The Mathemagix language

Table of contents

Chapter 1
Introduction to the Mathemagix language

Why develop or learn yet another programming language? At the start of the Mathemagix project during the late nineties, the state of the art concerning computer algebra systems was unsatisfactory at least for two reasons:

- There were no high quality general purpose free computer algebra systems.
- With the notable exception of Axiom and Aldor, there were no computer algebra system with a language that could be compiled.

In the beginning, the Mathemagix language was very much inspired by Axiom and Aldor, but as our ideas and implementations evolved, there were more and more differences. When Axiom and Aldor ultimately became free, our project had reached a stage in which it was interesting to further develop these new ideas.

In its present state, Mathemagix is a strongly typed functional language for computer algebra and computer analysis, which can both be intepreted and compiled. Strong typing means that every expression in the language admits a unique type, including the types themselves. For instance, the types of `2` and `"hello"` might be `Integer` and `String`, the type of the function `(x:Integer) :-> (x*x:Integer)` would be `Integer -> Integer`, and the type of `Integer` would be `Class`. A language is said to be functional if functions can be treated as basic objects on the same level as, say, numbers.

The requirement that programs be strongly typed has its pros and cons. On the one hand, it puts some burden on the user, since the user must carefully specify the type of every newly introduced symbol. For instance, evaluation of the expression `x*y` in a shell session will not work directly, since we first have to specify the types of `x` and `y`. Also, there may be some loss of flexibility. For instance, in more classical computer algebra systems, it is easy to construct vectors with entries of different types, such as `[ 2, "hello", x+y ]`. In Mathemagix, such expressions will only make sense if the entries can be casted into a common supertype.

On the other hand, specifying clean types for all newly introduced notations makes the semantics of the language far more robust and simplifies the task of writing compilers for the language which transform the source code into highly efficient executables. For instance, what do we actually mean by an expression such as `x*y`? Is this just a symbolic expression or rather an element of the polynomial ring $\mathbb{Z}[x, y]$? Is the multiplication necessary commutative, or not? Clean typing of all declarations is a way to make potential implicit assumptions of this kind more explicit. The increased robustness in the semantics makes it also easier to develop large mathematical libraries.

Furthermore, whereas the memory layout of data can only be determined at run time for untyped languages, this kind of information and other assumptions on data are available at compile time in strongly typed languages. This opens the route to all kinds of optimizations which usually make strongly typed languages one order of magnitude faster than their untyped homologues. This is particularly important in the case of Mathemagix: besides symbolic and algebraic objects, we are also interested in the manipulation of objects of a more analytic natures, such as the numeric integration of differential equations. In order to be competitive with standard numerical libraries, an optimizing compiler is a prerequisite.

Why did we not chose for an existing strongly typed language with an optimizing compiler? There are a certain number of more traditional languages which we have considered. First of all, there are the languages OCAML, HASKELL, and more recently SCALA, which all belong to the family of so called ML-style languages. We also mentioned the AXIOM and ALDOR systems which are based on different paradigms. Other well-known languages which we considered are C++ and SCHEME.

In section 2 of our paper "*Overview of the* MATHEMAGIX *type system*", we have outlined our main motivations for developing a new language. To go short, we want a language which adequately reflects the overloading present in traditional mathematical notation. For instance, depending on the context, the operator + acts on numbers, polynomials, matrices, etc.

Conceptually speaking, we also believe that the prototype of a function declaration is analogous to a mathematical definition or the statement of a mathematical theorem, whereas the the implementation of the function is analogous to giving a proof. The type system of MATHEMAGIX intends to make the declarations of function prototypes as precise as "operational part" of the statement of a mathematical definition or theorem. One simple example of this guiding principle is the following declaration of the cube function:

```
forall (R: Ring) cube (x: R): R == x*x*x;
```

This declaration clearly corresponds to a mathematical definition:

DEFINITION. *Given a ring R, and an element $x \in R$, we define the cube of x by* $\mathrm{cube}(x) = x \cdot x \cdot x$.

Notice that this is far more precise than simply declaring `cube(x) == x*x*x`. Similarly, the declaration of the function

```
forall (R: Real_Closed_Field)
complex_roots (p: Polynomial R): Vector Complex R == {
  ...
}
```

corresponds to the mathematical theorem that for any real closed field R and any polynomial $p \in R[x]$, we may compute the vector of all complex roots of p. Although this declaration is precise enough from the operational point of view, we may actually refine the prototype as follows:

```
forall (R: Real_Closed_Field)
complex_roots (p: Polynomial R): (v: Vector Complex R | #v = deg p) == {
  ...
}
```

This refinement would correspond to the mathematical statement that R[i] is algebraically closed (and that we may actually compute the roots of polynomials). However, MATHEMAGIX is only intended to be a compiler and not a mathematical theorem prover. Therefore, the mathematical property `#v = deg p` will not be rigourously proven, but only verified to hold for concrete inputs.

Chapter 2

Simple examples of Mathemagix programs

2.1. Hello world

Let us start with the famous "hello world" example, which is written as follows in Mathemagix:

```
include "basix/fundamental.mmx";
mmout << "Hello world!" << lf;
```

By default, only very few types are available in Mathemagix. The first line is therefore needed in order to make various standard types available, such as strings and input/output ports.

The example program can be run in two ways: by compiling it using the Mathemagix compiler `mmc` and then running the resulting executable, or by interpreting it using the Mathemagix interpreter. In the appendix getting and installing Mathemagix it is briefly described how to download and install Mathemagix. For more information, we refer to our website `www.texmacs.org`.

2.1.1. Compiling and running the program

Assuming that the above program was saved in a file `hello_world.mmx`, we may compile and execute the program in a shell session as follows:

```
Shell] mmc hello_world.mmx
Shell] ./hello_world
Hello world!
Shell]
```

2.1.2. Running the program in the interpreter

Alternatively, we may directly run the program in the Mathemagix interpreter `mmi`:

```
Welcome to Mathemagix 1.0.1
This software falls under the GNU General Public License
It comes without any warranty whatsoever
http://www.mathemagix.org
(c) 2010-2012
```

```
Mmx] include "hello_world.mmx"

 Hello world!

Mmx]
```

When including files with external C++ functionality for the first time in the interpreter, the interpreter first has to compile some glue in order to use this functionality. This happens in particular for the file `basix/fundamental.mmx`. Whenever the interpreter is compiling some glue, it displays a message which disappears as soon as the compilation is complete.

2.2. FIBONACCI SEQUENCES

Another classical example is the computation of Fibonacci sequences. A simple implementation using a recursive function goes as follows:

```
include "basix/fundamental.mmx";
fib (n: Int): Int ==
  if n <= 1 then 1 else fib (n-1) + fib (n-2);
```

Notice that the programmer has to specify explicit types for the arguments and return type of the function. In our example both the argument and the return value are machine integers of type `Int`. It is also possible to implement a faster non recursive algorithm which returns an integer of arbitrary precision:

```
include "numerix/integer.mmx";
fib (n: Int): Integer == {
  a: Integer := 1;
  b: Integer := 1;
  for k: Int in 2 to n do {
    c: Integer == a + b;
    a := b;
    b := c;
  }
  return b;
}
```

2.3. MERGE SORT

One more involved example is to provide a generic implementation of the merge sort algorithm:

```
include "basix/fundamental.mmx";

category Ordered == {
  infix <=: (This, This) -> Boolean;
}

forall (T: Ordered)
merge_sort (v: Vector T): Vector T == {
  if #v <= 1 then return v;
  v1: Vector T == merge_sort v [0, #v quo 2];
  v2: Vector T == merge_sort v [#v quo 2, #v];
  r : Vector T := [];
  i1: Int := 0;
  i2: Int := 0;
  while i1 < #v1 or i2 < #v2 do
    if i1 < #v1 and (i2 >= #v2 or v1[i1] <= v2[i2]) then {
      r << [ v1[i1] ];
      i1 := i1 + 1;
    }
    else {
      r << [ v2[i2] ];
      i2 := i2 + 1;
    }
  return r;
}
```

This routine `merge_sort` can be applied to any vector whose entries are of a type `T` with an ordering `infix <=: (T, T) -> T`. For instance, the instructions

```
mmout << merge_sort ([ 3, 2, 1, 5, 4, 4, 7 ]) << lf;
mmout << merge_sort ([ "bob", "alice", "carl" ]) << lf;
```

yield the output

```
[1, 2, 3, 4, 4, 5, 7]
["alice", "bob", "carl"]
```

Chapter 3

Declarations and control structures

3.1. Declaration and scope of variables

Constants are defined using the keyword ==:

```
welcome: String  == "Welcome to Mathemagix!";
ok?    : Boolean == sunny? and warm?;
```

Mutable variables can be defined and modified using the keyword :=, as in the following example:

```
i: Int := 1;
while i <= 10 step i := i + 1 do
  mmout << 10 << " * " << i << " = " << 10 * i << lf;
```

The skope of a variable corresponds to the innermost block delimited by { and } in which the variable is defined. For instance:

```
i: Int == 1;
if cond? then {
  i: Int == 2;
  foo (i, i);
}
mmout << i << lf; // i contains 1 at this point
```

Global variables admit the entire file as their skope. More complex scoping rules which apply in the case of multiple file projects or in presence of modules will be discussed in the chapter about programming in the large.

Regular identifiers should match the regular expression `[a-zA-Z_]+[a-zA-Z0-9_$?]*`. That is, the names of constants, variables, function names, macros and types should

- only contain letters, digits and the special characters _, $ and ? ; and
- start with a letter or _ or $.

In addition, it is customary to use lowercase identifiers for constants, variables and functions, and to capitalize the first letter of each word in identifiers for types (e.g. `Integer` or `Sparse_Vector`). Moreover, identifiers for boolean variables and predicates are usually suffixed by ?.

Besides regular identifiers, MATHEMAGIX supports various special identifiers, such as `infix +` for addition. These will be discussed in more detail in the section about identifiers.

3.2. DECLARATION OF SIMPLE FUNCTIONS

The declaration of functions will be discussed in more detail in the chapter about functions. Simple function declarations admit the following syntax:

```
function_name (arg_1: Type_1, ..., arg_n: Type_n): Ret_Type == body;
```

For instance:

```
cube (n: Int): Int == n*n*n;
```

It should be noticed that this declaration is actually equivalent to the following declaration of `cube` as a "function constant":

```
cube: Int -> Int == (n: Int) :-> (n*n*n: Int);
```

As in the case of ordinary variables, functions can be mutable:

```
foo (n: Int): Int := n*n;
foo := (n: Int) :-> (n*n*n: Int);
```

The `return` statement can be used to exit the function with a given return value. For instance:

```
search_index (item: Int, v: Vector Int): Int == {
  for i: Int in 0..#v do
    if v[i] = item then return i;
  return -1;
}
```

3.3. MACROS

MATHEMAGIX provides the keyword `==>` for the declaration of macros. Since names of types can sometimes be rather long, macros are often used in order to abbreviate them. For instance:

```
POL ==> Polynomial Rational;
p: POL == polynomial (1, 2, 3);
```

Since such abbreviations are usually local to a file, it is customary to declare all macros at the start of the file and to use the `private` keyword in order to keep them private to the file. For instance:

```
include "numerix/rational.mmx";
include "algebrabix/polynomial.mmx";
include "algebrabix/matrix.mmx";

private {
  POL ==> Polynomial Rational;
  MAT ==> Matrix Rational;
}

// new routines on rational polynomials and matrices
```

It is also customary to capitalize all letters in names of macros.

Remark 3.1. Macros with arguments are not yet supported by the compiler, but planned.

3.4. Conditional statements

Conditional statements are of one of following two forms

```
if condition? then then_body
if condition? then then_body else else_body
```

The bodies of the then-part and the else-part can either be single instructions or blocks of instructions braced into {...}. For instance, we may write

```
if done? then {
  clean_up ();
  return;
}
```

```
fib (n: Int): Int == {
  if i <= 1 then return 1;
  else return fib (n-1) + fib (n-2);
}
```

Notice that the `if-then-else` construct can also be used as an expression:

```
mmout << "Take a " << (if warm? then "shirt" else "jacket") << lf;
```

3.5. Simple pattern matching

It often occurs that a list of actions has to be undertaken depending on the value of some expression. This kind dispatching can be achieved using the `match` instruction which has the syntax

```
match expression with match_body
```

where `match_body` is a sequence of cases of the following form:

```
case case_pattern do case_body
```

For instance:

```
fib (n: Int): Int ==
  match n with {
    case 0 do return 1;
    case 1 do return 1;
    case _ do return fib (n-1) + fib (n-2);
  }
```

This is a simple example of the general mechanism of pattern matching, which will be discussed in more details in the chapter about abstract data types.

3.6. LOOPS

Loops are constructed as follows:

```
loop_modulator_1 ... loop_modulator_n do loop_body
```

where the loop modulators are among one of the following five types:

```
for variable: T in values
for instruction
while condition?
until contition?
step instruction
```

A simple example of how to use the `for-in` modular is the following:

```
for i: Int in 1 to 10 do
  mmout << i << " * " 10 << " = " << i * 10 << lf;
```

The expression `1 to 10` is an example of a "generator". For more information on such objects, we refer to the section on generators. The `for-in` loop is equivalent to the following one which uses the `for`, `while` and `step` modulators:

```
for i: Int := 1 while i <= 10 step i := i + 1 do
  mmout << i << " * " 10 << " = " << i * 10 << lf;
```

The `for` modulator (without `in`) is really syntactic sugar: the above code is essentially the same as

```
i: Int := 1;
while i <= 10 step i := i + 1 do
  mmout << i << " * " 10 << " = " << i * 10 << lf;
```

except that the scope of the variable `i` is bound to the body of the loop when using the `for` modulator. Similarly, the `step` modulator could have been moved to the body of the loop:

```
for i: Int := 1 while i <= 10 do {
  mmout << i << " * " 10 << " = " << i * 10 << lf;
  i := i + 1;
}
```

However, this way of writing things is slightly longer and less readable. Furthermore, this kind of rewriting becomes less straightforward in presence of `continue` instructions (see below).

The condition of the `while` modulator is tested each time before executing the body of the loop. By contrast, the condition of the `until` modulor is tested only at the end of the loop. For instance, in the following loop, the body is executed at least one time:

```
until i > 10 do {
  mmout << "i= " << i << lf;
  i := i + 1;
}
```

It should also be noticed that modulators of the same type can very well be used several times. For instance, the following loop will output the numbers 4, 10 and 18:

```
v1: Vector Int == [1, 2, 3];
v2: Vector Int == [4, 5, 6];
for x1: Int in v1
for x2: Int in v2 do
  mmout << x1 * x2 << lf;
```

In this last example, even though `v1` and `v2` are not of type `Generator Int`, there exists a prefix operator `@` from `Vector Int` to `Generator Int` which allow us to write `for x1: Int in v1` and `for x2: Int in v2`.

3.7. Loop interruption and continuation

The execution of a loop can be interrupted using the `break` instruction. As soon as a `break` instruction is encountered, execution will resume at the end of the loop. For instance, the following loop will only output the numbers one until five:

```
for i: Int in 1 to 10 do {
  if i = 6 then break;
  mmout << i << lf;
}
```

In a similar way, the `continue` instruction interrupts the execution of the body of the loop, but continues with the execution of the next cycle. For instance, the following code will display all numbers from one to ten, except for the number six:f the loop. For instance, the following loop will only output the numbers one until five:

```
for i: Int in 1 to 10 do {
  if i = 6 then continue;
  mmout << i << lf;
}
```

3.8. EXCEPTIONS

Exceptions in MATHEMAGIX are handled using a conventional try-catch mechanism. This mechanism resides on three keywords:

- The "`raise exception`" instruction with one argument `exception` of an arbitrary type `T` is used in order to raise an exception of type `T`.
- The "`try try_body`" instruction protects the block `try_body` of instructions against exceptions, by allowing the user to provide exceptions handlers for exceptions of various types.
- Any number of "`catch (exception: T) catch_T_body`" instructions can occur at the end of the `try_body`. Whenever an exception of type `T` occurs, it is handled by the corresponding exception handler `catch_T_body`. Exceptions which could not be catched are propagated further outwards.

A typical example of a safe routine for printing values of a partially defined function is

```
print_values (f: Double -> Double): Void == {
  for x: Double in -5.0 to 5.0 do {
    try {
      y: Double == f x;
      mmout << x << "\t" << y << lf;
      catch (err: String) {
        mmout << x << "\t" << err << lf;
      }
    }
  }
}
```

A typical example of a corresponding partially defined function is

```
foo (x: Double): Double == {
  if x <= -2.0 then return x + 2.0;
  if x >=  2.0 then return x - 2.0;
  raise "out of range";
}
```

Applying `print_values` on `foo` yields the following output

```
-5      -3
-4      -2
-3      -1
-2      0
```

```
-1      out of range
0       out of range
1       out of range
2       0
3       1
4       2
5       3
```

Standard MATHEMAGIX libraries usually raise errors of the type `Exception` instead of `String`. For this reason, one might wish to replace the type of `err` by `Exception`. In that case the routine `foo` should be replaced by

```
foo (x: Double): Double == {
  if x <= -2.0 then return x + 2.0;
  if x >=  2.0 then return x - 2.0;
  raise exception ("out of range", x);
}
```

The second argument of `exception` allows the user to specify a reason for the exception, such as an offending value or a line in the source code which triggered the exception.

3.9. COMMENTS

MATHEMAGIX supports two types of comments. Short comments start with `//` and extend to the end of the physical line:

```
class Complex (R: Ring) {
  re: R;  // real part of the complex number
  im: R;  // imaginary part of the complex number
  ...
}
```

Long multi-line comments should be braced into `/{ ... }/` and can be nested:

```
y == hacked_function x; /{ FIXME:
  This is a dangerous hack /{ which occurs only on the first of april }/
  A skilled extraterrestian should be able to fix it
}/
```

Chapter 4

Expressions

4.1. Identifiers

4.1.1. Regular identifiers

Identifiers are used as names for variables, functions, macros, types and categories. Regular identifiers should match the regular expression `[a-zA-Z_]+[a-zA-Z0-9_$?]*`. That is, identifiers should

- only contain letters, digits and the special characters `_`, `$` and `?` ; and
- start with a letter or `_` or `$`.

In addition, it is customary to use the following guidelines when choosing names:

- Use lowercase names for variables and functions.
- For names of types and categories, capitalize the first letter of each word categories (e.g. `Integer` or `Ordered_Group`).
- Capitalize all letters in macro names.
- Use the `?` suffix for names of predicates.

Besides the regular identifiers, MATHEMAGIX allows the programmer to use several types of special identifiers for the names of operators and special objects.

4.1.2. Operators

First of all, identifiers corresponding to the built-in operators (see section 4.3 below) are formed by prefixing them by one of the keywords `prefix`, `postfix`, `infix` and `operator`. For instance:

```
postfix ! (n: Int): Int == if n=0 then 1 else n * (n-1)!;
```

Similarly, the instruction

```
mmout << map (infix *, [ 1, 2, 3 ], [ 4, 5, 6 ]) << lf;
```

prints the vector $[4, 10, 19]$. The operator

```
operator []: (t: Tuple T) -> Vector T;
```

is used as a shorthand for the constructor of vectors (so that we may write `[ 1, 2, 3 ]` instead of `vector (1, 2, 3)`). Similarly, the operator `postfix []` is used for accessing entries of vectors and matrices.

Remark 4.1. TODO: in place operators formed with the keyword `inplace`.

4.1.3. Named access operators

In addition to the builtin operators, any regular identifier `id` can also be turned into a postfix operator `postfix .id`. For instance, when defining a class `Point` by

```
class Point == {
  x: Double;
  y: Double;
  constructor point (x2: Double, y2: Double) == {
    x == x2;
    y == y2;
  }
}
```

MATHEMAGIX automatically creates two such postfix operators for accessing the fields `x` and `y`:

```
postfix .x: Point -> Double;
postfix .y: Point -> Double;
```

Hence, we may define an addition on points using

```
infix + (p: Point, q: Point): Point ==
  point (p.x + q.x, p.y + q.y);
```

Additional operators similar to `postfix .x` and `postfix .y` can be defined outside the class

```
postfix .length (p: Point): Double ==
  sqrt (square p.x + square p.y);
```

Given a point `p`, we then write `p.length` for its length as a vector.

4.1.4. Other identifiers

The MATHEMAGIX keywords, such as `while`, `class`, etc. can also be turned into identifiers by prefixing them with `keyword`. Hence, `keyword while` stands for the keyword `while`. This notation is mainly using during formal manipulations of MATHEMAGIX programs.

More generally, any valid string can be turned into an identifier by putting it between quotes and prefixing it by the keyword `literal`. For instance, `literal "sqrt"` is equivalent to the regular identifier `sqrt`, `literal "+"` is equivalent to the infix operator `infix +`, and `literal "_+_"` is an identifier which can only be written using the keyword `literal`.

We finally notice that `this` is a special identifier which denotes the underlying instance inside a class method.

4.2. Literal constants

Mathemagix provides three types of literal constants: string literals, integer literals and floating literals. In addition, there are several important constants, such as `true` and `false`, which are really identifiers from the syntactic point of view.

4.2.1. Literal strings

Short string constants are either written inside a pair of double quotes "...":

```
mmout << "Hello world" << lf;
```

Double quotes and backslashes inside strings should be escaped using backslashes:

```
quote_char    : String == "\"";
backslash_char: String == "\\";
```

Long string constants which avoid this kind of escaping can be formed using the delimiters /"..."/, as in the following example:

```
hello_world_example: String == /"
include "basix/fundamental.mmx";
mmout << "Hello world!" << lf;
"/
mmout << hello_world_example << lf;
```

4.2.2. Literal integers

An integer literal is a sequence of digits, possible preceded by a minus sign. It matches the regular expression `[-]?[0-9]+`. Examples are: `123456789123456789`, `-123`. The user should define a routine

```
literal_integer: Literal -> T;
```

in order to allow literal integers to be interpreted as constants of type `T`. The file `basix/int.mmx` of the standard library defines the routine

```
literal_integer: Literal -> Int;
```

which allows literal integers to be interpreted as machine integer constants. Arbitrary precision integers are supported by importing

```
literal_integer: Literal -> Integer;
```

for `numerix/integer.mmx`.

4.2.3. Literal floating point numbers

A literal floating point constant is a sequence of digits with a decimal point inside, an optional sign and an optionalexponent. It matches the regular expression

[-]?[0-9]+[.][0-9]+[[eE][-]?[0-9]+]?

The user should define a routine

```
literal_floating: Literal -> T;
```

in order to allow literal floating poiunt numbers to be interpreted as constants of type `T`. In particular, the files `basix/double.mmx` and `numerix/floating.mmx` from the standard library define the routines

```
literal_floating: Literal -> Double;    // in basix/double.mmx
literal_floating: Literal -> Floating;  // in numerix/floating.mmx
```

For instance,

```
zero : Double   == 0.0;
pi   : Double   == -3.14159;
funny: Floating == 1.23456790123456789012345678890e2012;
```

Notice that `0.` is not permited: one must write `0.0`.

4.2.4. Special constants

Some constants are encountered so frequently, that it is useful to mention them here, even though they are really identifiers from the syntactic point of view:

- The boolean constants `false` and `true`.
- The standard input, output and error ports `mmin`, `mmout` and `mmerr`.
- Several special control symbols for formatted output:
  - `lf` for linefeed.
  - `indent` for indenting in.
  - `unindent` for indenting out.
  - `hrule` for a horizontal ruler.

4.3. OPERATORS

Table 4.1 summarizes all standard MATHEMAGIX operators, together with their binding forces. For instance, the expression

```
a*b + c > d
```

is naturally parsed as $((a\ b)+c)>d$. The operators can roughly be divided into four groups:

1. Infix operators such as `+` apply to one argument on the left and one argument on the right.
2. Prefix operators such as negation `prefix !` apply to one argument on the right.
3. Postfix operators such as the factorial `postfix !` apply to one argument on the left.
4. Other special operators, such as `operator []` for writing vectors `[ 1, 2, 3 ]`.

There are also some special postfix operators, such as function application `postfix ()` and named access operators such as `postfix .x` (see section 4.1.3). Most operators are infix operators, so infix notation is assumed to be the default, unless stated otherwise. In the remainder of this section, we will quickly survey the intended purpose of the various operators.

Assignment operators	`==`, `:=`, `+=`, `-=`, `*=`, `/=`, `<<=`, `>>=`, `==>`, `:=>`
Function expressions	`lambda`, `:->`
Input/output operators	`<<`, `>>`, `<<<`, `>>>`, `<<*`, `<<%`
Logical implication	`=>`, `<=>`
Logical disjunction	`or`, `\/`, `xor`
Logical conjunction	`and`, `/"`
Relations	`=`, `<`, `>`, `<=`, `>=`, `!=`, `!<`, `!>`, `!<=`, `!>=`, `:`, `in`
Type conversion	`:>`, `::`, `::>`
Arrows	`->`, `~>`
Ranges	`..`, `to`, `downto`
Addition and subtraction	`+`, `-`, `@+`, `@-`
Multiplication and division	`*`, `/`, `@*`, `@/`, `div`, `quo`, `rem`, `mod`, `@`, `><`, `%`, `&`
Prefix operators	`!`, `++`, `--`, `-`, `@-`, `@`, `#`, `&`
Operate on	empty string
Power	`^`
Postfix operators	`++`, `--`, `!`, `'`, `‘`, `~`, `#`, `()`, `[]`
Tuples and vectors	`()`, `[]`

Table 4.1. Overview of all MATHEMAGIX operators listed by increasing binding force.

4.3.1. Assignment operators

The operators `==` and `:=` are used for declarations of constants and mutable variables, as described in the sections about the declaration of variables and functions. The operator `==>` is used for macro definitions (see the section about macro declarations). The operator `:=>` is reserved for future use.

The operator `:=` can always be used for the assignment of mutable variables. The operators `+=`, `-=`, `*=`, `/=`, `<<=` and `>>=` are not yet exploited in the standard libraries, but there intended use is “assignment of the left hand expression with the result of the corresponding outplace operator applied to both arguments”. For instance, the instruction

```
a += b;
```

should considered to be equivalent to the assignment

```
a := a + b;
```

Remark 4.2. As a future extension of the compiler, we also intend to support assignment to tuples, in order to assign several mutable variables at once. For instance,

```
(a, b) := (b, a)
```

should swap the variables `a` and `b`, and

```
(a, b) += (x, y)
```

should respectively increase `a` and `b` with `x` and `y`.

4.3.2. Function expressions

The special operators `:->` and `lambda` are used for writing functions directly as expressions. The expressions

```
(a_1: T_1, ..., a_n: T_n) :-> (val: Ret_Type)
lambda (a_1: T_1, ..., a_n: T_n): Ret_Type do val
```

can both be used as a notation for the function with arguments `a_1`, ..., `a_n` of types `T_1`, ..., `T_n`, which returns the value `val` of type `Ret_Type`.

4.3.3. Input/output operators

The operators `<<` and `>>` are respectively used for sending data to an output port and retrieving data from an input port. The same notation is useful in analogous circumstances, such as appending data to a vector or popping data from a stack.

The operators `<<<` and `>>>` are used for sending and retrieving data in binary form. This allows for instance for the implementation of efficient communication protocols between different processes on the same or distant computers. The operators `<<*` and `<<%` are reserved for future use.

Remark 4.3. Sometimes, we also use the operators `<<` and `>>` as shift operators. For instance, given a power series `f` in z, we might write `f << n` as a shorthand for the multiplication of `f` with z^n. However, it should be noticed that the binding force of `<<` and `>>` is not really appropriate for this type of use (a binding force similar to the one of multiplication would be better for this kind of use), so one carefully has to put brackets whereever necessary in this case. In future versions of MATHEMAGIX, this kind of overuse of notations might be discouraged.

4.3.4. Logical operators

The operators `=>`, `<=>`, `\/`, `/"` stand for the standard logical connectors $\Rightarrow$, $\Leftrightarrow$, $\wedge$ and $\vee$. The prefix operator `prefix !` stands for logical negation $\neg$. These operators are functions which can be redefined by the user, so both arguments are evaluated in case of the logical connectors.

Mathemagix also provides the built-in operators `or` and `and` which must take arguments of type `Boolean`. Moreover the second argument of `or` is evaluated only if the first argument evaluates to `false`. Similarly, the second argument of `and` is evaluated only if the first argument evaluates to `true`.

Remark 4.4. The operators `=>`, `<=>`, `\/`, `/"` and `!` can also be useful for bitwise operations on numbers, even though the binding force is someone inappropriate for this kind of use. One might also want to use `/"` as a notation for exterior products, again with an inappropriate binding force. In future versions of Mathemagix, this kind of overuse of notations might be discouraged.

4.3.5. Relations

The operators `=`, `<` , `>`, `<=` and `>=` correspond to the standard mathematical relations $=$, $<$, $>$, $\leqslant$ and $\geqslant$. When prefixing these relations with `!`, we obtain the operators `!=`, `!<`, `!>`, `!<=`, `!>=` which correspond to their negations $\neq$, $\not<$, $\not>$, $\not\leqslant$ and $\not\geqslant$.

In computational contexts, mathematical relations often admit a very asymmetric nature: typically, it can be very easy to prove inequality, but very hard to prove equality. It can even happen that we have an algorithm for proving inequality, but no known algorithm for proving equality. This is for instance the case for the class of so called exp-log constants, constructed from the rational numbers using the field operations, exponentiation and logarithm. In contexts where equality testing is hard, it is therefore useful to make a notational distinction between various types of equality, such as proven equality, probable equality, syntactic equality, etc.

In Mathemagix, the intention of the notations `=`, `<`, `>`, `<=` and `>=` is that they stand for proven relations. On the other hand, the negations `!=`, `!<`, `!>`, `!<=` and `!>=` are intended to be mere shortcuts for their (not necessarily proven) negations. Hence, `a != b` should always be equivalent to `!(a = b)`. We are working on a comprehensive set of additional relations for proven negations; they will probably be denoted by `=/`, `</`, `>/`, `<=/`, `>=/`. As an additional rule, it is our intention that `<=` (resp. `>=`) is always equivalent to the disjunction of `<` (resp. `>`) and `=`. Thus `a <= b` should always be equivalent to the statement `a < b or a = b`.

The operator `:` should be read "is of type". For instance, `x: T` stands for "`x` is of type `T`". The operator `in` occurs inside the `for`-`in` construct (see the section about loops).

4.3.6. Type conversion

The operator `:>` can be used to convert an expression of a given type into another, provided that an appropriate converter was defined. More precisely, assume that `expr` has type `S` and that we defined a converter `convert: S -> D`. Then `expr :> D` stands for the explicit conversion of `expr` into an expression of type `D`. More information about type conversions can be found in the section on explicit type conversions and user defined converters.

4.3.7. Arrows

The operator `->` is used as an efficient notation for function types, such as `Int -> Int`. One typical use case of this notation is when a function is passed as an argument to another function:

```
iterate (f: Int -> Int, n: Int) (k: Int) ==
  if n = 0 then k else iterate (f, n-1) (f k);
```

The operator ~> is mainly used as a notation for key-value bindings whenever we explicitly wish to create a table with given entries. For instance:

```
basic_colors: Table (String, Color) ==
  table ("red" ~> rgb (1, 0, 0),
         "green" ~> rgb (0, 1, 0),
         "blue" ~> rgb (0, 0, 1),
         "white" ~> rgb (1, 1, 1));
```

or

```
forall (T: Type)
invert (t: Table (T, T)): Table (T, T) ==
  table (t[key] ~> key | key: T in t);
```

4.3.8. Ranges

There are three standard kinds of range operators:

`start to end`	Range from `start` up to `end` included
`start .. end`	Range from `start` up to `end` not included
`start downto end`	Range from `start` down to `end` included

4.3.9. Arithmetic operations

The standard arithmetic operations `+`, `-`, `*`, `/` and `^` stand for addition, subtraction, multiplication, division and powering. The `@`-prefixed variants `@+`, `@-`, `@*`, `@/` stand for $\oplus$, $\ominus$, $\otimes$ and $\oslash$. Notice that `-` and `@-` can either be infix or prefix operators.

MATHEMAGIX provides the additional operators `div`, `quo`, `rem` and `mod` for division-related operations in rings which are not necessarily fields. The operator `div` stands for (usually partially defined) exact division. For instance, `numerix/integer.mmx` provides the operation

```
infix div: (Integer, Integer): Integer;
```

but `5 div 3` is undefined and might raise an error. The operators `quo` and `rem` stand for quotient and remainder in euclidean domains. Hence, we should always have

$$a = (a \operatorname{quo} b)\, b + (a \operatorname{rem} b).$$

The operator `mod` stands for modular reduction, so that the return type is usually different from the source types. For instance `5 mod 3` would typically belong to `Modular (Int, 3)` or `Modular (Integer, 3)`.

There are a few other operations with the same binding force as multiplication. The append operator `><`, also denoted by $\bowtie$, is typically used for appending strings, vectors and table. For instance `"a" >< "b"` yields `"ab"`. The operator `infix @` is used for functional composition, whereas the operators `%` and `&` are reserved for future use.

4.3.10. Prefix operators

The standard prefix operators in MATHEMAGIX are `prefix !` (negation ¬), `prefix ++` (increment), `prefix --` (decrement), `prefix -` (unary −), `prefix @-` (unary ⊖), `prefix @` (explode), `prefix #` (size), `prefix &` (reserved for future use).

In addition, operator application of the form `sin x` parses in a similar way as when `sin` behaves as a prefix operator. For instance, `sin cos x` should be parsed as $\sin(\cos(x))$.

4.3.11. Postfix operators

The standard postfix operators in MATHEMAGIX are `postfix ++` (post increment), `postfix --` (post decrement), `postfix !` (factorial !), `postfix '` (quote or derivative), `postfix ‘` (unquote), `postfix ~` and `postfix #` (reserved for future use).

`++`, `--`, `!`, `'`, `‘`, `~`, `#`, `()`, `[]`

In addition, MATHEMAGIX provides the special postfix operators `postfix ()` and `postfix []` for which we are allowed to put additional arguments between the brackets. Hence, `postfix ()` stands for the traditional notation of function application, whereas `postfix []` is typically used as an accessor for compound data structures. Notice that `f(g)(x)` is parsed as $(f(g))(x)$, whereas `f g x` is parsed as $f(g(x))$.

Using the operator `postfix ()`, we may generalize the classical notation for function application to user defined types, such as vectors of functions:

```
postfix () (v: Vector (Int -> Int), x: Int): Vector Int ==
  [ f x | f: Int -> Int in v ];
```

4.3.12. Tuples and vectors

The reserved special operator `operator ()` is used for building tuples of expressions (of possibly different types), such as `(1, "hello")`. The special operator `operator []` is used as a notation for explicit vectors, such as `[ 1, 2, 3 ]`, but it might be used for other purposes.

4.4. Generators

Generators are an elegant way for representing a stream of data of the same type. For instance, the expression `1 to 10` of type `Generator Int` allows us to write

```
for i: Int in 1 to 10 do
  mmout << i << " * " << 7 << " = " << 7*i << lf;
```

MATHEMAGIX provides several constructs for forming generators.

4.4.1. Range generators

There are three standard kinds of range operators:

`start to end`	Range from `start` up to `end` included
`start .. end`	Range from `start` up to `end` not included
`start downto end`	Range from `start` down to `end` included

4.4.2. The explode operator

Many container types come with a prefix operator `@` which returns a generator. For instance, given a vector `v` of type `Vector T`, the expression `@v` has type `Generator T`. Whenever `expr` is an expression such that `@expr` has type `Generator T`, we are still allowed to use `expr` as the `in`-part of the `for-in` construct. For instance:

```
for i: Int in [ 2, 3, 5, 7, 11, 13, 19 ] do
  mmout << i*i << lf;
```

4.4.3. The such that construct

One other important construct for forming generators is the "such that" operator `|`. Given a generator `g` of type `Generator T`, the expression

```
(var: T in g | predicate? var)
```

stands for the generator of all items in `g` which satisfy the predicate `predicate?`. For instance, consider the following naive implementation of the predicate `prime?` which checks whether a number is prime

```
prime? (n: Int): Boolean == {
  for i: Int in 2..n do
    if i rem i = 0 then return false;
  return true;
}
```

Then we may display the vector of all prime numbers below 1000 using

```
mmout << [ p: Int in 1 to 1000 | prime? p ] << lf;
```

Notice that this vector is constructed from the expression

```
(p: Int in 1 to 1000 | prime? p)
```

of type `Generator Int` using the bracket operator `operator []`.

4.4.4. The where construct

The vertical bar `|` can also be used as the "where" operator, using the following syntax:

```
(expr var | var: T in g, predicate_1? var, ..., predicate_n? var)
```

Here `g` is again a generator of type `Generator T`, `expr var` any expressions which involves `var`, and `predicate_1? var`, ..., `predicate_n? var` an arbitrary number of predicates which involve `var`. If `expr var` has type `U`, then the resulting expression has type `Generator U`. For instance,

```
mmout << [ i^2 | i: in 1 to 100 ] << lf;
```

displays the vector of all squares of numbers from 1 to 100, and

```
mmout << [ i^2 | i: in 1 to 1000, prime? (4*i + 3) ]
```

displays the square of each number i such that $4\,i+3$ is prime.

4.4.5. The where construct with multiple generators

MATHEMAGIX actually supports a generalization of the where construct with multiple generators and predicates at the right-hand side. This generalization is best illustrated with an example:

```
mmout << [ p^i | p: Int in 1 to 1000, prime? p,
                 i: Int in 1 to 10, p^i < 1000 ] << lf;
```

This code will print the unordered vector of all prime powers below 1000.

4.4.6. Matrix notation

A special where notation `||` is used for generators which allow to build rows of matrices or similar two dimensional structures. Again, this notation is best illustrated with an example. Assuming that the file `algebramix/matrix.mmx` was included, the expression

```
[i+j | i: Int in 0 to 9 || j: Int in 0..10]
```

computes the following matrix:

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\ 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 \\ 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \end{bmatrix}$$

4.5. Mappers

Most MATHEMAGIX containers implement a mapping construct `map`. This construct is used for applying one or more functions to all entries of one or more containers.

Two simple examples for containers with a single parameter are

```
mmout << map (square, [ 1, 2; 3, 4 ]) << lf;
mmout << map (infix *, [ 1, 2, 3], [4, 5, 6]) << lf;
```

These instruction respectively output the matrix $\begin{pmatrix} 1 & 4 \\ 9 & 16 \end{pmatrix}$ and the vector $[4, 10, 18]$.

In the case of containers with more than one type parameter, one usually has to provide one mapping function for every such parameter. Consider for instance the following table:

```
t: Table (Int, String) == table (3 ~> "Hello", 4 ~> "Hi", 8 ~>
"Bonjour");
```

Then the instruction

```
mmout << map (square, prefix #, t) << lf;
```

prints the table $[9 \rightsquigarrow 5, 16 \rightsquigarrow 2, 64 \rightsquigarrow 7]$.

Syntactically speaking, the construct `map` is an ordinary identifier. For instance, assuming that we defined a container `Complex R` (see the section on how to define your own containers), a unary mapper for this container can be defined as follows:

```
forall (R1: Ring, R2: Ring)
map (f: R1 -> R2, z: Complex R1): Complex R2 ==
  complex (f z.re, f z.im);
```

When providing your own containers, it is actually important to define unary mappers of this kind, because such mappers automatically induce converters between containers of the same kind but with different type parameters. For instance, given a converter from `R1` to `R2`, the above mapper for complex numbers automatically induces a converter from `Complex R1` to `Complex R2`. This allows the user to write

```
z: Complex Rational == complex (1, 2);
```

In general, such a converter is constructed whenever the user provides a unary mapper which takes one mapping function for each parameter on input together with a single container.

Remark 4.5. We notice that the existence of a unary mapper is mandatory if a program both uses the container in an generic and in a specialized way, and if conversions between the generic and specialized versions of the container indeed occur in the program. For instance, some mathematical library `lib.mmx` might provide a generic function

```
forall (R: Ring)
conj (z: Complex R): Complex R == complex (z.re, -z.im);
```

Now assume that we a client program `client.mmx` which only works with complex numbers of type `Complex Double` and which has specialized this type for better performance. In memory, this means that such complex numbers are represented by pairs of double precision numbers rather than pairs of pointers to double precision numbers numbers as would be the case for generic complex numbers. However, the routine `conj` from `lib.mmx` *a priori* only applies to generic complex numbers, so conversions between the specialized and the generic view are necessary if we want to use this routine in `client.mmx`. As soon as the required unary mapper is defined, these conversions are automatic.

Chapter 5

Functions

5.1. Function declarations

5.1.1. Basic function declaration and application

The most basic form of a function declaration in Mathemagix is

```
fun_name (arg_1: Type_1, ..., arg_n: Type_n): Ret_Type == fun_body
```

The function name `fun_name` can be an arbitrary identifier, such as `hello`, `test?` or `infix +`. Using the keywords `:->` or `lambda`, it is also possible to construct function expressions. For instance, the following three declarations are equivalent:

```
cube (x: Int): Int == x*x*x;
cube: Int -> Int == (x: Int) :-> (x*x*x: Int);
cube: Int -> Int == lambda (x: Int): Int do x*x*x;
```

We recall that Mathemagix provides two syntaxes for function application: the classical syntax `f(x)` and the operator syntax `f x`. In the case of nested applications, these two syntaxes use a different grouping: whereas `f(x)(y)` parses as `(f(x))(y)`, the expression `f g x` should be parsed as `f(g(x))`.

5.1.2. Tuple and generator arguments

Functions with an arbitrary number of arguments of a given type can be formed by using so called tuple types. More precisely, assume a function declaration of the form

```
fun_name (arg_1: Type_1, ..., arg_n: Type_n, x: Tuple X): Ret_Type ==
fun_body
```

Then the function `fun_name` applies to n first arguments of types `Type_1` until `Type_n` and k optional arguments which are all of type `X`. For instance, the routine

```
extract (v: Vector Double, t: Tuple Int): Vector Double ==
  [ v[i] | i: Int in [t] ];
```

can be used to extract the vector of all entries with given indices from a given vector. Hence,

```
v: Vector Double == [ 1.0, 2.0, 6.0, 3.14, 2012.0 ];
mmout << extract (v, 1, 2, 3, 3, 0) << lf;
```

prints the vector $[2.0, 6.0, 3.14, 3.14, 1.0]$.

In a similar way, one may pass a generator instead of a tuple as a last argument of a function.

5.1.3. Recursive functions

Functions definitions are allowed to be recursive, such as the following routine for computing the factorial of an integer:

```
postfix ! (n: Integer): Integer ==
  if n <= 1 then 1 else n * (n-1)!;
```

Functions which are defined in the same scope are also allowed to be mutually recursive. An example of mutually recursive sequences are Hofstadter's female and male sequences F_n and M_n defined by $F_0 = 1$, $M_0 = 0$ and

$$\begin{aligned} F_n &= n - M_{F_{n-1}} \\ M_n &= n - F_{M_{n-1}}, \end{aligned}$$

for $n > 0$. They can be implemented in MATHEMAGIX as follows:

```
F (n: Integer): Integer == if n = 0 then 1 else n - M F (n-1);
M (n: Integer): Integer == if n = 0 then 0 else n - F M (n-1);
```

In large multiple file programs, it sometimes happens that the various mutually recursive functions are defined in different files, and thus in different scopes. In that case, prototypes of the mutually recursive functions should be defined in a file which is included by all files where the actual functions are defined. Protypes of functions are declared using the syntax

```
fun (arg_1: Src_1, ..., arg_n: Src_n): Dest;
```

In case of the above example, we might define prototypes for `F` and `M` in a file `common.mmx`:

```
F (n: Integer): Integer;
M (n: Integer): Integer;
```

Next, a first file `female.mmx` would include `common.mmx` and define `F`:

```
include "common.mmx";
F (n: Integer): Integer == if n = 0 then 1 else n - M F (n-1);
```

In a second file `male.mmx`, we again include `common.mmx` and define `M`:

```
include "common.mmx";
M (n: Integer): Integer == if n = 0 then 0 else n - F M (n-1);
```

5.1.4. Dependent arguments and return values

Besides mutually recursive function definitions, MATHEMAGIX also allows for dependencies among the arguments of a function and dependencies of the return type on the arguments. Although the dependencies among the arguments may occur in any order, mutual dependencies are not allowed.

For instance, the following routine takes a ring together with an element of this ring on input and displays the first n powers of this element:

```
first_powers (x: R, R: Ring, n: Int): Void == {
  p: R := x;
  for i: Int in 1 to n do {
    mmout << i << " -> " << p;
    p := x * p;
  }
}
```

In this example, the type `Ring` of `R` is a category. We refer to section 5.4.1 and the chapter about categories for a declaration of this category and more details on how to use them.

The following code defines a container `Fixed_Size_Vector (T, n)` for vectors with entries of type `T` and a fixed size `n: Int`.

```
class Fixed_Size_Vector (T: Type, n: Int) == {
  mutable rep: Vector T;
  constructor fixed_size_vector (c: T, n: Int) == {
    rep == [ c | i: Int in 0..n ];
  }
}
```

The return type `Fixed_Size_Vector (T, n)` of the constructor `fixed_size_vector` depends on the argument `n` to the same constructor.

5.2. Functional programming

Functions are first class objects in MATHEMAGIX, so they can consistently be used as arguments or return values of other functions. They can also be declared locally inside other functions or used as constant or mutable fields of user defined classes.

5.2.1. Functions as arguments

A simple example of a function which takes a function predicate as an argument is

```
filter (v: Vector Int, pred?: Int -> Boolean): Vector Int ==
  [ x: Int in v | pred? x ];
```

The `map` construct systematically exploits this possibility to use functions as arguments. For instance, the following instruction prints the vector $[4, 10, 18]$:

```
mmout << map (infix *, [1, 2, 3], [4, 5, 6]) << lf;
```

5.2.2. Functions as return values

A typical example of a function which returns another function is

```
shift (x: Int): Int -> Int == (y: Int) :-> (x+y: Int);
```

This kind of functions are so common that MATHEMAGIX provides a special syntax for it, which generalizes the syntax of basic function declarations:

```
fun_name (arg_11: Type_11, ..., arg_1n1: Type_1n1)
         ...
         (arg_k1: Type_k1, ..., arg_knk: Type_knk): Ret_Type == fun_body
```

This syntax allows to simplify the definition of `shift` into

```
shift (x: Int) (y: Int): Int == x+y;
```

5.2.3. Functions as local variables

In a similar way that functions can be used as arguments or return values of other functions, it is possible to locally define functions inside other functions. One typical example is

```
shift_all (v: Vector Int, delta: Int): Vector Int == {
  shift (x: Int): Int == x + delta;
  return map (shift, v);
}
```

Recursion and mutual recursion are still allowed for such local function declarations. For instance, we may generalize Hofstadter's example of female and male sequences by allowing the user to choose arbitrary initial conditions:

```
FM (n: Int, init_F: Int, init_M: Int): Int == {
  F (n: Integer): Integer == if n = 0 then init_F else n - M F (n-1);
  M (n: Integer): Integer == if n = 0 then init_M else n - F M (n-1);
  return F n;
}
```

5.2.4. Mutable functions

As in the case of ordinary variables, functions can be declared to be mutable, using the syntax

```
fun_name (arg_1: Type_1, ..., arg_n: Type_n): Ret_Type := fun_body
```

In that case, the function can be replaced by another function during the execution of the program:

```
foo (n: Int): Int := n*n;
foo := (n: Int) :-> (n*n*n: Int);
```

In section 5.6 below, we will how the mechanism of conditional overloading can exploit this possibility to dynamically replace functions by others.

5.3. Discrete overloading

In classical mathematics, operators such as + are heavily overloaded, in the sense that the same notation can be used in order to add numbers, polynomials, matrices, and so on. One important feature of Mathemagix is that it has powerful support for overloaded notations, which allows the programmer to mimick classical mathematical notations in a faithful way.

The simplest mechanism of *discrete overloading* allows the user to redefine the same symbol several times with different types. For instance,

```
dual: Int    == 123;
dual: String == "abc";
```

In this case, the variable `dual` can both be interpreted as a machine integer and as a string. For instance, assuming the above declaration, the following code is correct:

```
hop : Int    == dual + 1;
hola: String == dual >< "def";
```

Indeed, in the definition of `hop` (and similarly for `hola`), the code `dual + 1` only makes sense when `dual` is of type `Int`, so the correct disambiguation can be deduced from the context. On the other hand, the instruction

```
mmout << dual << lf;
```

is ambiguous and will provoke an error message of the compiler. In such cases, we may explicitly disambiguate `dual` using the operator `:>`. Both the following two lines are correct:

```
mmout << dual :> Int << lf;
mmout << dual :> String << lf;
```

In case of doubt, successions of disambiguations, such as `123 :> Integer :> Rational` can be useful in order to make the meaning of an expression clear.

Of course, overloading is most useful in the case of functions, and the mechanism described above applies in particular to this case. For instance, we may define

```
twice (x: Int): Int == 2*x;
twice (s: String): String == s >< s;
```

Then we may write

```
mmout << twice 111 << lf;
mmout << twice "hello" << lf;
```

Overloaded functions can very well be applied to overloaded expressions. For instance, the expression `twice dual` admits both the types `Int` and `String`. We may thus write

```
plok: Int == twice dual;
mmout << twice dual :> String << lf;
```

It should also be noticed that both the arguments and the return values of functions can be overloaded. For instance, we may overload `twice` a third time

```
twice (x: Int): String == twice as_string x;
```

After this, the expression `twice 111` can both be interpreted as the number 222 and as the string `"111111"`.

5.4. Parametric overloading

5.4.1. The forall construct

Besides discrete overloading, MATHEMAGIX also features parametric overloading, based on the `forall` construct. In this case, the overloaded value no longer takes a finite number of possible types, but rather an infinite number of possible types which depend on one or more parameters.

The general syntax for making one or more parametrically overloaded declarations is

```
forall (param_1: Type_1, ..., param_n: Type_n) declarations
```

The parameters `param_1`, ..., `param_n` are usually types themselves, in which case their types are so called *categories*. For instance, consider the following declaration:

```
forall (R: Ring) cube (x: R): R == x*x*x;
```

It states that for any type `R` which has the structure of a ring, we have a function `cube: R -> R`. Parametrically overloaded functions such as `cube` are also called templates. The conditions for `R` to be a ring are stated by declaring the category `Ring`. One possible such declaration is the following:

```
category Ring == {
  convert: Int -> This;
  prefix -: This -> This;
  infix +: (This, This) -> This;
  infix -: (This, This) -> This;
  infix *: (This, This) -> This;
}
```

In this case, any type `R` with the usual operations $+, -, \cdot$ and a converter `Int -> R` will be considered to be a ring. The mere presence of these operations in the current context is sufficient: the compiler does not check any of mathematical ring axioms.

Assuming a context in which both the types `Int` and `Double` are present, we may apply the template `cube` as follows:

```
mmout << cube 123 << lf;       // applies cube: Int -> Int
mmout << cube 1.0e100 << lf;   // applies cube: Double -> Double
```

Although this is usually not necessary, it is sometimes useful to make the values of the parameters explicit, for instance in order to make expressions less ambiguous. This can be done using the # infix operator. For instance,

```
mmout << cube#Int (123) << lf;
mmout << cube#Double (1.0e100) << lf;
```

5.4.2. Grouping forall statements

It often happends that several generic routines share the same parameters. In that case, they can be grouped together in a common `forall` block. For instance, given a ring R, assume that we are developing a container `Tangent R` for arithmetic in $R[\varepsilon]/(\varepsilon^2)$ (see also the section on container classes):

```
class Tangent (R: Ring) == {
  b: R;  // base point (coefficient of 1)
  s: R;  // slope      (coefficient of epsilon)
  constructor tangent (b2: R) == { b == b2;  == 0; }
  constructor tangent (b2: R, s2: R) == { b == b2; s == s2; }
}
```

Then we may define a ring structure on `Tangent R` using

```
forall (R: Ring) {
  convert (i: Int): Tangent R ==
    tangent (i :> R);
  prefix - (x: Tangent R): Tangent R ==
    tangent (-x.b, -x.s);
  infix + (x: Tangent R, y: Tangent R): Tangent R ==
    tangent (x.b + y.b, x.s + y.s);
  infix - (x: Tangent R, y: Tangent R): Tangent R ==
    tangent (x.b - y.b, x.s - y.s);
  infix * (x: Tangent R, y: Tangent R): Tangent R ==
    tangent (x.b * y.b, x.b * y.s + x.s * y.b);
}
```

5.4.3. Additional assumptions on parameters

It often happens that template parameters need to fulfill several requirements, such as being a ring and an ordering at the same time. MATHEMAGIX provides the keyword `assume` for this purpose. For example:

```
forall (R: Ring)
assume (R: Ordering)
operator <= (x: Tangent R, y: Tangent R): Boolean ==
  x = y or x.b < y.b;
```

Such additional assumptions can naturally be included in `forall` blocks:

```
forall (R: Ring) {
  ...
  infix * (x: Tangent R, y: Tangent R): Tangent R ==
    tangent (x.b * y.b, x.b * y.s + x.s * y.b);

  assume (R: Ordering)
  operator <= (x: Tangent R, y: Tangent R): Boolean ==
    x = y or x.b < y.b;
}
```

5.4.4. Partial specialization

It frequently occurs that for specific values of the template parameters, the generic implementation of the template can be further improved. For instance, consider the following generic implementation of a power operations on field elements:

```
forall (F: Field)
infix ^ (x: F, i: Int): F == {
  if i = 0 then return 1;
  else if i > 0 then {
    s: F == square (x^(i quo 2));
    if i rem 2 = 0 then return s;
    else return x * s;
  }
  else return (1 :> F) / x^(-i);
}
```

This implementation uses binary powering and is more or less as efficient as it can get for elements in a generic field. However, in the "field" `Floating` of arbitrary precision floating point numbers, we have fast implementations of the operations `exp` and `log`, so the following implementation is even more efficient in this specific case:

```
infix ^ (x: Floating, i: Int): Floating == {
  if x > 0 then
    return exp (i * log x);
  else if x < 0 then {
    if i rem 2 = 0 then return (abs x)^i;
    else return -(abs x)^i;
  }
  else {
    if i >= 0 then return 0;
    else return 1.0 / 0.0;
  }
}
```

MATHEMAGIX allows the above two implementations to happily coexist, thanks to the mechanism of partial specialization. Without this mechanism, any expression of the form `x^i` with `x: Floating` and `i: Int` would be ambiguous, since both implementations of `infix ^` allow for the interpretation of `x^i` as an expresson of type `Floating`. The idea behind partial specialization is that we always prefer the most particular (i.e. "best") implementation, in this case the second one.

In general, a first template (or function) is more particular than a second one, if any possible type (i.e. by substituting concrete values for the parameters) of the first template is also a possible type of the second one. For instance, the first above implementation of `infix ^` is not more particular than the second one, since the the type `(Rational, Int) -> Rational` of `(infix ^) # Rational` is not a possible type of the second implement of `infix ^`.

It should be noticed that the relation "is more particular than" is only a partial ordering. For instance, none of the two following routines is more particular than the other one:

```
forall (R: Ring) mul (i: Int, x: R): R == (i :> R) * x;
forall (R: Ring) mul (x: R, i: Int): R == x * (i :> R);
```

Applying the function `mul` to two elements of type `Int` would therefore be ambiguous. This ambiguity can be removed by implementing the routine

```
mul (i: Int, j: Int): Int == i * j;
```

Indeed, this routine is more particular than each of the two generic implementations of `mul`, so it will be the preferred implementation whenever `mul` is applied to two elements of type `Int`.

It should be noticed that the relation "is more particular than" does not necessarily mean that some of the parameters have to be substituted by actual values in order to become "more particular". For instance, consider the prototypes of two templates for the computation of determinants:

```
forall (R: Ring)  det: Matrix R -> R;
forall (F: Field) det: Matrix F -> F;
```

Then the second template is more particular than the first one, so it will be the preferred implementation when computing the determinant of a matrix with entries in a field.

5.5. Type conversions

5.5.1. Implicit conversions

In Mathemagix, the special operator `convert` is used for type conversions. For instance, given an integer `x: Integer` and a converter

```
convert: Integer -> Rational;
```

we may use the expression `x :> Rational` in order to explicitly convert `x` into a rational number.

The operator `convert` is used for implicit type conversions only under the following particular circumstances:

- During the declaration of variables. With `x: Integer`, we may thus write

  ```
  a: Rational == x;
  b: Rational := square x + 3;
  ```

- During assignments:

```
b: Rational := x;
b := x * x;
```

- When returning from a function:

```
f (x: Integer): Rational == x + 1;
g (x: Integer): Rational == {
  if x >= 0 then return x;
  else return 1/x;
}
```

- Inside the `for-in` construct:

```
for x: Rational in 1 to 10 do
  mmout << x << " -> " << 1/x << lf;
```

5.5.2. Explicit conversions

Except for the above special cases, MATHEMAGIX does not perform any implicit conversions. For instance, even if we have an implicit converter, then application the following function cannot be applied to an expression of type `Integer`:

```
foo (x: Rational): Rational == x + 1/x;
```

Nevertheless, using the mechanism of parametric overloading, we may define `foo` in the following way so as to make this possible:

```
forall (F: To Rational)
foo (x_orig: F): Rational == {
  x: Rational == x_orig :> Rational;
  x + 1/x;
}
```

Here `To T` stands for the following parameterized category:

```
category To (T: Type) == {
  convert: This -> T;
}
```

The new version of `foo` cannot only be applied to expressions of type `Integer`, but to any expression of a type `F` with a converter `convert: F -> Rational`.

The above way of adapting function declarations so as to accept convertable arguments is so common that MATHEMAGIX provides a special syntax for it. This syntax allows us to simplify the second declaration of `foo` into

```
foo (x :> Rational): Rational == x + 1/x;
```

We call this mechanism *a priori* type conversion of function arguments.

A similar syntax may be used for *a posteriori* type conversion of the return value:

```
bar (i: Integer) :> Integer == 1 - i;
```

This definition is equivalent to

```
forall (T: From Integer)
bar (i: Integer): T == (1 - i) :> T;
```

where

```
category From (F: Type) == {
  convert: F -> This;
}
```

5.5.3. On the careful use of type conversions

The mechanisms of *a priori* and *a posteriori* type conversions are powerful, but one should be careful not to abuse them. For instance, at a first sight, it may be tempting to allow for *a priori* type conversions for all routines on rational numbers:

```
infix + (x :> Rational, y :> Rational): Rational == ...;
infix - (x :> Rational, y :> Rational): Rational == ...;
infix * (x :> Rational, y :> Rational): Rational == ...;
infix / (x :> Rational, y :> Rational): Rational == ...;
...
```

Indeed, this would immediately give us support for the notation `x + 1` whenever `x` is a rational number. However, this kind of abuse would quickly lead to ambiguities, since it also allows the addition on rational numbers to be applied to two integers. Although many of these ambiguities are automatically resolved by the partial specialization mechanism, they tend to become a serious source of problems in more voluminous mathematical libraries with many types and heavily overloaded notaions.

Besides the semantic correctness issue, there is also a performance issue: the compiler has to examine all possible meanings of ambiguous expressions and then determine the preferred ones among them. It is therefore better to reduce potential ambiguities as much as possible beforehand. In the above case, this can for instance be achieved by using the following declarations instead:

```
infix + (x: Rational, y :> Rational): Rational == ...;
infix - (x: Rational, y :> Rational): Rational == ...;
infix * (x: Rational, y :> Rational): Rational == ...;
infix / (x: Rational, y :> Rational): Rational == ...;
...
infix + (x :> Rational, y: Rational): Rational == ...;
infix - (x :> Rational, y: Rational): Rational == ...;
infix * (x :> Rational, y: Rational): Rational == ...;
infix / (x :> Rational, y: Rational): Rational == ...;
...
```

In order to avoid the same kind of ambiguity as in the `mul` example from section 5.4.4, we will also have to provide the routines

```
infix + (x: Rational, y: Rational): Rational == ...;
infix - (x: Rational, y: Rational): Rational == ...;
infix * (x: Rational, y: Rational): Rational == ...;
infix / (x: Rational, y: Rational): Rational == ...;
...
```

In fact, most converters of a type `T` into `Rational` are actually compositions of a converter of `T` into `Integer` and the standard converter of `Integer` into `Rational`. Therefore, an even better idea is to replace the block of declarations with *a priori* conversions by

```
infix + (x: Rational, y :> Integer): Rational == ...;
infix - (x: Rational, y :> Integer): Rational == ...;
infix * (x: Rational, y :> Integer): Rational == ...;
infix / (x: Rational, y :> Integer): Rational == ...;
...
infix + (x :> Integer, y: Rational): Rational == ...;
infix - (x :> Integer, y: Rational): Rational == ...;
infix * (x :> Integer, y: Rational): Rational == ...;
infix / (x :> Integer, y: Rational): Rational == ...;
...
```

Indeed, besides the fact that we eliminate all possible ambiguities in this way, the above routines also admit more efficient implementations. In a similar way, for container types such as `Polynomial R`, we usually have special implementations for scalar operations:

```
forall (R: Ring) {
  infix + (p: Polynomial R, c :> R): Polynomial R == ...;
  infix - (p: Polynomial R, c :> R): Polynomial R == ...;
  infix * (p: Polynomial R, c :> R): Polynomial R == ...;
  ...
  infix + (c :> R, p: Polynomial R): Polynomial R == ...;
  infix - (c :> R, p: Polynomial R): Polynomial R == ...;
  infix * (c :> R, p: Polynomial R): Polynomial R == ...;
  ...
}
```

The compiler has been optimized so as to take advantage of the reduced amount of ambiguities when overloading operations in this way. This should lead to an appreciable acceleration of the compilation speed, provided that the programmer adopts a similar style when using the mechanism of *a priori* type conversions.

5.6. Conditional overloading

5.6.1. Conditional overloading of constant functions

Until now, we have only considered overloading based on the types of expressions. The mechanism of conditional overloaded allows us to overload functions based on dynamically evaluated conditions on values. Let us start with the simple example of Fibonacci numbers:

```
fib (n: Int): Int ==
  if n <= 1 then 1 else fib (n-1) + fib (n-2);
```

This example can be reimplemented using the mechanism of conditional overloading as follows:

```
fib (n: Int): Int == fib (n-1) + fib (n-2);  // general implementation
fib (n: Int | n <= 1): Int == 1;             // overloaded version
```

The idea here is to first specify a general implementation of the function, which can later be adapted to special cases. The overloaded versions of the function are potentially parts of other files.

In general, the syntax for a conditionally overloaded function declaration is

```
fun (arg_1: T_1, ..., arg_n: T_n | cond_1, ..., cond_k): R == body
```

where `cond_1`, ..., `cond_k` are conditions in `arg_1`, ..., `arg_n`. Inside the body of the overloaded function (or template), the special identifier `former` may be used as a name for the previous (fallback) version of the function (or template), before the overloading took place. In particular, the above overloaded declaration of `fun` is equivalent to

```
fun (arg_1: T_1, ..., arg_n: T_n): R ==
  if cond_1 and ... and cond_k then body
  else former (arg_1, ..., arg_n);
```

In the case when the function was not declared before, the function `former` simply raises an error whenever we call it. Hence, the first definition of the function is recommended to be an unconditional one, as in the introductory example `fib`.

Remark 5.1. The conditionally overloaded declarations are processed exactly in the same order as the declarations appear in the source file. In the case when these declarations are spread over several files, only the ones which explicitly occur in the inclusions of the current file matter, and they will be processed in the same order as we did the inclusions. In a given context, let

```
fun (arg_1: T_1, ..., arg_n: T_n |
     conds_11, ..., conds_1k1): R == body_1
...
fun (arg_1: T_1, ..., arg_n: T_n |
     conds_q1, ..., conds_qkq): R == body_q
```

be the sequence of all conditionally overloaded declarations of the same function symbol `fun` with a fixed type, ordered in the above way. Then the above sequence of overloaded declarations is equivalent in that context to the single declaration

```
bar (arg_1: T_1, ..., arg_n): R == {
  if      conds_q1 and ... and conds_qkq then body_q
  else if ...
  else if conds_11 and ... and conds_1k1 then body_1
  else    raise some_error;
}
```

A similar remark applies in the case of function templates.

5.6.2. Conditional overloading of mutable functions

The mechanism of conditional overloading also applies in the case of mutable functions, but with a slightly different semantics. The general syntax for a conditionally overloaded mutable function declaration is

```
fun (arg_1: T_1, ..., arg_n: T_n | cond_1, ..., cond_k): R := body
```

In the present case, such a declaration is equivalent to

```
fun := lambda (former: (T_1, ..., T_n) -> R): (T_1, ..., T_n) -> R do
         (lambda (arg_1: T_1, ..., arg_n: T_n): R do
            if arg_1 and ... and arg_n then body
            else former (arg_1, ..., arg_n)) (fun);
```

At initialization, `fun` contains a function which throws an error message for all inputs. There are two important difference with the semantics described in the previous section:

1. Nothing prevents the user to modify the value of `fun` elsewhere in the program; after all, `fun` is a mutable variable.

2. In the case when the conditionally overloaded mutable function declarations are spread over several files, the current value of the mutable function is stored in a unique global variable which is common to all files. In particular, its value does not depend on the context, and only depends on the order in which the various files in the project are initialized (and on any other assignments of the mutable function that might occur; see the previous point).

Let us illustrate this difference with an example. Assume that we have four files `a.mmx`, `b.mmx`, `c.mmx` and `d.mmx` with the following contents:

`a.mmx`.

```
f (i: Int): Int == 1;
```

`b.mmx`.

```
include "a.mmx";
f (i: Int | i = 2): Int == 2;
b (): Void == mmout << [ f 1, f 2, f 3 ];
```

`c.mmx`.

```
include "a.mmx";
f (i: Int | i = 3): Int == 3;
c (): Void == mmout << [ f 1, f 2, f 3 ];
```

`d.mmx`.

```
include "b.mmx";
include "c.mmx";
b();
c();
```

Execution of the program `d.mmx` yields

```
[ 1, 2, 1 ]
[ 1, 1, 3 ]
```

If we replace `==` by `:=` in all overloaded declarations of `f`, then we obtain the output

```
[ 1, 2, 3 ]
[ 1, 2, 3 ]
```

In other words, in the first case, each individual file is only aware of the overloaded declarations that occurred in the file itself and in all recursively included files. In the second case, `f` is a global mutable variable which is shared by all files.

In an interactive editor such as TEXMACS, conditional overloading of mutable functions is very useful, because we may use it to customize the behaviour of common editing actions as a function of the context. For instance, key presses might be handled by a global function

```
key_press (key: String) := insert_character (key);
```

Handlers for particular keys may then be defined *whereever* appropriate

```
key_press (key: String | key = "enter") := insert_newline ();
```

Similarly, special behaviour may be defined inside particular contexts. For instance, in a computer algebra session, pressing "enter" should evaluate the current input:

```
key_press (key: String | key = "enter", inside_shell_input? ()) :=
  evaluate_current_input ();
```

Of course, attention should be paid to the declaration order: the most general routines should be declared first if we don't want them to be overridden.

5.6.3. Conditional overloading of function templates

The conditional overloading mechanism also applies to (constant) function templates. The syntax for a conditionally overloaded function template declaration is

```
forall (P_1: C_1, ..., P_p: C_p)
tmpl (arg_1: T_1, ..., arg_n: T_n | cond_1, ..., cond_k): R == body
```

Mutable function templates are not supported.

5.6.4. Performance issues

In case of the mechanisms of discrete and parametric overloading, the actual resolution of the overloaded expressions (that is, the process of assigning disambiguous meanings to all subexpressions) is done during the compilation phase. This makes this kind of overloading very efficient: no matter how many times a function is overloaded, applying the function to actual values is as efficient as if the function were overloaded only once.

The mechanism of conditional overloading is more dynamic: the conditions under which a particular code gets executed are tested only at run time. Although the mechanism offers some flexibility that cannot be provided by purely static overloading mechanisms, the programmer has to be aware of this potential performance penalty. We also notice that some of the mechanisms for pattern matching to be described in the chapter on abstract data types rely on conditional overloading, and may thus suffer from a similar performance penalty.

Chapter 6

Classes

6.1. Declaration of new classes

The user may define new classes using the keyword `class`. A simple example of a user defined class is the following:

```
class Point == {
  x: Double;
  y: Double;
  constructor point (x2: Double, y2: Double) == {
    x == x2;
    y == y2;
  }
}
```

Declarations of variables inside the class correspond to declarations of the internal data fields of that class. In addition, it is possible to define constructors, destructors and methods (also called member functions).

6.2. Data fields

Declarations of variables inside the class correspond to declarations of data fields for that class. The data field with name `x` can be accessed using the postfix operator `.x`. For instance, we may define an addition on points as follows:

```
infix + (p: Point, q: Point): Point == point (p.x + q.x, p.y + q.y);
```

By default, data fields are read only. They can be made read-write using the keyword `mutable`, as in the following example:

```
class Point == {
  mutable {
    x: Double;
    y: Double;
  }
  constructor point (x2: Double, y2: Double) == {
    x == x2;
    y == y2;
  }
}
```

Assuming the above definition, the following code would be correct:

```
translate (p: Alias Point, q: Point): Void == {
  p.x := p.x + q.x;
  p.y := p.y + q.y;
}
```

Notice that the user may define additional postfix operators of the form `.name` outside the class, which will behave in a similar way as actual data fields. For instance, defining

```
postfix .length (p: Point): Double == sqrt (square p.x + square p.y);
```

we may write

```
mmout << point (3.0, 4.0).length << lf;
```

6.3. CONSTRUCTORS AND DESTRUCTORS

In order to be useful, a user defined class should at least provide one constructor. By convention, constructors usually carry the same name as the class, in lowercase. For instance, in the above example, the unique constructor for the class `Point` carried the name `point`. Nevertheless, the user is free to choose any other name.

In the body of the constructor, the user should provide values for each of the data fields of the class, while preserving the ordering of declarations. Constructors are also required to be defined inside the class itself. Nevertheless, the function name of the constructor can be overloaded outside the class. For instance, we may very well define the function

```
point (): Point == point (0.0, 0.0);
```

outside the class, which behaves as if it were a constructor.

The default destructors for class instances are usually what the user wants in MATHEMAGIX, except when some special action needs to be undertaken when an instance is destroyed (such as saving some data to a file before destruction). Destructors are defined as functions with no arguments and no return type using the keyword `destructor`. For instance, the following modification of the class `Point` allows the user to monitor when points are destroyed:

```
class Point == {
  x: Double;
  y: Double;
  constructor point (x2: Double, y2: Double) == {
    x == x2;
    y == y2;
  }
  destructor () == {
    mmout << "Destroying " << x << ", " << y << lf;
  }
}
```

6.4. Methods

Special methods on class instances can be defined inside the class using the keyword `method`. For instance, a method for transposing the x and y coordinates might be defined as follows:

```
class Point == {
  x: Double;
  y: Double;
  constructor point (x2: Double, y2: Double) == {
    x == x2;
    y == y2;
  }
  method reflect (): Point == point (y, x);
}
```

We may apply the method using the postfix operator `.reflect`:

```
mmout << point (1.0, 2.0).reflect () << lf;
```

Inside the body of a method, we notice that the data fields of the class can be accessed without specifying the instance, which is implicit. For instance, inside the definition of `reflect`, we were allowed to write `point (y, x)` instead of `point (this.x, this.y)`, where `this` corresponds to the underlying instance which is implicit. Similarly, other methods can be called without the need to specify the underlying instance.

6.5. Containers

Containers such as vectors or matrices can also be declared using the `class` keyword, using the syntax

```
class Container (Param_1: Type_1, ..., Param_n: Type_n) == container_body
```

As is the case of the `forall` keyword, the parameters are allowed to depend on each other in an arbitrary order, although cyclic dependencies are not allowed. The parameters may either be types (in which case their types are categories; see below) or ordinary values.

For instance, we may define complex numbers using

```
class Complex (R: Ring) == {
  re: R;
  im: R;
  constructor complex (x: R) == { re == x; im == 0; }
  constructor complex (x: R, y: R) == { re == x; im == y; }
}
```

Notice that the user must specify a type for the parameter `R`. In this case, we require `R` to be a ring, which means that the ring operations should be defined in `R`. Here `Ring` is actually an example of a *category* (see the chapter on categories for more details), which might have been as follows:

```
category Ring == {
  convert: Int -> This;
  prefix -: This -> This;
  infix +: (This, This) -> This;
  infix -: (This, This) -> This;
  infix *: (This, This) -> This;
}
```

6.6. USER DEFINED CONVERTERS

When introducing new classes, one often wants to define converters between the new class and existing classes. For instance, given the above container `Complex R`, it is natural to define a converter from `R` to `Complex R`. Depending on the desired transitivity properties of converters, there are three important types of converters: ordinary converters, upgraders and downgraders. We also recall that appropriate mappers defined using the `map` construct automatically induce converters (see the section about the map construct).

6.6.1. Ordinary converters

Ordinary converters admit no special transitivity properties. They are defined using the special identifier `convert` and usually correspond to casts. A typical such converter would be the cast of a double precision number of type `Double` to an arbitrary precision number of type `Floating` and *vice versa*:

```
convert: Double -> Floating;
convert: Floating -> Double;
```

6.6.2. Upgraders

Upgraders usually correspond to constructors. For instance, with the example of the container `Complex R` in mind, it is natural to define a converter from any ring `R` to `Complex R` by

```
forall (R: Ring) upgrade (x: R): Complex R == complex x;
```

This definition is equivalent to

```
forall (R: Ring) convert (x :> R): Complex R == complex x;
```

In other words, upgraders are left transitive: whenever we have a type `T` with a converter from `T` to `R`, then the upgrader also defines a converter from `T` to `Complex R`. For instance, we automatically obtain a converter from `Integer` to `Complex Rational`.

6.6.3. Downgraders

In contrast to upgraders, downgraders are right transitive. Downgraders correspond to type inheritance in other languages such as C++, but with the big advantage that the inheritance is abstract, and not related to the internal representation of data. For instance, with the example class `Point` from the beginning of this section and some reasonable implementation of a class `Color` in mind, consider the class

```
class Colored_Point == {
  p: Point;
  c: Color;
  constructor colored_point (p2: Point, c2: Color) == {
    p == p2;
    c == c2;
  }
}
```

Then the method `.postfix p` provides us with a downgrader from `Colored_Point` to `Point`:

```
downgrade (cp: Colored_Point): Point == cp.p;
```

Notice that this definition is equivalent to

```
convert (cp: Colored_Point) :> Point == cp.p;
```

Given any converter from `Point` to another type `T`, the downgrader automatically provides us with a converter from `Colored_Point` to `T`. For instance, given the converter

```
convert (p: Point): Vector Double == [ p.x, p.y ];
```

we automatically obtain a converter from `Colored_Point` to `Vector Double`.

6.7. Flattening

In Mathemagix, instead of implementing pretty printing functions for new user defined classes, we rather defining flattening functions, which compute syntactic representations for instances of the new classes. More precisely, given a user defined class `T`, the user can define a function

```
flatten: T -> Syntactic;
```

Mathemagix implements a default pretty printer for Expressions of type `Syntactic`.

In fact, any Mathemagix type `T` comes with such a flattening function. In particular, a default implementation is provided automatically when declaring a new class, but the default function can be overridden by the user. For instance, with the container `Complex R` as before, we may define a flattener for complex numbers by

```
forall (R: Ring)
flatten (z: Complex R): Syntactic ==
  flatten (z.re) + flatten (z.im) * syntactic ('mathi);
```

Here `'mathi` stands for the standard name for the mathematical constant i, and addition and multiplication of syntactic expressions are provided by `basix/syntactic.mmx`. The advantage of using the flattening mechanism is that Mathemagix takes care of some elementary simplifications when printing syntactic expressions. For instance, the complex number $1 - \mathrm{i}$ will be printed as expected and not as something similar to $1 + (-1\,\mathrm{i})$.

Chapter 7

Unions and free data types

7.1. Introductory example

Mathemagix classes provide a simple way to introduce new data types with specified data fields, methods, constructors and an optional destructor. Mathemagix structures provide a convenient tool for the definition of unions and more general free data types.

One typical example of a structure is the type of finite integer sequences, defined as follows:

```
structure Sequence == {
  null ();
  cons (head: Integer, tail: Sequence);
}
```

Any such sequence is a formal expression of the form

$$\mathrm{cons}(a_1, \mathrm{cons}(a_2, \ldots \mathrm{cons}(a_n, \mathrm{null}()))),$$

where $a_1, \ldots, a_n$ are integers. The symbols `null` and `cons` are called the *constructors* for the structure and any structure of the type `Sequence` is always either of the form `null ()` or `cons (h, t)`, with `h: Integer` and `t: Sequence`. The declaration of the structure `Sequence` automatically gives rise to predicates

```
null?: Sequence -> Boolean;
cons?: Sequence -> Boolean;
```

which allow the user to determine the kind of structure. The symbols `head` and `tail` induce partially defined *accessors*

```
postfix .head: Sequence -> Integer;
postfix .tail: Sequence -> Sequence;
```

If a structure `s: Sequence` is of the form `cons (h, t)`, then `cons? s` holds and we may obtain `h` and `t` via the expressions `s.head` and `s.tail` respectively.

For instance, the following function can be used in order to compute the length of a sequence:

```
prefix # (s: Sequence): Int ==
  if null? s then 0 else #s.tail + 1;
```

Pattern matching provides us with an alternative way to do this:

```
prefix # (s: Sequence): Int ==
  match s with {
    case null () do return 0;
    case cons (_, t: Sequence) do return #t + 1;
  }
```

MATHEMAGIX also provides the following syntax for doing the same thing:

```
prefix # (s: Sequence): Int == 0;
prefix # (cons (_, t: Sequence)): Int == #t + 1;
```

Notice that this syntax relies on the mechanism of conditional overloading (see the section on conditional overloading).

Likewise classes, structures can be parameterized. The `List T` container is a typical example of a parameterized structure which generalizes the structure `Sequence`:

```
structure List (T: Type) == {
  null ();
  cons (head: T, tail: List T);
}
```

7.2. STRUCTURES

The general syntax for the declaration of a structure is the following:

```
structure S == {
  cons_1 (field_11: T_11, ..., field_1n1: T_1n1);
  ...
  cons_k (field_k1: T_k1, ..., field_knk: T_knk);
}
```

This declaration in particular induces the declaration of functions

```
cons_1: (T_11, ..., T_1n1) -> S;
...
cons_k: (T_k1, ..., T_knk) -> S;
```

These functions `cons_1`, ..., `cons_k` are called the *constructors* for `S`, and all instances of `S` are expressions which are built up freely using these constructors. Hence, any instance `x` has the form `cons_i (y_1, ..., y_ni)` for exactly one index `i` and `y_1: T_i1`, ..., `y_ni: T_ini`. The declaration of `S` also declares *inspection* predicates

```
cons_1?: S -> Boolean;
...
cons_k?: S -> Boolean;
```

such that `cons_i? x` holds if and only if `x` is of the form `cons_i (y_1, ..., y_ni)`. The declaration of `S` finally induces the declaration of partially defined *accessors*

```
postfix .field_11 : S -> T_11;
... ...
postfix .field_knk: S -> T_knk;
```

Whenever `x: S` is of the form `cons_i (y_1, ..., y_ni)` then `y_j` can be retrieved *via* the expression `x.field_ij`. Whenever `x` is not of this form, the expression `x.field_ij` is undefined and may provoke an error or worse.

Example 7.1. A structure of the above kind is called a *union* precisely then when each constructor takes exactly one argument; in that case, `S` corresponds to the union of the types `T_11`, ..., `T_k1`.

Parameterized structures are defined in a similar way as ordinary structures using the syntax

```
structure S (Param_1: C_1, ..., Param_p: C_p) == {
  cons_1 (field_11: T_11, ..., field_1n1: T_1n1);
  ...
  cons_k (field_k1: T_k1, ..., field_knk: T_knk);
}
```

Such declarations induce declarations of constructors, inspection predicates and accessors in the same way as their unparameterized homologues.

In additional to the constructors, inspection predicates and accessors, declarations of (parameterized) structures also give rise to the following additional functions and constants for more low level introspection:

```
postfix .kind: S -> Int;
cons_1_kind?: Int -> Boolean;
...
cons_k_kind?: Int -> Boolean;
cons_1_kind: Int;
...
cons_k_kind: Int;
```

If `x` is of the form `cons_i (y_1, ..., y_iki)`, then `x.kind` is just the integer `i-1`. The predicate `cons_i_kind?` just checks whether the input integer is equal to `i-1`, so that `cons_i? x` holds if and only if `cons_i_kind? x.kind` holds. Finally, `cons_i_kind` is equal to the integer `i-1`.

7.3. Extensible structures

Under some circumstances, it is useful to add additional constructors to structures *a posteriori*.

For instance, assume that we are writing a compiler for some language and that the expressions of the language are represented by a structure. It might be that someone else would like to define a language extension but still use as much as possible the existing functions in the compiler for all kinds of manipulations of expressions. The simplest solution would then be to extend the expression type *a posteriori* with some new constructors provided by the extended language, modulo customization of existing functions on expressions whenever necessary.

Another important example is the design of a flexible type for symbolic expressions. Symbolic expression types are usually unions of various basic expression types, such as literals, integers, function applications, sums, products, etc. Whenever someone develops a library for a new kind of mathematical objects, say differential operators, then it is useful if these new objects can be seen as a new kind of symbolic expressions.

In MATHEMAGIX, extensible structures can be declared using the syntax

```
structure S := {
  cons_1 (field_11: T_11, ..., field_1n1: T_1n1);
  ...
  cons_k (field_k1: T_k1, ..., field_knk: T_knk);
}
```

The only distinction with respect to the mechanism from the previous section is that we are now allowed to further extend the structure using the following syntax:

```
structure S += {
  extra_1 (added_11: X_11, ..., added_1n1: X_1n1);
  ...
  extra_l (added_l1: X_l1, ..., added_lnl: X_lnl);
}
```

An arbitrary number of such extensions can be made after the initial declaration of `S` and these extensions may occur in different files (provided that the file with the initial declaration is (directly or indirectly) included in each of these files).

Whenever we extend a structure in the above way the corresponding new constructors, inspection predicates and accessors are automatically defined. The lower level inspection routines are also automatically extended, although the actual values of `extra_1_kind`, ..., `extra_l_kind` now depend on the initialization order, and are therefore harder to predict. Nevertheless, the set of all these kinds is always of the form $\{0, ..., r-1\}$, where r is the total number of constructors.

7.4. PATTERN MATCHING

The constructors of a structure can also be used as building bricks for so called *patterns*. For instance, given the structure

```
structure Sequence == {
  null ();
  cons (head: Integer, tail: Sequence);
}
```

from the beginning of this section, the expression

```
cons (_, cons (_, tail_tail: Sequence))
```

is a pattern which corresponds to all sequences of length at least two, and with an explicit name `tail_tail` for the typed name of the tail of the tail of the sequence.

More generally, there are six kinds of patterns:

1. Structural patterns, of the form `cons (p_1, ..., p_n)` where `cons` is a constructor with arity `n` of some structure, and `p_1`, ..., `p_n` are other patterns.
2. User defined patterns, to be described in the next section.
3. Wildcards of the form `var: T`, where `var` is the name of a variable and `T` a type.
4. Unnamed wildcards of the form `_: T`, where `T` is a type.
5. The unnamed and untyped wildcard `_`.
6. Ordinary expressions.

One may define a natural relation "matches" on expressions and patterns, and if an expression matches a pattern, then there exists a binding for the wildcards which realizes the match. For instance, the expression `cons (1, cons (2, cons (3, null ())))` is said to match the pattern `cons (_, cons (_, tail_tail: Sequence))` for the binding `tail_tail` $\to$ `cons (3, null ())`.

Patterns can be used as arguments of the `case` keyword inside bodies of the `match` keyword. Whenever the pattern after `case` matches the expression after `match`, the wildcards are bound using the bindings of the match, and can be used inside the body of the `case` statement. For instance, we may use the following function in order to increase all terms of a sequence with an even index:

```
even_increase (s: Sequence): Sequence ==
  match s with {
    case cons (x: Integer, cons (y: Integer, t: Sequence)) do
      return cons (x, cons (y+1, even_increase t));
    case _ do
      return s;
  }
```

Patterns can also be used as generalized arguments inside function declarations. In that case, the declaration is considered as a special kind of conditionally overloaded function declaration. For instance, the declaration

```
prefix # (cons (_, t: Sequence)): Int == #t - 1;
```

is equivalent to

```
prefix # (s: Sequence | cons? s): Int == {
  t: Sequence == s.tail;
  #t - 1;
}
```

Again, the bindings of potential matches are used as values for the wildcards, which become local variables inside the function body.

7.5. USER DEFINED PATTERNS

Besides the patterns which are induced by constructors of structures, new patterns may be defined explicitly by the user. The syntax for pattern declaration is as follows:

```
pattern pat_name (sub_1: T_1, ..., sub_n: T_n): PT == pat_body
```

where the body `pat_body` consist of a finite number of cases of the form

```
case pat_case do {
  sub_1 == expr_1;
  ...
  sub_n == expr_n;
}
```

where `pat_case` is a pattern. Assuming this declaration of `pat_name`, any patterns `p_1`, ..., `p_n` of types `T_1`, ⋯, `T_n` give rise to a pattern `pat_name (p_1, ..., p_n)` of type `PT`. This pattern is matched if and only if one of the patterns `pat_case` in the various cases for `pat_name` is matched (with all occurrences of `sub_1`, ..., `sub_n` in `pat_case` replaced by `p_1`, ..., `p_n`). In that case, we privilege the first case which matches, and bind `var_i` to `expr_i` whenever `sub_i` is of the form `var_i: T_i`.

In a similar way as for structures, replacing `==` by `!=` or `+=` in the declaration of the above pattern allows for the declaration of an extensible pattern resp. an actual extension of it. The above declaration of the pattern `pat_name` also gives rise to a generalized inspection predicate

```
pat_name?: PT -> Boolean;
```

and functions

```
postfix .sub_1: PT -> T_1;
...
postfix .sub_n: PT -> T_n;
```

which behave in a similar way as accessors of structures.

For instance, for sequences $[x_1, ..., x_{2n}]$ which really represent association lists $[x_1 \rightsquigarrow x_2, ..., x_{2n-1} \rightsquigarrow x_{2n}]$, we may use the following pattern for retrieving the first association $x_1 \rightsquigarrow x_2$:

```
pattern assoc (key: Integer, val: Integer, next: Sequence): Sequence ==
  case cons (k: Integer, cons (v: Integer, n: Sequence)) do {
    key  == k;
    val  == v;
    next == n;
  }
```

The implementation of the predicate `assoc?` is equivalent to

```
assoc? (s: Sequence): Boolean == cons? s and cons? s.next;
```

and the implementations of the accessors `postfix .key`, `postfix .val` and `postfix .next` are equivalent to

```
postfix .key (s: Sequence): Integer == s.head;
postfix .val (s: Sequence): Integer == s.tail.head;
postfix .next (s: Sequence): Sequence == s.tail.tail;
```

7.6. Syntactic sugar for symbolic expressions

As mentioned in the introduction of section 7.3, one important special case where extensible structures are useful is for the definition of a flexible type `Symbolic` for symbolic expressions. This type is essentially a union type. For instance, we might start with

```
structure Symbolic := {
  sym_undefined ();
  sym_boolean (boolean: Boolean);
  sym_literal (literal "literal": Literal);
  sym_compound (compound: Compound);
}
```

and further extend `Symbolic` whenever necessary:

```
structure Symbolic += {
  sym_integer (integer: Integer);
}
```

```
structure Symbolic += {
  sym_rational (rational: Rational);
}
```

The prefix `sym_` provides us with a clean syntatic distinction between a "symbolic expression which contains an object of type `T`" and a mere object of type `T`. However, it would be simpler to write `integer?` instead of `sym_integer?` as an inspection predicate, and it is somewhat cumbersome to implement addition of symbolic integers using

```
infix + (sym_integer (i: Integer), sym_integer (j: Integer)): Symbolic ==
  sym_integer (i + j);
```

For this reason, Mathemagix provides us with a special operator `::` to be used for structure constructors with one argument. For instance, the constructor `sym_integer` would rather be declared using

```
structure Symbolic += {
  sym_integer (integer :: Integer);
}
```

This declaration automatically declares the synonym `integer?` for `syn_integer?` and also introduces the simplified notation `var :: Integer` for the pattern `sym_integer (var: Integer)`. Assuming the further implementation of a constructor

```
convert (i: Integer): Symbolic == sym_integer i;
```

we may then simplify the declaration of our addition on symbolic integers into

```
infix + (i :: Integer, j :: Integer): Symbolic == i + j;
```

In a similar way, MATHEMAGIX provides support for an operator `::>` which allows us to mimick the notation `:>` used for *a priori* conversions for their symbolic wrappers. This notation is best illustrated with the example of an "symbolic converter from symbolic integers to symbolic rational numbers":

```
pattern sym_as_rational (as_rational ::> Rational): Symbolic := {
  case sym_rational (x: Rational) do as_rational == x;
  case sym_integer (x: Integer) do as_rational == x :> Rational;
}
```

This allows us for instance to write

```
infix + (i :: Integer, x ::> Rational): Symbolic == i + x;
```

7.7. FAST DISPATCHING

We already noticed that the mechanism of conditional overloading induces a performance overhead (see the section on performance issues concerning conditional overloading). In the case of basic operations on symbolic expressions, such as additions, which are overloaded dozens if not hundreds of times, this performance penalty is too significant.

Fortunately, the structure `Symbolic` as described in the previous section is essentially an extensible union type. Given a unary conditionally overloaded operation `foo` on symbolic expressions, the appropriate routine to be called for a given input `x` can often be determined as a function of the integer `x.kind` only. This makes it possible to use a fast lookup table instead of the usual conditional overloading mechanism in this particular case. In order to do so, we have to declare the operation `foo` using

```
foo (x: dispatch Symbolic): Symbolic := default_implementation
```

Operations with higher arities and operations which involve other types are treated in a similar way: the type of each argument involved in the fast table lookup should be preceded by the keyword `dispatch`. For instance,

```
infix + (x: dispatch Symbolic, y: dispatch Symbolic): Symbolic := ...;
postfix [] (x: dispatch Symbolic, i: Int): Symbolic := ...;
```

Notice that the size of the dispatch table is equal to the product of the number of structure constructors of all arguments on which we perform our dispatching. For this reason, we do not allow dispatching on more than two arguments.

Mathemagix also provides the keyword `disjunction` as a special case of user defined patterns which is compatible with the above dispatching mechanism. A typical example of a disjunction is

```
disjunction sym_scalar (scalar :: Scalar): Symbolic := {
  sym_boolean _;
  sym_integer _;
  sym_rational _;
}
```

Chapter 8

Categories

8.1. Categories

Generic programming in MATHEMAGIX is based on the notion of a *category*. When writing generic functions (also called templates) or classes (also called containers), it is usually necessary to make assumptions on the type parameters.

Consider for instance the container `Polynomial R` for univariate polynomials. It is natural to use an internal representation for which the leading coefficient is non zero. The mere existence of an instance "zero" of type `R` constitutes a condition on `R`. Similarly, the ring operations on polynomials are naturally defined in terms of the ring operations on coefficients in `R`. Hence, `R` should at least be a ring (even though more structure might be needed for certain operations, such as the computation of greatest common divisors).

Categories are defined using the following syntax:

```
category Category_Name == category_body;
```

The body of a category consists of a block of prototypes of functionality required by the category. For instance, a typical definition of the `Ring` category is the following:

```
category Ring == {
  convert: Int -> This;
  prefix -: This -> This;
  infix +: (This, This) -> This;
  infix -: (This, This) -> This;
  infix *: (This, This) -> This;
}
```

The special type `This` corresponds to the carrier of the category. A type `T` is said to *satisfy* the category `Category_Name` in a given context (and we write `T: Category_Name`) if the context provides the required functionality in the body of the category, with `This` replaced by `T`. In a context where `basix/fundamental.mmx` has been included, we thus have `Int: Ring`, since the context provides us with operations

```
convert: Int -> Int;
prefix -: Int -> Int;
infix +: (Int, Int) -> Int;
infix -: (Int, Int) -> Int;
infix *: (Int, Int) -> Int;
```

However, we no not have `String: Ring`, since there is no implementation of `infix -` on strings.

8.2. MATHEMATICAL PROPERTIES

It should be noticed that the notion of category satisfaction is purely syntactic: we simply look whether all prototypes in the body of the category are implemented in the current context, but we no not check any of the customary properties that the names of the category might suggest. For example, the type `Double` satisfies the category `Ring`, since the operations

```
convert: Int -> Double;
prefix -: Double -> Double;
infix +: (Double, Double) -> Double;
infix -: (Double, Double) -> Double;
infix *: (Double, Double) -> Double;
```

are all defined in `basix/double.mmx`. However, these operations are not associative: we typically have `(1.0e100 + 1.0) - 1.0e100 != 1.0`, due to rounding errors.

From the programming point of view this is really a feature, since it is desirable that straightforward implementations of containers such as `Polynomial R` can be instantiated for `R` $\to$ `Double`. However, from the mathematical point of view, the code cannot certified to be correct under all circumstances.

Since MATHEMAGIX aims to be an efficient general purpose mathematical programmer language rather than an automatic theorem prover, we have integrated no support for checking mathematical relations. Nevertheless, nothing prevents the user to informally introduce dummy functions which correspond to mathematical properties which are intended to be satisfied. For instance, the user might replace the definition of a ring by something like

```
category Ring == {
  convert: Int -> This;
  prefix -: This -> This;
  infix +: (This, This) -> This;
  infix -: (This, This) -> This;
  infix *: (This, This) -> This;
  associative_addition: This -> Void;
  commutative_addition: This -> Void;
  additive_inverse: This -> Void;
  associative_multiplication: This -> Void;
  ...
}
```

The idea is that each of these additional functions stands for a mathematical property. For instance, `associative_addition` would stand for the property that $(x + y) + z = x + (y + z)$ for all x, y, z in the carrier. A dummy implementation

```
associative_addition (x: T): Void == {}
```

of `associative_addition` corresponds to the assertion that the corresponding mathematical property is satisfied for `T`. However, this assertion is not checked by the compiler, and simply admitted on good faith.

8.3. Inheritance

It frequently happens that we want to declare new categories which are really extensions of already existing categories. For instance, the following category `Field` is really the extension of the category `Ring` with a division:

```
category Field == {
  convert: Int -> This;
  prefix -: This -> This;
  infix +: (This, This) -> This;
  infix -: (This, This) -> This;
  infix *: (This, This) -> This;
  infix /: (This, This) -> This;
}
```

A shorter and more comprehensive way to define this category is

```
category Field == {
  This: Ring;
  infix /: (This, This) -> This;
}
```

In other words, in our specification of the functionality of categories, we allow for prototypes of the form

```
This: Other_Category;
```

which really correspond to inheriting all functionality from another category. Multiple inheritance is allowed in the same way. For instance, assuming

```
category Ordered == {
  infix <=: (This, This) -> Boolean;
}
```

we may define

```
category Ordered_Ring == {
  This: Ring;
  This: Ordered;
}
```

8.4. Parameterized categories

Categories are allowed to be parameterized. The syntax for defining parameterized categories is similar to the syntax for the definition of containers:

```
category Cat (Param_1: Type_1, ..., Param_n: Type_n) == category_body
```

The parameters are allowed to depend on each other in an arbitrary order, although cyclic dependencies are not allowed. The parameters may either be types or ordinary values.

Two examples of parameterized categories `To F` and `From T` were already encountered in the section on type converters:

```
category To (T: Type) == {
  convert: This -> T;
}
```

```
category From (F: Type) == {
  convert: F -> This;
}
```

Another typical example of a parameterized category is

```
category Vector_Space (K: Field) == {
  This: Abelian_Group;
  infix *: (K, This) -> This;
}
```

8.5. PLANNED EXTENSIONS

One planned extension for categories concerns default implementations of certain methods. For instance, in the category `Ring`, the operation `infix -` is really redundant, since it admits a reasonable default implementation

```
infix - (x: This, y: This): This == x + (-y);
```

In the future, MATHEMAGIX should be able to use such default implementations except when a better method is explicitly provided by the user. Notice that this requires a mechanism for specifying the required functionality for default methods. This is not completely trivial, since it should also be possible to provide a default implementation of `prefix -` in terms of `infix -`:

```
prefix - (x: This): This == (0 :> This) - x;
```

Of course, this should not lead to infinite loops if neither `prefix -` nor `infix -` is present.

8.6. EFFICIENCY CONSIDERATIONS

Chapter 9

Interface with C++

9.1. Foreign imports and exports

Mathemagix both allows the user to import functionality from C++ template languages and to export Mathemagix functions and classes back to C++. The syntax for importating and exporting functionality is as follows:

```
foreign cpp import import_body;
foreign cpp export export_body;
```

The import and export bodies essentially contain dictionaries between C++ names and Mathemagix names of various classes, functions and other variables or constants.

In the case of foreign imports, it is usually necessary to specify options which should be passed to the compiler and to the linker, as well as some include statements or macro definitions which are necessary in order to compile the imported code. This can be done using the following keywords:

`cpp_flags`	:	flags that should be passed to the C++ compiler
`cpp_libs`	:	flags that should be passed to the C++ linker
`cpp_include`	:	C++ files that should be included in order to compile the imported code
`cpp_preamble`	:	macro definitions which are necessary to compile the imported code

A typical example of the use of these keywords occurs at the start of the file `basix/int.mmx`:

```
foreign cpp import {
  cpp_flags    "`basix-config --cppflags`";
  cpp_libs     "`basix-config --libs`";
  cpp_include  "basix/int.hpp";
  cpp_preamble "#define int_literal(x) as_int (as_string (x))";
  ...
}
```

The C++ compiler and linker flags are taken to be the results of the shell command `basix-config` which outputs the appropriate flags as a function of the user's environment. The header file `basix/int.hpp` contains various utility functions for machine integers which are imported into Mathemagix. The additional macro `int_literal` is also imported as the constructor of machine integers from literal integers.

9.2. Importing classes from C++

Inside the body of a `foreign cpp import` statement, the C++ class `my_class` can be imported into Mathemagix under the name `My_Class` using

```
class My_Class == my_class;
```

For instance, we may import the class `integer` as `Integer` using

```
class Integer == integer;
```

Any imported class `my_class` is required to provide the following standard routines:

```
bool operator == (const my_class&, const my_class&);
  // equality predicate
bool operator != (const my_class&, const my_class&);
  // inequality predicate
bool exact_eq (const my_class&, const my_class&);
  // predicate for syntactic equality
bool exact_neq (const my_class&, const my_class&);
  // predicate for syntactic equality
bool hard_eq (const my_class&, const my_class&);
  // predicate for hard equality (of pointers for reference counted
objects)
bool hard_neq (const my_class&, const my_class&);
  // predicate for hard inequality (of pointers for reference counted
objects)
nat hash (const my_class&);
  // a hash code compatible with operator ==
nat exact_hash (const my_class&);
  // a hash code compatible with exact_eq
nat hard_hash (const my_class&);
  // a hash code compatible with hard_eq
syntactic flatten (const my_class&);
  // a flattener for objects of type my_class
```

In practice, some of these routines can usually be derived from the others. In the file

`basix/defaults.hpp`

one may find various macros to this effect, such as `TRUE_TO_EXACT_IDENTITY_SUGAR`.

9.3. IMPORTING CONTAINERS FROM C++ AND EXPORTATION OF CATEGORIES

In a similar way, a C++ container class `my_container<param_1, ..., param_n>` can be imported as a MATHEMAGIX container class with name `My_Container` using

```
class My_Container (P_1: Cat_1, ..., P_n: Cat_n) ==
  my_container (P_1, ..., P_n);
```

The parameters of imported C++ containers are necessarily type parameters, and it is required to specify the types of these parameters, which are categories. For instance, in `numerix/complex.mmx`, the container class `complex<R>` of complex numbers is imported using

```
class Complex (R: Ring) == complex R;
```

In the case when the categories of the parameters are non trivial (e.g. in the above example where `Ring` is the category of the parameter `R`), it is necessary to specify a dictionary between the members of the category and their corresponding names in C++.

Indeed, in the C++ implementation of an addition on complex numbers, we add the real and imaginary parts using the C++ operator `+`. At some place, we have to specify that this C++ operator `+` corresponds to the MATHEMAGIX operator `infix +`. This can be done by *exporting* the category `Ring` to C++:

```
foreign cpp export {
  category Ring == {
    convert: Int -> This == keyword constructor;
    prefix -: This -> This == prefix -;
    infix +: (This, This) -> This == infix +;
    infix -: (This, This) -> This == infix -;
    infix *: (This, This) -> This == infix *;
  }
}
```

This example illustrates the general syntax for the exportation of categories to C++: every field

```
mmx_function: (S_1, ..., S_n) -> D;
```

in the definition of the category is replaced by a declaration

```
mmx_function: (S_1, ..., S_n) -> D == cpp_function;
```

where `cpp_function` is the C++ name corresponding to the function `mmx_function`.

9.4. Importing variables and functions from C++

Inside the body of a `foreign cpp import` statement, one may import a C++ function or variable with name `cpp_name` into MATHEMAGIX under the name `mmx_name` using the syntax

```
mmx_name: Type == cpp_name;
```

The type `Type` of the imported function or variable should be specified on the MATHEMAGIX side. For instance, some of the basic operations on strings can be imported using

```
foreign cpp import {
  prefix #: String -> Int == N;
  postfix []: (String, Int, Int) -> String == postfix ();
  infix *: (String, String) -> String == infix *;
  infix ><: (String, String) -> String == infix *;
  infix <<: (Alias String, String) -> Alias String == infix <<;
}
```

The standard ports `mmin`, `mmout` and `mmerr` for input, output and errors are imported using

```
foreign cpp import {
  mmin : Port == mmin;
  mmout: Port == mmout;
  mmerr: Port == mmerr;
}
```

9.5. IMPORTING TEMPLATE FUNCTIONS FROM C++

Template functions can be imported from C++ into MATHEMAGIX using the same syntax as for the importation of ordinary functions, by putting all declarations inside a `forall` block. For instance, the following basic imported functions on vectors were extracted from `basix/vector.mmx`:

```
foreign cpp import {
  forall (C: Type) {
    prefix #: Vector C -> Int == N;
    postfix []: (Vector C, Int) -> C == postfix [];
    postfix []: (Alias Vector C, Int) -> Alias C == postfix [];
    postfix []: (Vector C, Int, Int) -> Vector C == range;
    reverse: Vector C -> Vector C == reverse;
    infix ><: (Vector C, Vector C) -> Vector C == append;
  }
}
```

Inside the `forall` block it is possible to impose additional constraints on the parameters using the `assume` statement. For instance, we may extend the imported functions on vectors as follows:

```
foreign cpp import {
  forall (C: Type) {
    prefix #: Vector C -> Int == N;
    postfix []: (Vector C, Int) -> C == postfix [];
    postfix []: (Alias Vector C, Int) -> Alias C == postfix [];
    postfix []: (Vector C, Int, Int) -> Vector C == range;
    reverse: Vector C -> Vector C == reverse;
    infix ><: (Vector C, Vector C) -> Vector C == append;

    assume (C: Abelian_Group) {
      prefix -: Vector C -> Vector C == prefix -;
      infix +: (Vector C, Vector C) -> Vector C == infix +;
      infix -: (Vector C, Vector C) -> Vector C == infix -;
    }
  }
}
```

Here `Abelian_Group` stands for the following category:

```
foreign cpp export {
  category Abelian_Group == {
    convert: Int -> This == keyword constructor;
    prefix -: This -> This == prefix -;
    infix +: (This, This) -> This == infix +;
    infix -: (This, This) -> This == infix -;
  }
}
```

9.6. Exporting to C++

Classes, functions and variables can be exported back to C++ using the same syntax as for importations. For instance, the class `Point` from the chapter about user defined classes can be exported together with some basic routines as follows

```
foreign cpp export {
  class Point == point;
  point: (Double, Double) -> Point == keyword constructor;
  postfix .x: Point -> Double == get_x;
  postfix .y: Point -> Double == get_y;
}
```

In order to use C++ template functions with `Point` as a parameter, the required operations from the category of that parameter should be exported to C++ as well. For instance, assume that we wish to use the operations + and − from the previous section on vectors of points. Then we first have to define the abelian group operations on `Point`:

```
convert (i: Int): Point == point (as_double i, as_double i);
prefix - (p: Point): Point == point (-p.x, -p.y);
infix + (p: Point, q: Point): Point == point (p.x + q.x, p.y + q.y);
infix - (p: Point, q: Point): Point == point (p.x - q.x, p.y - q.y);
```

We next have to export these operations to C++, while making sure that the C++ names of the operations correspond to the exported names of the operations of the category `Abelian_Group`:

```
foreign cpp export {
  convert: Int -> Point == keyword constructor;
  prefix -: Point -> Point == prefix -;
  infix +: (Point, Point) -> Point == infix +;
  infix -: (Point, Point) -> Point == infix -;
}
```

Remark 9.1. In the current implementation of the compiler, the mandatory operations `operator ==`, `operator !=`, `exact_eq`, `exact_neq`, `hard_eq`, `hard_neq`, `hash`, `exact_hash`, `hard_hash` and `flatten` of MathemAgix-compatible C++ types are *not* exported to C++. For instance, assuming that the user redefined the default flattening routine for `Point`, this routine should be exported explicitly to C++ if we want vectors of points to be printed correctly.

Chapter 10

Large multiple file programs

10.1. General principles

When using Mathemagix for the development of large mathematical programs which usually consist of lots of files, one should keep in mind one important design principle:

> For any Mathemagix file `my_program.mmx` which is to be compiled into a binary, the set of all dependencies for `my_program.mmx` can be deduced from the source code in `my_program.mmx`.

Similarly,

> For any Mathemagix file `helper_program.mmx` which is only to be compiled into an object file, the set of all dependencies for `helper_program.mmx` can be deduced from the source code in `helper_program.mmx`.

As a consequence of these principles, the compilation process for large projects is easy to understand: the programmer should just make sure to carefully include the correct include files for every individual file in the project. The compiler will then take care of determining the various dependencies and compiling the files in the project in the right order and in parallel whenever possible.

Notice that these rules are *not* satisfied in several other programming languages such as C or C++: in this case, there is usually a makefile for the project which describes the dependencies and all kinds of compiler and linker flags which are necessary for building the executable. Mathemagix on the other hand does not require any particular configuration or makefiles in order to compile multiple file projects. Also the number of command line options is strongly reduced with respect to languages such as C or C++.

In the case when some of the functionality of a Mathemagix program is imported from C++, the user should use the keywords `cpp_include`, `cpp_flags` and `cpp_libs` in order to specify the dependencies on the C++ code and the particular compiler and linker flags to be used in order to compile the C++ code. We refer to the chapter on interfacing with C++ for more details.

10.2. Inclusion of files

Other files can be included into a given Mathemagix file using the keyword `include`. Mathemagix provides three modes for the inclusion of other files: ordinary public inclusion, private inclusion and virtual inclusion.

10.2.1. Public inclusion

When including a file `a.mmx` into the file `b.mmx` using

```
include "a.mmx";
```

all public functions, classes and categories from the file `a.mmx` are made available in the file `b.mmx`. Moreover, for any chain of public inclusions `b.mmx` ← `c_1.mmx` ← ⋯ ← `c_n.mmx` (we say that `b.mmx` is indirectly and publicly included by `c_n.mmx`), the public functionality of `a.mmx` is also available in `c_n.mmx`.

Public inclusions thus induce a dependency of `b.mmx` on `a.mmx` as well as of `c_n.mmx` on `a.mmx` for any file `c.mmx` which indirectly includes `b.mmx`. When building a large project in parallel, this means that both `b.mmx` and `c.mmx` will have to be recompiled whenever a change occurs in the public interface of `a.mmx`.

10.2.2. Private inclusion

In order to reduce the number of dependencies inside a large project, it is possible to use the mechanism of private inclusions whenever appropriate. When including a file `a.mmx` into the file `b.mmx` using

```
private include "a.mmx";
```

all public functions, classes and categories from the file `a.mmx` are made available in the file `b.mmx`. However, the functionality of `a.mmx` remains hidden for any other file `c.mmx` which directly or indirectly includes `a.mmx` (except when `c.mmx` includes `a.mmx` itself, or *via* some other file different from `b.mmx`, of course). On the other hand, if `a.mmx` indirectly and publicly includes a file `A.mmx`, then `A.mmx` remains an indirect (although private) inclusion of `b.mmx`, so all public functionality of `A.mmx` is also available in `b.mmx`.

In summary, a private inclusion of `a.mmx` in `b.mmx` still induces a dependency of `b.mmx` on `a.mmx`, but the transitivity of the inclusion relation is broken.

10.2.3. Virtual inclusion

When using the mechanism of private inclusion, we still introduce dependencies for the build process: whenever the public interface of the included file `a.mmx` changes, the file `b.mmx` where the inclusion occurred needs to be recompiled.

On some occasions, none of the functionality of `a.mmx` is actually needed in `b.mmx` or any other file which includes `b.mmx` (directly or indirectly). Indeed, it may happen that we just want to ensure that some initialization code present in `a.mmx` is executed before we start executing `b.mmx`. For instance, the file `a.mmx` might initialize some table which was declared in a file `A.mmx` which is included both by `a.mmx` and by `b.mmx`, and such that the code in `b.mmx` will only work correctly after initialization of this table.

Whenever the above situation occurs, we may perform a virtual inclusion of `a.mmx` into `b.mmx`

```
virtual include "a.mmx";
```

Virtual inclusions do not create any dependencies for the build process, but they do influence the list of files which need to be compiled and linked, as well as the order in which the various files of the project are executed (see section 10.3 below).

10.2.4. Circular inclusions

Whatever inclusion modes are used, no circular chain of inclusions is allowed in Mathemagix.

In the case when some code in a file `b.mmx` depends on some code in the file `a.mmx` and *vice versa*, the programmer will have to explicitly include prototypes for the required functionality in one of the files. Class prototypes are declared using the syntax

```
class Simple_Class;
class Container (P_1: T_1, ..., P_n: T_n);
```

Function prototypes are declared using the syntax

```
fun (arg_1: T_1, ..., arg_n: T_n);
```

10.3. Order of execution

- Duplicate inclusions. Initialization only once, for the first inclusion.

Chapter 11

Using the Mathemagix compiler

11.1. Introduction

The basic procedure for compiling and running a Mathemagix program `toto.mmx` is very simple: first compile the program using

```
mmc toto.mmx
```

This will create a binary `toto` which can be run using

```
./toto
```

This method still works if the program `toto.mmx` depends on many other files. In that case Mathemagix will automatically determine and compile all dependencies. Moreover, on modern computers the compiler will automatically compile as many of the dependencies as possible in parallel. Furthermore, if you modify one of the dependencies and recompile `toto.mmx`, then only those dependencies which were affected by your change will be recompiled. In order to monitor this process more precisely and get a rough idea about what the compiler is doing, you may compile `toto.mmx` using the option `--verbose`:

```
mmc --verbose toto.mmx
```

Whenever the user develops another program `booh.mmx` with an overlapping set of dependencies with `toto.mmx`, then compiling both programs `toto.mmx` and `booh.mmx` will compile the overlapping dependencies only once (or at most once, in case of recompilations).

In order to avoid unnecessary recompilations, the compiler uses a cache, which is stored by default in the directory `~/.mathemagix/mmc` or `~/.mathemagix/mmc-`*`version`*. Sometimes, you may wish to clean the cache and recompile all dependencies from scratch. This can be done using

```
mmc --clean-cache
```

This kind of cleaning may also be necessary whenever you replace your compiler by a newer development version.

11.2. Compiler flags

The compiler only supports few global flags to influence the compilation process. On the one hand, this is important in order to take optimal advantage out of the compiler cache, and thus make the compilation process as fast as possible. On the other hand this is part of the general design philosophy of Mathemagix, in which control is delegated to the programs themselves rather than to an external build system.

General purpose compiler flags are the following:

`--optimize`. This flag is used for the generation of optimized (but possibly larger) code.

`--gdb`. This flag should be used in order to generate code which can be debugged using GDB.

`--static`. This flag should be used for the creation of static binaries.

11.3. Other compiler options

The compiler supports a few other general purpose options:

`--color`. This option will print messages issued by the compiler using colors.

`--progressive`. Disable parallelism for the compilation process.

`--test-compile` *files*. Test the compilation of a list of *files*.

`--test-run` *files*. Both compile and run a list of *files*.

`--threads` *max*. Compile using at most *max* threads or processes.

For debugging purposes, it is sometimes useful to inspect the contents of intermediate files which are generated by the compiler. This can be done using the following options:

`--keep-mmh`. Keep all public interfaces of the MATHEMAGIX files.

`--keep-cpp`. Keep all intermediate C++ files generated by the compiler.

`--keep-o`. Keep all intermediate object files generated by the compiler.

Any C++ files which are kept in this way may be edited by the user. In order to resume the compilation process from these modified versions, you may use the option `--from-cpp`.

Hackers may also want to experiment with various debugging options for the compiler itself. These options are all of the form `--debug-`*feature*. Currently supported debugging options are:

`--debug-apply`, `--debug-build`, `--debug-categories`, `--debug-coerce`,
`--debug-compile`, `--debug-compiled`, `--debug-convert`, `--debug-cpp`,
`--debug-cse`, `--debug-csed`, `--debug-declare`, `--debug-exe`,
`--debug-flat-control`, `--debug-flattened-control`, `--debug-globals`,
`--debug-glue`, `--debug-implicit`, `--debug-inline`, `--debug-inlined`,
`--debug-instantiate`, `--debug-instantiated`, `--debug-new`,
`--debug-new-categories`, `--debug-phase`, `--debug-trace`, `--debug-uncurried`,
`--debug-uncurry`, `--debug-unnest`, `--debug-unnested`.

Chapter 12

Using the Mathemagix interpreter

This section describes how to use the MATHEMAGIX interpreter `mmi` available from the `mmcompiler` package.

12.1. Terminal interface

From a terminal, the MATHEMAGIX interpreter is launched with the following command:

```
$ mmi
------------------------------------------------------------
|:*)              Welcome to Mathemagix 1.0.2             (*:|
|----------------------------------------------------------|
|  This software falls under the GNU General Public License  |
|          It comes without any warranty whatsoever          |
|                http://www.mathemagix.org                   |
|                      (c) 2010-2012                         |
------------------------------------------------------------
1]
```

At first run the following message should last several seconds, according to the performances of your computer:

```
mmi: compiling glue...
```

This means that the fundamental librairies are being compiled. At next run, the message still appears but lasts just the time needed to check that these compiled libraries are up to date.

Whenever the interpreter has been compiled with the GNU readline library, several shortcuts are available such as

- `[Ctr-A]`, `[Ctr-E]` to go at the beginning or end of the instruction line,
- ↑, ↓ for the previous and next instruction line,
- `[tab]` for name completion,
- `[Meta-return]` to add an extra line to the current instruction line. Note that on several platforms the `[Meta]` key is binded to `[Alt]`. On the Mac OS X terminal, unless the `[Alt]` key has been specifically set so from the terminal preferences menu, one must type `[esc]` and then `[return]`.

For more information and customization of the keyboard interface, please refer to the documentation of the GNU readline library.

Within an interactive session, ending a line with a ';' actually means finishing with a null instruction. As a consequence this extra ';' prevents from printing the output of the previous instruction.

```
1] 1+1
2
2] 1+1;
3]
```

In order to quit the interpreter one can type `[Ctr-d]`, or call the function `exit` that takes the return value of the `mmi` command as an argument.

```
1] exit 0
```

12.2. Color mode

For a short list of terminals (xterm, xterm-color, xterm-256color), the color mode of the interpreter can be activated by adding the following option to the command line:

```
$ mmi --color
------------------------------------------------------------
|:*)              Welcome to Mathemagix 1.0.2             (*:|
|----------------------------------------------------------|
|  This software falls under the GNU General Public License  |
|          It comes without any warranty whatsoever          |
|                http://www.mathemagix.org                   |
|                      (c) 2010-2012                         |
------------------------------------------------------------
1]
```

The `--color` option can be activated by default by setting the global environment variable `MMX_COLOR_MODE` to `yes`. For instance, if your default shell is BASH, then you might want to add the following line to your `$HOME/.profile` file in order to get the color mode permanently:

```
export MMX_COLOR_MODE="yes"
```

In this case, colors can be punctually disabled as follows:

```
mmi --no-color
```

Supported terminals are specified in `basix/src/formatting_port.cpp`.

12.3. Other interpreter options

The behavior of the interpreter can be modified according to the following command line options:

`--quiet`. Disable printing the banner and prompt.

`--quit`. Quit the interpreter after replaying a session.

`--replay`. Replay the previous session.

`--texmacs`. Enable the TeXmacs interface (to be used by TeXmacs only).

`--time`. Display compilation and executing timings.

`--type`. Display types of computed expressions.

`--verbose`. Enable verbose mode.

The command line option `--help` summarizes the usage of `mmi`, and `--version` returns the current version of `mmi`.

The following options are passed to the compiler `mmc` for compiling dynamic librairies at runtime: `--diff`, `--gdb`, `--keep-cpp`, `--no-cache`, `--no-warnings`, `--optimize`, `--timings`, `--verbose`.

12.4. Builtin help command

Signatures and source locations of functions can be obtained via the `help` command as follows:

```
1] help infix +
+ : (Int, Int) -> Int --- mmx/basix/mmx/int.mmx:30:10
+ : (Double, Double) -> Double --- mmx/basix/mmx/double.mmx:33:10
+ : (Syntactic, Syntactic) -> Syntactic ---
mmx/basix/mmx/syntactic.mmx:33:8
2] fib (n: Int): Int == if n <= 1 then 1 else fib (n-2) + fib (n-1)
3] help fib
fib : Int -> Int --- /Users/lecerf/.mathemagix/mmi/input_2.mmx:0:43
```

The latter source location corresponds the actual file where the input command is temporarily saved in. Help on types is also available:

```
5] help Vector
Vector : Type -> Class --- mmx/basix/mmx/vector.mmx:15:8
```

12.5. File inclusion

File inclusion is performed via the `include` function, as for the compiler. If the file to be included or one of its dependencies contains `foreign` declarations, for importing or exporting `C++` functions, then the necessary dynamic librairies are automatically compiled and loaded. During the compilation the message `mmi: compiling glue...` is displayed. This compilation might take time, but it is only performed once.

```
1] include "numerix/integer.mmx"
2] 40!
815915283247897734345611269596115894272000000000
```

In case you are sure that all the dynamic libraries you are going to use in an interpreter session are already compiled then you might want to use the `--no-glue` option in order to discard checking if these dynamic libraries are up to date.

12.6. LOW LEVEL DEBUGGER

If the compiler and interpreter have been compiled with passing the `--enable-verify` option to the `configure` script, then a low level debugger is made available by adding `--exe-debugger` to the `mmi` command. The features of the debugger are rather limited but are essentially useful to understand casual bugs, and to display the actual builtin `C++` types being used. The interactive commands of this debugger are the following:

`I.` set interactive mode.

`i.` unset interactive mode.

`E.` set display of expressions.

`e.` unset display of expressions.

`V.` set display of values.

`v.` unset display of values.

`n.` go directly to next step.

`s.` step into the intermediate.

Appendix A
Getting and installing Mathemagix

Extensive information on how to download Mathemagix and install the software is available on our website `www.texmacs.org`. For the moment, we recommend compilation from the sources. In order to go short, the following steps should be followed in order to obtain a working version of the compiler `mmc` and the interpreter `mmi`:

1. Verify that all dependencies are installed on your system. In particular, it is recommended that you have recent versions of Libtool, Readline, Gmp, Mpfr and TEXMACS installed on your computer. You will also need Svn to check out the source code.

2. In a shell session, check out the most recent development version of Mathemagix using Svn, by issuing the command

   ```
   svn checkout svn://scm.gforge.inria.fr/svn/mmx
   ```

   This should create a directory `mmx`. Go to this directory, using

   ```
   cd mmx
   ```

3. Configure and build the software using the commands

   ```
   ./configure
   make
   ```

4. Assuming root privileges, install the software in `/usr/local` using

   ```
   make install
   ```

5. Enjoy using the software!

Remark A.1. If you want to install the software at a non standard location *`install-dir`* (which is in particular the case if you do not have root privileges on your machine), then you should replace the configuration line in step 3 by

```
./configure --prefix=install-dir
```

Of course, this requires that the directory *`install-dir`*`/bin` is in your path.

Remark A.2. Alternatively, you may not install the software at all and run the software directly from the place where it was built. In that case, you may run the command

```
source set-devel-path
```

after step 3 instead of doing a `make install`. This will add the necessary directories to your `PATH`. However, you will have to rerun the script every time that you open a new shell session.

Appendix B

History of Mathemagix

B.1. Original design goals

During the late ninetees, our wish for a new general purpose computer algebra language was motivated by two main reasons: the quasi-absense of free computer algebra systems and the non-existence of sufficiently general compiled computer algebra languages. At that time, there were very few free computer algebra systems around. The systems Axiom, Maxima and Reduce, which are all free nowadays, carried proprietary licences by then. I found the Axiom system and its successor Aldor especially inspiring, and I originally intended to write something close to these.

Concrete plans for the Mathemagix project started in 1998, around the same time as the development of TeXmacs, which was originally intended as the interface for Mathemagix. Our original design goals were the following:

Strong typedness. Mathemagix should be strongly typed, with support for discrete and parameterized overloading, generic objects, compile-time type checking and, possibly, built-in support for expression types which interact with the type system.

High level control structures. Mathemagix should ultimately support high level control structures, like coroutines, generators, exceptions, continuations, etc. In the future we also wish to consider parallellism.

Runtime efficiency. This is a really a long-term goal, since writing a compiler is not a short-term objective. Nevertheless, the possibility to write a compiler which produces efficient code should be kept in mind. In particular, the language should support directives for controlling memory layout and inlining in a way which is naturally compatible with the type system.

Reusability of extern libraries. Before achieving runtime efficiency of Mathemagix itself, we aim to achieve runtime efficiency through the extensive reuse of existing dedicated libraries written in other languages. Mathemagix should therefore implement transparent mechanisms for reusing extern libraries and in particular C++ template libraries. Special care should be taken of garbage collection.

Good scalability. It should be possible to develop large computer algebra systems using Mathemagix in a natural and modular way. Special attention should be paid to constructs for programming in the large and the type system should naturally allow extensions of types and code.

B.2. History of the implementations

Since I had only one or two months every year to spend on the development of MATHEMAGIX, I have hesitated a lot about the most efficient way to have "at least something working" in which I could test some of my mathematical algorithms, possibly written in C or C++. One important, but rather unrealistic, development goal was to be able to improve the system gradually, and implement the harder aspects of the type system in an incremental way.

My first failed attempt to directly write a compiler occurred between 1999 and 2002. I directly intended to integrate support for continuations, which made debugging quite complex, and the overall project too hard to be realistic with little development time available.

For my second attempt, which started in 2003, I decided to start with the implementation of an interpreter, with the additional requirement that it should be very easy to integrate existing C++ libraries, and in particular some of my own libraries for relaxed power series and computable real numbers. This new interpreter was called `mmx-shell` and came with a uniform typed extension facility `mmx-extend` for gluing external C++ libraries. In parallel, we started the development of a standard C++ library MMXLIB.

At the start of the ANR GECKO project in 2005, the new interpreter was severely reorganized into the interpreter `mmx-light`. The aim was to reach a far more modular design, such that MATHEMAGIX would become a bunch of separate package, with possible interdependencies, and a standardized way to compile and install packages (based on AUTOTOOLS). In particular, the interpreter and the glue were designed to be as independent as possible, making it *a priori* possible to use the glue for another language with a completely different syntax. In retrospect, this has been somewhat of a waste of time, but I have always been playing with the thought of deriving a SCHEME implementation from MATHEMAGIX, so that we might also use it as an extension language for $\TeXmacs$.

However, the ultimate type system of MATHEMAGIX is hard to implement in a dynamically typed interpreter. Indeed, expressions can be essentially overloaded, and, contrary to C++, the correct unambiguous type of an expression can not necessarily be determined at the level of the parent of the expression (e.g. when using it as an argument to a function call). Instead of endlessly hacking an inperfect interpreter, I therefore restarted the implementation of the current `mmc` compiler around 2007. The new compiler was directly written in MATHEMAGIX itself, using the existing interpreter, and I rather quickly managed to produce a compiler which could compile itself. The possibility to declare functions inside functions and easily construct expressions and vectors turned out to be a huge accelating factor.

At the time of writing (november 2012), the compiler `mmc` has reached a quite stable status which makes it possible to write non trivial projects with it. In the meantime, Grégoire LECERF has developed the interpreter `mmi`, which is just another backend for the compiler. Besides the compiler itself, the `automagix`, `caas`, `mcoq` and `mmail` packages can be compiled using `mmc`. In the near future, we intend to experiment writing packages which rely more heavily on the support of categories and templates.

However, some parts of the implementation of `mmc` have become a bit hacky, so it is time for a partial rewrite at least. In the future, we intend to build a robust API for typed disambiguous programs which are manipulated a lot inside the compiler. After this reorganization, it should be easier to write a high quality optimizer. We also intend to replace the C++ backend by a C backend and `mmi` by a backend with JIT support.

Another point which remains quite puzzling is to have *some* support for untyped expressions, whether this support exists directly in the compiler, or in a separate system. Indeed, a typical end user of a computer algebra system may want to compute $x + y$ in a shell without specifying the types of x and y, or define the cube function simply by `cube(x) == x*x*x`. In 2012, we therefore started the developement of a new library for symbolic computation named `caas`. This library comes with a new untyped interpreter, but with a language which is similar to the official language recognized by the compiler. Future investigations will learn us how well the untyped and the typed view of the world can be integrated. One other main reason behind `caas` is that it could become a reasonable replacement for `mmx-light`, and a suitable light weight front-end for use in education.

B.3. History of the type system

A first draft for the type system was developed during this period (199*), largely inspired by Axiom and Aldor.

A first major change in the language occurred in 2001, after a discussion with Dan Grayson. He convinced me that the Axiom/Aldor way to import modules is suboptimal, since it requires the user to do a lot of bookkeeping of when to import what. Also, an often heard complaint about the Axiom system was that the hierarchy of categories is rather rigid.

For these reasons, I decided to introduce the `forall` construct in Mathemagix. At that point, I realized that it would be a good idea to associate explicit types to ambiguous expressions. The main task of the compiler would thus be to transform "ambiguously typed expressions" into "unambiguously typed expressions". I later realized that this point of view is very close to the "système F", introduced by Girard. However, the design of a compiler which performs the above disambiguization was (and partly still remains) a non trivial challenge.

Moreover, in early versions of the language, I also wanted support for implicit type conversions. It turned out that this requirement really introduced too many sources of ambiguity into the language, which made it really hard to design a comprehensive set of rules and primitives for which interpretations to prefer over which other interpretations. In 2011, we therefore decided to remove implicit conversions from the language, which made it possible to greatly simplify the compiler. For similar reasons, Stephen Watt decided to remove implementations from Axiom in is implementation of the Aldor language.

It should be noticed that, in the case of Mathemagix, removal of implicit conversions was is not a big sacrifice. Based on the `forall` primitive, an acceptable substitute was found, which essentially obliges the programmer to specify which arguments to functions accept implicit conversions. This is cleaner anyway, in our model where all declarations should be typed very precisely.

Notice that a sketch of the type system of Mathemagix can be found in my paper Overview of the Mathemagix type system.

Appendix C

The Emacs mode for Mathemagix

To edit MATHEMAGIX code with EMACS, the `mmx` mode can be used. It features automatic indentation and syntax highlighting. The necessary EMACS code is available in the file

`mmxlight/emacs/mmx-mode.el`

in the MATHEMAGIX source distribution and can be installed as follows:

```
cp mmx/mmxlight/emacs/mmx-mode.el $HOME/.emacs.d
```

The following should be added in the file `.emacs` in order to automatically activate the MATHEMAGIX mode when loading a file with suffix `.mmx` or `.amx`:

```
(setq load-path
  (append load-path (list (expand-file-name "~/.emacs.d"))))
(setq auto-mode-alist
  (append auto-mode-alist '(("\.mmx\|\.amx" . mmx-mode))))
(autoload 'mmx-mode
  "mmx-mode.el" "Major mode for editing Mathemagix files" t)
```

Appendix D

Guidelines about coding style

There a few rules about coding style that we try to follow ourselves in the Mathemagix libraries written in our own language. Although these rules are not mandatory, the readability of your code should be easier for others if you follow them.

D.1. Naming conventions

In general, we try to avoid abbreviations when chosing names for global variables, functions, classes and categories, and choose short (often one letter) names for local variables. For instance:

```
hamming_distance (i: Int, j: Int): Int == ...;
```

Of course, standard mathematical functions such as `exp`, `log`, `cos`, etc. carry their traditional names. We also recall the following general conventions from the section about regular identifiers:

- Use lowercase names for variables and functions.
- For names of types and categories, capitalize the first letter of each word categories (e.g. `Integer` or `Ordered_Group`).
- Capitalize all letters in macro names.
- Use the `?` suffix for names of predicates.

D.2. Indentation

The following indentation rules are implemented both in TeXmacs and in the Emacs mode for Mathemagix. In both cases, you may use the ↹ key for indenting the current line.

Blocks of code are usually indented by two spaces. For instance:

```
if x < y then {
  z: T == x;
  x := y;
  y := z;
}
```

Multiple line bodies of keywords are enclosed between braces `{` and `}`, whereas one line bodies are simply indented whenever we put them on separate lines:

```
if weather = "cold" then
  mmout << "take a coat!" << lf;
else if weather = "hot" then
  mmout << "a T-shirt will suffice" << lf;
else
  mmout << "syntax error in weather; please call Meteo France" << lf;
```

Input/output operators are indented as follows:

```
mmout << "first line" << lf
      << "second line" << lf;
```

Functions or vectors with many arguments are indented as follows:

```
mmout << beginners_function (a, b, c, d, e, f, g, h,
                             i, j, k, l, m, n, o, p,
                             q, r, s, t, u, v, w, x) << lf;
v: Vector Int == [ 10000, 10001, 10010, 10011, 10100, 10101, 10110,
10111,
                   11000, 11001, 11010, 11011, 11100, 11101, 11110, 11111
];
```

Sometimes, the readability can be enhanced by using *ad hoc* indentation rules:

```
if a = 1 then return 2*x + 3*y + 4*z;
         else return 3*x + 4*y + 2*z;
```

D.3. SPACING RULES

There are a few less strict rules concerning whitespace management:

- Function application takes one space before the bracket `(` and after every `,`

  ```
  r: Int == foo_bar (x, y, z);
  ```

  and similarly for data access using `postfix []`:

  ```
  val: Val == my_table [key];
  ```

  When the name of the function or data structure is particularly short, we may omit the space before `(` or `[`:

  ```
  mmout << f(x) + g(y) + h(z) << lf;
  mmout << v[1] * v[2] * v[3] << lf;
  ```

- Long mathematical expressions which do not fit on one line, such as

  ```
  return [ (-b - sqrt (square b - 4*a*c)) / (2*a),
           (-b + sqrt (square b - 4*a*c)) / (2*a) ];
  ```

are usually split over several lines, by introducing some auxiliary variables

```
disc: C == square b - 4*a*c;
return [ (-b - sqrt disc) / (2*a), (-b + sqrt disc) / (2*a) ];
```

- When defining a succession of variables, align the definitions on the `:` and `==` (or `:=`) symbols, if this does not leave to much whitespace:

```
Ints ==> Vector Int;
x1: Int  == a + 2*b + 3*c;
x2: Int  == p + 2*q + 3*r;
v : Ints == [ x1, x2 ];
```

- Use whitespace around mathematical infix operations whenever this enhances readability:

```
x1: R == a + 2*b + 3*c;
x2: R == x[1] * x[2]^2 * x[3]^5 + 123 * x[1]^7 * x[2]^11;
```

Index