Common Lisp, Typing and Mathematics*

Francis Sergeraert†

August 2001

1 Introduction.

Common Lisp is seldom used by mathematicians and this is an anomaly. The high level of quality of Common Lisp comes from a simple reason: Common Lisp is a *mathematical* object, certainly one of the most beautiful and *potentially* productive existing at this time. Common Lisp is a descendant of the fantastic λ-calculus, designed by Church to produce one of the deepest *mathematical* results of the last century, namely the negative answer to Hilbert's Entscheidung Problem: no algorithm can determine whether an arbitrary mathematical statement is true, false or undecidable.

Then, by a quite indirect route, λ-calculus became a fascinating programming language, Lisp. Lisp's birth is old, at the end of the fifties, and forty years later it is clear it remains the most advanced "common" programming language, "common" meaning *with an ANSI norm*, in this case the... Common Lisp norm. The Common Lisp language is so advanced that it is not so easy to use it for serious applications: a reasonable lucidity about its complex and far-reaching structure is required; so that the initiation stage in the learning process of this language is a little hard. The present paper is a tutorial to introduce *at the mathematicians* the main components of Object Oriented Programming in Common Lisp. The extraordinary work of Guy Steele and his X3J13 committee during the years 1980-1994 led to a powerful language, on one hand, but on the other hand taking seriously account of standard *practical constraints* in *actual* programming work. Common Lisp is now widely used for complex industrial applications, mainly because of the powerful *dynamic* tools that are available; but it is not very used for scientific applications, in particular for mathematical applications, and this must be corrected.

No *concrete* language can be completely formalized, but the mathematical precision of the definition of Common Lisp gives its user much security, the precision which is required for sophisticated mathematical applications, in particular

*This text was written for a three hours satellite talk at the 2001 EACA Congress at Ezcaray.

†Institut Fourier, Université Grenoble I.
`Francis.Sergeraert@ujf-grenoble.fr`

when complicated data-sharings are necessary. Another aspect of Common Lisp is without any equivalent with the other languages: the user may *freely* design his application at the best programming level, sometimes at a very high level, directly handling rich and complex mathematical structures; for example it is explained in Section 5.2 that three simple lines of Lisp code are sufficient to explain to the machine that a simplicial group is simultaneously a Kan space and a Hopf algebra, sharing a common graded differential coalgebra structure, but with some added data, namely two simplicial morphisms describing the simplicial group structure. Other times, in the *same* program, you may on the contrary work at a very low level, in an assembler-like style. The language is carefully *stratified* to help the developer to remain lucid and at ease about these questions. The Common Lisp *macro generator*, without any equivalent in other common languages, allows also the programmer to easily use his low level code when he is on the contrary working at a high level.

Which is a good element for appreciation is the ability of a language to be quickly adapted to new *mandatory* evolutions; at this time, Object Oriented Programming (OOP) must be integrated in a language claimed *common*, and Common Lisp succeeded in doing it so nicely that it can even be used to *describe* what OOP exactly is. The CLOS (Common Lisp Object System) chapter of [11] is one of the most amazing part of Computer Science. The general *dynamic* feature of Common Lisp has in particular been kept: the user may dynamically change the class of his objects; he may also redefine dynamically the classes without loosing the objects of this class; and the user may freely decide exactly what happens for these objects. If this is not yet flexible enough, the Metaobject Protocol (MOP), not yet normalized, but already widely used, allows our user to freely define his own OOP system; CLOS is in fact only a particular application of MOP, see [6].

This paper is devoted to a few didactical examples to help mathematicians to understand how Common Lisp and more precisely CLOS can be used for mathematical applications. It is organized as follows. Firstly, a small set of artificial examples is used to explain the general structure of CLOS. The main work in OOP, and certainly the most difficult one, is at the initialization stage of the instances (objects). Even if a user does not intend to use the current Common-Lisp, studying how the initialization process is designed in this environment will explain to him the good points of view in this domain.

Then a more significant example is described to show the actual workstyle when CLOS is used in a non-beginner job. Because the OOP subject is strongly related to *typing*, a possible description of what typing *could be* is used as a theme of an exercise showing how CLOS allows to implement the consequences of this description. This example is very simple and should be a good tutorial to simultaneously understand the initial points of OOP under Common Lisp and what typing is, at least from the point of view of the *actual* programmer.

The next section is more mathematically oriented: it is explained how the basic mathematical *categories*, sets, magmas and monoids, can be directly implemented in CLOS, without meeting any kind of difficulty. The possibility for the Lisp user to arbitrarily mix OOP and high level functional programming (lexical closures) gives at once the required tools. The specialized programming environments such as Axiom, Gap, Magma... so can be skipped and the mathematician may freely work at an arbitrary level *in the programming language itself*, keeping a perfect control of the environment, without being bothered by a restrictive environment, seldom well-adjusted to an arbitrary new *research* work.

It was the case of the author and his colleagues when the central problems of Algebraic Topology were considered from a computational point of view [8, 7]. The resulting Lisp program Kenzo [4] shows a concrete example of use of CLOS for a relatively large implementation work[1]. The Kenzo program is not at all didactic, it is an actual program able to produce mathematical results that are unreachable otherwise. Some points of the experience acquired by Kenzo's authors are discussed in the last section and in this way the reader will see a few more concrete points of CLOS.

2 CLOS (= Common Lisp Object System).

Strictly speaking, the notion of *Common Lisp Object System*, CLOS in short, does not make sense anymore. The first definition of Common Lisp, known as Common-Lisp-1984 [10], did not have any "object system". Typing was already strong in CL-1984, at least if wished by the user; standard types were defined and the initial type system was freely extendable by the user, but the now classical notions of *classes*, *instances*, *generic functions* and *methods* were not yet available. The pre-ANSI version of Common Lisp, namely Common-Lisp-1990, contained a *proposal* for an OOP system, called Common Lisp Object System [11, Ch. 28, pp 770-864]; the pre-versions of this object system were already widely used at this time by the Lisp programmers. Finally, the ANSI specification [1] of Common Lisp (1994) completely integrated the so-called Common Lisp Object System in the very definition of the language. The standard Lisp programmer is now assumed working according to the spirit of OOP, constantly using classes, instances of classes, methods and so on, like in C++ or Java[2]. But the perfect integration of the general ideas of OOP with the so powerful Common Lisp gives its user many capabilities which are at this time outside of scope with other languages, mainly when *functional programming* is involved. However we continue to call CLOS the part of the ANSI Common Lisp language devoted to OOP.

This object system in Common Lisp coexists with the old typing system and also the old *structure* system, which allowed the user to define, construct and use the classical *record* objects, with several fields, typed or not. From a hierarchical point of view, the typing system and the structure system are *subsystems* of the object system: special predefined classes correspond to the main classical data types, and the same for the structured objects. But normally the current Lisp programmer mainly works with the object system, organizing his workspace in classes, subclasses, methods and so on.

[1] 16000 Common Lisp lines and a 340pp documentation.
[2] With a slightly different terminology.

The Common Lisp object system is elegant and powerful. As usual in Common Lisp, it is organized so that the user keeps a very large freedom. A *standard style* is carefully designed, but the user may go far from this standard style if special situations are encountered.

2.1 A quick comparison with C_{++} and Java.

A few indications are given in this section about the main OOP features of Java, C_{++} and CLOS. In Java, the classes are "above" the methods (member functions): the specific functions for a class are defined *inside this class* so that the class and member function hierarchies are more or less *parallel.* This leads sometimes to artificial classes, typically the `System` class and the `Math` class, when this *rule* becomes non-sense in particular situations. The *member functions* are available under C_{++} too.

In C_{++} you may also *overload* the functions, in particular the member functions. No member function in CLOS[3], only an overloading feature. But three other features are provided by CLOS which make particularly convenient and elegant the developer work:

- The initialization process of the objects is essential in OOP; the general CLOS organisation of this work is really wonderful, in particular conceptually very simple, leading the programmer towards the good points of view, see Section 2.3 for a small introduction to this point.
- The *method qualifiers* give much flexibility to organize the work of all the *applicable* methods; numerous simple examples in this paper.
- The functional basis of Common Lisp allows the user to easily install *functional slots* (members) of any sort, in particular *compiled closure slots*, dramatically extending the programming scope; Section 4 gives striking applications of this feature; in particular the instances may be themselves *funcallable*, in other words the instances become also functional objects, see Section 3 for a typical illustration.

In CLOS, the *basic ingredients* are clearly distinguished:

- The *class system*, mainly used through the `defclass` statement, allows the user to define the structure of his *instances* (objects), in particular through the notion of *subclass* (derived class).
- The *function type* is *primitive*; a function is an object which can be *funcalled* (called) for some work about some *arguments.*

[3] A C_{++} or Java member function is nothing but a CLOS generic function where the first argument is given *before* the function reference, with an implicit `with-slots` for this particular argument; such a feature can be easily added to the environment with the Metaobject Protocol; in general CLOS prefers to keep the maximal symmetry between the critical arguments.

- The *packaging system* allows the developer to precisely define what parts of his source code are normally directly known and/or reachable by a user of his program, obtaining in particular the equivalent of the `private` C++ code.

Then, if the *class system* and the *functional* organization of Common Lisp are to be simultaneously considered, the particular notions of *generic functions* and *methods* must be used.

- A *generic function* is a functional object the exact work of which will depend on the class of its arguments.
- The code for *some* specific work of a generic function corresponding to *some* particular class distribution of its arguments and *some qualifier* is a *method* object; each method is defined by a `defmethod` statement.
- In general *several* methods are involved when a *generic function* is invoked, they are called the *applicable* methods. CLOS gives its user a large freedom to organize these methods with respect to each other, to cover any possible situation, in particular when a sophisticated chronology of their work is required. This feature has no equivalent in C++ or Java, this is obtained by the *method qualifiers*.

In this way, the CLOS *methods* are roughly in a *three*-dimensional array where each entry is associated to a generic function, first index, the class distribution of the parameters, second index, and the method qualifier, third index.

The CLOS method qualifiers have no equivalent in C++ or Java; a predefined qualifier organization is provided, already rather rich, sufficient for most of the ordinary cases[4]. If still more sophisticated combinations of methods are required, the `define-method-combination` function allows the developer to freely extend this organization.

For example, and this will be detailed in Section 4, a mathematician who installs under CLOS the traditional mathematical categories must organize his work as follows:

- The `defclass` statements will be used to define what a *set* object is, what a *group* object is, what a *chain complex* object is, and so on.
- The `defgeneric` statements will be used to define functions working on these objects, but these "functions" are traditionnally called in this case *functors* in mathematics; therefore one `defgeneric` statement for the *sum* functor, another `defgeneric` for the *product* functor, another `defgeneric` for the *classifying space* functor, and so on.
- Finally each generic function will have various methods to adapt the generic function to specific cases; for example the product of two objects of some

[4]For example quite sufficient for the Kenzo program.

category is also an object of this category with the corresponding structure to be defined. Therefore one product method for the *sets*, another product method for the *magmas*, another product method for the *monoids*, and so on. The `call-next-method` and `change-class` functions will allow these methods to *possibly* refer to the *less specific* ones.

2.2 Very simple CLOS examples.

Classes can be defined with the traditional hierarchy and methods may be defined which will be called or not according to the argument classes. The general `call-next-method` function allows the user to invoke shadowed methods.

Let us define[5] a simple class `C1` and a subclass `C2`.

```
> (DEFCLASS C1 ()
    ((sl1 :initarg :sl1 :reader sl1))) ✠
#<STANDARD-CLASS C1>
> (DEFCLASS C2 (C1)
    ((sl2 :initarg :sl2 :reader sl2))) ✠
#<STANDARD-CLASS C2>
```

The `C2` class is a *subclass* of the `C1` class. A `C1`-object, in Lisp you must say a `C1`-*instance*, has only one component, you must say in Lisp one *slot*, labelled `sl1`, and a `C2`-instance has one further slot labelled `sl2`. Because of the `:initarg` slot-options, these slots may be initialized through the keyword arguments `:sl1` and `:sl2`. We create a `C1`-instance and a `C2`-instance.

```
> (setf i1 (make-instance 'c1 :sl1 1)) ✠
#<C1 @ #x20d3ce72>
> (setf i2 (make-instance 'c2 :sl1 11 :sl2 22)) ✠
#<C2 @ #x20be0a7a>
```

You see, and this is the *default display*, only the class of the instance and its machine address are displayed. We would like to display our simple instances with their slots[6]. In general every object is displayed through the generic function `print-object`, and the user is advised to write specific *methods* for this generic function to obtain the desired display. Just to illustrate the general organisation of methods in CLOS, we will define three `print-object` methods to "print" a `C1`- or `C2`-instance, to show the `sl1` and possibly the `sl2` slot(s). The `C2`-method calls the `C1`-method through `call-next-method`, and the `:after` method allows to print the '>' terminal.

[5]The small Lisp statements showed for illustration have really been run under Allegro Common Lisp and may be repeated under any ANSI Common Lisp; the Lisp prompt is here '>'; the maltese cross ✠ corresponds to the <Return> key asking for the evaluation of the typed-in statement; usually the symbols are keyed in lower case, and Lisp displays the answers with upper case.

[6]This is not standard: frequently the slots are numerous and complicated, and it is not sensible to display the instances with all their slots.

```
> (DEFMETHOD PRINT-OBJECT ((c1 c1) stream)
    (declare (type stream stream))
    (format stream "#<~S sl1=~S" (class-name (class-of c1)) (sl1 c1))
    c1) ✠
#<STANDARD-METHOD PRINT-OBJECT (C1 T)>
> (DEFMETHOD PRINT-OBJECT :after ((c1 c1) stream)
    (declare (type stream stream))
    (format stream ">")) ✠
#<STANDARD-METHOD PRINT-OBJECT :AFTER (C1 T)>
> (DEFMETHOD PRINT-OBJECT ((c2 c2) stream)
    (declare (type stream stream))
    (call-next-method)
    (format stream " sl2=~S" (sl2 c2))
    c2) ✠
#<STANDARD-METHOD PRINT-OBJECT (C2 T)>
```

The `format` statement is analogous to the traditional `printf` of C. It is frequent[7] to display a Lisp object with a string beginning by '`#<`'. Now we redisplay our existing `C1`- and `C2`-instances.

```
> (list i1 i2) ✠
(#<C1 sl1=1> #<C2 sl1=11 sl2=22>)
```

We can play to change the class of these instances.

```
> (change-class i1 'c2 :sl2 2) ✠
#<C2 sl1=1 sl2=2>
> (change-class i2 'c1) ✠
#<C1 sl1=11>
```

You see CLOS has taken the most natural decisions, but we will explain later how the process can be freely customized by the programmer.

There is also the possibility of `:around` methods, encapsulating the so-called *primary* ones:

```
> (DEFMETHOD PRINT-OBJECT :around ((c1 c1) stream)
    (declare (type stream stream))
    (format stream "*")
    (call-next-method)
    (format stream "*")
    c1) ✠
#<STANDARD-METHOD PRINT-OBJECT :AROUND (C1 T)>
```

[7] In principle a *non-readable* display must begin with '`#<`', but in this case, because of the complete display, in fact it could be read...

```
> (DEFMETHOD PRINT-OBJECT :around ((c2 c2) stream)
    (declare (type stream stream))
    (format stream "+")
    (call-next-method)
    (format stream "+")
    c2) ✠
#<STANDARD-METHOD PRINT-OBJECT :AROUND (C2 T)>
```

And we see the role of these methods:

```
> (list i1 i2) ✠
(+*#<C2 sl1=1 sl2=2>*+ *#<C1 sl1=11>*)
```

Again it is interesting to deduce from this example the chronology of the calls of the five user-defined `print-object` methods. The standard organization allows the user to define *primary* and *auxiliary* methods, the last ones being *qualified* by the keyword `:before`, `:after` or `:around`; the methods are *specialized* by giving arbitrary combination of classes of the arguments; here, only the first argument was specialized, but we will see later natural situations where several arguments are specialized. Simple but mathematically coherent rules, in particular based on topological sorting, explain what methods work in a particular case and in what order. Furthermore these possibilities may be freely modified or extended by the user through the `define-method-combination` function.

2.3 Initializing instances.

It is well known the main part and the main difficulties of the OOP job are in the initialization work for the instances. Let us examine the general organization of the CLOS initialization process.

The essential components of the initialization process in CLOS are the following generic functions:

- The *standard* `make-instance` allocates the memory space for the instance to be created and calls `initialize-instance` to initialize it;
- The *standard* `initialize-instance` in fact calls `shared-initialize` to do the initialization work.

The role of `shared-initialize` is the following. Frequently the initialization work to be done is essentially the same when an instance is created, or *re*initialized, or when its class is changed, or finally when the ambient class is itself redefined. If possible, the user is advised to define this common work in `shared-initialize` methods. If on the contrary there are significant differences between these cases, the user may write particular methods for the following generic functions:

- `initialize-instance`: called by *make-instance* to initialize a just created instance; the user can extend and/or modify the standard initialization by writing specific methods for some classes.

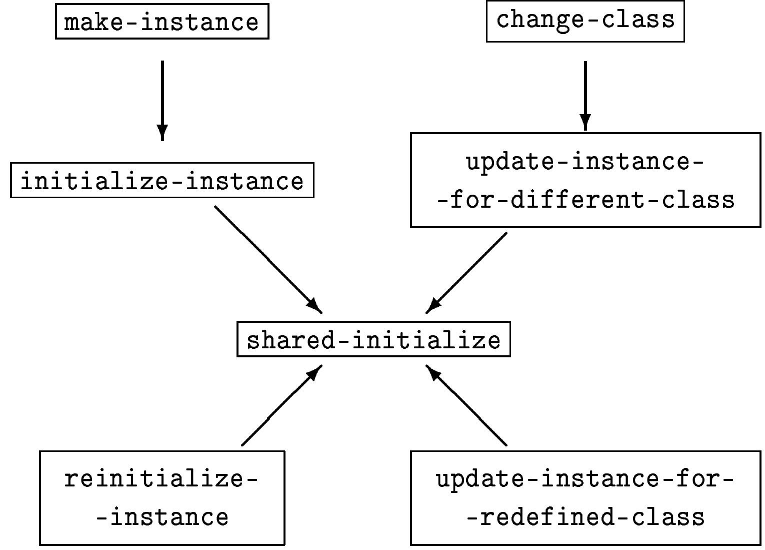


Figure 1: The main initialization generic functions.

- `reinitialize-instance`: called by the user when he wants to "refresh" some instance; sometimes it is the same as initializing from scratch, sometimes not; in the first case it is better to write an appropriate `shared-initialize` method; in the second case, the user must write specific methods for `reinitialize-instance`.

- `change-class`: called by the user when the class of an instance is *changed*; this function asks for a kind of conversion. Normally this function calls the generic function `update-instance-for-different-class` and the user may write methods for the latter. A simple example follows.

- `defclass`: if a `defclass` is in fact a *redefinition* of the class, each time an instance of the old version is considered, the generic function `update-instance-for-redefined-class` is applied, allowing in principle to coherently continue the work.

- `make-instance`: the user can also entirely redefine the creation and initialization process.

Writing appropriate methods for these generic functions, the user can freely define the initialization work, clearly distinguishing the various levels from each other; if sufficient, the programmer may partially or totally use the *standard style* without any further work. You understand that combining these methods with the user-defined class hierarchy, combining also with the various possibilities of *auxiliary methods* (`:before`, `:after`, `:around`), the user has a fantastically large workspace. However, once this structure is reasonably understood, it is not difficult to be used, and quickly very powerful.

2.4 The example of "change-class".

The various situations that are met in the artificial simple example of this section around the `change-class` generic function allow to easily understand the general CLOS organisation of the initialization work.

```
> (DEFCLASS E12 ()
    ((sl1 :initarg :sl1 :reader sl1)
     (sl2 :initarg :sl2 :reader sl2))) ✠
#<STANDARD-CLASS E12>
> (DEFCLASS E13 ()
    ((sl1 :initarg :sl1 :reader sl1)
     (sl3 :initarg :sl3 :reader sl3))) ✠
#<STANDARD-CLASS E13>
> (setf ins (make-instance 'e12 :sl1 1 :sl2 2)) ✠
#<E12 @ #x20bc46b2>
> (change-class ins 'e13) ✠
#<E13 @ #x20c95952>
> ins ✠
#<E13 @ #x20c95952>
> (sl1 ins) ✠
1
> (sl3 ins) ✠
Error: The slot SL3 is unbound in the object #<E13 @ #x20c95952>
       of class #<STANDARD-CLASS E13>.
 [condition type: UNBOUND-SLOT]
```

No relation at all between both classes `E12` and `E13`; there is a common slot *name*, namely `sl1`, but this does not imply any relation between both classes. With an exception, if ever the class of an instance is changed from `E12` to `E13`, this common slot name will be considered by the standard `change-class` which in fact calls the standard `update-instance-for-different-class`: this method transmits the slots with the same *names* in both classes, gives up the other slots in the source instance, and lets unbound the other slots of the target instance. Here, the `sl1` slot has been kept, the `sl2` slot has been given up, and the `sl3` slot in the result is let unbound, which implies the error when this slot is read by the `:reader` method `sl3`. Note also the pseudo-new instance is at the same machine address, but the small invisible steps of the garbage-collector continually modify the actual addresses, which is made obvious by the value of the symbol `ins` locating our *unique* but *variable* instance.

Let us convert again our instance, but with an additional keyword argument.

```
> (change-class ins 'e12 :sl2 22) ✠
#<E12 @ #x20b55342>
> (sl2 ins) ✠
22
```

The default methods are sufficient to pass initial arguments to `change-class` to obtain partial initializations of the new *version* of the *same* instance. Let us consider now the situation where for some strange "theoretical" reason, when the class is changed from `E12` to `E13`, the value of the `sl3` slot must be computed as the product of the `sl2` slot of the source instance multiplied by some number included in the `change-class` data. This is obtained as follows.

```
> (DEFMETHOD UPDATE-INSTANCE-FOR-DIFFERENT-CLASS :after
      ((source e12) (target e13) &key (multiplier 1))
    (declare (type number multiplier))
    (setf (slot-value target 'sl3)
      (* (sl2 source) multiplier))) ✠
#<STANDARD-METHOD UPDATE-INSTANCE-FOR-DIFFERENT-CLASS :AFTER (E12 E13)>
> (change-class ins 'e13 :multiplier 4) ✠
#<E13 @ #x20b588e2>
> (list (sl1 ins) (sl3 ins)) ✠
(1 88)
```

Because we have used an `:after` method, the standard *primary* method is firstly called, solving the `sl1` transfer. And after this is done, our auxiliary method works. Note in particular how elegant is the handling of the extra multiplier argument; because this keyword argument is used in our method, it becomes available for the user when a `change-class E12` $\rightarrow$ `E13` happens.

We want finally to consider the situation where a conversion `E13` $\rightarrow$ `E12` must be installed, where the new version of the instance must have both slots redefined to zero. In this case the old version of the instance is useless, so that we can directly obtain this conversion by a `change-class` method.

```
> (DEFMETHOD CHANGE-CLASS :after ((ins e13) (class (eql 'e12)) &rest rest)
    (declare (ignore rest))
    (setf (slot-value ins 'sl1) 0
          (slot-value ins 'sl2) 0)) ✠
#<STANDARD-METHOD CHANGE-CLASS :AFTER (E13 (EQL E12))>
> (change-class ins 'e12) ✠
#<E12 @ #x2069248a>
> (list (sl1 ins) (sl2 ins)) ✠
(0 0)
```

3 What typing is.

The small examples of the previous section were artificial, and we want to consider now a *plausible* situation. We want to use CLOS to implement a simple coherent *typing system*, allowing the user to simultaneously use arbitrary types and high levels of *functional programming*, both subjects being furthermore strongly dependent on each other.

What is typing? If a *mathematical* definition of typing is wished, many definitions are possible, and one of them is to be illustrated by a small CLOS program. Note this *new* typing system will be installed while the standard one remains alive in our environment.

Typing consists in giving the programmer the ability of *partial* descriptions of his algorithms. Let us call $\mathcal{A}$ the *universe* of all the machine objects, $\mathcal{A}$ for anything. Any algorithm can be viewed as a map $\mathcal{A} \rightarrow \mathcal{A}$ *not everywhere defined*. Typically the Euclid algorithm is defined on the set $\mathbb{N}_* \times \mathbb{N}_*$ of pairs of positive integers, and returns such an integer. The set $\mathbb{N}_*$ is a subset of $\mathcal{A}$, and the latter contains also the set $\mathcal{L}$ of lists of any length; among these lists, some of them are made of two positive integers; let us call $\mathcal{L}(\mathbb{N}_*, \mathbb{N}_*)$ the corresponding subtype. The type specification (signature) for the Euclid algorithm E is then $E : \mathcal{L}(\mathbb{N}_*, \mathbb{N}_*) \rightarrow \mathbb{N}_*$.

Speaking so, we have used a few subsets of $\mathcal{A}$. Such a simple example could imply some wrong ideas. On one hand, one could think two different types should be disjoint, but it is not the case for $\mathcal{L}$ and $\mathcal{L}(\mathbb{N}_*, \mathbb{N}_*)$. If not disjoint, one could then require one of the considered types is included in the other one; for example $\mathcal{L}(\mathbb{N}_*, \mathbb{N}_*) \subset \mathcal{L}$ and it seems sensible to organize the types as a large *oriented graph* describing more and more finely the various sets of objects the user works with. Simple typing systems usually are of this sort.

Another idea has a much larger scope; it consists in deciding a type is nothing but some subset of $\mathcal{A}$, where the *membership property* may be verified by an algorithm. In other words we start only with two predefined types, $\mathcal{A}$ and $\mathcal{B}$; the second one, the *Boolean* type, has only two objects, $\top$ and $\bot$, implemented in Lisp as the symbols `T` and `NIL`. It is then interesting to define a type as an algorithm $\mathcal{A} \rightarrow \mathcal{B}$ *everywhere* defined; in other words, the algorithm defining a type has the special signature $\mathcal{A} \rightarrow \mathcal{B}$.

This is sufficient in simple programming, but fails as soon as *functional programming* must be considered. In fact we meet the Russel paradox or if you prefer the Gödel theorem. Let us call $\mathcal{T}$ the type of... types, that is the set of functional objects $\alpha : \mathcal{A} \rightarrow \mathcal{B}$. Then no algorithm can verify the membership of $\mathcal{T}$! Let us assume τ is such an algorithm; therefore $\tau : \mathcal{A} \rightarrow \mathcal{B}$ is everywhere defined and $\tau(\alpha) = \top$ if and only if α defines a type, that is, $\alpha : \mathcal{A} \rightarrow \mathcal{B}$ is also everywhere defined. Then one could design the subtype $\mathcal{T}'$ made of the algorithms $\alpha \in \mathcal{T}$ such that $\alpha(\alpha) = \bot$, in other words the type of "typing algorithms" α that are *not* "element" of the type associated with α. It would be easy to write down the algorithm τ' corresponding to the new type $\mathcal{T}'$:

$$\tau'(\alpha) \quad = \quad \texttt{if } \tau(\alpha) \;\; \texttt{then not } \alpha(\alpha) \\ \texttt{else } \bot$$

The algorithm τ' firstly examines whether its argument α is an algorithm $\mathcal{A} \rightarrow \mathcal{B}$; if yes, the algorithm α may work on any object, in particular on itself, and τ' returns the opposite of $\alpha(\alpha)$; otherwise the answer is negative. Once the algorithm $\tau : \mathcal{A} \rightarrow \mathcal{B}$ is available, then the object $\tau' : \mathcal{A} \rightarrow \mathcal{B}$ is available too, but Cantor and Russel remarked there is no possible answer for $\tau'(\tau')$: the answer of $\tau(\tau')$ should certainly be $\top$, so that we obtain the contradictory relation $\tau'(\tau') = \texttt{not } \tau'(\tau')$. Therefore the algorithm τ may not exist.

The traditional (pseudo-) solution for this difficult problem consists in *delaying* the examination of the type of functional objects. If an object α is claimed to be an algorithm $\mathcal{T}_1 \to \mathcal{T}_2$, this cannot be verified by a *general* algorithm ; instead, we must *wait* for an actual work of the algorithm α: if the computation of $\alpha(\omega)$ is asked for, some process can *then* verify the argument ω is really in the type $\mathcal{T}_1$; if yes the computation of $\alpha(\omega)$ is started and the result ω' is examined in turn, verifying the relation $\omega' \in \mathcal{T}_2$. In other words it is not possible to verify the type of a functional object *in general*; the only possible verification consists, each time the algorithm works, in verifying that the argument and the result have the correct type; this "verification" is not at all complete, it can be applied only to a finite number of calls of the algorithm α.

This organization must be recursive: the source and/or target types could in turn be functional, so that the verifications $\omega \in \mathcal{T}_1$ and $\omega' \in \mathcal{T}_2$ maybe must also be "delayed" with the meaning just explained. In particular if ω and ω' are both functional, then the functional object ω *will* probably *be* used when the object ω' will work. It is only at this time the correctness of the claimed types for ω, at least for this particular use, may be verified. Examples of this sort are showed in this paper.

In this way, when a whole program is executed, all the *particular uses* of the functional objects imply that the type of arguments and result are verified, so that when the program is finished, for all the invocations of functional objects, types of input and output are confirmed. Thinking a little about this situation finally leads to the following conclusion: as far as they have been used, the type rules about functional objects have been satisfied *for this specific run*. A quite satisfactory conclusion.

To illustrate the Common Lisp object system, we take as exercise subject the implementation of such an organization in Common Lisp, using the main components of CLOS.

3.1 Classes as structured types.

A *class* is firstly a structured type. Each *instance* (element) of a *standard* class has several *slots* (components, fields, members), and each slot has various properties described in the definition of the class. The *builtin* classes, corresponding to old classical types (integers, symbols, ...), are not standard.

The classes are organized in a hierarchical way: a class may be or not a *subclass* of another one, and this defines an *order* between the defined classes. This order must be coherent but in general it is not *total*, that is, for two classes $\mathcal{C}_1$ and $\mathcal{C}_2$, they may or may not be compared. But any class is a subclass of the maximal type $\mathcal{A}$, which contains any object, in particular the corresponding class object itself; to prove this point, we assign this class object to the symbol `universe` and verify the object so located is of the type so described; this maximal class $\mathcal{A}$ corresponds to the set (type) $\mathcal{A}$ of the previous section.

```
> (setf universe (find-class 't)) ✠
#<BUILT-IN-CLASS T>
```

```
> (typep universe universe) ✠
T
```

The *types* of the proposed organization for typing in this paper are pointed out by the letters `TP` only. This is necessary because we have to simultaneously work with three typing systems:

1. The old one of Common Lisp, still present in our workspace; we call it the *type-system*;
2. The *class-system* which is the main constituent of CLOS;
3. The theoretical proposal of our exercise; it is called the `TP`-*system* in this text.

3.2 The TP-class.

Each `TP`-type is described by an instance of the `TP` class now to be defined. We define this class as follows:

```
> (DEFCLASS TP ()
    ((name :type symbol :initarg :name :initform (gensym) :reader name))
    (:metaclass funcallable-standard-class)) ✠
#<FUNCALLABLE-STANDARD-CLASS TP>
```

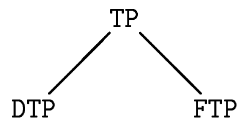
At this time, only one slot called `name` is defined for an instance of `TP`. Four *slot-options* have been used with the following meanings:

- The `:type` option explains the `name` slot must contain a `symbol`;
- The `:initarg` option says the `name` slot may be initialized through the keyword `:name`;
- The `:initform` option says that if the `name` slot is not otherwise initialized, it must be automatically initialized by the `gensym` function, a predefined Lisp function which creates from scratch a certainly *new* symbol;
- Finally the `:reader` option asks CLOS to construct a *method* named `name` allowing the user to call this method to obtain the value of the slot.

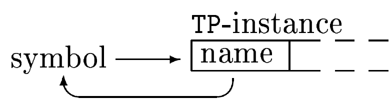
The `:metaclass` class option is explained later.

A *type descriptor* of the `TP` system is always an instance of the `TP`-class. The corresponding type is functional or not; if functional the type descriptor is in fact an instance of the `FTP` class, a subclass of the `TP` class, else the type descriptor is an instance of the `DTP` class (discriminant type), another subclass of the `TP` class. The situation is here particularly simple: a descriptor of type is *always* a `TP` instance, and in fact always an instance of only one subclass, either `FTP`, or `DTP`. Much more complicated situations between classes and subclasses can be designed under

CLOS, but this simple situation allows us to show the main points of the nature of CLOS. In other words, the class diagram is this one.



The type descriptor is firstly some `TP` instance, but it is convenient in this situation to locate the descriptor through a symbol and this is the reason of the `name` slot: this slot contains the symbol which in principle locates the `TP`-type descriptor. So that we have two (almost) "symmetric" pointers: the symbol points to the descriptor and the `name` slot of this descriptor points to this symbol; this is nothing but an *explicit* C++-`this` method. If ever the user is not really concerned by the symbol, the `gensym` will automatically generate a symbol for coherency.



The method `name` may now be applied to a `TP`-descriptor to obtain the associate symbol.

We have explained in the previous section the role of the `print-object` generic function. Here, our strategy consists in simply locating `TP`-types through symbols to be considered as *labels*, so that it is natural to display such a type by the associate symbol.

```
> (DEFMETHOD PRINT-OBJECT ((tp tp) stream)
    (format stream
        "#<~S ~S>"
      (class-name (class-of tp)) (name tp))) ✠
#<STANDARD-METHOD PRINT-OBJECT (TP T)>
```

Which is finally displayed through the (implicit) call of this method could be for example `#<DTP INTEGER>`; it is a tradition in Lisp to begin the display of a *non-readable* object by '`#<`'; this notion is carefully defined in ANSI Common-Lisp, but it is not the subject of this text. The value of `(name tp)` is the associate *name* of the type-descriptor; the value of `(class-of tp)` is the `class` of `tp`, therefore the *class-object* `DTP` or `FTP`, and the `class-name` is the symbol naming this class. We do not want to explain here the technical details about the `format` Lisp function, close to the traditional `printf` C function, but fantastically more flexible.

It was explained a little earlier the symmetry property between a `TP`-type descriptor and the symbol locating it. It is a little painful for the user to manage the necessary pointers, but an appropriate method can be used to make automatic the process during the initialization stage.

```
> (DEFMETHOD INITIALIZE-INSTANCE :after ((tp tp) &rest rest)
    (declare (ignore rest))
    (set (name tp) tp)) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE :AFTER (TP)>
```

The `:after` *qualified* method shown above means that *after* the standard initialization process, something must be done: the symbol in the `name` slot must be bound to the `TP`-instance itself.

3.3 The DTP-class.

We define now the subclass `DTP` (discriminant type) of the class `TP`.

```
> (DEFCLASS DTP (TP)
    ((sub :type list :accessor sub)
     (sup :type list :accessor sup))
    (:metaclass funcallable-standard-class)) ✠
#<FUNCALLABLE-STANDARD-CLASS DTP>
```

In the first line, the pseudo-argument `TP` indicates the new class is a subclass of the class `TP`. Two new slots are defined, so that a `DTP`-instance will have three slots because of the `name` slot already defined for a `TP`-instance:

- The `sub` slot (sub-types) is a list of the types that are known equal or smaller than the one which is described by the `DTP`-instance.
- The `sup` slot (super-types) is a list of the types that are known equal or larger than the one which is described by the `DTP`-instance.

Note that `:accessor` methods (`sub` and `sup`) have been required for the corresponding slots; this means these methods will allow to *read* the corresponding slot but they can also be used to *write* it or *update* it. In fact, these slots must in general be modified after the creation, to maintain the coherence of the `TP`-system.

We create now the minimal and the maximal type objects of the `TP`-system, namely the `void` and `any` types. A special initialization work is done here because the general initialization process will assume these types are *already* defined in the environment.

```
> (MAKE-INSTANCE 'DTP :name 'void) ✠
#<DTP VOID>
> (SETF
     (sub void) '(void)
     (sup void) '(void any)) ✠
(VOID ANY)
```

```
> (SET-FUNCALLABLE-INSTANCE-FUNCTION void
    #'(lambda (obj)
        (declare (type t obj))
        (the boolean nil))) ✠
#<Interpreted Function (unnamed) @ #x20b9e882>
```

Firstly the `DTP`-instance to be assigned to the `void` symbol is created by the call of the generic function `make-instance`, then the `sub` and `sup` slots are defined, and finally, because of the `:metaclass` class option, it is possible to associate a functional object to this instance, namely the function which always return `nil`; this is nothing but the characteristic function of the void type in our environment. Example of use of this function:

```
> (funcall void 'anything) ✠
NIL
```

Because of the `funcall`, the functional object associate to the object pointed by the symbol `void` is called with the symbol `anything` as argument. Whatever is the argument, the answer is `nil`. We can verify the pointer symmetry between the `name` slot and the corresponding symbol:

```
> void ✠
#<DTP VOID>
> (name void) ✠
VOID
```

We do exactly the same work for the `any` type, without showing the corresponding part of the session, perfectly symmetric of the `void` work.

The general process of initialization of a `DTP`-instance can now be defined. Firstly we need an `add-relation` function, allowing us to *add* a new order relation to the environment, something like "`integer` $<$ `number`", *and all the consequent relations.* This function *updates* the `sub` and `sup` slots of the involved `DTP`-instances with the `union` Lisp function.

```
> (DEFMETHOD ADD-RELATION ((dtp1 symbol) (dtp2 symbol))
    (the list
      (with-slots (sub) (eval dtp1)
        (declare (type list sub))
      (with-slots (sup) (eval dtp2)
        (declare (type list sup))
        (dolist (item sub)
          (declare (type symbol item))
          (setf (sup (eval item))
            (union (sup (eval item)) sup)))
        (dolist (item sup)
          (declare (type symbol item))
          (setf (sub (eval item))
            (union (sub (eval item)) sub)))
        (list dtp1 dtp2))))) ✠
#<STANDARD-METHOD ADD-RELATION (SYMBOL SYMBOL)>
```

A new `:after` method can then be defined this time for the `DTP`-type. Again because this is an `:after` method, the defined process is *added* to the standard initialization process, in particular giving the allocation of the instance, the initialization of arguments through initargs and initforms.

```
> (DEFMETHOD INITIALIZE-INSTANCE :after ((dtp dtp)
                                         &key prdc
                                         (dsub '(void))
                                         (dsup '(any)))
    (declare
     (type (function (t) boolean) prdc)
     (type list dsub dsup))
    (with-slots (name sub sup) dtp
      (declare
       (type symbol name)
       (type list sub sup))
      (setf
         sub (union dsub (list name))
         sup (union dsup (list name)))
      (dolist (item dsub)
        (declare (type symbol item))
        (add-relation item name))
      (dolist (item dsup)
        (declare (type symbol item))
        (add-relation name item)))
    (set-funcallable-instance-function dtp prdc)) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE :AFTER (DTP)>
```

This method for the generic function `initialize-instance` uses not only the `dtp`-instance to be initialized, but also the keyword optional arguments `:prdc` (predicate), `dsub` (direct subtypes) and `dsup` (direct supertypes). This mechanism works as follows: if an `initialize-instance` method uses a keyword argument, this argument is available to the user when it creates an instance of the corresponding class; usually this argument is used for a small work to be done during the initialization stage. Artificial example:

```
> (defclass cc () ((slot :initarg :slot :reader slot))) ✠
#<STANDARD-CLASS CC>
> (defmethod initialize-instance :after ((cc cc) &key incslot)
    (incf (slot-value cc 'slot) (* 2 incslot))) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE :AFTER (CC)>
> (setf cc-instance (make-instance 'cc :slot 3 :incslot 5)) ✠
#<CC @ #x20cc5e3a>
> (slot cc-instance) ✠
13
```

The `:slot` argument, because it is an `:initarg`, is used to ordinarily initialize the unique slot named `slot` of a `c`-instance; but the `:after initialize-instance` method, then adds twice the other keyword argument `:incslot`.

For the initialization of a `dtp`-instance, the `:prdc` argument is used to define the predicate function defining the type, which is associated to the the `dtp`-instance and used when the `dtp`-instance is *funcalled*; the `:dsub` and `:dsup` arguments allow the user to give the list of *direct* sub- and super-types; the method then computes, with the help of `add-relation`, all the sub- and super-types. If not used, these arguments default to an obvious value.

Let us define the `TP`-version of the boolean type.

```
> (MAKE-INSTANCE 'DTP
    :name 'BOOLEAN
    :prdc #'(lambda (obj)
              (declare (type t obj))
              (the boolean
                (if (member obj '(nil t))
                    t nil)))) ✠
#<DTP BOOLEAN>
> (funcall boolean 't) ✠
T
> (funcall boolean 'true) ✠
NIL
```

We use now the standard number types `number`, `integer` and `fixnum` to give examples of use of the `:dsub` and `:dsup` arguments.

```
> (make-instance 'dtp :name 'number :prdc #'numberp) ✠
#<DTP NUMBER>
> (make-instance 'dtp :name 'fixnum
     :prdc #'(lambda (obj)
               (declare (type t obj))
               (the boolean
                 (typep obj 'fixnum)))
     :dsup '(number)) ✠
#<DTP FIXNUM>
> (make-instance 'dtp :name 'integer :prdc #'integerp
                      :dsub '(fixnum) :dsup '(number)) ✠
#<DTP INTEGER>
> (sup fixnum) ✠
(INTEGER FIXNUM NUMBER ANY)
> (funcall fixnum 3) ✠
T
> (funcall fixnum 3.3) ✠
NIL
> (funcall number 3.3) ✠
T
```

In particular the relations “`fixnum` $\leq$ `integer`” and “`integer` $\leq$ `number`” have implied “`fixnum` $\leq$ `number`”. Note in these examples, the `prdc` slots for `number` and `integer` have been defined through symbols, for example `#'numberp` points to the functional value of the symbol `numberp`, in this case a predefined Lisp function examining whether its argument is a number. On the contrary, the `prdc` slot of the `fixnum` `TP`-instance is a function constructed in the call of `make-instance`.

The user may also construct several new types, and *after*, explicitly using the add-relation, define the order relations between them.

```
> (make-instance 'dtp
    :name 't1
    :prdc #'(lambda (obj)
              (declare (type t obj))
              (the boolean (eql obj 1)))) ✠
#<DTP T1>
> (make-instance 'dtp
    :name 't12
    :prdc #'(lambda (obj)
              (declare (type t obj))
              (the boolean
                (if (member obj '(1 2))
                    t nil)))) ✠
#<DTP T12>
```

and in the same way the types T123 (three objects 1, 2 and 3) and T1234 (four objects 1, 2, 3 and 4) are constructed, without using the :dsub and :dsup arguments. Then the user can maintain the structure of his type set; the mapcar shows the list of the sup slots of the just defined types.

```
> (mapcar #'sup (list t1 t12 t123 t1234)) ✠
((T1 ANY) (T12 ANY) (T123 ANY) (T1234 ANY))
> (add-relation 't1 't12) ✠
(T1 T12)
> (mapcar #'sup (list t1 t12 t123 t1234)) ✠
((T1 T12 ANY) (T12 ANY) (T123 ANY) (T1234 ANY))
> (add-relation 't123 't1234) ✠
(T123 T1234)
> (mapcar #'sup (list t1 t12 t123 t1234)) ✠
((T1 T12 ANY) (T12 ANY) (T123 T1234 ANY) (T1234 ANY))
> (add-relation 't12 't123) ✠
(T12 T123)
> (mapcar #'sup (list t1 t12 t123 t1234)) ✠
((T12 T1 T123 T1234 ANY) (T12 T123 T1234 ANY) (T123 T1234 ANY) (T1234 ANY))
```

Which is striking in such a process is the fact that the order relations must be explained to the machine by the user, again a case of a necessary *intuitionistic* work! Of course it would be easy to design a simple generator of enumerative types which would itself determine the order relations that are satisfied, but, because of Cantor, Russel and Gödel, this is definitively impossible for general types.

3.4 The FTP and SF classes.

Now we process the FTP-class, devoted to the functional types. To simplify the presentation, we consider only the functions $\mathcal{T}_1 \to \mathcal{T}_2$, in other words the functions with exactly *one* argument. It is easy to prove this is theoretically sufficient, but practically it is not. The standard trick consists in considering a function with two arguments as a function defined for a list of two elements. The further technicalities for the case of an arbitrary number of arguments are far from the subject of this paper, so that we do not want to consider them[8].

The FTP-class is defined as follows:

```
> (DEFCLASS FTP (TP)
    ((sorc :type symbol :initarg :sorc :reader sorc)
     (trgt :type symbol :initarg :trgt :reader trgt))
    (:metaclass funcallable-standard-class)) ✠
#<FUNCALLABLE-STANDARD-CLASS FTP>
```

This time, two slots, `sorc` (source) and `trgt` (target), are added to the `name` slot of the underlying TP-structure; these slots must be two TP-types, in fact two symbols locating them, according to our organization. Note these source and target types can be discriminant or functional.

A functional object will be an instance of the class SF, for *safe-function*, safe because the types of argument and result are systematically verified when the function works:

```
> (DEFCLASS SF ()
    ((sorc :type symbol :initarg :sorc :reader sorc)
     (trgt :type symbol :initarg :trgt :reader trgt))
    (:metaclass funcallable-standard-class)) ✠
#<FUNCALLABLE-STANDARD-CLASS SF>
```

The definition of FTP and SF are almost the same! In the case of an FTP-instance, the associated function will examine whether some object is in this type. In the case of an SF-instance, the associated function will be the functional object itself; the slots `sorc` and `trgt` are then additional information about the correct types of argument and result.

To explain the main part of the work to be done now, let us consider a functional type $\mathcal{T}_1 \to \mathcal{T}_2$ and a particular function f declared to be $f : \mathcal{T}'_1 \to \mathcal{T}'_2$; in what cases, this particular f is in the functional type $\mathcal{T}_1 \to \mathcal{T}_2$? Only one solution: we must have the order relations: $\mathcal{T}_1 \leq \mathcal{T}'_1$ and $\mathcal{T}'_2 \leq \mathcal{T}_2$, and the reader understands now why we took care of these order relations. We therefore need a `subtypep` Lisp function to compare types. If both arguments are discriminant types, the answer is read in a `sup` slot; if both arguments are functional, the appropriate comparisons must be applied to the respective sources and targets; if the types are not of the same nature, one being discriminant and the other one being functional, the answer is certainly negative[9]. Lisp translation:

[8] Common Lisp is also there fantastically more advanced than the other current languages: the numerous types of argument transfer that are available give the user a great flexibility to manage the various cases; they are allowed only because of the great *mathematical* precision of the definition of the language structure.

[9] It would be easy to work only with (pseudo-) "discriminant" types, even for functional objects, but the organization chosen here makes more obvious the deep difference of nature between both kinds of types; in particular in this way, a DTP-type and an FTP-type can never be compared.

```
> (DEFUN SUBTYPE-P (tp1 tp2)
    (declare (type symbol tp1 tp2))
    (the boolean
      (etypecase (eval tp1)
        (dtp
         (etypecase (eval tp2)
           (dtp (if (member tp2 (sup (eval tp1)))
                    t nil))
           (ftp nil)))
        (ftp
         (etypecase (eval tp2)
           (dtp nil)
           (ftp (and (subtype-p (sorc (eval tp2)) (sorc (eval tp1)))
                     (subtype-p (trgt (eval tp1)) (trgt (eval tp2))))))))) ✠
SUBTYPE-P
```

We cannot try the `subtype-p` function if the initialization work for `FTP`-instances is not finished. Remember in general a `TP`-instance must also be *funcallable* to verify the type of any object. The standard initialization work is therefore completed as follows.

```
> (DEFMETHOD INITIALIZE-INSTANCE :after ((ftp ftp) &rest rest)
    (declare (ignore rest))
    (set-funcallable-instance-function ftp
       #'(lambda (obj)
           (declare (type t obj))
           (the boolean
             (and (typep obj 'sf)
                  (subtype-p (sorc ftp) (sorc obj))
                  (subtype-p (trgt obj) (trgt ftp))))))) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE :AFTER (FTP)>
```

This time the associated function is entirely deduced from the `sorc` and `trgt` slots, available *after* the standard initialization work. You see the *only* objects of type some `FTP`-descriptor are `SF`-instances, and the appropriate order relations must be verified between source, target and the considered functional type. Now we can create a few `FTP`-instances and do the obvious tests for `subtype-p`.

```
> (subtype-p 'fixnum 'integer) ✠
T
> (subtype-p 'fixnum 'fixnum) ✠
T
> (subtype-p 'integer 'fixnum) ✠
NIL
> (make-instance 'ftp
   :name 'fii
   :sorc 'integer
   :trgt 'integer) ✠
#<FTP FII>
```

```
> (make-instance ’ftp
   :name ’fnf
   :sorc ’number
   :trgt ’fixnum) ✠
#<FTP FNF>
> (subtype-p fii fixnum) ✠
NIL
> (subtype-p fii fixnum) ✠
NIL
> (subtype-p fii fnf) ✠
NIL
> (subtype-p fnf fii) ✠
T
```

There remains to define the initialization of the `SF`-instances and to make them work. An `SF`-instance is funcallable and the associate function will be the ordinary function the user intends to define, with the type verifications automatically added. Lisp translation:

```
> (DEFMETHOD INITIALIZE-INSTANCE :after ((sf sf) &key f)
    (declare (type (function (t) t) f))
    (set-funcallable-instance-function sf
      (with-slots (sorc trgt) sf
        #’(lambda (arg)
            (declare (type t arg))
            (unless (funcall (eval sorc) arg)
              (error "The argument ~S should be of type ~S."
                arg sorc))
            (the t
              (let ((rslt (funcall f arg)))
                (unless (funcall (eval trgt) rslt)
                  (error "The result ~S should be of type ~S."
                    rslt trgt))
                rslt)))))) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE :AFTER (SF)>
```

Note the extra `:f` initialization argument. The obvious test:

```
> (setf 2+ (make-instance ’sf
             :sorc ’integer
             :trgt ’integer
             :f #’(lambda (n) (+ n 2)))) ✠
#<SF @ #x20cb326a>
> (funcall 2+ 5) ✠
7
> (funcall 2+ 5.5) ✠
Error: The argument 5.5 should be of type INTEGER.
```

3.5 Improvements.

But it is a little painful to use the `make-instance` method with its keywords, so that we use another aspect of the Common Lisp environment, the *macro generator* to make automatic the appropriate call to `make-instance`. We detail the work for the creation of `SF`-instances.

```
> (DEFMACRO MAKE-SF (sorc trgt f)
    `(make-instance 'sf
       :sorc ,sorc :trgt ,trgt :f ,f)) ✠
MAKE-SF
> (macroexpand
     '(make-sf 'number 'integer #'(lambda (n) (+ n 5))))) ✠
(MAKE-INSTANCE 'SF :SORC 'NUMBER :TRGT 'INTEGER :F #'(LAMBDA (N) (+ N 5)))
> (setf 5+ (make-sf 'number 'integer
                   #'(lambda (n) (+ n 5)))) ✠
#<SF @ #x20bbf6da>
> (funcall 5+ 6) ✠
11
> (funcall 5+ 6.5) ✠
Error: The result 11.5 should be of type INTEGER.
```

The `defmacro` must be considered as defining a *source text translator* converting a call to `make-sf` into some `make-instance`. The translation mechanism is *tested* by using the `macroexpand` Lisp function. Then the `5+` function is really created and then used. This time the error is in the result type.

Constructing the `DTP` and `FTP` instances can be processed in the same way. We do not show the details.

To explain the coherence of this type system, we add another stage of verification: a discriminant type is essentially a function $\mathcal{A} \rightarrow \mathcal{B}$, so that the functional type of this function should be also verified. Let us solve this challenge. We slightly modify the `DTP`-initializer.

```
> (DEFMETHOD INITIALIZE-INSTANCE :after ((dtp dtp)
    ... ...
    (set-funcallable-instance-function dtp
      (make-sf 'any 'boolean prdc))) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE :AFTER (DTP)>
```

As an example, let us consider the frequent case of a user who forgets the positive answer of the `member` function is not the boolean `T`:

```
> (member 3 '(1 2 3 4 5)) ✠
(3 4 5)
```

Such a user could erroneously define an enumerative type as follows:

```
> (make-dtp 'colors
    #'(lambda (obj)
        (member obj '(red orange yellow green blue indigo violet)))) ✠
#<DTP COLORS>
> (funcall colors 'black) ✠
NIL
> (funcall colors 'blue) ✠
Error: The result (BLUE INDIGO VIOLET) should be of type BOOLEAN.
```

and you see the type error… during the type verification is detected and a clear message is displayed. This cannot be applied to the `boolean` type itself, because an infinite loop would be generated. You see also in such a case the functional object associated with an instance (`DTP`) is not purely functional but another (`SF`)-instance which in turn has an associated functional object.

3.6 The function `compose`.

It is traditional in discussions between Lisp and C++ or Java programmers to give the example of the `compose` function to make obvious how C++ is weak when *functional programming* is involved; see at [3] how the "official" solution in C++ is technically difficult. We give here a simple Lisp solution, furthermore at once valid for any data types.

Firstly we construct the `pair` and `any-sf` types[10]:

```
> (make-dtp 'pair
            #'(lambda (obj)
                (and (consp obj)
                     (consp (cdr obj))
                     (null (cddr obj))))) ✠
#<DTP PAIR>
> (make-dtp 'any-sf
            #'(lambda (obj)
                (typep obj 'sf))) ✠
#<FTP ANY-SF>
```

Then the `2-sf-that-may-be-composed` type may be defined[11].

[10] Exercise: a previous version of this paper had the claimed "subtle" definition of the `any-sf` type: `(make-ftp 'any-sf 'void 'any)`; this really defines a type, but the `compose` `SF`-instance would not have the `any-sf` type, why? Analyzing precisely the difference between both definitions for `any-sf` is quite interesting and shows that any *strict* typing system cannot be entirely satisfactory, some *freedom* must necessarily be given to the programmer ; it is exactly the point of view of the Common Lisp designers.

[11] Exercise: Design a `make-subdtp` constructor to make automatic the generation of the `:dsup` argument and using the predicate associated to the initial type.

```
> (make-dtp '2-sf-that-may-be-composed
    #'(lambda (obj)
        (and (funcall pair obj)
             (funcall any-sf (first obj))
             (funcall any-sf (second obj))
             (subtype-p (trgt (second obj)) (sorc (first obj)))))
   :dsup '(pair)) ✠
#<DTP 2-SF-THAT-MAY-BE-COMPOSED>
```

Now the `compose` functional object is constructed where the types at any level are verified.

```
> (setf compose
    (make-sf '2-sf-that-may-be-composed 'any-sf
             #'(lambda (arg)
                 (let ((sf2 (first arg))
                       (sf1 (second arg)))
                   (make-sf (sorc sf1) (trgt sf2)
                            #'(lambda (arg)
                                (funcall sf2 (funcall sf1 arg)))))))) ✠
#<SF @ #x20ba9602>
> (setf 4+ (funcall compose (list 2+ 2+))) ✠
#<SF @ #x20bae5a2>
> (funcall 4+ 5) ✠
9
> (funcall 4+ 5.5) ✠
Error: The argument 5.5 should be of type INTEGER.
```

More subtle example: the composed function is in the right type with respect the argument, but a *used* function is not.

```
> (setf 3/2+ (make-sf 'integer 'integer
                      #'(lambda (x) (+ x 3/2)))) ✠
#<SF @ #x20bca3ba>
> (setf 3+ (funcall compose (list 3/2+ 3/2+))) ✠
#<SF @ #x20bcff8a>
> (funcall 3+ 6) ✠
Error: The result 15/2 should be of type INTEGER.
```

The last type-error example; this time, the `compose` function itself observes its argument does not have the required type.

```
> (funcall compose (list compose compose)) ✠
Error: The argument (#<SF @ #x206b77e2> #<SF @ #x206b77e2>)
       should be of type 2-SF-THAT-MAY-BE-COMPOSED.
```

4 CLOS and Mathematical Structures.

We give a small example to explain how CLOS can easily be used to implement the classical *mathematical structures.* Most often, an object of some *type* in mathematics is a structure with several components, frequently of *functional* nature. In this small presentation, we consider the case of a user who wants to handle *sets*, *magmas*, *associative* magmas and *monoids* ; these simple particular cases are sufficient to understand how CLOS gives the right tools to process mathematical structures. In the following section, we will describe how these simple methods have been used in [4] to implement the main structures of *Algebraic Topology* such as *chain complexes*, *simplicial sets*, *simplicial groups*, *Hopf algebras*, and also the various *morphisms* between these objects.

4.1 Sets.

We define a `SET` class whose instances correspond to the *sets* of the classical *set theory.*

```
> (DEFCLASS SET ()
    ((name :type symbol :initarg :name :initform (gensym) :reader name)
     (prdcf :type function :initarg :prdcf :reader prdcf)
     (cmprf :type function :initarg :cmprf :reader cmprf))) ✠
#<STANDARD-CLASS SET>
```

In this organization, a set is made of three slots. The `name` slot has the same role as for the types of the previous section: it is only an auxiliary tool to locate easily the sets; the slot contains the symbol that locates the set, and we make automatic this organization:

```
> (DEFMETHOD SHARED-INITIALIZE :after ((set set) slot-names &rest rest)
    (declare (ignore slot-names rest))
    (set (name set) set)) ✠
#<STANDARD-METHOD SHARED-INITIALIZE :AFTER (SET T)>
```

The appropriate `print-object` method; when a *set* or an object of a subclass will be displayed, the output will show the corresponding class and the `name`:

```
> (DEFMETHOD PRINT-OBJECT ((set set) stream)
    (declare (type stream stream))
    (format stream "#<~S ~S>" (class-name (class-of set)) (name set))
    (the set set)) ✠
#<STANDARD-METHOD PRINT-OBJECT (SET T)>
```

The second slot of a set, namely the `prdcf` slot (predicate-function), contains a function which can be called to examine whether some *arbitrary* object is an element of the set. To use easily this slot, we define two functions, `owns` and `in`; the first one may answer yes or no, that is, `t` or `nil`; and the second one generates an *error* if the membership relation is not satisfied[12].

```
> (DEFUN OWNS (set obj)
    (declare
     (type set set)
     (type t obj))
    (the boolean
      (funcall (prdcf set) obj))) ✠
OWNS
> (DEFUN IN (set obj)
    (declare
     (type set set)
     (type t obj))
    (if (funcall (prdcf set) obj)
        obj
      (error "The object ~S is not in ~S."
        obj set))) ✠
IN
```

The explanation of the third slot `cmpr` is given a little later; we construct the set `N` of non-negative integers.

```
> (MAKE-INSTANCE 'set
    :name 'N
    :prdcf #'(lambda (obj)
               (declare (type t obj))
               (the boolean
                 (and (integerp obj)
                      (>= obj 0))))
    :cmprf #'=) ✠
#<SET N>
```

The symbol `N` locates the constructed set:

```
> N ✠
#<SET N>
```

and the function `owns` allows the user to examine the membership relation:

```
> (owns N +1) ✠
T
> (owns N -1) ✠
NIL
```

The `cmprf` slot is essential in this context. There are frequently strong differences between an element of a set *as thought by the mathematician* and its possible machine representation<u>s</u>, the *s* being important. The notion of *equality* in mathematics is "primitive" and rarely *logically* considered by mathematicians. On the

[12] The sections about our two main didactical examples, typing and mathematical categories, are entirely *independent*; but the lucid reader will observe the root class of the second application, namely the `set` class, can after all be also considered as a typing system in a mathematical context!

contrary, this notion is crucial in Computer Science, and Common Lisp is by far the most precise language from this point of view[13]. Here, we want to let the user freely decide how equality is defined between the elements of his sets, and this is the role of the `cmprf` slot (comparison function). For example, the comparison between elements in the set `N` is done by the Lisp predefined function `#'=`, in particular appropriate to compare integers. Again we define a function `cmpr` to easily use the `cmprf` slot of a set.

```
> (DEFUN CMPR (set elmn1 elmn2)
    (declare
     (type set set)
     (type t elmn1 elmn2))
    (the boolean
      (funcall (cmprf set)
               (in set elmn1) (in set elmn2)))) ✠
CMPR
```

Note the use of the `in` function to verify that the compared elements are really in the considered set. Now we can compare two elements of `N`.

```
> (cmpr N 4 9) ✠
NIL
> (cmpr N 4 -9) ✠
Error: The object -9 is not in #<SET N>.
```

Let us construct now the set `Z/5`, that is the set of *integers modulo 5*. In our context, the most elegant method to implement this set consists in admitting a representation of an element of `Z/5` by an *arbitrary* integer, possibly negative or `> 4`, and to define the comparison with the help of the `mod` function, a predefined Lisp function.

```
> (MAKE-INSTANCE 'SET
    :name 'Z/5
    :prdcf #'integerp
    :cmprf #'(lambda (elmn1 elmn2)
               (= 0 (mod (- elmn1 elmn2) 5)))) ✠
#<SET Z/5>
> (cmpr Z/5 4 9) ✠
T
```

This time the comparison between 4 and 9 is positive: these machine integers are *different* representations of the *same* mathematical object. You understand it is easy in this framework to define *quotient* sets.

[13]In particular all the standard predefined Lisp functions allow the user to *freely* define the equality relation to be used for every particular call.

4.2 Magmas.

A magma is a set provided with a law of composition without any particular required property. It is a set with an additional ingredient, a function able to work on two elements of the magma and returning another element, their composition according the composition law. It is natural in this context to define the `magma` *subclass* of the `set` class; any `magma` instance is a set with a further slot, the `lawf` slot.

```
> (DEFCLASS MAGMA (set)
    ((lawf :type function :initarg :lawf :reader lawf))) ✠
#<STANDARD-CLASS MAGMA>
```

Again a function is added to the environment allowing the user to easily refer the `lawf` slot of a magma.

```
> (DEFMETHOD LAW ((magma magma) &rest rest)
    (case (length rest)
      (1 (in magma (first rest)))
      (2 (in magma
             (funcall (lawf magma)
                      (in magma (first rest))
                      (in magma (second rest)))))
      (otherwise
       (error "Non-correct arguments in:~@
               (LAW ~S~{ ~S~})." magma rest)))) ✠
#<STANDARD-METHOD LAW (MAGMA)>
```

We intend in general to allow the user to refer the *law* defining a magma with an *arbitrary* number of elements, at least if this makes sense. In such a case, CLOS allows the user to define a generic function, here the `law` generic function, with one mandatory argument, called here `magma` and any number of other arguments put together in a list reachable through the symbol `rest`[14]. The ingredient `(magma magma)` in the parameter list has two meanings: the first `magma` names the first parameter, and the second `magma` explains the corresponding argument must be of *class* `magma`, otherwise the method is not applicable.

For a magma without any further claimed property, it is sensible to allow one argument, and then this argument is returned, or two arguments and then the *product* of these arguments according to the composition law is returned. Otherwise an error message is displayed. Note how the `in` function is used to verify the correct type of the arguments.

We have defined the *set* `N` and it is possible to transform it now into a *magma*. Note how the standard initialization process of an instance allows the user to obtain such a conversion without any further work.

[14]More precisely, the symbol `&rest` *marks* in the parameter list a new zone, in this case with a symbol locating the "other" arguments; the Lisp programmer almost always chooses the symbol `rest` to locate this list.

```
> (change-class N 'magma
                :lawf #'(lambda (n1 n2)
                          (* n1 (- n2 3)))) ✠
#<MAGMA N>
```

The *law* which is defined is $(n_1, n_2) \mapsto n_1(n_2 - 3)$. The obvious trials:

```
> (law N 4 5) ✠
8
> (law N 4 -5) ✠
Error: The object -5 is not in #<MAGMA N>.
> (law N 1 0) ✠
Error: The object -3 is not in #<MAGMA N>. ✠
> (law N 4) ✠
4
> (law N 4 5 6) ✠
Error: Non-correct arguments in:
(LAW #<MAGMA N> 4 5 6).
```

The constructed magma in fact is not correct; let us construct a classical correct one:

```
> (make-instance 'magma
    :name 'Q
    :prdcf #'rationalp
    :cmprf #'=
    :lawf #'+) ✠
#<MAGMA Q>
> (law Q 3/2 2/3) ✠
13/6
> (law Q 3/2 2/3 4/5) ✠
Error: Non-correct arguments in:
(LAW #<MAGMA Q> 3/2 2/3 4/5).
```

This is simply the rational set `Q` provided with the standard addition law. The last result is non-satisfactory: the law is *associative* and it should be possible to make operate this law on an arbitrary number of arguments. The solution consists in defining a new class `A-MAGMA` (associative magma). No new slot in this class: membership of the new class only means the object, some magma, has an associative law.

```
> (DEFCLASS A-MAGMA (magma) ()) ✠
#<STANDARD-CLASS A-MAGMA>
```

But this new class may be used to define a new method for the generic function `law`; in this new case, an arbitrary *positive* number of arguments can be used. A simple recursive process defines the new method from the old one: if the argument number is three or more, the computation is decomposed, otherwise the *next* method is called; the law is associative and this definition is coherent.

```
> (DEFMETHOD LAW ((a-magma a-magma) &rest rest)
    (if (> (length rest) 2)
        (law a-magma
             (in a-magma (first rest))
             (apply #'law a-magma (rest rest)))
      (call-next-method))) ✠
#<STANDARD-METHOD LAW (A-MAGMA)>
```

You see in particular how the very basic Lisp function `apply` allows to recall the same method with one argument removed. We inform now the environment that our magma `Q` is associative:

```
> (change-class Q 'a-magma) ✠
#<A-MAGMA Q>
```

which allows us to compute the composition in `Q` of an arbitrary number of elements. Because the `magma` method will eventually be called, a possible type fault is intercepted.

```
> (law Q 1/2 2/3 3/4 4/5) ✠
163/60
> (law Q 1/2 2/3 0.75 4/5) ✠
Error: The object 0.75 is not in #<A-MAGMA Q>.
```

4.3 Monoids.

And the process can be continued for ever and ever, allowing the user to enrich as far as necessary the considered mathematical structures. Here, the following step consists in considering the structure of *monoids*. A monoid is an associative magma with a unit. Only a `unit` slot is to be added to the `a-magma` structure. It is impossible in general to verify the claimed *unit* e satisfies the required property, but however the property $e * e = e$ can be tested; doing this test through a `shared-initialize` method allows it to be used in a `make-instance`, or in a `change-class`, or in a `reinitialize-instance` as well. The membership of the monoid may be also tested.

```
> (DEFCLASS MONOID (a-magma)
    ((unit :initarg :unit :reader unit))) ✠
#<STANDARD-CLASS MONOID>
> (DEFMETHOD SHARED-INITIALIZE :after ((monoid monoid) slot-names &rest rest)
    (let ((unit (unit monoid)))
      (unless (owns monoid unit)
        (error "Sorry, the claimed unit ~S is not in ~S."
          unit monoid))
      (unless (cmpr monoid unit (law monoid unit unit))
        (error "Sorry, ~S does not look like a unit in ~S."
          unit monoid)))) ✠
#<STANDARD-METHOD SHARED-INITIALIZE :AFTER (MONOID T)>
```

```
> (change-class Q 'monoid :unit 1) ✠
Error: Sorry, 1 does not look like a unit in #<MONOID Q>.
> (reinitialize-instance Q :unit 0) ✠
#<MONOID Q>
```

Note the error about the unit is intercepted *after* the standard initialization process, so that our `Q` is then become a `monoid`, as observed in the error message, but a wrong monoid; it is natural in this situation to use `reinitialize-instance` to redefine the object. A little more sophisticated use of `:around` methods would allow us to verify the coherence `before` changing class, and to refuse it if detected non-coherent.

It is interesting to convert `Z/5`, currently a set, into a monoid with the unit `10`, why not.

```
> (change-class Z/5 'monoid
                :lawf #'+
                :unit 10) ✠
#<MONOID Z/5>
> (cmpr Z/5 10 (law Z/5 10 10)) ✠
T
```

In a monoid, the composition law may logically be called for *zero* argument; in this case, the *unit* is the result. A new method for the generic function `law` is therefore stacked.

```
> (DEFMETHOD LAW ((monoid monoid) &rest rest)
    (if (= 0 (length rest))
        (unit monoid)
      (call-next-method))) ✠
#<STANDARD-METHOD LAW (MONOID)>
> (list (law Q) (law Z/5)) ✠
(0 10)
```

This looks a little artificial because `(law Q)` and `(unit Q)` are in fact synonymous, but think of the frequent case where you have to process a statement like `(apply #'law Q some-list)` where the last argument is a *computed* list, which could be sometimes empty; with our new `monoid` method for the `law` generic function, even if the value of `some-list` is the empty list, the correct process is applied.

4.4 What about the morphisms?

So far we have only worked with the *objects* of the mathematical categories of sets, magmas, associative magmas, monoids. What about the *morphisms*? The solution is easy, but needs a little lucidity about the usual *compatibility* properties with ambient structures. We only sketch a possible solution, the expansion of which being obvious.

The root of our classes of morphisms is the class of morphisms between sets.

```
> (DEFCLASS SET-MRPH ()
    ((sorc :type set :initarg :sorc :reader sorc)
     (trgt :type set :initarg :trgt :reader trgt)
     (f :type function :initarg :f :reader f))) ✠
#<STANDARD-CLASS SET-MRPH>
```

A `set-mrph` instance is essentially a *source*, some set, a *target*, some set, and a function mapping any object of the source to an object of the target. Again the standard *functional* properties of Common Lisp give the developper the right environment. We could also hide the `f` slot into the special hidden functional slot of a *funcallable* instance, as we did for the `TP`-instances in Section 3; it is only a question of taste.

We need something to be able to "funcall" a `set-mrph`:

```
> (DEFUN ? (mrph elmn)
    (...
     code which:
     1) verifies elmn is in the source of mrph;
     2) applies the f slot of mrph to elmn to compute the image;
     3) verifies this image is in the target of mrph;
     4) returns the image.
     )) ✠
?
> (? some-set-mrph some-source-element) ✠
...
```

Now a magma-morphism is nothing but a set-morphism satisfying the standard compatibility properties with the magma structures of source and target. So that:

```
> (DEFCLASS MAGMA-MRPH (set-mrph) ()) ✠
#<STANDARD-CLASS MAGMA-MRPH>
```

and the same for `a-magma-mrph`, `monoid-mrph` and so on. Note this organization allows the user to clearly distinguish a *set*-morphism between magmas from a *magma*-morphism between the same magmas; in the first case the compatibility properties are not satisfied or maybe only non-required; in the second case the compatibility properties are assumed satisfied. Such morphisms are intensively used in the Kenzo program and considered in the following section.

4.5 What about functors?

Working with categories, the mathematician inevitably will have to work with functors. In this organisation a tower of compatible functors is nothing but *one* generic function with various methods corresponding to the different cases. For example the *cartesian product* functor is defined for sets, magmas, associative magmas, monoids, and many other categories. It is sufficient to implement *all these functors* in *one* generic function, as follows.

```
> (DEFMETHOD PRODUCT ((set1 set) (set2 set))
    (make-instance 'set
      :name (intern (format nil "~S-PRDC-~S" (name set1) (name set2)))
      :prdcf #'(lambda (obj)
                 (declare (type t obj))
                 (the boolean
                   (and (listp obj)
                        (= 2 (length obj))
                        (owns set1 (first obj))
                        (owns set2 (second obj)))))
      :cmprf #'(lambda (pair1 pair2)
                 (declare (type list pair1 pair2))
                 (the boolean
                   (and (cmpr set1 (first pair1) (first pair2))
                        (cmpr set2 (second pair1) (second pair2))))))) ✠
#<STANDARD-METHOD PRODUCT (SET SET)>
> (product N Z/5) ✠
#<SET N-PRDC-Z/5>
> (cmpr N-prdc-Z/5 '(4 5) '(9 5)) ✠
NIL
> (cmpr N-prdc-Z/5 '(4 5) '(4 10)) ✠
T
> (cmpr N-prdc-Z/5 '(4 5) '(-4 10)) ✠
Error: The object (-4 10) is not in #<SET N-PRDC-Z/5>.
```

You see how clear and natural is the code of the `product` method. It is only a question of constructing the appropriate `prdcf` and `cmprf` slots, some functions, from the corresponding slots of the arguments `set1` and `set2`; the process is always the same, so that the Lisp *closures*, a subtle notion not available in C++ or Java[15], give the user the natural tool, even he does not know exactly what a closure is!

For the richer structures of `magma`, `a-magma` and `monoid`, it is sufficient to write down new specific methods for the *same* generic function, defining the additional work to be done, using the previous work thanks to `call-next-method`.

```
> (DEFMETHOD PRODUCT ((magma1 magma) (magma2 magma))
    (change-class (call-next-method) 'magma
                  :lawf #'(lambda (pair1 pair2)
                            (list
                             (law magma1 (first pair1) (first pair2))
                             (law magma2 (second pair1) (second pair2)))))) ✠
#<STANDARD-METHOD PRODUCT (MAGMA MAGMA)>
> (DEFMETHOD PRODUCT ((magma1 a-magma) (magma2 a-magma))
    (change-class (call-next-method) 'a-magma)) ✠
#<STANDARD-METHOD PRODUCT (A-MAGMA A-MAGMA)>
> (DEFMETHOD PRODUCT ((monoid1 monoid) (monoid2 monoid))
    (change-class (call-next-method) 'monoid
                  :unit (list (unit monoid1) (unit monoid2)))) ✠
#<STANDARD-METHOD PRODUCT (MONOID MONOID)>
```

[15] available in Maple since the Release 5, but no OOP in Maple...

```
> (SETF Q2 (product Q Q)) ✠
#<MONOID Q-PRDC-Q>
> (SETF Q4 (product Q2 Q2)) ✠
#<MONOID Q-PRDC-Q-PRDC-Q-PRDC-Q>
> (law Q4 '((1/2 2/3) (3/4 4/5))
          '((5/6 6/7) (7/8 8/9))) ✠
((4/3 32/21) (13/8 76/45))
> (unit Q4) ✠
((0 0) (0 0))
```

Note also the applicability rule for methods of a generic function implies the product of a monoid by a magma, for example, will fortunately be a magma.

```
> (product Q N) ✠
#<MAGMA Q-PRDC-N>
```

5 CLOS and the Kenzo program.

The Kenzo program is the first significant *machine program* about classical Algebraic Topology. It is not only a program implementing various *known* algorithms; *new* methods have been developped to *transform* the main "tools" of Algebraic Topology, mainly the spectral sequences, not at all *algorithmic* in the traditionnal organisation, into actual *computing* methods.

5.1 An example of Kenzo work.

Let us show a simple example to illustrate which is possible with this program. The homology group $H_5\Omega^3\text{Moore}(\mathbb{Z}_2, 4)$[16] is "in principle" reachable thanks to old methods, see [2], but experience shows even the most skilful topologists meet some difficulties to determine it, see [7, 9]. With the Kenzo program, you construct the Moore space.

```
> (setf m4 (moore 2 4)) ✠
[K1 Simplicial-Set]
```

The program returns the Kenzo-object #1, a simplicial set, that is, a combinatorial version of the Moore space which is asked for, and this object is assigned to the symbol `m4`. Then you construct the third loop-space of this Moore space.

```
> (setf o3m4 (loop-space m4 3)) ✠
[K15 Simplicial-Group]
```

[16]The space $\text{Moore}(\mathbb{Z}_2, 4)$ is a "canonical" space having only non-trivial homology in dimension 4, namely $\mathbb{Z}_2$, and $\Omega^3\text{Moore}(\mathbb{Z}_2, 4)$, its third loop space, is the space of continuous maps from the 3-sphere S^3 to this Moore space; the challenge is to determine the fifth homology group of this functional space.

The combinatorial version of the loop space is *highly* infinite: it is a combinatorial version of the space of *continuous* maps $S^3 \to \text{Moore}(\mathbb{Z}_2, 4)$ but functionnally coded as a small set of functions in a simplicial-group object, that is, a simplicial set with an added group structure compatible with the simplicial structure. Finally the fifth homology-group is asked for.

```
> (homology o3m4 5) ✠
Homology in dimension 5 :
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
Component Z/2Z
---done---
```

and the result $H_5\Omega^3\text{Moore}(\mathbb{Z}_2, 4) = \mathbb{Z}_2^5$ is obtained in 1m30s with a PC 400MHZ. In natural situations a little more complicated, the Kenzo program has already computed new homology groups unreachable so far with "classical" Algebraic Topology, even from a theoretical point of view.

5.2 Kenzo classes.

Figure 2 shows the class diagram of Kenzo objects. The situation is to be compared with which was explained in Section 4 about the most elementary mathematical structures. The lefthand part of the class diagram is made of the main mathematical categories that are used in combinatorial Algebraic Topology. A *chain complex* is a graded differential module; an *algebra* is a chain complex with a compatible multiplicative structure, the same for a *coalgebra* but with a comultiplicative[17] structure. If a multiplicative and a comultiplicative structures are added and if they are compatible with each other in a natural sense, then it is a *Hopf algebra*, and so on.

The hopf-algebra and simplicial-group classes are typical cases where a *multi-heritage* situation is met; we show the *actual* Kenzo definitions of these classes.

```
(DEFCLASS HOPF-ALGEBRA (coalgebra algebra)
  ())

(DEFCLASS SIMPLICIAL-GROUP (kan hopf-algebra)
   ((grml :type simplicial-mrph :reader grml1)
    (grin :type simplicial-mrph :reader grin1)))
```

You see the definition of the hopf-algebra class is particularly striking; it explains that a Hopf-algebra is nothing but an algebra *and* a coalgebra; the compatibility conditions between both structures *cannot* be verified by a program and they necessarily depend on the programmer's "lucidity". In the same way, a *simpli-*

[17]That is, some *co*operator $A \to A \otimes A$.

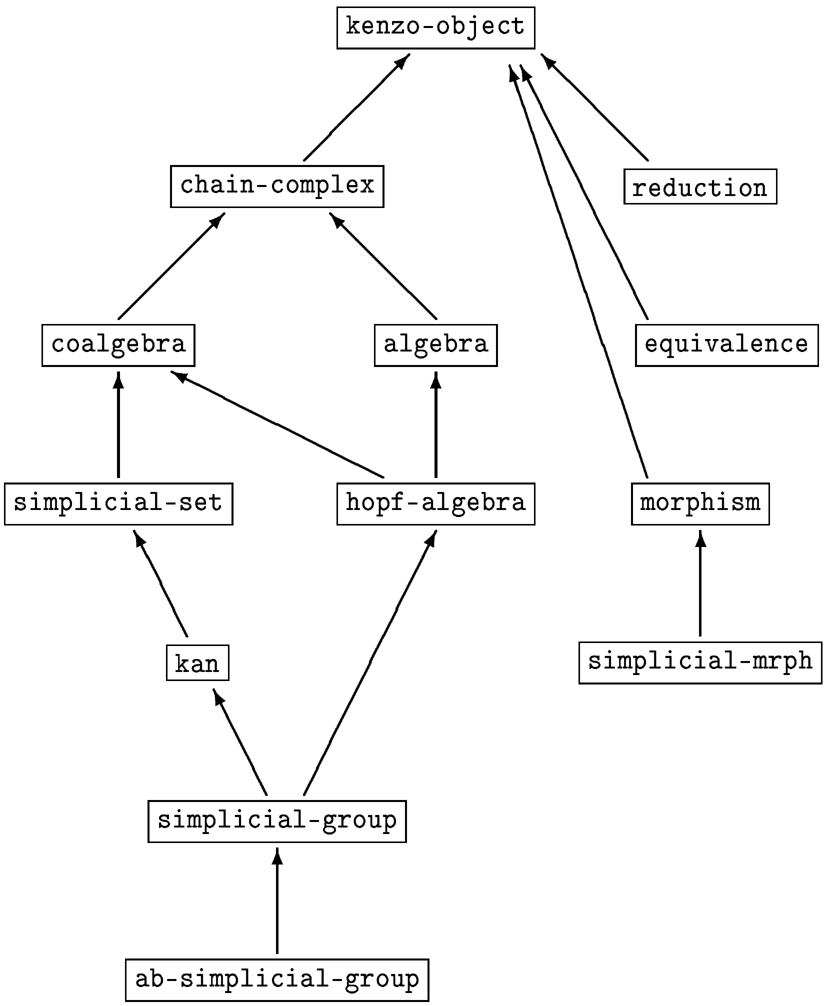
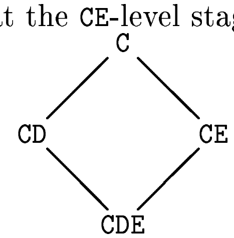


Figure 2: The Kenzo class diagram.

cial group is a kan object and a hopf-algebra object sharing some common data, namely a coalgebra structure, with two further slots, grml (group multiplication) and grin (group inversion), those slots being some simplicial morphisms.

In such a multi-heritage situation, it is important the call-next-method function works as hoped-for. Look at this artificial situation just to show the process; The C class has two subclasses CD and CE, which have in common the subclass CDE; the artificial initialize-instance methods let you verify that call-next-method *remembers its story* when deciding what really the *next method* must be. Here, when processing the CD-level, call-next-method "remembers" the process was initiated from the CDE-level, so that the CE-level stage is not forgotten.




```
> (defclass C () ()) ✠
#<STANDARD-CLASS C>
> (defclass CD (C) ()) ✠
#<STANDARD-CLASS CD>
> (defclass CE (C) ()) ✠
#<STANDARD-CLASS CE>
> (defclass CDE (CD CE) ()) ✠
#<STANDARD-CLASS CDE>
> (defmethod initialize-instance ((c c) &rest rest)
    (print "C-initialization")) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE (C)>
> (defmethod initialize-instance ((cd cd) &rest rest)
    (print "beginning CD-initialization")
    (call-next-method)
    (print "finishing CD-initialization")) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE (CD)>
> (defmethod initialize-instance ((ce ce) &rest rest)
    (print "beginning CE-initialization")
    (call-next-method)
    (print "finishing CE-initialization")) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE (CE)>
> (defmethod initialize-instance ((cde cde) &rest rest)
    (print "beginning CDE-initialization")
    (call-next-method)
    (print "finishing CDE-initialization")) ✠
#<STANDARD-METHOD INITIALIZE-INSTANCE (CDE)>
> (make-instance 'C) ✠
"C-initialization"
#<C @ #x212184da>
> (make-instance 'CD) ✠
"beginning CD-initialization"
"C-initialization"
"finishing CD-initialization"
#<CD @ #x21220e8a>
```

```
> (make-instance 'CE) ✠
"beginning CE-initialization"
"C-initialization"
"finishing CE-initialization"
#<CE @ #x2122698a>
> (make-instance 'CDE) ✠
"beginning CDE-initialization"
"beginning CD-initialization"
"beginning CE-initialization"  ←—←—←—!!!
"C-initialization"
"finishing CE-initialization"
"finishing CD-initialization"  ←—←—←—!!!
"finishing CDE-initialization"
#<CDE @ #x2122c03a>
```

And you may also play with the *auxiliary* `:before`, `:after` and `:around` methods to order as you like the various initialization steps. As a typical example, when the essential part of the initialization work of any `kenzo-object` is done, then the object is *finally* pushed in a list which is used later as explained in the next section. This is obtained as follows.

```
(DEFMETHOD INITIALIZE-INSTANCE :after ((k kenzo-object) &rest rest)
  (push k *k-list*))
```

In this way this is done if and only if the initialization work is successfully finished, even for the more specialized structures: if for example the specialized initialization work for a simplicial set fails and stops on error, then the pushing statement concerning the weakest structure is not run.

5.3 Optimizing computations.

The Kenzo program is certainly one of the most *functional* programs ever written down. It is frequent that several thousands of functions are present in memory, each one being *dynamically* defined from other ones, which in turn are defined from other ones, and so on. In this quite original situation, the same calculations are frequently *asked again*. To avoid repeating these calculations, it is better to store the results and to systematically examine for each calculation whether the result is already available.

Because of this situation, it is very important not to have *several copies* of the same function; otherwise it is impossible for one copy to guess some calculation has already been done by another copy. This is a very important question in this program, so that the following strategy has been used. Each Kenzo object has a rigorous *definition*, stored as a list in the `dfnt` slot of the object. This is the main reason of the top class `kenzo-object`: making easier this process. The actual definition of the `kenzo-object` class:

```
(DEFCLASS KENZO-OBJECT ()
  ((idnm :type fixnum :reader idnm)
   (dfnt :type list :reader dfnt)
   (prpr :type list :reader prpr)
   (cmmn :type list :reader cmmn)))
```

Then, when any `kenzo-object` is to be considered, its *definition* is constructed and the program firstly looks in `*k-list*` whether some object corresponding to this definition already exists; if yes, no `kenzo-object` is constructed, the already existing one is simply returned. Look at this small example where we construct the second loop space of S^3, then the first loop space, and then again the second loop space. In fact the initial construction of the second loop space required the first loop space, and examining the identification number `K??` of these objects shows that when the first loop space is later asked for, Kenzo is able to return the already existing one.

```
> (setf s3 (sphere 3)) ✠
[K372 Simplicial-Set]
> (setf o2s3 (loop-space s3 2)) ✠
[K380 Simplicial-Group]
> (setf os3 (loop-space s3 1)) ✠
[K374 Simplicial-Group]
> (setf o2s3-2 (loop-space s3 2)) ✠
[K380 Simplicial-Group]
> (eq o2s3 o2s3-2) ✠
T
```

The last statement shows the symbols `o2s3` and `o2s3-2` points to the same machine address. In this way we are sure any kenzo-object has no duplicate, so that the memory process for the values of numerous functions cannot miss an already computed result. Let us look some `dfnt` slots:

```
> (dfnt o2s3) ✠
(LOOP-SPACE [K374 Simplicial-Group])
> (dfnt (k 374)) ✠
(LOOP-SPACE [K372 Simplicial-Set])
> (dfnt (k 372)) ✠
(SPHERE 3)
```

You see in this way the history of the construction process can be freely examined by the user, which is important in the development stage.

5.4 Delaying initializations.

The complete structure of a Kenzo object is extremely complicated, and many components are often useless. Another CLOS feature is therefore used to avoid the maybe non-necessary initialization works. The following artificial example explains how this is possible; it is a kind of *autoloading* mechanism, elegant, easy to be used, and useful to avoid initializing needless slots. We assume a `C` class, where each `C` object has two slots, `sl1` and `sl2`; the first one is necessary, but the second one would be the result of a *complex* process here simulated as being 1000 times the value of the first one.

```
> (DEFCLASS F ()
    ((sl1 :type integer :initarg :sl1 :reader sl1)
     (sl2 :type integer :reader sl2))) ✠
#<STANDARD-CLASS F>
> (DEFMETHOD SLOT-UNBOUND (class (fi f) (slot-name (eql 'sl2)))
    (declare (ignore class))
    (setf (slot-value fi 'sl2) (* 1000 (sl1 fi)))
    (sl2 fi)) ✠
#<STANDARD-METHOD SLOT-UNBOUND (T F (EQL SL2))>
> (SETF FI (make-instance 'f :sl1 23)) ✠
#<F @ #x213a7b8a>
> (SLOT-BOUNDP fi 'sl2) ✠
NIL
> (sl2 fi) ✠
23000
> (SLOT-BOUNDP fi 'sl2) ✠
T
```

You see the generic function `slot-unbound` is available which is called by the error manager when a non-initialized slot is asked for. The standard process finally does generate an error. But the user can write specialized methods for this generic function, allowing him instead to initialize the missing slot by some process using the available information. You see the initialization process lets uninitialized the `sl2` slot of the `F`-instance located by `fi`, but when this slot is asked for, the "right" value is in fact returned! A new examination by `slot-boundp` shows the slot is now bound.

This process is extremely convenient to organize the data as a living object where each time some missing component is questionned, an automatic "repairing process" is started, computing the missing information. The process may be recursive, so that if, in the repairing process, some other datum is again missing, an other repairing process is recursively started, and so on.

This possibility is intensively used in the Kenzo program. Look at this small experience. Firstly we reinitialize the environment by `cat-init`. When the fourth loop space $\Omega^4 S^5$ is constructed, you see only 26 Kenzo objects are present in the environment. Then the homology group $H_2\Omega^4 S^5$ is asked for. The answer, $\mathbb{Z}_2$ is quickly obtained, but the number of present Kenzo objects is now 504; an enormous set of `slot-unbound` calls has generated the construction of 478 new Kenzo objects, necessary to do the calculation. Furthermore a `:before` method had been added just to count the number of `slot-unbound` calls, a convenient debugging trick; you see the homology calculation has recursively generated 240 `slot-unbound` calls.

```
> (cat-init) ✠
---done---
```

```
> (setf s5 (sphere 5)) ✠
[K1 Simplicial-Set]
> (setf o4s5 (loop-space s5 4)) ✠
[K21 Simplicial-Group]
> (length *k-list*) ✠
26
> (setf counter 0) ✠
0
> (defmethod slot-unbound :before (class instance slot)
    (declare (ignore class instance slot))
    (incf counter)) ✠
#<STANDARD-METHOD SLOT-UNBOUND :BEFORE (T T T)>
> (homology o4s5 2) ✠
Homology in dimension 2 :
Component Z/2Z
---done---
> (length *k-list*) ✠
504
> counter ✠
240
```

5.5 Mixing low level and high level programming.

Computing time is crucial for the applications of the Kenzo program. The complexity of the implemented algorithms is highly exponential, so that the developer must carefully consider how he can improve the computing time of the written down Lisp code. In particular, if the heart of the program may be written close to the machine language, large amounts of computing time can be saved. But conversely this must not penalize the *readability* and the *modularity* of the program.

Which is striking with the current version of Common Lisp is the possibility of easily mixing *low level* and *high level* programming. The features about OOP previously described in this paper show how Common Lisp is powerful in high level programming, allowing the user to directly handle the sophisticated objects of Algebraic Topology such as chain complexes, products and coproducts, Hopf algebras, simplicial sets and simplicial groups.

But on the other hand, the Kenzo program intensively uses the low level part of the Common Lisp language, that is, the quasi-assembler language which is the very root of the language, such as the popular (?) `car`, `cdr`, and `cons`. This is possible thanks to the Common Lisp *macrogenerator*, already mentioned Section 3.5. Let us consider the case of the type `absm`, that is, *abstract simplex.* These objects are really the most elementary consituents of the Kenzo geometric objects, and they are so intensively used, billions of times for every significant Kenzo run, that you *must not* use CLOS for these kernel structures. Kenzo defines the `absm` type as follows:

```
(DEFUN ABSM-P (object)
   (declare (type any object))
   (the boolean
      (and (consp object)
           (eq :absm (car object))
           (typep (cdr object) 'iabsm))))

(DEFTYPE ABSM () '(satisfies absm-p))
```

The `absm-p` function explains an `absm` is a `cons` (pair) where the lefthand component is the keyword `:absm` and the righthand one is an `iabsm`, that is, an *internal* `absm`; in the same way, elsewhere in the program, it is explained an `iabsm` is again a `cons` where the righthand component is anything and the lefthand component is a fixnum coding a *degeneracy operator*. Most of computations in Algebraic Topology are in fact low level computations about degeneracy operators where such an operator is a decreasing list of small integers, like `(5 2 0)`; because this list is *strictly* decreasing, it can be represented by the fixnum 37 because $37 = 2^5 + 2^2 + 2^0$, so that all the standard calculations about degeneracy operators become fine calculations *at the bit level* on binary fixnums. But Common Lisp has all the predefined functions to do such a job, so that the programmer can efficiently work according to this strategy. A considerable memory space is saved so and furthermore the calculations are much faster.

If a degeneracy operator is to be extracted from an `absm`, the `dgop` macro is used:

```
> (DEFMACRO DGOP (absm)
   `(the dgop (cadr (the cadr ,absm))) ✠
DGOP
> (macroexpand '(dgop argument)) ✠
(THE DGOP (CADR (THE ABSM ARGUMENT)))
```

which explains that in fact the call of `dgop` is synonymous with a call of the assembler-like `cadr`, but the types of argument and result are verified:

```
> (dgop (absm 37 'something)) ✠
37
> (dgop 'not-an-absm) ✠
Error: object "NOT-AN-ABSM" is not of type "ABSM".
[condition type: PROGRAM-ERROR]
```

When the program is compiled, the compiler firstly translates the source code when a macro call is found, so that it is an assembler-like statement which is compiled; furthermore an appropriate compiler option allows the compiled code to ignore or not the type verifications through the '`the`' statements. When the program is finalized for production work, of course these type verifications are discarded to save computing time. You see in this way the Lisp code is *readable*, this code being firstly translated in low level Lisp statements, therefore very efficiently compiled, without loosing if necessary the type verifications.

References

[1] ANSI Common-Lisp.
`www.xanalys.com/software_tools/reference/HyperSpec/`

[2] Gunnar Carlsson and R. James Milgram. *Stable homotopy and iterated loop spaces.* in [5], pp 505-583.

[3] `compose.cpp`.
`http://www.boost.org/libs/compose/compose.hpp.html`

[4] Xavier Dousson, Julio Rubio, Francis Sergeraert and Yvon Siret. *The Kenzo program.* `http://www-fourier.ujf-grenoble.fr/~sergerar/Kenzo/`

[5] *Handbook of Algebraic Topology* (Edited by I.M. James). North-Holland (1995).

[6] Gregor Kiczales, Jim des Rivières and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[7] Julio Rubio, Francis Sergeraert. *Constructive Algebraic Topology.* Talk at the EACA Conference of Tenerife, 1999. To appear in *Bulletin des Sciences Mathématiques.*

[8] Francis Sergeraert. *The computability problem in algebraic topology.* Advances in Mathematics, 1994, vol. 104, pp 1-29.

[9] Francis Sergeraert. $\maltese_k$*, objet du* 3^e *type.* Gazette des Mathématiciens, 2000, vol. 86, pp 29-45.

[10] Guy L. Steele Jr. *Common Lisp.* Digital Press, 1984.

[11] Guy L. Steele Jr. *Common Lisp, second edition.* Digital Press, 1990.