy in functional programming.

Best method ?? **Problem still widely open!!**

Auxiliary natural question:

Let $\boldsymbol{E}$ be an implemented set.

Does there exist an effective well-ordering for $\boldsymbol{E}$?

Macro-Generation.

Macro-Generation = Macro-Assembly =

= Intermediary tool between

High-Level language and Machine (Assembly) language.

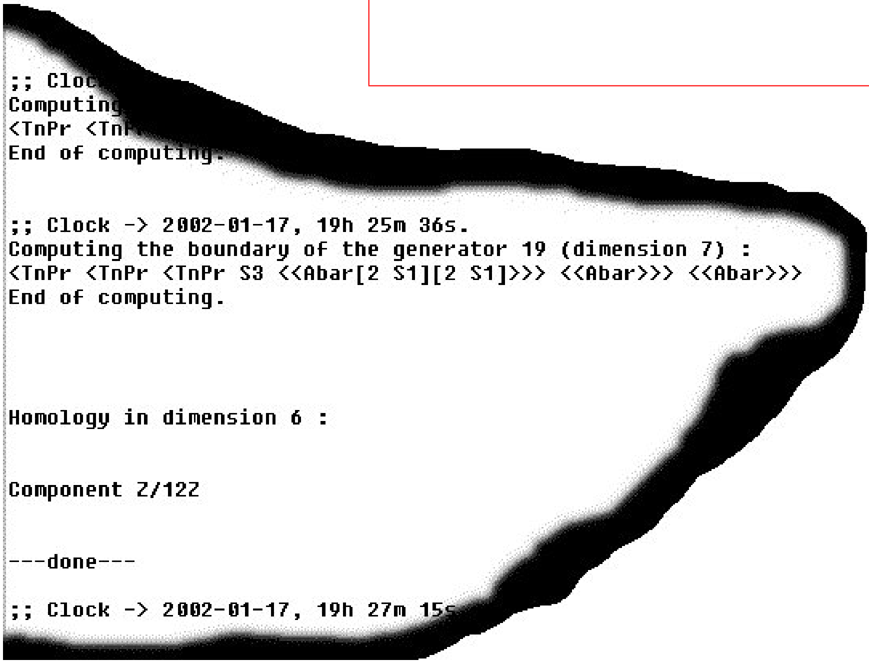
Runtime is critical in Implemented Algebraic Topology,

because of unavoidable exponential complexity.

Macro-generation is very useful to make compatible:

1. **Sensible readability.**
2. **Efficient compiled code.**

The END



```
;; Cloc
Computing
<TnPr <TnP
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---

;; Clock -> 2002-01-17, 19h 27m 15s
```

Francis Sergeraert, Institut Fourier, Grenoble, France
Castro-Urdiales, January 9-13, 2006