Galileo 97 Reference Manual. Version 2.0 [*]

A. Albano, G. Antognoni, G. Baratti, G. Ghelli, and R. Orsini*

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa – Italy
albano@di.unipi.it
*Università di Venezia
Corso di Laurea in Scienze dell'Informazione
Via Torino 155, 30170 Mestre – Italy

January 2000

Abstract

Galileo 97 is a statically–scoped functional conceptual language. Functions are first class values which can be passed as parameters, returned as values and embedded in data structures.

Galileo 97 is an interactive language. A session is a dialogue of questions and answers with the Galileo 97 system.

Galileo 97 is a statically and strongly typed language. Every expression has a type, which is determined statically. The type of an expression is usually automatically inferred by the system, without needing of type definitions. The Galileo 97 type system guarantees that any expression that can be typed will not generate type errors at run–time.

Galileo 97 has a rich collection of data types and type constructors. Moreover, the user can define new abstract data types, which are indistinguishable from system predefined types.

Galileo 97 has subtyping. If a type `T` is a subtype of a type `T'`, then a value of `T` can be used as an argument of any operation defined for values of `T'`, but not vice versa because the subtype relation is a partial order.

Galileo 97 has parametric polymorphism and bounded parametric polymorphism. Polymorphism is the ability of a function to handle values of many types.

Galileo 97 has control mechanism for exceptions and their handling.

Galileo 97 is a database programming language and supports the abstraction mechanisms of object databases.

This manual describes a main memory implementation of Galileo 97 running on Macintosh and PC-Windows.

[*]This work has been supported in part by grants from the C.E.C. under ESPRIT BRA No.6309 (FIDE2: Fully Integrated Data Environment), the Progetto finalizzato "Sistemi informatici e calcolo parallelo" of C.N.R. under grant No. 93.1502.PF69, and by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica" .

Contents

1 Introduction

Galileo 97 is an expression language: each construct is applied to values to return a value.

Galileo 97 is an interactive language: the system repeatedly prompts for inputs and reports the results of computations; this interaction is said to happen at the top level of evaluation. At the top level one can evaluate expressions or perform declarations. This feature allows the interactive use of Galileo 97 without a separate query language.

Galileo 97 is an higher order language, in that functions are denotable and expressible values of the language. Therefore, a function can be embedded in data structures, passed as a parameter and returned as a value.

Galileo 97 is a safe language. Each denotable value of the language possesses a type. Besides predefined types, type constructors exist to define new types, from predefined or previously defined types. They are: pairs, tuple, sequence, discriminated union, function and semi-abstract types. When defining semi-abstract types, the set of possible values can be restricted with assertions. In general, any expression has a type that can be statically determined, so that every type violation can be detected by textual inspection (static type checking). Although any statically detectable error could also be detected at run–time, the language has been designed to be statically type checkable for the following basic reasons: firstly, programs can be safely executed disregarding any information about types; secondly, the language offers considerable benefits in testing and debugging applications, since the type-checker detects a large class of common programming errors without the need to execute programs, while errors at run–time could be detected only by providing test data that cause the error to be raised.

Galileo 97 has subtyping. If a type `T` is a subtype of a type `T'`, then a value of `T` can be used as an argument of any operation defined for values of `T'`, but not vice versa because the subtype relation is a partial order.

Galileo 97 has parametric polymorphism and bounded parametric polymorphism. Polymorphism is the ability of a function to handle values of many types. Bounded parametric polymorphism has the effect of integrating parametric polymorphism with subtyping, and it provide more expressive power than either notion taken separately.

Galileo 97 has an exception-trap mechanism, which allows programs to handle system and user generated exceptions uniformly. Exceptions can be selectively trapped, and exception handlers can be specified.

Galileo 97 supports the abstraction mechanisms of an object data model. Classes are the mechanism to represent a database by means of modifiable sequences of interrelated objects, which are the computer representation of certain facts of entities of the world that is being modeled. Predefined assertions on classes are provided and the operators for including or eliminating elements in a class are automatically defined.

This manual describes the current main memory implementation of Galileo 97 running on Macintosh and PC-Windows.

The next sections show an example of a session with the system. The transcript of the interaction is shown enclosed in solid lines.

2 Getting started

The next sections explain how to install Galileo 97 under Macintosh OS X or MS-Windows. The two versions are the same as far as the language is concerned, but they offer different functionalities.

2.1 Installing Galileo 97 on a Macintosh OS X

To install the Galileo application copy it onto the hard disk and double-click on it.

2.2 Installing Galileo 97 on a PC

At least the following are required to use the application:

- a processor 386;
- Windows 3.1, 95 or XP;
- 2MB of RAM and 4MB of swap file.

The Galileo disk contains:

- the application code `Galileo.exe`;
- the application `Starter.exe` to use the first time;
- the file `Galileo.res`
- a folder with examples.

To install the application:

- create a folder on the disk;
- copy the files into the folder;
- execute the application `Starter.exe` to generate six files with a prefix `SG` in the application folder. Once the application `Starter` has been executed, it can be removed.

2.3 Interacting with the system

The Galileo 97 application is opened with a double-click on its icon.

The Windows version opens a window with the title `Galileo 97` and a menu bar. To execute a text, open a window on the file, select the text and choose `Execute` from the Galileo 97 menu.

The Macintosh version opens one special `"Galileo 97 Session"` window. An expression terminated by a ";" is executed by hitting the `Execute` key. Several expressions can be typed, and the system will execute them after the `Execute` key is pressed (see Preferences).

Once the system is loaded, the user types phrases, which can be *expressions, top-level declarations* or *commands*, and the system prints back a result in the case of an expression, and an acknowledgment in the other cases. This cycle is repeated over and over, until the system is exited.

Every top–level phrase is terminated by a semicolon. A phrase can be distributed on several lines by breaking it at any position where a blank character is accepted. Several phrases can also be written on the same line.

```
3*2+7;

   13 : int

it+1;

   14 : int

let x := 2*3;

  > x = 6 : int
```

The previous phrases are numerical expressions. When an expression is evaluated, the system responds with the value of the expression, a ":", and the type of the expression. If the value is a functional value, then the value is substituted by the string `fun`.

The variable `it` is always bound to the value of the last expression evaluated, and is not affected by declarations or by computations which for some reason fail to terminate. The `it` variable is useful in an interaction, when used to access values which have "fallen on the floor" because they have been computed as expressions but have not been bound to any variable.

The phrase `let x:= e` is an example of a top-level declaration: it binds the identifier `x` to the value of `e` in the global environment. From this moment, unless `x` is redefined, every occurrence of `x` will be substituted by the associated value.

Currently, there is no provision in the Galileo 97 system for editing and storing programs or definitions given during a session.[1] For this reason Galileo 97 schemas are usually prepared in text files and then loaded and tested interactively.

A Galileo 97 source file looks like a set of top–level phrases, terminated by a semicolon. A file containing Galileo 97 expressions can be loaded by the command `load "Name"`. *The file must be in the application folder*, unless the command is given through the corresponding menu entry.

With the command `outputoff` the system prints only the result of an expression, but not the effects of declarations. The command `outputon` restores the normal system behavior. Finally, with the command `:TypeIde` the system prints the definition of the user defined `TypeIde`.

[1]With the Mac version the user can save the current state at any time with the command `dump "Name"`, and restore it later in the same or a following session with the command `restore "Name"`.

3 Lexical Matters

The following lexical entities are recognized: *identifiers, keywords, numbers, strings, delimiters*, and *comments*. See Appendix I for a definition of their syntax.

An identifier is a sequence of letters, numbers, and underscores `_` that do not start with a digit.

Keywords are lexically like identifiers, but they are reserved and cannot be used for program variables. The keywords are listed in Appendix B.

Numbers are integers or reals. Integers are sequences of digits. The form of real numbers is described in Section 4.5.

Strings are sequences of characters delimited by a string quote `"`.

Delimiters are one–character punctuation marks like `","` and `";"` which never stick to any other character. No space is ever needed around them. Left parentheses are `(`, `[`, and {; right parentheses are `)`, `]` and }. Moreover, some characters stick to the inner side of parentheses to form compound parentheses like `(|` and `|)`. See Appendix I for details.

Any sequence of legal characters enclosed between `%` is a *comment*.

4 Expressions

Galileo 97 is an expression–based language. Expressions denote values, and have a type which is statically determined.[2] All standard programming constructs (conditionals, declarations, functions, etc.), are packaged into expressions yielding values. Strictly speaking there are no statements: even side–effecting operations return values. All these different kinds of expressions are described in this section.

It is always meaningful to supply arbitrary expressions as arguments to functions (when the type constraints are satisfied), or to combine them to form larger expressions in the same way that simple constants can be combined.

4.1 Unknown

The expression `unknown` denotes the only value of the language, of type `none`, which belongs to any type. It can be used as an expression of any type to denote an unknown value of that type. The value `"unknown"` is returned when this value is used by an operator which tries to extract some information from it. For instance, the expression `unknown + 3` returns `unknown:int`. The operator:

```
isunknown : any -> bool
```

returns `false` for any value different from `unknown`, `true` otherwise.[3]

The equality operator returns `true` if both the operands are `unknown`, `false` otherwise; any other comparison operator returns `unknown` if one of its operands is `unknown`.

4.2 Nil

The expression `nil` denotes the only value, besides `unknown`, of type `null`. The only operator on `nil` is the equality operator.

4.3 Booleans

The predefined constants `true, false` denote the respective Boolean values. Boolean operators are:

```
Not        :bool -> bool
And, Or    :(bool # bool) -> bool     (infixes)
```

`And` and `Or` evaluate only one argument when it is sufficient to determine the result. If an argument is `unknown` the result is `unknown`.

4.4 Integers

Operations on integers are:

[2]More precisely, expressions have a set of types, as discussed in 7.

[3]Any Galileo 97 value has type `any`.

```
~2,~1,0,1,2,        :int                 (constants)
~                   :int -> int
+, -, *, div, mod   :int # int -> int    (infixes)
>, >=, <=, <, <>    :int # int -> bool   (infixes)
```

Negative integers are written `~3`, where `~` is the complement function, while `-` is the difference. Integers assume values in the range `~maxint` to `maxint`, where `maxint` = +2,147,483,647; when an integer operation produces a number outside this interval, the exception `"integer overflow"` is generated. Moreover, exceptions can be generated by integer division `div` and module `mod`, which fail with string `"division by zero"` when their second argument is zero.

4.5 Real Numbers

Real numbers can assume values in the range `~maxreal` to `maxreal`, where `maxreal` $= 1.7 \times 10^{38}$.

```
syntax:     RealNum ::=
                Integer"."[Integer]["e"("+" | "-")Integer].
```

Real numbers have the type `real`.

Operations on real numbers are:

```
~2.,~1.,0.,1.1      :real                  (constants)
~                   :real -> real
+, -, *, /          :real # real -> real   (infixes)
>, >=, <=, <, <>    :real # real -> bool   (infixes)
intofreal           :real -> int
```

`intofreal` converts a real number to the nearest integer. If the absolute value of its argument is greater then `maxint`, it returns `maxint`

4.6 Strings

A string constant is a sequence of characters enclosed in quotes. By characters we mean full 8-bit codes in the range 0..255. Operations on string are:

```
"this is a string" :string                       (constants)
&                  :string # string -> string    (infix)
stringlength       :string -> int
extract            :string # int # int -> string (defined)
explode            :string -> seq string
implode            :seq string -> string
explodeascii       :string -> seq int
implodeascii       :seq int -> string
intofstring        :string -> int
realofstring       :string -> real
stringofint        :int -> string
match              :string # string -> bool      (defined)
printstring        :string -> null
```

`&`

maps two strings into a string which is their concatenation.

`stringlength`

returns the length of a string.

`extract`

extracts a substring from a string: the first argument is the source string, the second argument is the starting position of the substring in the string (the first character in a string is at position 1), and the third argument is the length of the substring. It fails with string `"extract"` if the numeric arguments are out of range. `extract` is a library function (see Section 12.1).

`explode`

maps a string into the list of its characters, each one being a 1-character string.

`implode` maps a sequence of strings into a string which is their concatenation.

`explodeascii`

is like `explode`, but produces a list of the integer representations of the characters contained in the string.

`implodeascii`

maps a list of integers onto a string containing the corresponding characters. It fails with the string `"implodeascii"` if the integers are negatives or greater than 255.

`intofstring`

converts a numeric string into the corresponding integer number; negative numbers start with `~`. It fails with string `"intofstring"` if the string is not numeric.

`realofstring`

converts a numeric string into the corresponding real number, negative numbers start with `~`. It fails with string `"realofstring"` if the string is not numeric.

`stringofint`

converts an integer into a string representation of the length needed; negative numbers start with `~`.

`match`

returns true if the first argument is equal to the second argument, the latter can contain special characters: `?` is about to any character; `*` refers to any string of characters; `#` refers to any numeric character; `@` refers to any alphabetic character. `match` is a library function (see Section 12.1).

4.7 Updatable References

Assignment operations act on reference cells. A reference value is an updatable pointer to another value. References are, together with classes, the only data which can be side effected; they can be inserted anywhere an update operation is needed in variables or data structures. Side effects on embedded references are reflected in all the structures which share them.

References are created by the operator `var`, updated by `<-` and dereferenced by `at`. The assignment operator always returns `nil`. Reference values have type `var T`, where `T` is the type of the value contained in the reference.

```
var      : a* -> var a*                  (constructor)
at       : var a* -> a*
<-       : var a* # a* -> null           (infix)
```

The function `at` is an example of a *polymorphic* function, i.e., a function that works uniformly over a class of arguments of different types. The type of `at` contains a *type variable* `a*`, indicating that any kind of value can be used. A type containing type variables is called *polymorphic*, otherwise it is called *monomorphic*. Several type variables can appear in a type, and every variable can appear several times, expressing contextual dependencies among values of that type.

Here is a simple example of the use of references. A reference to the number 3 is created and updated to contain 5, and its content is then examined.

```
let a := var 3;

>   a = var 3 : var int

a <- 5;

    nil : null

at a;

    5 : int
```

4.8 Pairs

A pair is a heterogeneous list of two values. The type of a pair is the cartesian product, denoted by the symbol `#`. Two pairs are equal if the elements are of the same type and the elements in the same position are equal.

```
syntax:         E1,E2
typing rule:    if E1: t1 and E2 : t2
                   then (E1,E2) : t1 # t2
```

The operations on pairs are:

```
    ,           :a*, b* -> (a* # b*)      (infix)
    fst         :(a* # b*) -> a*
    snd         :(a* # b*) -> b*
```

Parentheses can be used to express non–flat cartesian products. In the following example, the second expression is a pair, whose first component is a pair:

```
1,true;

    (1,true) : (int # bool)

(1,true),3;

    ((1,true),3) : ((int # bool) # int)

snd(it);

    3 : int
```

4.9 Records

A record is an unordered set of (identifier (attribute or label), denotable value) pairs. A record type is an unordered set of (identifier, type) pairs. Two record types are equal if the fields with equal labels have equal values. The special brackets [, and] are used to construct records and record types.

```
syntax:         [Id1 := E1; ...; Idn := En ]      n >= 0
typing rule:    if E1: T1 And ... And En : Tn
                   then [ Id1 := E1; ...; Idn := En ] :
                               [ Id1 : T1; ...; Idn : Tn ]
```

The empty record `[ ]` has type `[ ]`.

Records are equipped with the dot operator: `R.Exp` returns the value of `Exp` evaluated in the current environment extended with the set of pairs (identifier, value) of the record `R`. When `Exp` is an identifier `Id` of `R`, the dot operator is the standard extraction operator which returns the value associated with an identifier `Id` from a record `R`.[4] Here there is an example of construction of a two component record and uses of the dot operator:

```
let r := [ Name := "Paul"; Surname := "Brown" ];

> r = [ Name := "Paul"; Surname := "Brown" ]
       : [Name : string; Surname : string ]

r.Name;

   "Paul" : string
```

[4]In general, to parse `R.Exp` correctly, it must be written as `R.(Exp)`.

```
r.(Name & "_" & Surname);

   "Paul_Brown" : string
```

In addition to the operator that extracts the value of an identifier from a record, the following operators are provided to construct new records:

- `project` extracts from a record `R` a subset of the (identifier, value) pairs:

  `R project [`$\mathtt{A}_1$ `:`$\mathcal{S}_1$`; ...; `$\mathtt{A}_n$ `:`$\mathcal{S}_n$`]`

  where $\mathtt{A}_1, \ldots, \mathtt{A}_n$ are labels of `R` with types $\mathcal{T}_1 \subseteq \mathcal{S}_1, \ldots \mathcal{T}_n \subseteq \mathcal{S}_n$.

  The result is the record `[`$\mathtt{A}_1$ `:= `$\mathcal{E}_1$`; ...; `$\mathtt{A}_n$ `:= `$\mathcal{E}_n$`]` with type `[`$\mathtt{A}_1$ `:`$\mathcal{S}_1$`; ...; `$\mathtt{A}_n$ `:`$\mathcal{S}_n$`]`.

- `rename` changes the names of the identifiers; the unchanged identifiers and the new ones must all be different:

  `R rename (`$\mathtt{A}_1$ `=> A'`$_1$`; ...; `$\mathtt{A}_n$ `=> A'`$_n$`)`

  A label $\mathtt{A}_i$ may be a path $\mathtt{A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A}_i^{n_i}$; if $\mathtt{R.A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A}_i^{n_i}:\mathcal{T}$ for some type $\mathcal{T}$, then

  $$\mathtt{(R\ rename\ (A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A}_i^{n_i}\ \mathtt{=>\ A'}_i\mathtt{)).A'}_i\ :\mathcal{T}.$$

- `extend` adds a new set of pairs (identifier, value) to a record `R`; if an identifier $\mathtt{A}_i$ is present in `R`, `extend` replaces its value with one of a possibly unrelated type:

  `R extend [`$\mathtt{A}_i$ `:= Exp`$_i$`; ...; `$\mathtt{A}_n$ `:= Exp`$_n$`]`

  An expression $\mathtt{Exp}_i$ associated with a label may contain the pseudo-variable `me`, which denotes, recursively, the new record *after* it has been extended. If $\mathtt{Exp}_i$ is not a function, the type $\mathtt{T}_i$ of $\mathtt{Exp}_i$ must be specified as $\mathtt{A}_i\mathtt{:T}_i$ `:= Exp`$_i$.

- `times` concatenates the pairs (identifier, value) of two records $\mathtt{R}_1$ and $\mathtt{R}_2$ without common identifiers:

  $\mathtt{R}_1$ `times` $\mathtt{R}_2$

```
[ Name := "Paul"; Surname := "Brown" ] project [Name];

   [ Name := "Paul" ] : [Name : string ]

let R := [ Name := "Paul"] extend
         [BirthYear := 1950;
          Age := fun(CurrentYear:int):int is
                    CurrentYear - me.BirthYear] ;

>  R := [ Name := "Paul"; BirthYear := 1950; Age := fun  ] :
        [ Name :string; BirthYear :int; Age :int -> int ]

R.Age(1997);

   47 : int

[ Name := "Paul"; Surname := "Brown" ]
            rename ( Name => Nome; Surname => Cognome );

   [ Nome := "Paul"; Cognome := "Brown" ] :
       [Nome : string; Cognome : string ]

[ Name := "Paul" ] times [ Surname := "Smith"];

   [ Name := "Paul"; Surname := "Smith"] :
       [Name : string; Surname : string]
```

4.10 Variants

The strong typing rules of Galileo 97 do not directly allow one to write functions which return different kinds of values, e.g. integers and booleans. Similarly, sequences have to be homogeneous and cannot contain, say, strings and integers at the same time. These restrictions can be relaxed by using values of variant types.

Variants and records can be considered as complementary concepts: a record type is a labeled product of types, while a variant type is a labeled sum (disjoint union) of types. A variant type consists of a set of alternative values. It is different from the mathematical union of sets in that each value retains an inspectable *tag* (or label), indicating the alternative to which it belongs. Hence testing the label of a variant value is like testing its type from a finite set of possibilities. The special brackets `(|` and `|)` are used to delimit variants and variant types.

```
syntax:         (| Id := E |)
typing rule:    if E : T
                   then (| Id := E |) :(| Id : T |)
```

Two variant types are equal if the set of their pairs (tag, type) are equal.

Two basic operators are defined on variants: `is` tests the label of a variant value, and `as` returns the value contained in the variant.

Here there is an example of construction and operation on two variant values. The type of the first variant is inferred by the system, while the type of the second variant is specified by the user.

```
(|Integer := 3|);

   (|Integer := 3|) : (|Integer : int|)

it as Integer;

   3 : int

(|Integer := 6|) : (| Integer : int or Token : string|);

   (|Integer := 6|) : (| Integer : int or Token : string|)

it is Integer;

   true : bool
```

A very useful abbreviation concerning variants is the following: whenever we have a variant type with a field `a :null`, we can abbreviate that field specification to `a`, and whenever we have a variant value `(| a := nil |)`, we can abbreviate it to `(| a |)`. This is very convenient when defining enumeration types, which are variant types where only the labels are relevant and the types associated with them are not used. Below is an example of an enumeration value.

```
(|red|) : (|red or green or yellow|);

   (|red|) : (|red or green or yellow|)
```

The type `optional T` is an abbreviation for type `(|bound :T or unbound |)`.

4.11 Sequences

A sequence is a finite ordered collection of homogeneous elements, i.e. values with the same type. Sequences differ from sets in the ordering and multiplicity of elements. Sequences are enclosed in square brackets and their elements are separated by semicolons. The empty list is { }.

```
syntax:        {E1; E2; ... ; En}    n >= 0
typing rule    if E1: T And ... And En : T
                  then {E1; E2; ... ; En} :seq T
```

Sequences of any type can be created, e.g. sequences of strings, sequences of sequences of integers, sequences of functions, etc. Since each expression must have a type that is statically determinable, when an empty sequence is created, the type of the sequence must be specified, e.g. "`{ }:seq int`".

Two sequences are equal if the elements are of the same type, the cardinality of the sequences are equal, and the elements in the same position are equal.

The predefined operators on sequences are:

`append`	`:seq a* # seq a* -> seq a*`	`(infix)`
`avg`	`:seq int -> real`	
`avg`	`:seq real -> real`	
`count`	`:seq a* -> int`	
`difference`	`:seq a* # seq a* -> seq a*`	`(infix)`
`emptyseq`	`:seq a* -> bool`	
`first`	`:seq a* -> a*`	
`flatten`	`:seq seq a* -> seq a*`	
`get`	`:seq a* -> a*`	
`intersection`	`:seq a* # seq a* -> seq a*`	`(infix)`
`isin`	`:a* # seq a* -> bool`	`(infix)`
`known`	`:seq a* -> seq a*`	
`max`	`:seq int -> int`	
`max`	`:seq real -> real`	
`min`	`:seq int -> int`	
`min`	`:seq real -> real`	
`nth`	`:seq a* # int -> a*`	
`pick`	`:seq a* -> a*`	
`rest`	`:seq a* -> seq a*`	
`reverse`	`:seq a* -> seq a*`	
`setof`	`:seq a* -> seq a*`	
`sort`	`:seq a* # (a* # a* -> bool) -> seq a*`	
`sum`	`:seq int -> int`	
`sum`	`:seq real -> real`	
`union`	`:seq a* # seq a* -> seq a*`	`(infix)`
`::`	`:a* # seq a* -> seq a*`	`(infix)`

`append`

appends the elements of its left argument to the head of a sequence (its right argument).

`avg, min, max, sum`

can be applied to not empty sequences of numbers and return, the sum, minimum, maximum, average, of the numbers, respectively. If the parameter is a sequence with unknown values, the functions return the value `unknown`. The function `known` it is useful to ignore the unknown values.

`count`

returns the length of a sequence.

`emptyseq` tests whether the sequence is empty.

`first`

returns the first element of a non–empty sequence; it fails when applied to an empty sequence.

`flatten`

takes as its argument a sequence of sequences and combines all the elements in these sequences to form a single sequence. For instance, `flatten({{1;2};{3;4}})` returns `{1; 2; 3; 4}`.

`get`

returns the element of a sequence with one element;

`intersection, difference, union`

are an extension of the usual set operators on sequences: if the arguments have duplicates, the result is ordered according to the order of the first argument, and the duplicate elements are differentiated according to their position in the sequence. For instance:

`{1; 1; 2; 1} intersection {1; 2; 1}` gives `{1; 1; 2}`, and not `{1; 2; 1}`, in fact it is like
`{1'; 1''; 2; 1'''} intersection {1'; 2; 1''}`.
`{1; 1; 2; 1; 3} difference {1; 2; 1}` gives `{1; 3}`.
`{1; 1; 2; 1; 3} union {1; 2; 1}` gives `{1; 1; 2; 1; 3}`.

`isin`

tests whether an element is in a sequence.

`known`

returns the given sequence without the `unknown` elements.

`nth`

returns the n-th element in the sequence; it fails if the value of the second argoment is out of range.

`rest`

returns the given sequence without its first element; it fails when applied to an empty sequence;

`setof`

returns a sequence without duplicates.

`set`

tests whether a sequence has no duplicates.

`reverse`

returns a new sequence which contains the elements of the argument in the reverse order.

`sort`

takes a list and a binary predicate and returns a new list obtained by sorting its first argument according to the predicate.

`::`

(cons) adds an element to a list (e.g. `1::{2;3}` is the same as `{1;2;3}`).

The following constructs are available on sequences:

```
I In Q
TS where B
select E from TS
TS groupby [Id1 := E1; ...; Idn := En]
TS .* Q
each TS with B
some TS with B
TS project* [ I1 [:T1]; ...; In [:Tn] ]
TS extend* [ I1 [:T1] := Exp1; ...; In [:Tn] := Expn ]
TS rename* ( I1 => I'1; ...; In => I'n )
TS times* TS'
loop TS do E
```

where:

- `I, I1, I2` are identifiers;
- `Q` is a sequence of value of type $\mathtt{T}_Q$;
- `B` is a boolean expression;
- `TS` is a sequence of records, or objects, with type $\mathtt{T}_{TS}$;
- `E` is an expression with type $\mathtt{T}_E$.

`I In Q`

returns a sequence of records with a unique field named `I`, associated with the value of the corresponding element in `Q`. The expression has type `seq [I :`$\mathtt{T}_Q$`]`.

`TS where B`

returns the sequence of the values that satisfy the boolean expression `B`. The expression has type `seq` $\mathtt{T}_{TS}$.

`select E from TS`

returns the sequence of the values of the expression `E` evaluated for each element in a sequence `TS`. The expression has type `seq` $\mathtt{T}_E$.

`TS groupby [`$\mathtt{Id}_1$` := `$\mathtt{E}_1$`; ...; `$\mathtt{Id}_n$` := `$\mathtt{E}_n$`]`

returns a sequence with type

`seq [`$\mathtt{Id}_1$` :`$\mathtt{T}_{E_1}$`; ...; `$\mathtt{Id}_n$` :`$\mathtt{T}_{E_n}$`; partition:seq `$\mathtt{T}_{TS}$`]`

The elements of `partition` are those of `TS` with the same value of `[`$\mathtt{Id}_1$` := `$\mathtt{E}_1$`; ...;` $\mathtt{Id}_n$` := `$\mathtt{E}_n$`]`, which is evaluated for each element `e` of `TS` in the current environment extended with the components (attribute, value) of `e`.

The result is evaluated as follows:

1. for each element `e` of `TS` the record `[Id`$_1$ `:= E`$_1$`; ...; Id`$_n$ `:= E`$_n$`]` is evaluated in the current environment extended with the components (attribute, value) of `e` to produce a sequence `W` of pairs (`e`, `[Id`$_1$ `:= v`$_1$`; ...; Id`$_n$ `:= v`$_n$`]`);
2. the elements of `W` are partitioned in subsequences with the same value of `[Id`$_1$ `:= v`$_1$`; ...; Id`$_n$ `:= v`$_n$`]`. Let {`ei`} the sequence of the first component of the elements of a partition;
3. for each partition an element of the `groupby` result is produced with components `[Id`$_1$ `:= v`$_1$`; ...; Id`$_n$ `:= v`$_n$`; partition:= {ei}]`.

`TS .* Q`

is equivalent to `flatten(select Q from TS)`.

`each TS with B`

tests whether every element in `TS` satisfies `B`.

`some TS with B`

tests whether at least one element in `TS` satisfies `B`.

`project*`

returns the sequence of values obtained by applying the operator `project` to each element in `TS`.

`extend*`

returns the sequence of values obtained by applying the operator `extend` to each element in `TS`.

`rename*`

returns the sequence of values obtained by applying the operator `rename` to each element in `TS`.

`times*`

returns the sequence of values obtained by concatenating with the operator `times` each element in `TS` to all the elements in `TS'`.

`loop`

iterates on the elements in a sequence of tuples or objects to cause side-effects. It returns `nil`. The iteration can be halted with an exception or with the expression `exit`.

When the above constructs are used as part of an expression, to avoid parsing ambiguities, the construct should be enclosed in parenthesis. For instance:

```
let x:= (get Persons where Name = "Smith")
                  iffails failwith "Unknown person";
```

Here are some simple examples of the use of sequences:

```
{2; 10; 9; 2; 5};

   {2; 10; 9; 2; 5} :seq int

x In {2; 10; 9; 2; 5} where x > 5;

   {[x := 10];[x := 9]} :seq [x:int]

select x + 1
from x In {10; 9}
where x > 5 ;

   {11; 10} :seq int

select if x > 10 then x - 1 else x + 1
from x In {8;9;10;11;12} ;

   {9; 10; 11; 10; 11} :seq int

first ({1; 2} append {3; 4});

   1 : int

sort({(1,2);(5,6);(3,4);(1,3)},
      fun(x:(int#int),y:(int#int)) :bool is
           fst(x) <= fst(y) And snd(x) <= snd(y));

   {(1,2);(1,3);(3,4);(5,6)} :seq (int#int)

{[a:=1];[a:=2];[a:=1];[a:=3]} groupby [SameA:= a];

    {[SameA := 1; partition := {[a:=1];[a:=1]}];
     [SameA := 2; partition := {[a:=2]}];
     [SameA := 3; partition := {[a:=3]}]}
   :seq [SameA :int; partition :seq [a:int]]

{[a:=1];[a:=2];[a:=1];[a:=3]}
groupby [Less2:= a<2; Greater2 := a>2];

    {[Less2:=true; Greater2:=false; partition :={[a:=1];
                                                 [a:=1]}];
     [Less2:=false; Greater2:=false; partition :={[a:=2]}];
     [Less2:=false; Greater2:=true; partition :={[a:=3]}]}
  :seq [Less2 :bool;Greater2 :bool; partition :seq [a:int]]

{[a:=1];[a:=2];[a:=1];[a:=3]} .* {a};

    {1; 2; 1; 3} :seq int

{[a:= 1; b:={[b1:= "a"; b2:= "b"]}];
 [a:= 2; b:={[b1:= "c"; b2:= "d"]}]} .* b .* {b1};
```

```
    { "a"; "c" } :seq string
```

4.12 Functions

Functions are usually defined as follows

```
syntax:        let Id := fun(Id1:T1, ...,Idn:Tn) : T is E
typing rule:   if Id1:T1 And ... And Idn:Tn And E : T
                   then Id : (T1 # ... # Tn) -> T
```

but they can also be introduced in isolation by the use of *fun(lambda)–notation*.

A *fun-expression*, like the one above, denotes an unnamed function; it has a binder, preceded by `fun`, and a body, preceded by `is`. The binder is the formal parameter of the function. The body is an expression, and its value is the result of the function.

Functions can be constructed and applied to their arguments. They cannot be printed.

```
let Succ:= fun(x : int) : int is x+1;

>  Succ = fun : int -> int

Succ(3);

   4 : int

%    ApplyPositive takes a function as a parameter and returns a new
function %

let ApplyPositive := fun(f: int->int): int->int is
      fun(x:int) : int is
         if x <= 0 then failwith "Negative argument"
                   else f(x);

>  ApplyPositive = fun : (int -> int) -> (int -> int)

  %    This is an example of a function embedded in a data structure
which is later applied %

let Natural := [ Predecessor := var 0; Next := Succ ];

>  Natural = [ Predecessor := var 0; Next := fun ] :
                  [ Predecessor: var int; Next: int->int ]

Natural.Next(at Natural.Predecessor);

   1 : int
```

4.12.1 Polymorphic Functions

Polymorphic functions are functions whose operands (actual parameters) can have more than one type. The fact that a given functional form is the same for all types is expressed by universal quantification.

```
syntax:         let Id := fun[X1; ...;Xn](Id1:X1, ...,Idn:Xn)
                          : T is E                          n > 0
typing rule:    if X1 ... Xn are distinct And E : T
                then Id : all[X1; ...;Xn](X1 # ... # Xn) -> T
```

Here is the polymorphic identity function:

```
let Id := fun[T](a:T) : T is a;

> Id = fun : all [T <: any] T -> T
```

In this definition of `Id`, `T` is a type variable and `[T]` provides type abstraction for `T` so that `Id` is the identity function for all types. Type parameters are enclosed in square brackets, while typed arguments are enclosed in parentheses.

In order to apply this identity function to an argument of a specific type, we must first supply the type as parmeter and then the argument of the given type:

```
Id[int](3);

   3 : int
```

Another interesting example is the polymorphic function `twice`, which can double the effect of any function:

```
let twice := fun[T](f:T->T, x:T):T is f(f(x));

> twice = fun : all [T <: any] ( T -> T # T )  -> T

let succ := fun(x:int):int is x + 1;

> succ = fun : int -> int

let SuccSucc := twice[int];

> SuccSucc = fun : ( int -> int # int )  -> int

SuccSucc(succ, 3);

   5 : int
```

Another example is a polymorphic function to sort sequences of values:

```
let rec
QuickSort := fun [X] (s: seq X, less:(X # X) -> bool): seq X is
if emptyseq(s)
then s
        else use
             p := first(s) and
             r := rest(s)
             in QuickSort[X](
                   (select x from x In r where less(x, p)),
                    less)
                append {p}
                append QuickSort[X](
                         (select x from x In r where Not less(x, p)),
                          less);

let LessThan := fun(x:int, y :int):bool is x < y;
QuickSort [int] ({2;4;3;1;9;5}, LessThan);
```

For another kind of polymorphism, see Section 6.3.

4.13 Conditional

The syntactic form for a conditional expression is:

```
syntax:       if E1 then E2 else E3
typing rule:  if E1:bool And E2:T And E3:T
                   then if E1 then E2 else E3: T
```

the expression E_1 is evaluated to obtain a boolean value. If the result is `true` then E_2 is evaluated; if the result is `false` or `unknown` then E_3 is evaluated. The `else` branch must always be present.

The typing rule for the conditional states that the `if` branch must be a boolean, and that the `then` and `else` branches must have equal types.

4.14 Sequencing

When several side-effecting operations have to be executed in sequence, it is useful to use *sequencing*:

```
syntax:          (E1; ...; En)     n >=1
typing rule:     if En : T then (E1; ...; En) : T
```

(the parentheses are needed), which evaluates `E1`; ...; `En` in turn and returns the value of `En`. The type of a sequencing expression is the type of `En`.

4.15 While

The `while` construct can be used for iterative computations. The `while` construct has two parts: a test, preceded by the keyword `while` and a body, preceded by the keyword `do`. If the test evaluates to `true` then the body is executed, otherwise `nil` is returned. This process is repeated until the test yields `false` (if ever), an exception occurs or `exit` is evaluated.

```
syntax:       while E1 do E2
```

The result of a terminating `while` construct is always `nil`, hence it is only useful for its side effects.

```
typing rule:      if E1 : bool And E2 : T
                  then (while E1 do E2) : null
```

As an example, here is an alternative iterative definition of factorial:

```
let factorial := fun(n :int) :int is
      use count := var n
      and result := var 1
      in  (while Not(at count = 0) do
             (result <- at count * at result;
              count <- at count - 1);
              at result) ;

>  factorial = fun : int -> int
```

4.16 Case

The case construct provides a systematic way of structuring programs based on variant types. It is very common for a function to take an argument of some variant type and to do different things according to the variant label. The case construct is a convenient form to test the label of a variant and to bind the value to a local identifier.

```
syntax:     case E when
                   (| Id1 := Id1'. E1
                   or Id2 := Id2'. E2
                   ...
                   or Idn := Idn'. En |)

typing rule: if E : (| Id1 : T1 or ... or Idn : Tn |)
                      And for each label Idi' : Ti
                              imply that Ei : T
                   then (case E when (| ... |) ) : T
```

This says that every `E1` must have the same type, and if `E` has label `Idi`, then `Idi'` is bound to `E as Idi` and it can be used during the evaluation of `Ei`. In other words, `Ei` is evaluated as `use Idi' := (E as Idi) in Ei`.

The function `alltoint` converts the argument, of type `(| Integer :int or Token :string|)`, to an integer:

```
let alltoint := fun(x: (|Integer: int or Token: string|)): int is
       case x when
         (| Integer := i. i
         or Token:= t. intofstring(t) |);

>  alltoint = fun : (|Integer: int or Token: string|)  -> int
```

4.17 Scope Blocks

A scope block is a control structure which introduces new identifiers and delimits their scope. Scope blocks have the same function as begin-end constructs in Pascal–like languages, but they are different due to the fact that they are expressions returning values.

```
syntax:          use D in  E
typing rule:     if E : T
                   then (use D in E) : T
```

The use construct introduces new bindings in the `declaration` part which can be used in the `exp` part (and there alone). Newly introduced identifiers hide externally declared identifiers that have the same name for the scope of `exp`; the externally declared identifiers remain accessible outside `exp`.

The value returned by a scope block is the value of its expression part. Likewise, the type of a scope block is the type of its `exp` part. The various kinds of declarations are described in Section 6.

A truncated form of a scope block can be used at the top-level (i.e. it cannot be nested inside other expressions):

```
syntax:     let declaration;
```

This has the effect of binding at the top–level the identifiers defined in `declaration`; the scope of these identifiers is the whole history of interaction after that point, until those identifiers are redefined. When top–level identifiers are redefined, they become totally inaccessible, however their values might still be reachable by other paths.

4.18 Exceptions and Traps

An *exception* (also called a *failure*) can be raised by a system primitive or by a user program. When an exception is raised, the execution of the current expression is abandoned, and an *exception string* is propagated outward, tracing back along the history of function calls and expression evaluations which led to the exception. If the exception is allowed to propagate up to the top–level, the exception string is printed as a *failure message*:

```
    exception:      Failure: string
```

Note that the exception string is not the value of a failing expression: failing expressions have no value, and exception strings are manipulated by mechanisms which are independent of the usual value manipulation constructs. The exception string is often the name of the system primitive or user function which raised the exception.

User exceptions can be raised by the failwith and fail constructs:

```
syntax:       failwith E
```

The expression `E` above must evaluate to a string, which is the exception string. `fail` is equivalent to `failwith "fail"`. The expression `failwith E` has type `none`, which is compatible with every type, i.e. a failure can be generated without regard to the expected value of the expression which contains it.

The propagation of exceptions can only be stopped by one of the two *trap* constructs:

```
syntax:       TrapExpression::=
                      E iffails E"
                      E casefails E' E"
```

The result of the evaluation of `E iffails E"` is normally the result of `E`, unless it raises an exception, in which case it is the result of `E"`. The result of the evaluation of `E casefails E' E"` (where `E'` evaluates to a sequence of strings `ss`) is normally `E`, unless `E` raises an exception, in which case it is the result of `E"` whenever the exception string is one of the strings in `ss`; otherwise the exception is propagated.

Note that in the current implementation of Galileo 97, when an expression fails its side–effects are not undone, i.e. transactions are not implemented.

The rule for the trap constructs states that the exception handler must have the same type as the possibly failing expression, so that the resulting type is the same whether the exception is raised or not.

```
typing rule:        if E : T And E" : T And E' :seq string
                    then (E iffails E") : T
                    And (E casefails E' E") : T
```

5 Type Expressions

A type expression is either a basic type or the application of a type operator to a sequence of type expressions. Type operators are usually suffixes, except for `#`, `and`, `;`, `or` and `->`, which are infixes.

5.1 Type Operators

The predefined type operators are:

`null`	`Null type`	`Nonfix`
`none`	`Least general type`	`Nonfix`
`any`	`Most general type`	`Nonfix`
`bool`	`Boolean type`	`Nonfix`
`int`	`Integer type`	`Nonfix`

`real`	`Real type`	`Nonfix`
`string`	`String type`	`Nonfix`
`#`	`Pair type`	`Infix`
`;`	`Record type`	`Infix`
`or`	`Variant type`	`Infix`
`->`	`Function type`	`Infix`
`seq`	`Sequence type`	`Prefix`
`var`	`Reference type`	`Prefix`

6 Declarations

Declarations are used in scope blocks to establish a set of bindings of identifiers to values, called an *environment* (see section 4.17); in this section scope blocks are frequently used to illustrate the use of declarations.

Every declaration is said to import some identifiers, and to export other identifiers. The imported identifiers are the ones which are used in the declaration, and are usually defined outside the declaration (except in recursive declarations). The exported identifiers are the ones defined in the declaration, and that are accessible outside the declaration.

A declaration can be a *type binding*, a *value binding*, a *class binding*, a *parallel declaration*, a *recursive declaration*, or a *sequential declaration*.

```
syntax:      Declaration ::=
                  TypeBinding
                  ValueBinding
                  ClassBinding
                  SubclassBinding
                  ParallelDeclaration
                  RecursiveDeclaration
                  SequentialDeclaration
```

6.1 Type Bindings

Type bindings define types:

```
syntax:     TypeBinding ::=
                  SimpleTypeBinding
                  SemiAbstractTypeBinding
                  ObjectTypeBinding
                  ViewTypeBinding
                  ParallelTypeBinding
                  RecursiveTypeBinding
```

6.1.1 Simple Type Bindings

This simplest form of type definition introduces a name which stands for the associated type.

```
syntax       SimpleTypeBinding ::=
                  type TypeIde := TypeExp
```

For instance, in the following example the types `Date` and `[Day: int; Month: int; Year: int]` are the same type.

```
let type Date := [Day: int; Month: int; Year: int ];

>  type Date = [Day: int; Month: int; Year: int ]

let ADate := [Day := 3; Month := 8; Year:= 1985];

>  ADate = [Day := 3; Month := 8; Year := 1985]
| : [ Day :int; Month :int; Year :int ]

ADate:Date;

   [Day := 3; Month := 8; Year := 1985]: Date
```

6.1.2 Semi-abstract Type Bindings

The type of the values presented so far depends only on the structure of the values, i.e. a *structural equivalence* rule is adopted. In contrast, a user-defined semi-abstract type is always different from all other concrete or semi-abstract types; in addition, it differs from its representation type. New semi-abstract types are introduced with the following declaration:

```
syntax:      SemiAbstractTypeBinding ::=
                    type Id <-> RepType
                         [BeforeMK] [BeforeDrop]
             BeforeMK   ::=  "before" "mk" "(" Ide ")"
                                 {"if" BoolExp "do" Exp}1.
             BeforeDrop ::=  "before" "drop" "(" Ide ")"
                                 {"if" BoolExp "do" Exp}1.
```

The `BeforeMK`, and the `BeforeDrop` parts can be omitted. This declaration introduces the following bindings:

1. `Id` is bound to a new abstract type whose domain is isomorphic to the domain of the representation type `RepType`, which must be a concrete type, and may be restricted by the assertion `BoolExp`;
2. the identifiers `mkId`, and `repId` are bound to two primitive functions, automatically declared, to map values of the representation type into the semi-abstract type and vice versa;
3. the identifier `dropId` is bound to a primitive function to remove the type `Id` from a value of type `Id`. This function is useful to deal with object that can acquire and lose dynamically types, as described in the next section.

```
mkId   : RepType -> Id
repId  : Id  ->  RepType
dropId : Id  ->  null
```

The clauses `BeforeMK`, and `BeforeDrop` define triggers to invoke an action before the execution of the functions `mkId`, and `dropId` If the `BeforeMK` clause is present, `BoolExp` is a boolean expression on the values of `RepType` (denoted by the argument `Ide` of `mk`), which imposes a constraint on the values of the semi-abstract type. The constraint is controlled at execution time, before executing the `mkId` operation. If the `BoolExp` is true, then the `Exp` of the `do` clause is evaluated.

If the `BeforeDrop` clause is present, `BoolExp` is controlled before executing the `dropId` operation. If the `BoolExp` is true, then the `Exp` of the `do` clause is evaluated.

The following is the declaration of an abstract record type for dates. The identifier `this` in `mk(this)` is bound to the argument of `mkId`.

```
let type Date <->
      [ Day : int; Month : int; Year : int ]
     before mk(this)
     if  Not
        (use D := this.Day
         and M := this.Month
         and Y := this.Year
         in  M >= 1 And M <= 12 And Y >= 1983 And Y <= 2000
             And D <=  if M= 2
                       then if Y mod 4 = 0 then 29 else 28
                       else if M isin{4;6;9;11}
                             then 30 else 31 )
      do failwith "Illegal Date";

> type Date = -
|  mkDate = fun: [Day:int; Month:int; Year:int ] -> Date
|  repDate = fun : Date -> [Day :int; Month :int; Year :int ]
|  repDate = fun : Date -> null
```

Note that the representation type and the values of a semi-abstract data type are printed as `-`.

Once defined, new semi-abstract types have the same status as primitive types like `int` or `bool`: indeed, primitive types can be consistently regarded as predefined abstract types provided by the language.

Operator Overloading

A semi-abstract type imports all the primitive operators of the representation type. The set of imported operators on type `Id` depends on the representation type (see Appendix D).

The system overloads these operators which also work with values of the new type `Id`. For instance, consider the following semi-abstract type for weights:

```
let type Weight <-> int
    before mk(this)
    if Not this >= 0
    do failwith "Weight must be >= 0";
```

As a consequence of this declaration, the new type `Weight` inherits the operators on integers, in the sense that identifiers such as `-` and `+` are bound to two different functions: e.g., `-` is bound both to the primitive integer subtraction (`- :int # int -> int`) and to the overloaded weight subtraction (`- :Weight # Weight -> Weight`), which has been automatically declared by the system as follows:

```
 - := fun(x: Weight, y: Weight) : Weight is
              mkWeight(repWeight(x) - repWeight(y))
```

Notice that the weight subtraction is defined in terms of the original integer subtraction, but it also controls the trigger on weights, thus guaranteeing that weight subtraction yields proper weight values. Therefore, the expression:

```
use x := mkWeight(4)
and y := mkWeight(5)
in  x - y;
```

yields a run–time failure, since weights have to be non negative. In the application of overloaded identifiers, the type checker can to choose the right operator according to the type of the arguments: in the example above, the weight subtraction is chosen, since the identifier `-` is applied to a weight pair.

To define abstract types with a hidden representation and user-defined operations, objects types with `private` *attributes must be used, as shown below.*

6.1.3 Object Type Bindings

Objects are entities with an immutable identity, organized as acyclic graphs of *roles*. A role is an entry to access the object it belongs to, and is a first-class value of the language. Roles have both a behavior and a mutable state, which can be encapsulated to be queried and updated only by sending messages to the role. Role types are organized into an inclusion hierarchy with simple inheritance. Operators are defined to test for the existence of a certain role and to extend objects with new roles. As a first approximation, those familiar with objects can think of a role as an object of their favorite object-oriented language.

Roles are modeled using the so-called "object as record" analogy, adopted initially in Simula, and usually used in object database systems: Roles are essentially records with possibly special functional components `meth` to model methods; message passing is implemented as field selection.

The operator `<->` is used to define a new role type. Role types are represented as records, and methods can access the fields of a role using the predefined identifier `self`, if the role type is defined in a recursive environment.

A role type can have attributes defined as `Ide := Exp` (computed attributes). The value of the attribute `Ide` is the value of `Exp` evaluated at the time of the construction of the role value. In `Exp` the predefined identifier `self` cannot be used.

The operator `<->` is a *generative* type constructor, i.e. each role type definition produces a new type, different from any other type previously defined.

The signature $\Downarrow\mathcal{T}$ of an object type $\mathcal{T}$ is the set of label-type pairs of the messages which can be sent to its instances.

The following example shows the definition of the role type `Person`, with a method `WhoAreYou`:

```
let rec
type Person <->
          [ Code :string;
            Name :string;
            BirthYear :int;
            WhoAreYou:= meth() :string is
                        "My name is  " & self.Name ];
```

The definition of a role type $\mathcal{T}$ introduces the function mk$\mathcal{T}$ to construct values of type $\mathcal{T}$: the function parameter is a record type which is the representation type of $\mathcal{T}$ without the components describing methods or computed attributes.

An example of a construction of a role of type `Person` is:

```
let John := mkPerson ([ Name := "John Smith";
                        Code := "jhnsmt23h67";
                        BirthYear := 1967 ] );
```

Each application of the constructor mk$\mathcal{T}$ gives a value of type $\mathcal{T}$ with a different identity.

A method `m` *of a value* `O` *of a semi-abstract type* `T` *is activated by sending a message to* `O` *with the notation* `O.m(...)`*, if the method has parameters, otherwise with the notation* `O.m`*. The following example shows the use of the* `John` *method* `WhoAreYou` :

```
John.WhoAreYou;   returns "My name is John Smith"
```

Here is an example of role type with a computed attribute:

```
let TimeStamp :=
      use Stamp := var 0
      in fun () :int is (Stamp <- at Stamp + 1; at Stamp);

let type Order <->
        [ Code := TimeStamp();
          Product: [Code: int; Price: int];
          Customer: [Name: string; Address: string];
          Quantity: int];
```

Public and Private Properties

Methods and attributes, i.e. the properties of an object type, may be either *public* or *private*. A private property is accessible only inside the type definition. A public property may be used anywhere in the application. Properties are always public unless specifically declared private by placing the keyword `private` before the property name.

```
let rec
type Rectangle <->
        [ private base :int;
          private height :int;
          perimeter := meth() :int is
```

```
                      (self.height + self.base)*2;
          getBase := meth() :int is self.base;
          getHeight := meth() :int is self.height
        ]
      before mk(this)
      if Not this.base > 0 And this.height > 0
      do failwith "height and base must be > 0" ;
```

Since the constraint defined with the clause `before` is checked before the object construction, in the assertion the private attributes are accessible.

The parameter of the function mk$\mathcal{T}$, to construct values of type $\mathcal{T}$, is a record with attributes and types which are those of the representation type of $\mathcal{T}$, defined both public and private. For example:

```
let r := mkRectangle( [base := 20; height := 50 ] );
```

Virtual Methods

A virtual method is a method that has a declaration (signature), but no definition (implementation). The implementation is declared `virtual` and it will be given given later with a type defined defined by inheritance. An object type with a virtual method cannot be instantiated. For example:

```
let type Point := real # real;
let type Tile := Point # Point;

let type  Picture <->
            [ Class:string;
               Area:= meth() :real is virtual;
               Tile:= meth() :Tile is virtual];
```

Role Type Definition by Inheritance

Inheritance means any mechanism which allows something to be defined, typically an object type, by only describing how it differs from a previously defined one. Inheritance should not be confused with subtyping: subtyping is a relation between types such that when $T \subseteq U$, then any operation which can be applied to any value of type U can also be applied to any value of type T. The two notions are sometimes confused because, in object oriented languages, inheritance is usually only used to define object subtypes, and object subtypes can only be defined by inheritance. However, we will keep the two terms distinct and will use each of them with its proper meaning.

In Galileo 97 a role type $\mathcal{T}$ can be defined by inheritance from another role type $\mathcal{T}'$ as follows:

```
type 𝒯 <-> is 𝒯′ and ℋ
          [BeforeMK] [BeforeIn] [BeforeDrop]
BeforeIn ::= "before" "in" "(" Ide ["," Ide ] ")"
             "if" BoolExp "do" Exp1.
```

Type $\mathcal{T}$ *inherits* the $\mathcal{T}'$ attributes, i.e. both its instance variables and methods, as specified before: when an attribute is selected from an object role of type $\mathcal{T}$ it is first looked up inside the $\mathcal{T}$ role and then it is looked up in $\mathcal{T}'$, so that any $\mathcal{T}'$ attribute can be selected from a $\mathcal{T}$ type value.

Galileo 97 allows *strict* inheritance only: a $\mathcal{T}'$ attribute $\mathcal{A}_i$, with type $\mathcal{H}_i$, may be redefined in $\mathcal{T}$ only by specializing its type, that is the new type $\mathcal{H}_i'$ of $\mathcal{A}_i$ must be a subtype of $\mathcal{H}_i$. For this reason, the inheritance mechanism in Galileo 97 always produces an object subtype, i.e. in our example we will have $\mathcal{T} \leq \mathcal{T}'$.

Moreover, the following restrictions hold on private properties:

- in the type definition $\mathcal{T}$, public or private properties cannot be defined with the same name as private properties of the supertype $\mathcal{T}'$. This constraint implies that (a) private properties of the supertype cannot be redefined in a subtype, and (b) a private property of the supertype cannot be redefined public in a subtype;
- in the type definition $\mathcal{T}$, a private property cannot be defined with the same name as a public property of the supertype $\mathcal{T}'$. This constraint implies that a public property of the supertype cannot be redefined private in a subtype.

Another advantage of inheritance is that the a method of the subtype can be defined using the definition given for it in the supertype. The pseudo-variable `super` is statically bound, i.e. the method search for a message sent to `super` begins with the supertype of the type where the method is defined.

Here is an example of a role type defined by inheritance:

```
let rec
type Student <->
        is Person and
          [ StudentNumber :string;
            Faculty :string;
            WhoAreYou := meth() :string is
                         super.WhoAreYou &
                         " I am a student of  " &
                         self.Faculty ];

let rec
type Square <-> is Rectangle and
        [ perimeter := meth() :int is
                       self.getBase * 4
         ]
      before mk(this)
      if Not this.base = this.height
      do failwith "base and height of a square must be equal";
```

A subtype inherits the triggers of the super-type $\mathcal{T}'$, so the constructor mk$\mathcal{T}$ of values of type $\mathcal{T}$ raises both the trigger defined in the supertype $\mathcal{T}'$ and the one defined in the subtype $\mathcal{T}$.

Moreover the parameter of the constructor mk$\mathcal{T}$ is a record with the attributes defined both public and private in $\mathcal{T}$ and $\mathcal{T}'$. For this reason in the trigger defined in $\mathcal{T}$ both the private attributes of $\mathcal{T}$ and $\mathcal{T}'$ are accessible (see the trigger on type `Square`).

Finally, when a supertype method is redefined in a type $\mathcal{T}$, a value of type $\mathcal{T}$ uses the method given in $\mathcal{T}$ to answer a request to execute the method (*dynamic binding*).[5]

An Operator to Extend Objects with New Roles

When a role type $\mathcal{T}'$ is defined by inheritance from a type $\mathcal{T}$, besides the constructor `mk`$\mathcal{T}'$, the function `in`$\mathcal{T}'$ is also automatically generated, which extends an object from $\mathcal{T}$ to $\mathcal{T}'$, thus preserving its identity.

The function `in`$\mathcal{T}'$ has two parameters: the value of the object $\mathcal{O}$ to be extended, and a record which gives the attribute values of $\mathcal{T}'$ which are not inherited from $\mathcal{T}$.

If an attribute `A` of type `Ti` is redefined in the subtype with type `Ti'`, the function `in`$\mathcal{T}'$ checks that the value given for that attribute in the second parameter is a *subvalue* of the corresponding one in the first parameter, according to the following rule: `A`:`Ti` = `A`:`Ti'` (see Section 8).

If the `BeforeIn` clause is present, `BoolExp` is controlled before executing the `in`$\mathcal{T}'$ operation. If the `BoolExp` is true, then the `Exp` of the `do` clause is evaluated, and finally the `in`$\mathcal{T}'$ operation is executed.

For example, the object `John` with type `Person` may be extended with the type `Student` as follows:

```
let JohnAsStudent :=
       inStudent(John, [ StudentNumber := "0123";
                         Faculty := "Science"
                        ]);
```

Let us assume now that the following type is defined as well:

```
let rec
type Athlete <->
        is Person and
          [ Number:int;
            Sport: string;
            WhoAreYou:= meth() :string is
                        super.WhoAreYou &
                        ". I practice " &
                        self.Sport ];
```

We assume that an object can be extended with independent subroles and exhibit a different behavior according to the role through which it is accessed.

Let the object `John` be extended with the role type `Athlete`:

```
let AthleteJohn :=
        inAthlete(John, [Number := 245;
                         Sport := "tennis" ]);
```

The answer to the message `Number` sent to `StudentJohn` is a string, while the answer to the same message sent to `AthleteJohn` is an integer, likewise, there will be different answers to

[5]Galileo does not support multiple inheritance.

the message `WhoAreYou` sent to `StudentJohn` or to `AthleteJohn`. We say that `John`, `Student-John` and `AthleteJohn` are three different roles of the same object, of type `Person`, `Student` and `Athlete` respectively.

When an object is extended with new sub-roles which redefine a method of the super-role, a message can be sent to the super-role with two different notations to request a different search technique for the method to be used for answering the message:

- if the message is sent with the dot notation, the method is looked for in the receiving role and in its ancestors (*upward lookup*);
- if the message is sent with the `!` notation, the method is first looked for in all the descendants of the receiving role, visited in reverse temporal order, then in the receiving role, and finally in its ancestors (*double lookup*).

For example, the answer to the message `WhoAreYou` sent to `John` changes once the object has been extended with the role type `Student`, and once again after its extension with the role type `Athlete`. To receive from `John` always the same answer, irrespective of what extensions have occurred, the message must be sent with the notation `John!WhoAreYou`.

Other operators defined on objects and roles are:

- `dropT(Expr)`, to remove the role with type `T` and all the sub-roles of the object denoted by the expression `Expr`. If the object is still accessible by one of the removed roles, because the role has been previously bound to an identifier or because it has been used as a component of another value, a run–time failure will be generated when a message is sent to it.

  The function `dropT(Expr)`, like `mkT` and `inT'`, is automatically declared when a subrole type is defined.
- `Expr isalso T`, to test whether an object with the role denoted by the expression `Expr` has also the role type `T`; for example both the expressions `John isalso Athlete` and `StudentJohn isalso Athlete` are true.
- `Expr As T`, to coerce an object with the role denoted by the expression `Expr` to one of its possible roles `T`; for example `AthleteJohn As Student` returns the role with type `Student` of the object with role `AthleteJohn`.
- `Expr isexactly T`, to test the run–time role type of the role denoted by the expression `Expr`; for example `John isexactly Athlete` is false while `AthleteJohn isexactly Athlete` is true.

Note that the predicates `isalso`, `isexactly`, and the operator `As` do not generate a run–time failure if `Exp` denotes a dropped role of an object. For example, after a `dropAthlete(Athlete-John)`, `AthleteJohn isalso Athlete` returns false.

The operator `As` and message passing with the `"!"` notation are useful to define a role type where a method must inherit the method defined in the supertype (see the definition of method `WhoAreYou` in types `Student` and `Athlete`).

Finally, note that upward and double lookup are two different forms of dynamic binding, i.e. in both cases the method which is activated by the message cannot be determined generally at compile time, while static binding to the method of type `T` can be obtained through the `(obj As T)!msg` idiom. Let us consider the following function:

```
let foo := fun(x:Person) :seq string is
            {x.WhoAreYou;
             x!WhoAreYou;
            (x As Person)!WhoAreYou};
```

Let `JohnAsStudent` be bound to a value of type `Student`, which has been later extended with a role of type `ForeignStudent`, a subtype of `Student` which redefines the method `WhoAreYou`. The value returned by `foo(JohnAsStudent)` is a sequence of three answers produced by the methods defined in type `ForeignStudent` (*dynamic binding with double lookup*), by the method defined in type `Student` (*dynamic binding with upward lookup*), and by the method defined in type `Person` (*static binding*).

Method Lookup

When a message `m` is sent to an object `O`, two questions must be answered: (a) which method is used to answer the message? and (b) which is the semantics of the pseudo-variable `self` used in the method selected to answer the message `m`?

In traditional object-oriented languages, with objects that cannot change dynamically their type, the type `T` of an object `O` is fixed at creation time. When a message is sent to `O`, the method is searched first in the type `T`. If none is found, the method is searched in the supertype of `T`, and then the search continues up the supertype chain until the root type. The search will certainly stop because static typechecking ensures that the method has been defined in one of the super-types.

When a method contains a message to `self`, the search for the method for that message begins in the instance's type `O`, regardless of which type contains the method containing `self`.

For instance, let us consider the following object types and the object instances `p1` and `p2`:

```
let CurrentYear := 1997;

let rec
type Person <->
     [Name :string;
      BirthYear :int;
      Age := meth():int is
              CurrentYear - self.BirthYear;
      StringOf := meth():string is
                    "Name is " & self.Name &
                    ". Age is " & stringofint(self.Age) ]
and
type DeadPerson <-> is Person and
     [DeadYear :int;
      Age := meth():int is
              CurrentYear - self.DeadYear;
      StringOf := meth():string is
                    super.StringOf &
                    ". Dead year is " & stringofint(self.DeadYear) ];

let p1:= mkPerson([Name:= "Bob";  BirthYear:= 1950]);
let p2:= mkDeadPerson([Name:= "Jim";
                          BirthYear:= 1950;
                          DeadYear:= 1970]);
```

The results of evaluating the following expressions are:

```
p1.StringOf;  returns "Name is Jim. Age is 47"
p2.StringOf;  returns "Name is Jim. Age is 27. Dead year is 1970"
```

The expressions `p1.StringOf` and `p2.StringOf` show the effect of sending a message to `self`: both invoke the same method `StringOf`, which is found in the type `Person`, but they produce different results because of the message `self.Age` in method `StringOf` of `Person`. In the first case the result is the value of `Age` in `p1`, while in the second case the result is the value of `Age` in `p2`.

Galileo 97 supports objects which can dynamically acquire new types and exhibit plurality of behaviors, and the rules to select the method to be used for answering the message are those described previously: if the message is sent with the dot notation, the method is found with a *double lookup*, if it is sent with the exclamation mark notation, the method is found with a *upward lookup*.

When the method selected by a role `r` to answer a message `m` contains a message whose receiver is `self`, the interpretation of the self-reference depends on how the method has been found:

- if the method has been found in the downward lookup phase, hence in a type `T` which is a subtype of the type of `r`, then `self` is bound to the `r As T` role;
- if the method was found by a search in the supertype chain, then `self` is bound to the `r` role.

As an example of the `self` interpretation rule, let us consider the following definitions (Figure 1):

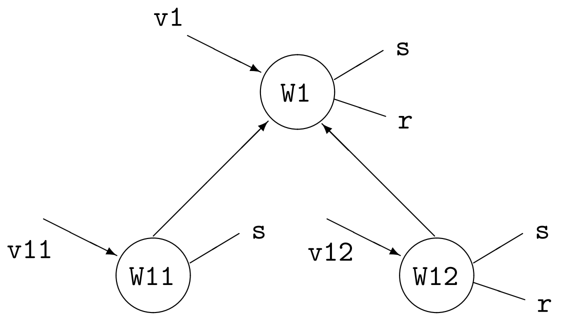

Figure 1: An example of type hierarchy

```
let rec type  W1 <-> [ s := meth() :int is 3;
                       r := meth() :int is self.s ];
let type      W11 <-> is W1 and [ s := meth() :int is 4 ];
let rec type  W12 <-> is W1 and [ s := meth() :int is 5;
                                  r := meth() :int is
                                       2*(self.s) ];
```

Let us construct a value `v1` of type `W11`, and send it the message `v1.r`:

```
let v1 := mkW11([ ]);
vi.r;                returns 4
```

`vi.r` returns 4 because the method for `r` is inherited from `W1`; here `self` is assigned type `W11` (dynamic binding), hence `self.s` returns 4.

Let us extend `v1` with the type `W12`, and send it again the message `v1.r`:

```
let v2 := inW12(v1,[ ]);
v1.r;                returns 10
```

This time the method for `r` is found in `W12` by history search, hence `self` is statically bound to type `W12`, hence `self.s` returns 2*5.

6.1.4 View Type Bindings

The operators on objects presented so far allow objects to be built, and roles to be added and dropped without affecting object identity. These operators allow one to model the dynamic behavior of real-world entities, and are also useful for dealing with the most common kind of schema evolution, i.e. attribute addition or specialization. In this situation, in fact, it is possible to introduce a new subtype of the old type and to extend the old values with the new information. The type correctness of the preexisting applications and data structures will not be affected, and it is even possible to decide to partially modify the behavior of an old application by specializing the behavior of some methods.[6]

However, the role mechanism cannot cope with the related problem of giving different views of the same object without affecting its behavior. This is because object extension actually modifies the object, and object extension can only modify an object in a very limited way (only field addition and specialization). If we call "real objects" those that have been explicitly constructed using the `mkT` or `inT` functions, what is needed is the possibility to define "virtual objects" by starting with objects and changing their interface while preserving their identity.

The virtual object mechanism we are going to describe has the following features:

- a virtual object has the same identity as the object (or objects) it is based on;
- a virtual object can add, remove, and rename fields of its base object; moreover, a virtual object can have its own instance variable, which is accessed by its own methods;
- a virtual object can be based on more than one base object;
- the behavior of an object is not affected by the existence of a related virtual object;
- a virtual object can be used exactly like a real object and vice versa, at least as long as the type rules described later on are satisfied;
- virtual objects allow one to update those components of the state of the real object which the virtual object allows one to view;

[6]More precisely, a preexisting application is affected by method specialization only if the application exploits the double lookup (*obj.msg*) form of message passing.

- a message to a virtual object to execute a method imported from an object returns an answer which is the same as if the message were directly sent to the object.

We will now show how virtual objects are defined, and discuss their types and how these types are related to object and tuple types.

Virtual object types

A virtual object role, which for the sake of simplicity we will sometimes call virtual object or virtual role, can be seen as a pair formed by the base object role and a mapping which may hide, rename, combine, or even add some fields (even state components) with respect to the original object role. More precisely, a virtual object role can generally be based on a set of base objects, with a mapping which manipulates their components and gives the external impression of a unique virtual entity.

Virtual object roles are typed thanks to a new type constructor, called `view`:

$<\mathcal{T}_1,\ldots,\mathcal{T}_m>$ `view` $[\mathtt{A}_1{:}\mathcal{S}_1;\ldots;\mathtt{A}_n{:}\mathcal{S}_n]$

where: $\mathcal{T}_1,\ldots,\mathcal{T}_m$ are role types; and $\mathtt{A}_1,\ldots,\mathtt{A}_n$ are labels; $\mathcal{S}_i$ are types. As a syntactic abbreviation, the type $\mathcal{S}_i$ of a label $\mathtt{A}_i$ can be omitted and it is assumed that it is the type of label $\mathtt{A}_i$ in one of the types $\mathcal{T}_1,\ldots,\mathcal{T}_m$, considered in the order.

Intuitively, the statement:

`O:` $<\mathcal{T}_1,\ldots,\mathcal{T}_m>$ `view` $[\mathtt{A}_1{:}\mathcal{S}_1;\ldots;\mathtt{A}_n{:}\mathcal{S}_n]$

means that `O` is a virtual object based on m object roles with types $\mathcal{T}_1,\ldots,\mathcal{T}_m$, whose signature is $[\mathtt{A}_1{:}\mathcal{S}_1;\ldots;\mathtt{A}_n{:}\mathcal{S}_n]$. As for object roles, the signature of a virtual object role is a sequence of label-type pairs which determines all its components, both proper and inherited, i.e. the messages which can be sent to the object (when the type is a functional type), and the components of its state (non functional types).

For this reason, for any object role type $\mathcal{T}$ with signature $\Downarrow\mathcal{T}$, the following type equivalence holds:

$<\mathcal{T}>$ `view` $[\Downarrow\mathcal{T}] \sim \mathcal{T}$

where $\mathcal{T}{\sim}\mathcal{S}$ means that $\mathcal{T}{\subseteq}\mathcal{S}$ and $\mathcal{S}{\subseteq}\mathcal{T}$, i.e. that values of each type can be considered as if they were values of the other one. The subtype relationship among real and virtual object types will be defined in Section 7.

Virtual object constructors

The operators to build virtual objects are: `project`, `rename`, `extend` and `times`.

These operators can be applied to sequences of objects using the notation `project*`, `rename*`, `extend*` and `times*`.

Project

`project` is used to hide properties from an object role.

`O project` $[\mathtt{A}_1\texttt{:}\mathcal{S}_1\texttt{;}\ldots\texttt{;}\mathtt{A}_n\texttt{:}\mathcal{S}_n]$

returns the object `O` with components $\mathtt{A}_1,\ldots,\mathtt{A}_n$ only. More precisely, if `O` is an object role with type

$<\mathcal{T}_1,\ldots,\mathcal{T}_m>$ `view` $[\mathtt{A}_1\texttt{:}\mathcal{S}'_1\texttt{;}\ldots\texttt{;}\mathtt{A}_n\texttt{:}\mathcal{S}'_n\texttt{;}\ \mathtt{B}_1\texttt{:}\mathcal{R}_1\texttt{;}\ldots\texttt{;}\mathtt{B}_l\texttt{:}\mathcal{R}_l]$,

and if each $\mathcal{S}'_i$ is a subtype of $\mathcal{S}_i$, then `O project` $[\mathtt{A}_1\texttt{:}\mathcal{S}_1\texttt{;}\ldots\texttt{;}\mathtt{A}_n\texttt{:}\mathcal{S}_n]$ returns the same object role `O` seen through type

$<\mathcal{T}_1,\ldots,\mathcal{T}_m>$ `view` $[\mathtt{A}_1\texttt{:}\mathcal{S}_1\texttt{;}\ldots\texttt{;}\mathtt{A}_n\texttt{:}\mathcal{S}_n]$.

The type $\mathcal{S}_i$ of a label $\mathtt{A}_i$ can be omitted, and it is then assumed to be the type $\mathcal{T}_i$ of $\mathtt{A}_i$ in `O`.

Notice that, even if `project` is formally defined on virtual objects alone, it can also be applied to real objects too, thanks to the type equivalence

$<\mathcal{T}>$ `view` $[\Downarrow\mathcal{T}] \sim \mathcal{T}$;

the same observation holds for all the other virtual object operators that will be presented, which can all be applied in the same way to real and virtual objects to produce virtual objects.

Example 1 *Let us consider the following definitions:*

```
let type
AnAddress := [ Street: var string;
               City: var string;
               Zip: var string;
               Country: var string ];

let rec type
Person <->
          [ Name: string;
            Income: var int;
            Address: AnAddress;
            BirthYear: int;
            Parents: [Father: optional Person;
                      Mother: optional Person ];
            WhoAreYou:= meth() :string is
                        "My name is  " & self.Name
          ];

let rec type
Employee <-> is Person and
          [ Salary: var int;
            Company: Company;
            WhoAreYou:= meth() :string is
                        super.WhoAreYou &
                        "I work with company" &
                        self.Company.Name
          ]
and type
Company <-> [ Name: string;
              Location: string;
              Revenue: string ];

let type PersonView :=
         <Person> view [ Name;
                         Address: [Street: var string;
                                   City: var string ];
                         WhoAreYou ];
```

Let `John` *be an object role of type* `Person`*,* `JohnView` *a virtual role of type* `PersonView`*,* `Foo` *and* `Goo` *two functions defined as follows:*

```
let JohnView :=
      John project [ Name;
                     Address: [Street: var string;
                               City: var string
                              ];
                     WhoAreYou];
let Foo := fun (x: Person) ...
let Goo := fun(x: PersonView):int is
              if x isalso Employee
                 then at (x As Employee).Salary
                 else at (x As Person).Income;
```

The following considerations apply:

- *By projecting* `Address` *to the indicated subtype of the original type, its* `Zip` *and* `Country` *components are hidden;*
- `John = JohnView` *returns true since the two identifiers denote the same object, even if* `John` *is seen through a view;*
- `John.WhoAreYou = JohnView.WhoAreYou` *returns true since the method executed to answer the message* `WhoAreYou` *sent to* `JohnView` *is the method defined for* `John`*;*
- *the component* `Street` *of the* `Address` *of* `JohnView` *can be updated and this will also effect* `John`*. Likewise, an update of* `John` *will have the same effect on* `JohnView`*;*
- *as will be explained in Section 7,* `Person` *is a subtype of* `PersonView`*, but not vice versa, hence* `Goo(John)` *would be typed, while* `Foo(JohnView)` *would not;*
- *let us assume that* `John` *has been extended with the role type* `Employee`*. The following virtual object:*

  ```
  let JohnEmplView :=
        (John As Employee)
        project [ Name;
                  Company: [Name: string;
                            Location: string ];
                  WhoAreYou];
  ```

has a type which is a supertype of `Employee`*, but it is not comparable with either* `Person` *or* `PersonView`*.*

□

Extend

`extend` is used to redefine an object structure and behavior by adding new fields with a computed value.

$\texttt{O extend [A}_1\texttt{:}\mathcal{S}_1\texttt{:=Expr}_1\texttt{;...;A}_n\texttt{:}\mathcal{S}_n\texttt{:=Expr}_n\texttt{; A}_{n+1}\texttt{:=Meth}_{n+1}\texttt{;...]}$

returns the object `O` extended with new fields whose value is specified by the expressions $\texttt{Expr}_i$ or by the methods $\texttt{Meth}_i$. If a label $\texttt{A}_i$ is present in `O`, `extend` overrides $\texttt{A}_i$ with a value of a possibly unrelated type. More precisely, if `O` has type

$\texttt{<}\mathcal{T}_1\texttt{,...,}\mathcal{T}_m\texttt{> view [B}_1\texttt{:}\mathcal{R}_1\texttt{;...;B}_l\texttt{:}\mathcal{R}_l\texttt{;A}_{\sigma(1)}\texttt{:}\mathcal{S}_{\sigma(1)}\texttt{;...;A}_{\sigma(k)}\texttt{:}\mathcal{S}_{\sigma(k)}\texttt{]}$

then the extension expression above has type:

$\texttt{<}\mathcal{T}_1\texttt{,...,}\mathcal{T}_m\texttt{> view [B}_1\texttt{:}\mathcal{R}_1\texttt{;...;B}_l\texttt{:}\mathcal{R}_l\texttt{;A}_1\texttt{:}\mathcal{S}_1\texttt{;...;A}_n\texttt{:}\mathcal{S}_n\texttt{]}.$

Note that this rule allows one to extend both real and virtual objects.

`extend` can be used to add both new fields and new methods, which belong to the virtual part of the object. To this aim, the expression associated with a label may contain the pseudo-variable `me`, which denotes, recursively, the whole virtual object *after* it has been extended. This `me` variable can be used much the same as `self` in the real object, but there is a difference. When a method defined in a role $\mathcal{R}$ but activated by inheritance by a message sent to a subrole $\mathcal{S}$, sends a message `msg` to `self`, then method lookup for `msg` starts from role $\mathcal{S}$. When any expression, message passing included, is applied to `me`, then `me` denotes the virtual object that has been created by the `extend` operation it is bound to, hence method lookup for `me.msg` starts from the virtual object where the method invoking `me.msg` is found. Technically, we say that `self` is *dynamically bound* to the role that receives the message, while `me` is *statically bound* to the virtual object that is created by the `extend` expression which `me` is bound to.

Example 2 *The following example shows how to redefine the structure and behavior of a person object; note that, in the definition of fields* `Mother` *and* `Father`*, using* `me` *or* `John` *makes no difference, while* `me` *is essential to access field* `Age` *inside* `WhoAreYou`*.*

```
let CurrentYear := 1997;
let rec AnotherJohnView :=
  (John extend [ Age := meth() :int is
                        CurrentYear - me.BirthYear;
                Mother:optional Person := me.Parents.Mother;
                Father:optional Person := me.Parents.Father;
                WhoAreYou := meth() :string is
                              (me As Person)!WhoAreYou &
                              "I am " & stringofint(me.Age) &
                              " years old"
              ]
   ) project [Name; Age; Mother; Father; WhoAreYou];
```

□

Rename

`rename` is used to change the name of properties of an object. If `O` has the labels $\mathtt{A}_1$,...,$\mathtt{A}_n$,$\mathtt{B}_1$,...,$\mathtt{B}_l$,

$\mathtt{O\ rename\ (A_1\ =>\ A'_1;\ \ldots;\ A_n\ =>\ A'_n)}$

returns `O` with the labels $\mathtt{A'}_1$, ..., $\mathtt{A'}_n$, $\mathtt{B}_1$, ..., $\mathtt{B}_l$, which must all be different. A label $\mathtt{A}_i$ may also be a path expression $\mathtt{A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A}_i^{n_i}$.

More precisely, if no $\mathtt{A}_i$ is a path expression and `O` has type:

$\mathtt{<}\mathcal{T}_1,\ldots,\mathcal{T}_m\mathtt{>\ view\ [B_1:}\mathcal{R}_1\mathtt{;\ \ldots;\ B_l:}\mathcal{R}_l\mathtt{;\ A_1:}\mathcal{S}_1\mathtt{;\ \ldots;\ A_n:}\mathcal{S}_n\mathtt{]}$

and $\mathtt{A'}_1$, ..., $\mathtt{A'}_n$, $\mathtt{B}_1$, ..., $\mathtt{B}_l$, contains no duplicate, then the renaming expression above has type:

$\mathtt{<}\mathcal{T}_1,\ldots,\mathcal{T}_m\mathtt{>\ view\ [B_1:}\mathcal{R}_1\mathtt{;\ \ldots;\ B_l:}\mathcal{R}_l\mathtt{;\ A'_1:}\mathcal{S}_1\mathtt{;\ \ldots;\ A'_n:}\mathcal{S}_n\mathtt{]}.$

More generally, when an $\mathtt{A}_i$ is a path expression, then it is required that $\mathtt{O.A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A}_i^{n_i}:\mathcal{T}$ for some $\mathcal{T}$, and, in this case,

$$\mathtt{O\ rename\ (A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A}_i^{n_i}\ \mathtt{=>\ A'}_i\mathtt{).A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A'}_i\mathtt{:}\quad \mathcal{T}.$$

It is required that, after renaming, no duplicate labels exist in the value $\mathtt{O.A}_i^1.\mathtt{A}_i^2.\ldots.\mathtt{A}_i^{n_i-1}$.

Example 3 *The following definition gives a view of a person object for Italian users:*

```
let rec
type UnIndirizzo := [ Via : var string;
                      Citta : var string]
and
JohnViewForItalians :=
        ((JohnView extend [ Indirizzo :UnIndirizzo :=
                                       [ Via :=  me.Address.Street;
                                         Citta := me.Address.City ];
                            Presentati := meth() :string is
                                          "Mi chiamo " & me.Name
                          ]
         )  rename (Name => Nome)
        )  project [Nome; Indirizzo; Presentati];
```

□

It may appear that `rename` is an operator that can be defined in terms of `extend` and `project`; for example; let `O` be an object with an attribute $\mathtt{A}_i$:

```
let r := O rename ( Ai => Ai' );

let r' := (O extend [ Ai' := meth() :... is me.Ai ])
               project [ <all attributes except Ai> ];
```

Actually, the two expressions above have a different meaning, since both `r'.A'`$_i$ and `r'!A'`$_i$ are equivalent to `O.A`$_i$, while `r.A'`$_i$ and `r!A'`$_i$ are generally different since `r` keeps upward and double lookup distinct.

For instance, let us assume that `John` has the type `Person`, and it has been extended with the type `Student`, where the method `WhoAreYou` has been redefined. Then the message `John.WhoAreYou` is answered using the method defined in `Student`, while `John!WhoAreYou` is answered using the method defined in `Person`. Now let us define two views:

```
let JohnViewOne:= John rename (WhoAreYou => IntroduceYourself);

let JohnViewTwo:= (John extend [ IntroduceYourself :=
                                    meth() :string is me.WhoAreYou ]
                  ) project [ <all attributes except WhoAreYou> ];
```

While the effect of `JohnViewOne.IntroduceYourself` and `JohnViewOne!IntroduceYourself` is different as it was for `John`, `JohnViewTwo.IntroduceYourself` and `JohnViewTwo!Introduce-Yourself` return the same value as `John.WhoAreYou`.

Times

`times` is used to create a virtual object by starting from two objects whose component names are all different.

```
O times O'
```

returns an object which contains the identities of both `O` and `O'`, and has the fields of `O` and `O'`.

More precisely, if:

O: <$\mathcal{T}_1$,...,$\mathcal{T}_m$> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$]
O': <$\mathcal{T}'_1$,...,$\mathcal{T}'_l$> view [B_1:$\mathcal{R}_1$;...;B_k:$\mathcal{R}_k$]

then `O times O'` has type:

<$\mathcal{T}_1$,...,$\mathcal{T}_m$,$\mathcal{T}'_1$,...,$\mathcal{T}'_l$> view [A_1:$\mathcal{S}_1$;...;A_n:$\mathcal{S}_n$;B_1:$\mathcal{R}_1$;...;B_k:$\mathcal{R}_k$].

For example, if $\mathcal{T}$ and $\mathcal{T}'$ are two real object role types with no common component names, then

O times O': <$\mathcal{T}$,$\mathcal{T}'$> view [$\Downarrow\mathcal{T}$;$\Downarrow\mathcal{T}'$].

This type indicates that the virtual object (a) can answer all the $\Downarrow\mathcal{T}$, $\Downarrow\mathcal{T}'$ messages, and (b) contains both a $\mathcal{T}$ and a $\mathcal{T}'$ object role, which can be recovered with the `obj As` $\mathcal{T}$ operator previously defined.

Example 4 *Let us consider the following object type definitions:*

```
let rec type
   Person <->
          [ Name: string;
            BirthYear: int;
            WhoAreYou:= meth() :string is
                        "My name is  " & self.Name
          ];

let rec type
   Student <->
        is Person and
          [ SNumber: string;
            Faculty: string;
            WhoAreYou := meth() :string is
                         super.WhoAreYou &
                         ". I am a student of  " &
                         self.Faculty
          ];

let rec type
   Athlete <->
        is Person and
          [ Code: int;
            Sport: string;
            WhoAreYou := meth() :string is
                         super.WhoAreYou &
                         ". I practice  " &
                         self.Sport
          ];
```

Let `s` *be an object with role types* `Student` *and* `Athlete`*. The following definition defines a virtual object which is a subtype of both the type* `Student` *and the type* `Athlete`*.*

```
let AStudentAthlete :=
        ( (s project [Name;  BirthYear; SNumber; Faculty] )
          times
         ((s As Athlete) project [Code; Sport] )
        ) extend [ WhoAreYou:= meth() :string is
                               (me As Student).WhoAreYou &
                               ". I practice  " & me.Sport
                 ];
```

□

6.1.5 Parallel Type Bindings

A set of type bindings can be introduced with the multiple type declaration D_1 `and` D_2. This construct exports the bindings exported by D_1 and by D_2. It is illegal to declare the same type identifier both in D_1 and D_2.

```
syntax:       ParallelTypeBinding ::=
                        TypeBinding and TypeBinding
```

```
let type Color := (| red or green or white |)
and type CardSymbol := (| hearth or club or diamond or spade |);

>  type Color = (| red or green or white |)
|  type CardSymbol = (| hearth or club or diamond or spade |)
```

6.1.6 Recursive Type Bindings

The operator `rec` is used to define recursive and mutually defined *semi-abstract* types. *Concrete types cannot be defined recursively*. The binding `rec TypeBinding` exports the identifiers exported by *TypeBinding*, and imports the identifiers imported by *TypeBinding* and the identifier exported by *TypeBinding*.

```
syntax:      RecursiveTypeBinding ::=
                  rec TypeBinding
```

```
let rec type
Person <->
  [ Age:= meth() :int is
         CurrentYear - self.BirthDate.Year;
    BirthDate: Date;
    Name: string;
    Children: var seq Person];

mkPerson(
   [Name := "Smith";
     BirthDate:=
        mkDate([Year:=1945; Month:= 3; Day:= 1]);
     Children := var({ } :seq Person)] );

   - : Person
```

A method can be defined in terms of the value of other attributes, using the identifier `self` to denote the current object value. In this case, the type definition must be recursive.

6.2 Value Bindings

Value bindings define values:

```
syntax:     ValueBinding ::=
                  [private] Ide := Exp
                  [private] Ide :=  derived Exp
```

The first declaration `Ide := Exp` import the identifiers used in `Exp` and export the identifier `Ide`.

`Ide := derived Exp` returns an environment where the only binding is between `Ide` and a "virtual" value, obtained by evaluating `Exp` whenever `Ide` is evaluated.

In a recursive or sequential declaration a value binding can be declared private (see Section 6.8).

```
let x := 3;

>   x = 3 : int

let v := derived x*x;

>   v = - : int

 v;

    9 : int
```

6.3 Bounded Polymorphic Functions

```
syntax:      let Id := fun[X1<:W1; ...;Xn<:Wn]
                         (Id1:X1, ...,Idn:Xn): T  is E     n > 0
typing rule: if X1 ... Xn are distinct And E : T
             then Id : all[X1<:W1; ...;Xn<:Wn]
                         (X1 # ... # Xn) -> T
```

We have seen the usefulness of subtyping and parametric polymorphism in different context, we now show that it is useful, and sometimes necessary, to merge them.

Let us consider the following identity function `PersonId`:

```
let PersonId := fun(p :Person):Person is p;

PersonId (Bob);

   - : Person
```

when `PersonId` is applied to an object `Bob` of type `Student`, a subtype of `Person`, it returns a value of type `Person` and so we lose information.

This problem can be solved by defining `PersonId` as polymorphic in the subtypes of `Person`; it takes a type which is any subtype of `Person`, then a value of that type, and returns a value of that type. For example, let `John` of type `Person`, `Bob` of type `Student`, and `Jim` of type `Athlete`, with `Student` and `Athlete` subtypes of `Person`:

```
let PersonId := fun[X <: Person](p :X):X is p;

PersonId [Person] (John);

   - : Person
```

```
PersonId [Student] (Bob);

   - : Student

PersonId [Athlete] (Jim);

   - : Athlete
```

Another example is the function `QuickSortOnK` to order sequences `s` with elements of type record `X` which have at least one attribute `k` of any type, using the function `QuickSort` defined previously in Section 4.12.1: In this case, the bounded parametric polymorphism allows one to express the condition that the elements of `s` can have any record type `X` which contains a field `k` of the type `Y` expected by the comparison function `less`. Note that the comparison function to pass to `QuickSort` can be created "on the fly" using the construct `fun ...is ...`.

```
let rec
QuickSortOnK := fun [Y<:any; X<:[k:Y]]
                    (s: seq X, less:(Y#Y) -> bool): seq X is
                QuickSort[X](s,
                             fun(x:X, y:X):bool is less(x.k,y.k)) ;

> QuickSortOnK = fun
| : all [Y <: any; X <: [ k :Y ] ] ( seq X # ( Y # Y )  -> bool )
        -> seq X

let LessThan := fun(x:int, y :int):bool is x < y;

> LessThan = fun
| : ( int # int )  -> bool

let rec
StrLessThan:= fun(x:string, y:string):bool is
   use rec lessThanAscii:= fun(x:seq int, y:seq int):bool is
           if emptyseq(x) then true
           else if emptyseq(y) then false
           else if first(x) = first(y)
                then lessThanAscii(rest(x), rest(y))
                else if first(x) < first(y)
                     then true
                     else false
   in lessThanAscii(explodeascii(x), explodeascii(y));

> StrLessThan = fun
| : ( string # string )  -> bool

QuickSortOnK [int; [k:int; v:string]]
                 ({[k := 4; v := "a"];
                   [k := 2; v := "b"];
                   [k := 3; v := "c"]}, LessThan);
```

```
{ [ k := 2; v := "b" ] ;
  [ k := 3; v := "c" ] ;
  [ k := 4; v := "a" ]  }
  : seq [ k :int; v :string ]

QuickSortOnK [string; [v:int; k:string]]
                  ({[v := 4; k := "a"];
                    [v := 2; k := "b"];
                    [v := 3; k := "c"]}, StrLessThan);

{ [ v := 4; k := "a" ] ;
  [ v := 2; k := "b" ] ;
  [ v := 3; k := "c" ]  }
  : seq [ v :int; k :string ]
```

In an object type definition a method can be defined polymorpic, as it is shown in the following trivial example:

```
let rec
Persons class
   Person <->
    [Name :string;
     Homonym := meth[X<:Person](a:X):bool
                is a.Name = self.Name];
```

6.4 Class Bindings

The word "class" is usually used with the meaning of "type" in object-oriented languages, but here it will be used with a different meaning. A class is a modifiable sequence of elements and a class definition has two different effects:

- it introduces the definition of the type T of its elements, which must be defined with the <-> constructor, therefore two elements of different classes have always different types although they may be defined with the same representation type (*intensional aspect*);
- it supplies a name to denote the modifiable sequence of the elements of type T currently in existence in the database (*extensional aspect*).

Classes provide a mechanism to represent a data base by means of modifiable collections of interrelated objects. An element of a class is an object which is the computer representation of certain facts of an entity of the world that is being modeled. Associations among entities are modeled by relating the corresponding objects, and not through the use of keys or other external references.

Besides semi-abstract types values characteristics, elements of classes can be destroyed, and can be constrained to be uniquely identified by certain attributes (keys).

Each class can be a *base class* or a *subclass*. A base class is defined independently of other classes; subclasses combine the class concept with that of type hierarchies and will be treated in the next section.

A class is introduced with the following declaration:

```
syntax:     ClassBinding::=
                   Id1 class
                       Id2 <-> Type
                       [BeforeMk] [BeforeDrop]
                   [key (Ide, ...,Ide)]
```

The `BeforeMk`, `BeforeDrop`, and `key` parts can be omitted. The declaration introduces the following bindings:

1. `Id2` is bound to a new semi-abstract type;
2. `Id1`is bound to a modifiable sequence of value of type `Id2`;
3. the identifiers `mkId2` and `repId2` are bound to two primitive functions similar to those for semi-abstract types with the difference that the function `mkId2` also inserts the constructed element into the modifiable sequence bound to the name of the class. When a value loses the type `Id2` it is also removed from the corresponding class.

The key constraint asserts that elements in a class must differ in the value of certain constant attributes. Note that if the key constraint is not specified, the insertion will be made even though the values of the attributes are equal to those of another object already present in the class. That is, elements in classes are always distinct objects, but the construction of an element will fail when the constraints are violated.

Since the name of a class denotes a sequence of all the elements of the class member type in the database, all the operators on sequences can be applied to classes.

```
let rec
Persons class
Person <->
   [ Name: string;
      BirthYear : int;
      Address: [PermanentAddress: string ];
      Age:= meth() :int is
            CurrentYear - self.BirthYear ]
key (Name);

mkPerson([Name:="Smith"; BirthYear:=1964;
            Address:=[PermanentAddress:="14 St, New York"] ]);

   - : Person
```

6.5 Subclass Bindings

Classes can be organized into a *subset* hierarchy. Let `C2` be a subclass, with member type `T2`, defined in terms of `C1`, with member type `T1`. `C1` can be either a base class or a subclass.

The following constraints hold:

1. *Structural constraint*: `T2` is defined by inheritance from `T1`, consequently `T2` $\subseteq$ `T1`.

2. *Extensional constraint*: If `c` is an element in `C2` then `c` is also an element in `C1`.

3. *Integrity constraint*: `C2` inherits the triggers and the key constraint in `C1`.

```
syntax:       SubClassBinding ::=
                   Id1 subset of Id2 class
                       Id3 <-> is Id4 and [ NewAttributes ]
                       [BeforeMk] [BeforeDrop] [BeforeIn]
                   [key (Ide, ..., Ide)]
```

`[ NewAttributes ]` may be the empty record `[ ]`.

The declaration introduces the same bindings as for base classes with the following differences:

1. To satisfy the extensional constraint, `mkId3` inserts the created value both in the subclass `Id1` and in the superclass `Id2`. In the current implementation `mkId3`, evaluates `mkId4` first, then it checks assertions on the subtype `Id3`, and finally inserts the value created in the superclass by `mkId4` in the subclass `Id1`. If a failure is generated, the value created in the superclass by the `mkId4` is removed.

2. `inId3` inserts into the subclass `Id1` a value already present in the superclass `Id2`.

3. When the type `T` is dropped from a value, the value is removed from the corresponding class and its subclasses.

```
let Students subset of Persons class
    Student <->
       is Person and
            [ StudentCode: int;
               Address:   [PermanentAddress: string;
                            TemporaryAddress: string] ]
    before mkStudent(this)
    if Not this.StudentCode > 0
    do failwith "StudentCode must be > 0"
    key (StudentCode);


inStudent
    ((get Persons where Name = "Smith"),
     [StudentCode := 253;
       Address:=
         [PermanentAddress := "14 St, New York";
           TemporaryAddress := "Walker Road, Berverton" ] ]);

   - : Student
```

6.6 Parallel Declarations

A set of bindings can be introduced with the multiple declarations D_1 `and` D_2: it exports the bindings of D_1 and of D_2. The construct is "parallel" in the sense that the bindings of D_1 are not imported by D_2 and vice versa. It is illegal to declare the same identifiers both in D_1 and D_2.

```
syntax:    ParallelDeclaration ::=
                   Declaration and Declaration
```

```
let type PositiveInt <-> int
         before mkPositiveInt(this)
         if Not this > 0
         do failwith "PositiveInt must be > 0"
and Departments class
    Department <-> [Name: string; Budget: int];
```

6.7 Recursive Declarations

The environment operator `rec D` permits recursive definitions: it exports the identifiers exported by `D`, and the bindings of `D` are imported by `D`.

```
syntax:    RecursiveDeclaration ::=
                   rec Declaration
```

```
let rec Factorial := fun(x: int): int is
            if x = 0 then 1 else x * Factorial(x-1);

>  Factorial = fun : int -> int

use x := 10 in Factorial(x);

   3628800 : int
```

Note that if `rec` were omitted, the identifier `Factorial` on the right of `:=` would not refer to its defining occurrence on the left of `:=`, but to some previously defined identifier (if any) in the surrounding environment.

A recursive declaration behaves correctly if used to define functions, classes, semi-abstract types, and recursive values of semi-abstract types; it cannot be used to defined recursive concrete values and types. For example, let us define a mandatory one-to-one association between two sets of objects:

```
let rec
Nations class
   Nation <->
      [ Name :string;
```

```
        Capital :Capital]
   key(Name)
and
Capitals class
   Capital <->
      [ Name :string;
        Nation :Nation]
   key(Nation);
```

To guarantee the properties of the association, the class elements must be created and eliminated in a single operation by redefining the operators `mk` and `drop` as follows:

```
let mkNationCapital:=
       fun(NNation:string, NCapital :string)
       :Nation # Capital is
       use rec
       TheNation := mkNation( [ Name := NNation;
                                Capital := TheCapital ] )
       and
       TheCapital := mkCapital( [ Name := NCapital;
                                  Nation := TheNation ] )
       in (TheNation, TheCapital);

let mkNation := nil;
let mkCapital := nil;

let dropNation :=
      fun(x: Nation) :null
      is (dropCapital(x.Capital);dropNation(x));

let dropCapital :=
      fun(x: Capital) :null
      is dropNation(x.Nation);
```

Note that a parallel declaration D_1 **`and`** D_2, with D_2 a recursive declaration `rec` D_3, must be written as D_1 **`and`** $\{$| `rec` D_3 |$\}$, since the binding power of the environment operators is **`and`** > `rec`.

6.8 Sequential Declarations

Cascaded, or *sequential*, declarations are provided by the environment operator `ext`. The latter differs from **`and`** in that, in the expression D_1 `ext` D_2, D_2 is evaluated in the current environment extended with the bindings of D_1. If an identifier is defined both in D_1 and D_2, then its value in the resulting environment will be that evaluated in D_2.

```
syntax:    SequentialDeclaration ::=
                    Declaration ext Declaration
```

```
let type ABirthYear <-> int
         before mk(this)
         if Not this>0
         do failwith "a birth year must be >0 "
ext rec type Person <->
               [ Name: string;
                 BirthYear: ABirthYear;
                 Age:= meth() :int is
                       CurrentYear - self.BirthYear ];
```

If an identifier is defined as `private` in D_1, then it can be used in D_2, but it is undefined in the resulting environment. For example the following definition:

```
let private y :=  5
ext x := derived 3 + y
and z := y * y;
```

introduces in the current environment only the bindings between `x` and the computed value `3 + y`, and `z = 25`, while the private identifier `y` is undefined outside the declaration.

The binding power of the environment operators is `and` > `rec` > `ext`.

7 Type Hierarchies

The Galileo 97 type system supports *inclusion polymorphism*: i.e. a subtype relation is defined on types such that if a type `t` is a subtype of `u` (written also `t` ⊆ `u`), then any value of type `t` can be used where a value of type `u` is expected; consequently

- Functions with argument of type `u` also accept a value of type `t` (*substitutability principle*).
- Control structures with components of type `u` also accept values of type `t` (e.g., the condition of an `if` expression may have a value whose type is a subtype of `bool`).
- Control structures which require a set of values of the same type (the values of the sequence constructor and the branches of conditional and case expressions), also accept values with a common supertype, if each value has a type comparable to the most general type of the previous ones. In this case, the resulting type is the most general type.

Subtyping for concrete types is only determined by the structure of type terms, while for semi-abstract types it is determined both by the structure of the representation types, and by an explicit user declaration.

The rules describing when two types are included, are as follows:

- Every type is included in itself.
- Every type is included in `any`.
- `int` is a subtype of `real`.
- The type `none` is included in every type.
- If `r` and `s` are record types, then `r` ⊆ `s` iff the set of identifiers of `r` *contains* the set of identifiers of `s`, and, if `r'` and `s'` are the types of a common identifier, then `r'` ⊆ `s'`. For instance the type

  ```
   [ Name:string;
     Address:[City:string; Street:string ] ]
  ```

  includes

  ```
  [ Name:string;
    Age:int;
    Address:[City:string; Street:string; Country:string ]]
  ```

- If `r` and `s` are variant types, then `r` ⊆ `s` iff the set of identifiers of `r` *is contained* in the set of identifiers of `s`, and, if `r'` and `s'` are the types of a common identifier, then `r'` ⊆ `s'`. For instance the type `(| Integer:int or Boolean:bool |)` includes `(| Integer:int |)`.

- If `seq r` and `seq s` are sequence types, then `seq r` ⊆ `seq s` iff `r` ⊆ `s`.
- A modifiable type `var r` is a subtype of another type var `var s` iff `r` and `s` are the same type.
- If `(r -> s)` and `(r' -> s')` are function types, then `(r -> s)` ⊆ `(r' -> s')` iff `r'` ⊆ `r` and `s` ⊆ `s'`. For instance the type

```
seq [Name:string; Age:int] -> [Name:string]
```

  includes

```
seq [Name:string] -> [Name:string; Count:int].
```

- if T_1 = `all[X1; ...;Xn] r -> s` and T_2 = `all[Y1; ...;Yn] r' -> s'` are polymorphic function types, then $T_1 \subseteq T_2$ iff `(r -> s)` ⊆ `(r' -> s')[Y1 <- X1, ..., Yn <- Xn]`, where `T[Y <- X]` denotes the substitution of `X` for all the free occurrence of `Y` in `T`.
- if T_1 = `all[X1<:Z1; ...;Xn<:Zn] r -> s` and T_2 = `all[Y1<:W1; ...;Yn<:Wn] r' -> s'` are bounded polymorphic function types, then $T_1 \subseteq T_2$ iff `W1` ⊆ `Z1`, ..., `Wn` ⊆ `Zn` and `(r -> s)` ⊆ `(r' -> s')[Y1 <- X1, ..., Yn <- Xn]`, where `T[Y <- X]` denotes the substitution of `X` for all the free occurrence of `Y` in `T`.
- Let `T` and `T'` be semi-abstract types. `T` ⊆ `T'` if the subtype relation is declared explicitly as follows:

  `type` $\mathcal{T}$ `<-> is` $\mathcal{T}'$ [`NewAssertions`]

  The assertions on `T` are those of `T'` and `NewAssertions`.
- A type `T <-> is T' and Tr NewAssertions`, is a subtype of `T'`. The assertions on `T` are those of `T'` and `NewAssertions`.

  For instance the type

```
type Thing <-> [Name:string]
```

  includes

```
type LivingThing  <-> is Thing and [BirthDate:Date]
```

- A type `T <-> Tr` is a subtype of `Tr`. If `Tr` is a record type with private properties, `T <-> Tr` is a subtype of `Tr` without the private properties. For example:

```
let rec
type Rectangle <->
        [ private base :int;
          private height :int;
          perimeter := meth() :int is
                          (self.height + self.base)*2;
          getBase := meth() :int is self.base;
          getHeight := meth() :int is self.height
         ]
      before mkRectangle(this)
      if Not this.base > 0 And this.height > 0
      do failwith "height and  base must be > 0" ;
```

is a subtype of

```
[ perimeter :int;
  getBase   :int;
  getHeight :int
 ]
```

- Type hierarchies with `view` types. A relation $\mathcal{T}$'$\subseteq\mathcal{T}$ means that any operation which can be applied to any value of type $\mathcal{T}$ can also be applied to any value of type $\mathcal{T}$'. A virtual object `O` with type:

  $\mathcal{T}$`= <`$\mathcal{T}_1$`,...,`$\mathcal{T}_m$`> view [`$\mathtt{A}_1$`:`$\mathcal{S}_1$`;...;`$\mathtt{A}_n$`:`$\mathcal{S}_n$`]`

  can be submitted to two kinds of operations:

  – message passing: `O.`$\mathtt{A}_i$, `O!`$\mathtt{A}_i$;
  – object extraction: `O As` $\mathcal{T}_j$.

  Hence, a type $\mathcal{T}$' is a subtype of $\mathcal{T}$ if it contains enough object identities and components to be able to deal with every object extraction and message passing operation which is supported by $\mathcal{T}$. More precisely:

  `<`$\mathcal{T}$`'`$_1$`,...,`$\mathcal{T}$`'`$_m$`> view [`$\mathtt{A}_1$`:`$\mathcal{S}_1$`;...;`$\mathtt{A}_n$`:`$\mathcal{S}_n$`]`
  $\subseteq$ `<`$\mathcal{T}_1$`,...,`$\mathcal{T}_r$`> view [`$\mathtt{B}_1$`:`$\mathcal{R}_1$`;...;`$\mathtt{B}_s$`:`$\mathcal{R}_s$`]`

  if (a) for each $\mathcal{T}_i$ exists a $\mathcal{T}$'$_i$ such that $\mathcal{T}'_i \subseteq \mathcal{T}_i$ and (b) for each pair $\mathtt{B}_i$:$\mathcal{R}_i$ there is a pair $\mathtt{A}_j$:$\mathcal{S}_j$ such that $\mathtt{A}_i = \mathtt{B}_j$ and $\mathcal{S}_i \subseteq \mathcal{R}_j$.

  The principle that an object view type specifies the object extraction and message passing operations which can be applied to a type, implies that the following two equivalences hold:

  $\mathcal{T}$ $\sim$ `<`$\mathcal{T}$`> view [`$\Downarrow\mathcal{T}$`]`
  `[`$\mathtt{A}_1$`:`$\mathcal{S}_1$`;...;`$\mathtt{A}_n$`:`$\mathcal{S}_n$`]` $\sim$ `<> view [`$\mathtt{A}_1$`:`$\mathcal{S}_1$`;...;`$\mathtt{A}_n$`:`$\mathcal{S}_n$`]`

It should be noted that the types `none` and `any` do not represent a relaxation of the strict typing of Galileo 97, but they can only be used for very particular purposes. In fact, functions which receive a parameter of type `any`, can be given any value as an actual parameter, but cannot extract any information from it, while a function with a parameter of type `none` can apply any operation on its parameter, but can receive as an actual value only the value `unknown`.

8 Equality

With object databases that support object identity there are three kinds of equality:

- *Identity equality* (*identical*) ("`==`"). This corresponds to the equality of references or pointers in conventional languages: two objects are identical if their identities are the same.
- *Shallow equality* ("`=`"). Two objects are shallow equal if they have the same run–time type and their states are identical. That is, it goes one level deep, and compares corresponding identities of the state components.
- *Deep equality* (*deep equal*). This is a purely value-based deep equality: two objects are deep equal if they have the same run–time type and their states are value-based deep equal.

In Galileo 97, every type is associated with an equality function, and the equality of two values is determined using the equality function associated with their static type. More precisely, `a = a'` is well typed if the type of `a` is either a supertype or a subtype of the type of `a'`, and `a` and `a'` are compared using the equality function associated with the supertype. As a consequence, the same pair of values can be equal or different according to the type it is accessed through. For example, the following two expressions would evaluate to `false` and `true`, respectively, since the `b` field would be ignored in the second comparison:

```
[a:=1; b:=2] = [a:=1; b:=3] ==> false
[a:=1; b:=2]:[a:int] = [a:=1; b:=2] ==> true
```

Equality on concrete data types operands (with the exception of updatable references, and functions) is *structural equality*: two values are equal when they are the same atomic object, or all their components are equal.

Equality on updatable reference values and functions is *identity* or *sameness*, i.e. pointer equality (e.g. `var 3 = var 3` is false, because a new location is created every time `var` is used).

Equality on semi-abstract data types operands is *sameness*.

The situation is slightly more complex with `view` types, which are to some extent intermediate between object and record types. In this case, the rule is that two values `O1` and `O2` belonging to type

`<`$\mathcal{T}_1$`,...,`$\mathcal{T}_m$`> view [A`$_1$`:`$\mathcal{S}_1$`;...;A`$_n$`:`$\mathcal{S}_n$`]`

are equal if:

`(O1 As` $\mathcal{T}_1$`) = (O2 As` $\mathcal{T}_1$`) And ...  And (O1 As` $\mathcal{T}_m$`) = (O2 As` $\mathcal{T}_m$`) And`
  `O1.A`$_1$ `= O2.A`$_1$ `And ...  And O1.A`$_n$ `= O2.A`$_n$.

Note that:

1. for each `A`$_i$ field the associated equality is used. Hence, methods are compared by identity (since they are functions), updatable fields are also compared by identity, and constant concrete fields are compared by value;

2. two records are equal in type `[ ...]` iff they are equal in type `<> view [ ...]` .

The points above highlight that equality on view values generalizes both role and record equalities. One may also expect that, whenever $T \sim S$, then $a : T = b$ is the same as $a : S = b$. However, this is not always true. As a counterexample, consider a pair of role types $\mathcal{S} \subseteq \mathcal{T}$. In accordance with the view types subtyping rules, the following equivalence holds:

```
<𝒮> view [ ] ~ <𝒯,𝒮> view [ ]
```

Now, let `s` be a role of type $\mathcal{S}$ and `t1,t2` be two different roles of type $\mathcal{T}$. Comparing `t1 times s` with `t2 times s` gives two different answers in the two types above, since only in the second case are the two virtual objects also compared with respect to the result of the operation `x As` $\mathcal{T}$, which gives two different results.

```
(t1 times s):<𝒮> view [ ] = (t2 times s) => true
(t1 times s):<𝒯,𝒮> view [ ] = (t2 times s) => false
```

The fact that two types that are equivalent with respect to subtyping are not equivalent with respect to equality is not very satisfactory, but could be avoided by adopting a more complex notion of equality, where two objects in type `<`$\mathcal{T}_1$`,...,`$\mathcal{T}_m$`> view [ ..]` are also compared with respect to the result of `O As` $\mathcal{S}$ for every supertype $\mathcal{S}$ of every $\mathcal{T}_i$. Choosing the best approach is a matter for further research.

Let us consider the following object type definitions to show how role comparison depends on the type used to access the roles:

```
let rec type
   Person <->
    [ Name: string;
      BirthYear: int;
      WhoAreYou:= meth() :string is
                  "My name is  " &  self.Name
    ] ;

let rec type
   Student <-> is Person and
    [ SNumber: string;
      Faculty: string;
      WhoAreYou := meth() :string is
                   super.WhoAreYou &
                   ". I am a student of " &
                   self.Faculty
    ] ;

let rec type
   Athlete <-> is Person and
    [ Code: int;
      Sport: string;
      WhoAreYou := meth() :string is
                   super.WhoAreYou &
                   ". I practice  " & self.Sport
    ];

let John := mkPerson ([ Name := "John Smith";
                        BirthYear := 1967 ]);

let JohnAsStudent :=
       inStudent(John, [ SNumber := "0123";
                         Faculty := "Science" ]);

let JohnAsAthlete :=
       inAthlete(John, [ Code := 123;
                         Sport := "Soccer" ]);

let NewAthlete := mkAthlete([Name := "John Smith";
                            BirthYear:=  1967;
                            Code := 2;
                            Sport := "Basket" ]);
```

The following expression

```
JohnAsStudent  = JohnAsAthlete;
```

is not well typed since `JohnAsStudent` and `JohnAsAthlete` are not subtypes of each other.

`John` and `JohnAsAthlete` can be compared if they are considered of type `Person` as follows, and the equality prdicates returns true:

```
(JohnAsStudent:Person)  = JohnAsAthlete;
```

Two virtual objects can be compared by identity using an appropriate view type. For example:

```
let type IdPerson := <Person> view [] ;

(JohnAsStudent project [SNumber;Name]:IdPerson)
       = (JohnAsAthlete project [Code;Name]:IdPerson);   is true
```

Two role values can be compared by ignoring their identity using an appropriate record type, which allows both the shallow and deep equality to be simulated. For example:

```
let type PersonRecord :=
         [Name:string; BirthYear: int] ;

(JohnAsStudent:PersonRecord)
      = (JohnAsAthlete:PersonRecord);  is true

(JohnAsStudent:Person)
    = NewAthlete;           is false

(JohnAsStudent:PersonRecord)
     = (NewAthlete:PersonRecord);      is true
```

9 Input-Output

Input-output is quite simple. There is an idea of an input stream and an output stream (initially connected to the session window), which are used implicitly by input and output operations.

The input stream is redirected to a file by:

`infile: string -> bool`

> returns `true` if the file with the specified name exists and it can be opened as an input file. If an end-of-file is found, or if an `infile("")` is executed, the input is returned to the stream connected to the session window.

`caninput: null -> bool`

> returns `true` if the input is from the session window or if there are characters to be read from a file. If it returns `false`, the input is returned to the stream connected to the session window.

The output stream is redirected to a file by:

`outfile: string -> bool`

> returns `true` if the file with the specified name can be opened as an output file. If an `outfile("")` is executed, the output returns to the stream connected to the session window.

Output functions are:

`printstring :string -> null`

> writes a string of characters.

`printint :int -> null`

> writes an integer.

`printreal :real -> null`

> writes a real.

`blanks :int -> null`

> writes n blanks.

`newlines :int -> null`

> writes n lines.

`classes :null -> seq string`

> writes the class and subclass names defined in the current environment.

There are two kinds of input operations: buffered with echo and unbuffered without echo.

9.1 Buffered Input

`getline :null -> string`

reads and returns a string of characters delimited by the current read pointer and terminated with the return key. If there are no characters to read, `getline` waits until a new line is typed. Characters are buffered until a line is composed (that is until return key is typed), and the backspace can be used to delete the last character typed.

`getchar :null -> char`

reads and returns the first character available on the input line. If there are no characters to read, `getchar` waits until a new line is typed.

Note that the characters written on a new line are consumed one at a time for each application of `getchar`.

`getint :null -> int`

reads and returns the integer value corresponding to the optional sign and digits read. If there are no digits to read, `getint` waits until they are typed. If the first character after the optional sign is not a digit, `getint` fails and does not consume the other characters in the line.

`getreal :null -> real`

reads and returns a real value. `getreal` consumes the maximum input string matching a legal real number. If there are no characters to read, `getreal` waits until they are typed. If the first character after the optional sign is not a digit, `getint` fails and does not consume the other characters in the line.

The following example shows an application of the above functions:

```
let rec
Students class
   Student <-> [ SNumber: int; BirthYear: int; Name: string;
                  Print := meth() :string is
                     implode({ self.Name; " ";
                               stringofint(self.SNumber); " ";
                               stringofint(self.BirthYear)}) ];

let ReadStudent := fun(): Student is
      use name := getline()
      in if name = " " then failwith "End of data"
            else mkStudent( [ Name:= name;
                              SNumber:= getint();
                              BirthYear := (getchar();
                                            getint()) ] );

let ReadStudents := fun() :int is
      use n := var 0
      in ((while caninput() do
              (ReadStudent(); n <- (at n) + 1)) iffails nil;
           at n);
```

```
let ReadStudentsFromWindow := fun(): int is
     (newlines(1);
      printstring("Insert data as follows:");
      newlines(1);
      printstring("Name <return> SNumber <blank>");
       printstring("BirthYear <return>");
      newlines(1);
      printstring("To terminate, add a line starting");
      printstring(" with a blank.");
      newlines(2);
      ReadStudents());

let ReadStudentsFromFile := fun(File:string): int is
      (if infile(File) then ReadStudents()
       else failwith "The file cannot be opened");

let WriteStudents := fun(): null is
     loop Students do (printstring(Print); newlines(1));

let WriteStudentsIntoWindow := WriteStudents;

let WriteStudentsIntoFile := fun(File:string): null is
      (if outfile(File) then (WriteStudents(); outfile(""); nil)
       else failwith "The file cannot be created");
```

9.2 Unbuffered Input

The function

```
inchar  :null -> char
```

returns the character typed by the user after the function application. The typed character is not shown at the terminal. It is used to write interactive programs such as those that are waiting for the selection of an option from a menu. For example:

```
let f := fun(): null is
      (printstring("Type a number between 0 and 9. ");
       newlines(2);
       use c := inchar()
       ext n := first(explodeascii(c)) - first(explodeascii("0"))
       in if (n >= 0) And (n <= 9)
           then (printstring("The number is "); printint(n))
           else printstring("You must type a digit!");
       newlines(2));
```

9.3 The Menu Library

The input/output functions have been used to define the `Menu Library` to construct and use two kinds of menu: simple and general (see Section 12.3).

A simple menu is a list of items (sentences) which can be selected by specifying the associated number. When a menu item is selected, the item number is returned.

A general menu is a list of items (sentences) which can be selected to display another menu or to apply a function.

Simple Menu

A simple menu is created with the function `mkSimpleMenu`, whose parameters are the menu title and the list of menu items; for example:

```
let myMenu :=
      mkSimpleMenu("Select a color", {"Red"; "Yellow"; "Green"});
```

The number of items is limited to 9.

The menu is displayed using the function `simpleInteract`:

```
simpleInteract(myMenu);

Select a color

1)   Red

2)   Yellow

3)   Green

Type a number
```

The function `simpleInteract` returns the selected item number.

General Menu

To define and use a general menu, we will now give an example to show how to use the `Menu` library.

Let us assume that we need a menu with the following items:

```
Main Menu

1)   Exit

2)   Display submenu

3)   Display main menu

Type a number
```

The submenu has has the following items:

```
Submenu

1)   Exit

2)   Print item id

3)   Print item name

4)   Display main menu

Type a number
```

First, the menu titles and menu items are defined as follows:

```
let m1 := mkMenu("Main Menu");
let m2 := mkMenu("Submenu");
```

The function `mkMenu` takes a string as its parameter and returns a menu without items, with title the string passed as a parameter.

The construction of a menu item is made with the function `mkItem`, which has the following parameters:

1. the item name;
2. the item `id`, a number that is different from any other item `id`;
3. the function to apply when the item is selected; the function parameter is the selected item (the function can be the no-operation `NullOperation` when the selection is used only to display another menu);
4. the final action, that is to exit (`(|quitMenu|)`) or to display another menu (`(|enter-Menu := <anotherMenu>|)`).

```
let i1 := mkItem("Exit",10,
                 (fun(x:menuItem):null is
                    (printstring("Ciao");
                     newlines(1))
                 ),
                 (|quitMenu|));

let i2 := mkItem("Display submenu",20,
                   (fun(x:menuItem):null is
                   printstring("Test submenu")),
                (|enterMenu := m2|));

let i3 := mkItem("Display main menu", 30,
                 NullOperation,
                 (|enterMenu:=m1|));

let i4 := mkItem("Exit ", 40,
                 (fun(x:menuItem):null is
                    (printstring("Ciao");
                    newlines(1))
                 ),
                 (|quitMenu|));

let i5 := mkItem("Print item id", 50,
                (fun(x:menuItem):null is
                   use c := x.id  in
                     (printstring("The item id is ");
                      printint(c);
                      newlines(1))),
                (|enterMenu := m2|));
```

```
let i6 := mkItem("Print item name", 60,
                 (fun(x:menuItem):null is
                 use c := x.name  in
                   (printstring("The item name is ");
                    printstring(c);
                    newlines(1))),
                 (|enterMenu := m2|));
```

The menu method `addItem` is used to add an item in a certain position (a number between 1 and 9) to a menu.

```
m1.addItem(i1, 1); m1.addItem(i2,2); m1.addItem(i3,3);

m2.addItem(i4,1); m2.addItem(i5,2);
m2.addItem(i6, 3); m2.addItem(i3, 4);
```

Once a menu has been constructed, it can be displayed using its method `interact`:

```
m1.interact;
```

A menu item is created as visible and active, but it can change status with the methods `setVisible`, `setInvisible`, `setActive`, and `setInactive`. When a menu item is inactive, it will be displayed but it cannot be selected. For example:

```
i4.setInactive;
m1.interact;

i4.setActive;
i5.setInvisible;
m1.interact;
```

10 Known Bugs and Limitations

The Galileo 97 interpreter presents the following bugs and limitations that will be removed in a future release:

- Concrete values and types (e.g. a record and a record type) must not be defined recursively; however, because of a bug in the type checker, the system does not prevent such definitions, which can enter the interpreter into a non terminating loop. With the other recursive definitions (functions, classes, objects) the system works correctly.
- A file to be loaded with the command `load` must be in the same folder as the Galileo application. The file content must be terminated with a carriage return.

11 Galileo 97 Publications List

Galileo 97 is the result of a redesign of the Galileo language [Alb83, ACO85, AGOP88] aimed at a better integration of an object mechanism into a database programming language.

The role mechanism in Galileo 97 is shaped over the one proposed for Fibonacci [ABGO93, AGO95].

An implementation technique for objects with roles is discussed in [ADG95].

The operators to construct virtual objects are discussed in [AAG99].

An overview of the language features is given in [AAB$^+$95, AAG99].

12 Examples

12.1 Operations on Strings

`extract :string # int # int -> string`

extracts a substring from a string: the first argument is the source string, the second argument is the starting position of the substring in the string (the first character in a string is at position 1), and the third argument is the length of the substring. It fails with string `"extract"` if the numeric arguments are out of range.

`match :string # string -> bool`

returns true if the first argument is equal to the second argument, the latter can contain special characters: `?` is about to any character; `*` refers to any string of characters; `#` refers to any numeric character; `@` refers to any alphabetic character.

```
let extract:=
    fun(x:string, f:int, L:int):string is
      use s:= explode(x)
      ext l:= count(s)
      in if l < f Or l < f + L - 1
           then failwith "extract"
           else use r:= var {nth(s,f)}
                and c:= var 1
                in if L = 1
                     then implode(at r)
                      else (while (at c) <= L - 1 do
                              (r<- nth(s, (at c) + f) :: (at r);
                               c <- (at c) + 1 );
                            implode(reverse(at r))
                           );

let match := fun(a: string, b:string):bool is
     use letters:= explode("qwertyuiopasdfghjklzxcvbnm") append
                   explode("QWERTYUIOPASDFGHJKLZXCVBNM")
     and  digits:= explode("1234567890")
     ext rec try:= fun(a :seq string, b :seq string):bool is
               if emptyseq(a)
                 then emptyseq(b)
                 else
                   if first(a) = "*"
                     then subsL(rest(a),b)
                     else
                       if emptyseq(b)
                         then false
                         else
                           if first(a) = "#"
                             then
                               if first(b) isin digits
                                 then try(rest(a),rest(b))
                                 else false
                             else
                               if first(a) = "@"
                                 then
                                   if first(b) isin letters
                                     then try(rest(a),rest(b))
                                     else false
                                 else
                                   if (first(a) = "?")
                                      Or (first(a) = first(b))
                                     then try(rest(a),rest(b))
                                     else false
     and subsL:= fun(a :seq string, b :seq string): bool is
           if try(a,b)
             then true
             else
               if Not emptyseq(b)
                 then subsL(a, rest(b))
                 else false
     in try(explode(b),  explode(a));
```

12.2 Date Library

The object type `Date` provides the following attributes and methods:

`Day :int`

`Month :int`

`Year :int`

returns the day, month, and year of a date.

`Print :string`

prints the date with the format `dd/mm/yyyy`.

`IsLeap :bool`

tests for leap years.

`DateToDays :int`

maps dates to integers. `DateToDays` maps the date 1/1/1 to the integer 1.

`LessEqDate :Date -> bool`

tests for temporal precedence.

`AddDays :int -> Date`

adds days to a date.

A value of type `Date` is constructed with the function:

`mkDate :int # int # int -> Date`

with parameters a day, a month, and a year.

The object type `MyDate` is defined by inheritance from `Date` with the redefinition of the method `Print` to print a date with the format `Month dd, yyyy`.

```
let rec
type Date <->
       [ Day :int;
         Month :int;
         Year :int;
         Print := meth() :string is
            implode({stringofint(self.Day);"/";
                     stringofint(self.Month);"/";
                     stringofint(self.Year)});
         private Leap := meth(Year :int) :bool is
           use isdiv := fun(x :int, y:int) :bool
                        is (x mod y = 0)
           in  if isdiv(Year, 100)
                 then isdiv(Year,400)
                 else isdiv(Year,4);
         IsLeap := meth() :bool is self.Leap(self.Year);
```

```
   private MonthToDays :=
            meth(Month :int, Year :int) :int is
              use MonthTable := {0;31;59;90;120;151;181;
                                 212;243;273;304;334;365}
              in
                 nth(MonthTable, Month) +
                 if (Month > 2) And (self.Leap(Year))
                    then 1 else 0;
   DateToDays := meth() :int is
     use PastYears := self.Year - 1
           %complete years between 1/1/1 and d%
      in  PastYears*365 +( PastYears div 4)  +
           (PastYears div 400) +
            self.MonthToDays(self.Month, self.Year) +
             self.Day - (PastYears div 100);
   LessEqDate := meth(d :Date) :bool is
      self.DateToDays <= d.DateToDays;
   private DaysToDate := meth(Days :int)  :Date is
      use Days := var (Days - 1)
      and E4Years := 365 * 4 + 1
      ext E100Years := E4Years * 25 - 1
      ext E400Years := E100Years * 4 + 1
      ext year := var (((at Days) div E400Years) * 400 + 1)
      in  (Days <- (at Days) mod E400Years;
          year <- at year + ((at Days) div E100Years) * 100;
          Days <- (at Days) mod E100Years;
          year <- at year + ((at Days) div E4Years) * 4;
          Days <- (at Days) mod E4Years;
          year <- at year + ((at Days) div 365);
          Days <- ((at Days) mod 365) + 1;
          use month := var 0
          in  (month <- ((at Days) div 30) + 1;
              while at Days <= self.MonthToDays(at month,
                                                  at year)
              do  month <- at month - 1;
              Days <- at Days - self.MonthToDays(at month,
                                                   at year);
              mkDate([Year := at year;
                       Month := at month;
                       Day := at Days ])) );
   AddDays := meth(Days :int) :Date is
     self.DaysToDate(self.DateToDays + Days)
   ]
before mk(this)
if Not(
   use y := this.Year
   and m := this.Month
   and d := this.Day
   and within := fun(V :int, m :int, M :int) :bool is
         if V >= m And V <= M
           then true
           else false
   in y >= 0 And
```

```
            within(m, 1, 12) And
            within(d, 1, if m=2
                           then if y mod 4 = 0
                                  then 28
                                  else 29
                           else if m isin {4;6;9;11}
                                  then 30
                                  else 31) )
       do failwith "wrong values for a data";

let mkDate :=
        fun(G:int, M:int, A:int):Date is
            mkDate( [Day := G; Month := M; Year := A] );

% Another Date type to print dates with the format
  "Month D, Y" %

let rec type
MyDate <->
     is Date and
      [Print := meth() :string is
          use MonthName := {"January";"February";"March";
                            "April"; "May";"June";"July";
                            "August";"September";"October";
                            "November";"December"}
           in
            implode({nth(MonthName,self.Month);" ";
                     stringofint(self.Day);", ";
                     stringofint(self.Year)});
        AddDays := meth(Days :int) :MyDate is
           inMyDate( (self As Date)!AddDays(Days), [ ] )

        ];

let mkMyDate :=
        fun(G:int, M:int, A:int):Date is
            mkMyDate( [Day := G; Month := M; Year := A] );
```

12.3 Menu Library

```
let rec type menuStruct <->
  [ title: string;
     items: var seq menuItem;
     private display := meth() :int is
       use c := var 0 in
         (newlines(2);
          printstring(self.title);
          newlines(1);
          loop (at self.items) do display(c);
          newlines(1);
          printstring("Type a number");
          newlines(1);
          at(c)
         );
     interact := meth() :null is
     % See the effect of changing getchar with inchar %
         use rec noBlanks := fun(x: string):string is
             use s := explode(x)
             in
               if first(s) = " "
                 then noBlanks(implode(rest(s)))
                 else x
         in
         if emptyseq(at self.items) then nil
          else
         use validItems := self.display
         ext char := var first (explodeascii(getchar()))
         ext one := first (explodeascii("1"))
         ext last := one + validItems - 1
          in (while ((at char < one) Or (at char > last)) do
                 (newlines(1);
                  printstring
                     (({"Retry: the digits must be between 1 and " &
                         noBlanks(stringofint(validItems))}));
                  newlines(1);
                  char <- first (explodeascii(getchar())) );
               use numHit := var (at char - first (explodeascii("0")))
                in loop i In (at self.items) do
                       (if i.IamNth(numHit) then
                            (i.operation(i);
                             if i.tailAction is enterMenu
                                then (i.tailAction as enterMenu)
                                       .interact
                                else nil;
                             exit)
                        else nil));
     addItem := meth(newItem: menuItem, position:int): menuItem is
         use rec addOrd := fun(itemNum:int, itemList: seq menuItem)
                             : seq menuItem is
                 if position <= itemNum then newItem :: itemList
                 else if emptyseq(itemList) then {newItem}
```

```
                   else first(itemList) :: addOrd(itemNum+1,
                                                  rest(itemList))
          in (if some (at self.items) with id = newItem.id
                 then failwith "duplicated item id"
                 else nil;
              self.items <- addOrd(1, at(self.items));
              newItem);
     remItem := meth(itemId: int): null is
         self.items <- (at self.items) where Not(id = itemId);
     itemWithId := meth(itemId: int): menuItem is
         (get x In (at self.items) where (x.id = itemId)).x
            iffails failwith "No item with such id";
     itemWithName := meth(itemName: string): menuItem is
         pick((at self.items) where name = itemName)
            iffails failwith "No item with such name"
  ]

and type menuItem <->
  [ name: string;
     id: int;
     operation: menuItem -> null;
     tailAction: (| quitMenu or enterMenu: menuStruct |);
     private active: var bool;
     private visible: var bool;
     setActive := meth() :null is self.active <- true;
     setVisible := meth() :null is self.visible <- true;
     setInvisible := meth() :null is self.visible <- false;
     setInactive := meth() :null is self.active <- false;
     IamNth := meth(c:var int): bool is
       (if (at self.visible) And (at self.active)
           then c <- at c - 1
           else nil;
        (at c) = 0);
     display := meth(c:var int):null is
     (if (at self.visible) then
        (newlines(1);
         if (at self.active) then
            (c <- at c + 1;
             printint(at c))
         else printstring("-");
         printstring(")");
         blanks(3);
         printstring(self.name);
         newlines(1)
        )
        else nil
     )
  ];


% Redefinition of the constructors  %

let mkMenu :=
      fun(title:string): menuStruct is
        mkmenuStruct ([title := title;
                       items := var({}:seq menuItem)]);

let mkItem := fun(name: string, id:int, op: menuItem -> null, tail:
                 (| quitMenu or enterMenu: menuStruct|))
                 : menuItem is
              mkmenuItem([name := name;
                          id := id;
                          operation := op;
                          tailAction := tail;
                          active := var true;
                          visible := var true]);

let NullOperation := fun (x:menuItem):null is nil;

% Declarations for the simple menu version %

let private lastIdSelected := var 0
ext
mkSimpleMenu :=
  use idOp := fun(i: menuItem): null is lastIdSelected <- i.id
  ext mkSimpleItem :=
        fun(name: string, id:int): menuItem is
        mkmenuItem([name := name;
                    id := id;
                    operation := idOp;
                    tailAction := (|quitMenu|);
                    active := var true;
                    visible := var true])
  in fun(title: string, items: seq string): menuStruct is
       mkmenuStruct ([title := title;
                      items :=  var
                             (use c := var 0
                             in select ( c <- (at c) + 1;
                                        mkSimpleItem(i, (at c)))
                                from i In items
                             )])
ext
 simpleInteract :=
      fun(m: menuStruct): int is
        (m.interact;
         at lastIdSelected);


% Elimination of the standard constructors %

let mkmenuStruct :=
     fun(x:[title : string; items : var seq menuItem]): none is
       failwith
       "To create a menuStruct please use mkSimpleMenu or mkMenu";

let mkmenuItem := fun(
   x:[name : string; id: int;
       operation: menuItem -> null;
       tailAction: (| quitMenu or enterMenu: menuStruct |);
       active: var bool; visible: var bool]): none is
     failwith "To create a menuItem please use mkItem";
```

12.4 Pictures

```
let type Point := real # real;
let type Tile := Point # Point;

let PrintPoint :=
    fun(p :Point) :null is
        (printstring("(");
         printreal(fst(p));
         printstring(",");
         printreal(snd(p));
         printstring(")") );

let PrintTile :=
    fun(t :Tile) :null is
        (printstring("(");
         PrintPoint(fst(t));
         printstring(",");
         PrintPoint(snd(t));
         printstring(")"));

let rec Pictures class
        Picture <->
             [ Class:string;
                Area:= meth() :real is virtual;
                Tile:= meth() :Tile is virtual;
                Print := meth() :null is
                            ( newlines(1);
                              printstring("   Class: ");
                              printstring(self.Class);
                              newlines(1);
                              printstring("   Area:  ");
                              printreal(self.Area);
                              newlines(1);
                              printstring("   Tile:  ");
                              PrintTile(self.Tile);
                              newlines(2))
              ];

let rec
Rectangles subset of Pictures class
       Rectangle <-> is Picture and
             [ Class := "RECTANGLE";
                Vertixes :Tile;
                Area := meth() :real is
                            use t := self.Vertixes
                            in (fst(snd(t))-fst(fst(t))) *
                                (snd(snd(t))-snd(fst(t)));
                Tile := meth() :Tile is
                             self.Vertixes
              ];
```

```
let rec
Circles subset of Pictures class
       Circle <-> is Picture and
             [ Class := "CIRCLE";
                Center :Point;
                Radius :real;
                Area := meth() :real is
                          use r := self.Radius
                          in r * r * 3.14;
                Tile := meth() :Tile is
                            use cx:= fst(self.Center)
                            and cy := snd(self.Center)
                            and r := self.Radius
                            in ((cx - r, cy - r),(cx + r, cy + r));
                Print := meth() :null is
                            (newlines(1);
                            (super.Print);
                            printstring("   Center: ");
                            PrintPoint(self.Center);
                            newlines(1);
                            printstring("   Radius: ");
                            printreal(self.Radius);
                            newlines(2))
             ];


let c := mkCircle([Center := (1.5, 2.5); Radius := 12.]);
let r := mkRectangle([Vertixes:=((2.,3.5),(11.2,17.1)) ]);

select Print
from Pictures ;

   Class:RECTANGLE
   Area: 125.12000
   Tile:((2,3.50000),(11.20000,17.10000))


   Class:CIRCLE
   Area: 452.16000
   Tile: ((-10.50000,-9.50000),(13.50000,  14.50000))
   Center:(1.50000,2.50000)
   Radius:12
```

12.5 Parts

This is a toy application used in “M. P. Atkinson and O. P. Buneman, Types and Persistence in Database Programming Languages, *ACM Computing Surveys*, 19(2):105-190, 1987” to compare different database programming languages. It represents a fragment of a manufacturing company’s parts and suppliers database; in particular, the way certain parts are made of other parts, either composite or base bought outside. The database schema is shown in Figure 2.

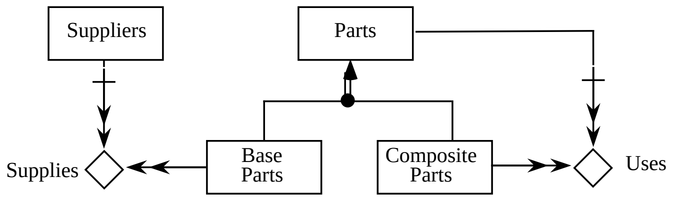

Figure 2: Database schema

```
let NewCode:=
        use Code := var 0
        in fun() :int is
            (Code <- at Code + 1;
             at Code);

let rec Parts class
    Part <->
        [  Name: string;
            Code:= NewCode();
            Cost:= meth() :int is virtual;
            Mass:= meth() :int is virtual;
            UsedIn:= meth() :seq  CompositePart is
               CompositeParts where
                  (some x In Components with x.Part  = self);
            BasePartComponents := meth() :seq BasePart is
                if self isalso BasePart
                    then {self As BasePart}
                    else use pc:= self As CompositePart
                          in setof(flatten(
                               select x.Part.BasePartComponents
                               from x In pc.Components
                                          )
                                  )
        ]

and BaseParts subset of Parts class
       BasePart <-> is Part and
              [PartCost: int;
                PartMass: int;
                Cost:= meth() :int is self.PartCost;
                Mass:= meth() :int is self.PartMass;
                SuppliedBy:=  meth() :seq Supplier is
                                    Suppliers
```

```
                                 where self isin (at PartsSold) ]
       before mk(this)
       if Not (this.PartCost > 0) And (this.PartMass > 0)
       do failwith "Cost and Mass must be > 0"

and CompositeParts subset of Parts class
        CompositePart <-> is Part and
              [AssemblyCost :int;
                Cost :=
                  meth() :int is
                     self.AssemblyCost +
                              sum(select (Part.Cost) * Quantity
                                  from self.Components );
                Mass :=
                  meth() :int is
                    sum(select (Part.Mass) * Quantity
                        from self.Components );
                Components: seq [ Part: Part; Quantity: int ] ]
       before mk(this)
       if Not this.AssemblyCost > 0
       do failwith "Assembly cost must be > 0"

and Suppliers class
        Supplier <->
            [ Name: string;
               Address: string;
               PartsSold: var seq BasePart
            ]
        before mk(this)
        if  use isSet := fun(x:seq BasePart):bool is setof(x) = x
            in  Not(count(at (this.PartsSold)) > 0
                And isSet(at (this.PartsSold)) )
        do failwith "Parts sold must be at least one and all different"
     key(Name) ;


% ********* Data ********* %

let p1:=mkBasePart([ Name := "Piston";
                     PartCost := 10000;
                     PartMass := 1000 ]);
let p2:=mkBasePart([ Name := "Cylinder";
                     PartCost := 10000;
                     PartMass := 500 ]);
let p6:=mkBasePart([ Name := "Wheel" ;
                     PartCost := 2222;
                     PartMass := 222 ]);
let pc2:=mkCompositePart([ Name := "Piston-Cylinder" ;
                           AssemblyCost :=  1000;
                           Components:=
                        {[ Part := p2; Quantity := 1 ];
                         [ Part := p1; Quantity := 1 ]}]);
```

```
let pcc1:=mkCompositePart(
               [ Name := "Car" ;
                 AssemblyCost := 13000 ;
                 Components:=
                    {[ Part := pc2; Quantity := 6 ];
                     [ Part := (p6 As Part); Quantity := 4 ]} ]);

mkSupplier([ Name :="Bendini";
             Address := "Via P.Tarditi 154, 11332, Biella";
             PartsSold := var {p1; p2} ]);
mkSupplier([ Name :="Landini";
             Address := "Via A. Lancia 1, 10121, Torino";
             PartsSold := var {p6} ]);
mkSupplier([ Name := "Landocci";
             Address := "Via A. Lancia 3, 10121, Torino";
             PartsSold := var ({ }:seq BasePart) ]);

% ******* Queries ******* %

% Q1: Find Name, Cost and Mass of the base parts which
      cost more than 100 %

select [Name := p.Name;
        Cost := p.PartCost;
        Mass := p.PartMass ]
from (p In BaseParts)
where p.PartCost > 100 ;

% Q2: Find the name of base parts of a composite part with
      a given name. %

select Name
from (get CompositeParts where Name = "Car").BasePartComponents;

select Name
from (get Parts where Name = "Car").BasePartComponents;

% Q3: Find  Mass  and  Cost  of a part with a given name.%

use p := get Parts where Name = "Car"
in [ Cost := p.Cost;
     Mass := p.Mass];
```

12.6 Library

A library has a set of books (one copy of each book), each identified by a unique "call number". Books may be loaned to borrowers, who may be faculty members. A book can normally be borrowed for two weeks and two one week extensions are allowed. Some books may be placed on restricted loan; these can only be taken out by faculty members for three days and cannot be renewed. In general, any borrower may have on loan at most five books at any time, and at most one short term loan book. A preliminary database schema is shown in Figure 3.

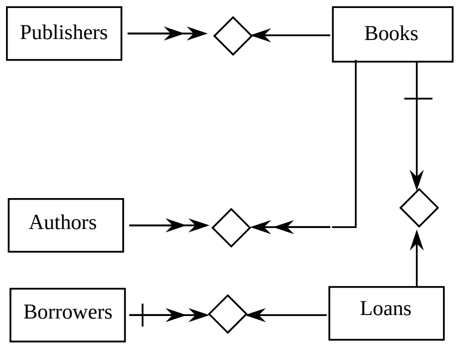

Figure 3: Database schema

```
% Load "Date library" %

let ToDay := mkDate(2, 2, 1995);
let type NoRenewals <-> int
            before  mk(this)
            if Not(this >= 0 And this <= 2)
            do failwith "A loan may be renewed only twice";

let rec
Authors class
    Author <->
        [ FirstName: string;
           LastName: string;
           Nationality :string ]
and
Publishers class
    Publisher <->
        [ Name :string;
          Address :string ]
key(Name)
and
Books class
    Book <->
      [ CallN :int;
        Publisher :Publisher;
        Authors :seq Author;
        Title :string;
        LoanedTo:= meth(theBorrower: Borrower) :Loan is
              (if theBorrower.NOutstandingLoans >= 5
                    then failwith "to many books on loan"
                    else if some Loans with Book = self
                           then failwith "Book not available"
                           else mkLoan(
                                  [Book := self;
                                    LoanedTo := theBorrower;
                                    DueDate :=
                                       var (ToDay.AddDays(14));
                                    RenewalsLeft :=
                                       var mkNoRenewals(2)
                                   ]     )
               )
       ]
key(CallN)
and
 Borrowers class
    Borrower <->
       [ Name :string;
         Address :var string;
         BooksBorrowed := meth() :seq [Book :Book;
                                       DueDate :Date;
                                       RenewalsLeft :NoRenewals] is
                select[ Book := Book;
                        DueDate := at DueDate;
                        RenewalsLeft := at RenewalsLeft ]
                from Loans where LoanedTo = self;
          NOutstandingLoans := meth() :int is
                                 count(Loans where LoanedTo = self)
        ]
and
Loans class
    Loan <->
      [ Book :Book;
        LoanedTo :Borrower;
        DueDate :var Date;
        RenewalsLeft : var NoRenewals]
key(Book);

% Data %

let a1 := mkAuthor([FirstName := "joe"; LastName := "stoy";
                     Nationality := "American"]);
let a2 := mkAuthor([FirstName := "jim"; LastName := "jen";
                     Nationality := "American"]);
let a3 := mkAuthor([FirstName := "nik"; LastName := "wir";
                     Nationality := "Swiss"]);
let e1 := mkPublisher( [Name :="sw"; Address := "b" ] );
let e2 := mkPublisher( [Name :="aw"; Address := "nj" ] );
let b1 := mkBook([ CallN := 1; Publisher := e1; Authors := {a2;a3};
                    Title := "pa" ]);
let b2 := mkBook([ CallN := 2; Publisher := e2; Authors := {a1};
                    Title := "ds" ]);
let u1 := mkBorrower( [Name :="aa"; Address := var "di"] );
let u2 := mkBorrower( [Name :="rb"; Address := var "di"] );

b1.LoanedTo(u1);
b2.LoanedTo(u2);
```

The database schema is then refined by specialization as shown in Figure 4.

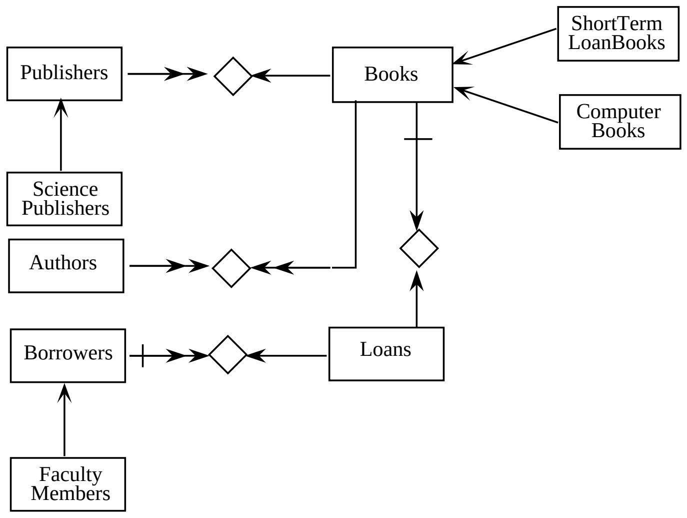

Figure 4: Database schema refinement

```
%  Schema refinement %

let rec
ComputerBooks subset of Books class
  ComputerBook <-> is Book and
     [ Publisher: SciencePublisher;
       CRSubjectCode :seq string]
and
ShortTermLoanBooks subset of Books class
  ShortTermLoanBook <-> is Book and
     [ ExpiryOfRestriction : Date;
       LoanedTo:=
          meth(theBorrower: Borrower) :Loan is
           if Not theBorrower isalso FacultyMember
            then failwith "Loan for FacultyMembers"
            else
              if (theBorrower As FacultyMember).HasShortLoan
                then failwith "One ShortTermLoanBook only is allowed"
                else
                use ALoan:=
                       ((self As Book)!LoanedTo) (theBorrower)
                 in  (ALoan.RenewalsLeft <- mkNoRenewals(0);
                      ALoan.DueDate
                                 <- ToDay.AddDays(3);
                      ALoan
                      )
```

```
      ]
and
SciencePublishers subset of Publishers class
  SciencePublisher <-> is Publisher and [ ]
and
FacultyMembers subset of Borrowers class
  FacultyMember <-> is Borrower and
         [ Phone : string;
           HasShortLoan := meth() :bool is
               some x In Loans with (x.LoanedTo = self)
                And (x.Book isalso ShortTermLoanBook)
          ];

let se1:= inSciencePublisher(e1, [ ]);
let sb1:= inComputerBook(b1,[Publisher := se1;
                             CRSubjectCode :=
                                {"Programming Language"}
                             ]);
let d1:= inFacultyMember(u1, [Phone := "887266"]);
let b3:= mkShortTermLoanBook
            ([ CallN:=3;
               Publisher:=
                 mkPublisher([Name := "Oxford University Press";
                             Address := "Oxford"]);
               Authors := {mkAuthor([FirstName := "A S";
                                     LastName:="Hornby";
                                     Nationality := "English" ]) };
               Title := "Oxford Dictionary of Current English";
               ExpiryOfRestriction:= mkDate(31, 12, 1996)
            ]);

b3.LoanedTo(d1);
```

A Lexical Classes

Ascii characters are classified into the following categories:

Illegal

Unacceptable in a source program. These characters are ignored, and a warning message is printed.

Eof

End–of–file character. It terminates an interactive session, or stops the process of reading a source Galileo file. A top level control– D is interpreted as Eof, thereby exiting the system.

Blank

Characters which are interpreted as a space character “”.

Digit

Digits.

Letter

Letters and other characters which can be used to form identifiers.

Symbol

A class of characters which can be used to form operators (see Appendix I).

Escape

Escape character in strings. It behaves like a Symbol outside strings.

Quote

String quotation character.

Delim

One–character punctuation marks and parentheses.

Moreover, any sequence of legal characters enclosed between `%` and `%` is a *comment*. Each comment is considered equivalent to a single space character.

```
nul Illegal;           soh Illegal;           stx Illegal;
etx Illegal;           eot Illegal;           enq Illegal;
ack Illegal;           bel Illegal;           bs  Illegal;
ht  Blank;             lf  Blank;             vt  Blank;
ff  Blank;             cf  Blank;             so  Illegal;
si  Illegal;           dle Illegal;           dc1 Illegal;
dc2 Illegal;           dc3 Illegal;           dc4 Illegal;
nak Illegal;           syn Illegal;           etb Illegal;
can Illegal;           em  Illegal;           sub Illegal;
esc Illegal;           fs  Illegal;           gs  Illegal;
rs  Illegal;           us  Illegal;           '\' Letter;
'!' Letter;            '"' Quote;             '#' Symbol;
```

```
‘$’ Letter;             ‘%’ Symbol;             ‘&’ Letter;
‘'’ Letter;             ‘(’ Delim;              ‘)’ Delim;
‘*’ Symbol;             ‘+’ Symbol;             ‘,’ Delim;
‘-’ Symbol;             ‘.’ Delim;              ‘/’ Symbol;
‘0’ Digit;              ‘1’ Digit;              ‘2’ Digit;
‘3’ Digit;              ‘4’ Digit;              ‘5’ Digit;
‘6’ Digit;              ‘7’ Digit;              ‘8’ Digit;
‘9’ Digit;              ‘:’ Symbol;             ‘;’ Symbol;
‘<’ Symbol;             ‘=’ Symbol;             ‘>’ Symbol;
‘?’ Letter;             ‘@’ Letter;             ‘A’ Letter;
‘B’ Letter;             ‘C’ Letter;             ‘D’ Letter;
‘E’ Letter;             ‘F’ Letter;             ‘G’ Letter;
‘H’ Letter;             ‘I’ Letter;             ‘J’ Letter;
‘K’ Letter;             ‘L’ Letter;             ‘M’ Letter;
‘N’ Letter;             ‘O’ Letter;             ‘P’ Letter;
‘Q’ Letter;             ‘R’ Letter;             ‘S’ Letter;
‘T’ Letter;             ‘U’ Letter;             ‘V’ Letter;
‘W’ Letter;             ‘X’ Letter;             ‘Y’ Letter;
‘Z’ Letter;             ‘[’ Delim;
‘]’ Delim;              ‘^’ Letter;             ‘_’ Letter;
                        ‘a’ Letter;             ‘b’ Letter;
‘c’ Letter;             ‘d’ Letter;             ‘e’ Letter;
‘f’ Letter;             ‘g’ Letter;             ‘h’ Letter;
‘i’ Letter;             ‘j’ Letter;             ‘k’ Letter;
‘l’ Letter;             ‘m’ Letter;             ‘n’ Letter;
‘o’ Letter;             ‘p’ Letter;             ‘q’ Letter;
‘r’ Letter;             ‘s’ Letter;             ‘t’ Letter;
‘u’ Letter;             ‘v’ Letter;             ‘w’ Letter;
‘x’ Letter;             ‘y’ Letter;             ‘z’ Letter;
‘{’ Delim;              ‘|’ Symbol;             ‘}’ Symbol;
‘~’ Symbol;
```

B Keywords

and	And	
any	append	as
As	assert	at
before	bool	
case	class	casefails
derived	div	do
drop		
each	else	end
elsefail	ext	exit
extend	extend*	
fail	failwith	from
fun		
get	groupby	
has		
if	iffails	in
In	int	is
isin	isalso	isexactly

key		
let	loop	meth
mk	mod	
none	Not	null
of	optional	or
Or		
project	project*	
quit		
real	rec	
rename	rename*	
seq	some	string
subset	select	
times*	then	type
use		
var	where	
when	while	with

C Predefined Identifiers

Costanti

`unknown`	`unknown` value
`nil`	the only value of type `null`
`true`	logic true
`false`	logic false

Functions

`append`	sequence append
`avg`	number sequence average
`blanks`	blanks output
`caninput`	test for input
`count`	sequence length
`emptyseq`	test for empty sequence
`explode`	string explosion
`explodeascii`	string to ASCII conversion
`first`	sequence head
`fst`	first element in a pair
`getchar`	buffered char input
`getint`	buffered integer input
`getline`	buffered string input
`getreal`	buffered real input
`implode`	string implosion
`implodeascii`	ASCII to string conversion
`inchar`	unbuffered char input
`infile`	to redirect input stream
`intofreal`	real to int conversion
`intofstring`	string to integer conversion
`isin`	sequence membership
`isunknown`	test for `unknown`
`known`	sequence without `unknown` elements
`min`	number sequence minimum
`max`	number sequence maximum
`newlines`	lines output
`outfile`	to redirect output stream
`printint`	integer output
`printreal`	real output
`printstring`	string output
`realofstring`	string to real conversion
`rest`	sequence tail
`reverse`	sequence reverse
`snd`	second element in a pair
`sort`	sequence sorting
`stringofint`	integer to string conversion
`stringofreal`	real to string conversion
`sum`	number sequence total

D Overloaded Operators

For each primitive or constructed type `T'`, the operators automatically overloaded when defining an semi-abstract type `T <-> T'` are listed in the table. The type of the overloaded operators is obtained substituting `T` for `T'`.

Type	*Operators*
int	`>, <, =, >=, <=, <>, +, -, *, /, ~, mod, div`
real	`>, <, =, >=, <=, <>, +, -, *, /, ~`
bool	`=, And, Or, Not`
string	`=`
null	`=`
any	
none	
record	`=, of, ., project, times, extend, rename`
variant	`=, is, as`
pair	`=`
function	{`application`}
sequence	`=, append, isin, ::, some, each`
record sequence	`=, append, isin, ::, some, each, .*` `project*, times*, extend*, rename*` `difference, intersection, union`
locations	`=, at, <-`

E Predefined Type Identifiers

`null`	Null Type	
`any`	Most general Type	
`none`	Least general Type	
`bool`	Boolean Type	
`int`	Integer Type	
`real`	Real Type	
`string`	String Type	
`->`	Function Space	Infix
`seq`	Sequence Type	Prefix
`var`	Modifiable Type	Prefix
`view`	View Type	Infix

F Precedence of Operators

Operators are given in order of decreasing precedence, and for the infix operators the associativity is also shown.

{application}	left associative
of . .* In	left associative
isin isalso As isexactly	left associative
is as	(suffix)
~ at	(prefix)
* / & mod div	left associative
+ -	left associative
= > < >= <= <>	left associative
Not	(prefix)
And	left associative
Or	left associative
::	right associative
append	left associative}
difference intersection union	left associative
project extend rename times project* extend* rename* times*	left associative
where	left associative
get	(prefix)
:	(suffix)
<-	left associative
,	right associative
iffails	left associative
groupby	left associative

G ASCII Codes

```
|  0 nul |  1 soh |  2 stx |  3 etx |  4 eot |  5 enq |  6 ack |  7 bel |
|  8 bs  |  9 ht  | 10 nl  | 11 vt  | 12 np  | 13 cr  | 14 so  | 15 si  |
| 16 dle | 17 dc1 | 18 dc2 | 19 dc3 | 20 dc4 | 21 nak | 22 syn | 23 etb |
| 24 can | 25 em  | 26 sub | 27 esc | 28 fs  | 29 gs  | 30 rs  | 31 us  |
| 32 sp  | 33  !  | 34  "  | 35  #  | 36  $  | 37  %  | 38  &  | 39  ’  |
| 40  (  | 41  )  | 42  *  | 43  +  | 44  ,  | 45  -  | 46  .  | 47  /  |
| 48  0  | 49  1  | 50  2  | 51  3  | 52  4  | 53  5  | 54  6  | 55  7  |
| 56  8  | 57  9  | 58  :  | 59  ;  | 60  <  | 61  =  | 62  >  | 63  ?  |
| 64  @  | 65  A  | 66  B  | 67  C  | 68  D  | 69  E  | 70  F  | 71  G  |
| 72  H  | 73  I  | 74  J  | 75  K  | 76  L  | 77  M  | 78  N  | 79  O  |
| 80  P  | 81  Q  | 82  R  | 83  S  | 84  T  | 85  U  | 86  V  | 87  W  |
| 88  X  | 89  Y  | 90  Z  | 91  [  | 92  \  | 93  ]  | 94  ˆ  | 95  _  |
| 96  ‘  | 97  a  | 98  b  | 99  c  |100  d  |101  e  |102  f  |103  g  |
|104  h  |105  i  |106  j  |107  k  |108  l  |109  m  |110  n  |111  o  |
|112  p  |113  q  |114  r  |115  s  |116  t  |117  u  |118  v  |119  w  |
|120  x  |121  y  |122  z  |123  {  |124  |  |125  }  |126  ˜  |127 del |
```

H Metasyntax

- Strings between quotes `"like this one"` are terminals (`""` is the empty string).
- Identifiers are non-terminals.
- `X Y` means `X` followed by `Y`.
- `X` | `Y` means `X` or `Y`.
- `[X]` means (`""` | `X`).
- {`X`} means (`""` | `X` {`X`}).
- {`X`}$_n$ means a sequence of at least n `X`.
- {`X /";"`} means (`""` | `X` | `X;X` | `X; ...; X`, that is the empty string, `X`, or a sequence of `X` separated by `;`.
- {`X/";"`}$_n$ means a sequence of at least n `X` separated by `;`.
- `(X)` means `X`.

I Syntax of Lexical Entities

```
Letter ::=
   "a" | .. | "z" | "A" | .. | "Z" |  "_".

Digit ::=
   "0" | .. | "9".

Symbol ::=
   "!" | "#" | "%" | "&" | "*" | "+" | "-" | "/" | ":" | "<" |
   "=" | ">" | "?" | "@" | "\" | "^" | "`" | "|" | "~" | "$".

Character ::= ... (see Appendix A for a list of legal characters)

Ide ::=
   Letter {Letter | Digit}.

Integer ::=
   {Digit}1.

Real ::=
    {Digit}1 "." [{Digit}] [("e" | "E") ("+" | "-") {Digit}1].

Number ::=
    Integer | Real

String ::=
   """ {Character} """.
```

J Syntax

Syntactic alternatives are in order of decreasing precedence.

```
Phrase ::=
   [Exp | "let" Decl | Command] ";".

Command ::=
   "load" String | "quit" | "outputoff" | "outputon" | ":" TypeIde.

SimpleExp ::=
   Ide |
   Number |
   String |
   "{" "}" ":" "seq" Type |
   "{" {Exp /";"} "}" |
   "[" {Ide ":=" Exp / ("and" | ";") } "]" |
   "(|" Ide [ ":=" Exp] "|)" |
   Exp, Exp |
   "(" {Exp /";"}1 ")".

Exp ::=
   SimpleExp |
   "assert" Exp ["elsefail" Exp] |
   Exp "(" {Exp /","} ")" |
   Exp "[" {Type /";"} "]" |
   Exp "[" {Type /";"} "]" "(" {Exp /","} ")" |
   Exp ":" Type |
   PrefixOp Exp |
   Exp InfixOp Exp |
   Ide "In" Exp |
   "exit" |
   "fail" |
   "failwith" Exp |
   "if" Exp "then" Exp "else" Exp |
   "while" Exp "do" Exp |
   "use" Decl "in" Exp |
   "case" Exp "when" "(|" {Ide ":=" Ide "." Exp /"or"}1 "|)" |
   Exp "iffails" Exp |
   Exp "casefails" Exp Exp |
   "fun" "("{ Ide ":" Type /","} ")" ":" Type "is" Exp |
   "fun" "[" {Ide ["<:" Type] /";"} "]"
         "("{ Ide ":" Type /","} ")" ":" Type "is" Exp |
   "select" Exp "from" Exp |
   Exp "where" Exp |
   "get" Exp |
   "some" Exp "with" Exp |
   "each" Exp "with" Exp |
   Exp"."Ide |
   Exp"!"Ide |
   "super."Ide |
   Exp "isalso" Ide |
   Exp "isexactly" Ide |
   Exp "As" Ide |
   "loop" Exp "do" Exp |
   Exp ".*" Exp |
   Exp "groupby" "[" {Ide ":=" Exp / ("and" | ";") } "]" |
   Exp ("extend" | "extend*")
                "[" {Ide [":" Type] ":="
                   ( Exp  |
                     "meth" "("{Ide ":" Type /","} ")"
                        ":" Type "is" Exp ) /";"}
                "]" |
   Exp ("project" | "project*") "[" {Ide [":" Type]  /";"} "]" |
   Exp ("times" | "times*") Exp |
   Exp ("rename" | "rename*") "(" {Ide[{"."Ide}] "=>" Ide /";"} ")".

Decl ::=
   ValBind |
   Decl "and" Decl |
   "rec" Decl |
   "type" TypeBind |
   Decl "ext" Decl.

ValBind ::=
   ["private"] Ide ":=" ( Exp | "derived" Exp) |
   %Ide "("{Ide ":" Type}")" ":" Type ":=" Exp |
   Ide "class" AbsTypeBind ["key" "(" {Ide / "," } ")" ] |
   Ide "subset" "of" Ide "class" SubAbsTypeBind
         ["key" "(" {Ide /"," } ")" ].

TypeBind ::=
   Ide ":=" Type | AbsTypeBind | SubAbsTypeBind.

AbsTypeBind ::=
   Ide "<->" [Type | ExtRecordType]
             [BeforeMK] [BeforeIn] [BeforeDrop]

SubAbsTypeBind ::=
   Ide "<->" "is" Ide ["and" ExtRecordType]
             [BeforeMK] [BeforeIn] [BeforeDrop]

Type ::=
   "seq" Type |
   "var" Type |
   "optional" Type |
   "["{Ide ":" Type /";"} "]" |
   "(|" {Ide ":" Type /"or"} "|)" |
   Ide |
   {Type /"#"}2 |
   Type "->" Type |
   "all" "["{Ide ["<:" Type] /";"} "]" Type "->" Type |
   "<" {Type /","} ">" "view" "[" {Ide [":" Type]/";"}"]".

ExtRecordType ::=
    "["{["private"] Ide (":" Type |
                        ":=" (Exp |
                              "meth" "("{Ide ":" Type /","} ")"
                                           ":" Type "is"  (Exp | "virtual")
                             ) /";"} "]".
BeforeMK   ::=  "before" "mk" "(" Ide ")"
                {"if" Exp "do" Exp}1.

BeforeIn   ::=  "before" "in" "(" Ide ["," Ide ] ")"
                {"if" Exp "do" Exp}1.

BeforeDrop ::=  "before" "drop" "(" Ide ")"
                {"if" Exp "do" Exp}1.
```

Acknowledgments

A preliminary implementation of a Galileo 97 subset was made by Milena Diotallevi. Diego Dellagiacoma has implemented the Windows interface, and the garbage collector for Windows. Their efforts and cooperation are much appreciated.

References

[AAB$^+$95] A. Albano, G. Antognoni, G. Baratti, G. Ghelli, and R. Orsini. Galileo 95. In *Terzo Convegno Nazionale su Sistemi Evoluti per Basi di Dati*, Ravello (SA), 28–30 Giugno, 1995.

[AAG99] A. Albano, G. Antognoni, and G. Ghelli. View Operations on Objects with Roles for a Statically Typed Database Language. *IEEE Transactions on Knowledge and Data Engineering*, 12(4):548–567, 2000.

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 39–51, Dublin, Ireland, 1993.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985. Also in S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[ADG95] A. Albano, M. Diotallevi, and G. Ghelli. Extensible objects for database evolution: Language features and implementation issues. In *Proc. of the Fifth Intl. Workshop on Data Base Programming Languages (DBPL), Gubbio, Italy*, 1995.

[AGO95] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *The VLDB Journal*, 4(3):403–439, 1995.

[AGOP88] A. Albano, F. Giannotti, R. Orsini, and D. Pedreschi. The type system of Galileo. In M. Atkinson, P. Bunemann, and R. Morrison, editors, *Data Types and Persistence*, Topics in Information Systems, pages 101–120, Berlin, 1988. Springer-Verlag.

[Alb83] A. Albano. Type hierarchies and semantic data models. In *ACM SIGPLAN '83: Symposium on Programming Languages Issues in Software Systems*, pages 178–186, San Francisco, 1983.