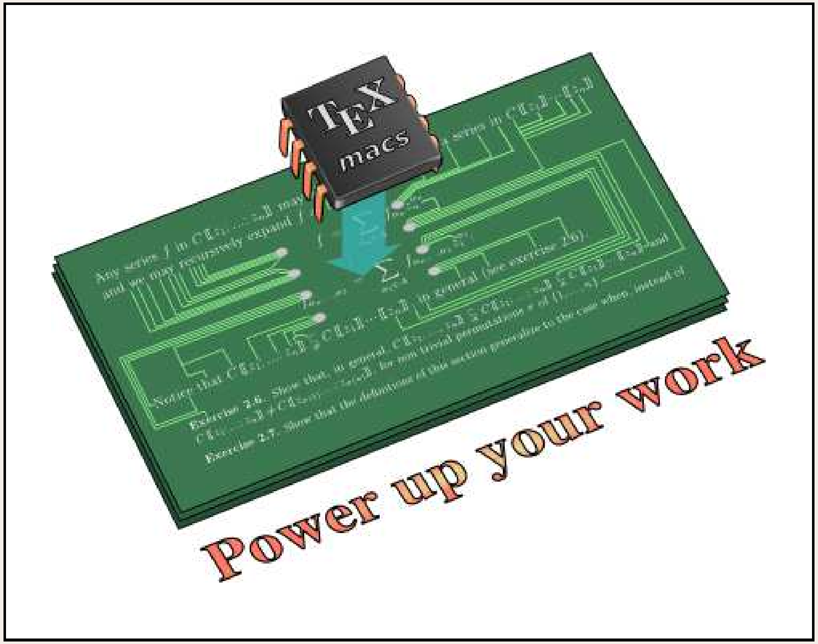
The MATHEMAGIX compiler



Joris van der Hoeven, Palaiseau 2011
http://www.TEXMACS.org

Motivation

- Existing computer algebra systems are slow for numerical algorithms

  $\rightsquigarrow$ we need a compiled language

- Low level systems (GMP, MPFR, FLINT) painful for compound objects

  $\rightsquigarrow$ we need a mathematically expressive language

- More and more complex architectures (SIMD, multicore, web)

  $\rightsquigarrow$ general efficient algorithms cannot be designed by hand

- Existing systems lack sound semantics

  $\rightsquigarrow$ we need mathematically clean interfaces

Motivation

- Existing computer algebra systems are slow for numerical algorithms

  ⇝ we need a compiled language

- Low level systems (GMP, MPFR, FLINT) painful for compound objects

  ⇝ we need a mathematically expressive language

- More and more complex architectures (SIMD, multicore, web)

  Non standard but efficient numeric types

  ⇝ general efficient algorithms cannot be designed by hand

- Existing systems lack sound semantics

  ⇝ we need mathematically clean interfaces

Motivation

- Existing computer algebra systems are slow for numerical algorithms

  ⇝ we need a compiled language

- Low level systems (GMP, MPFR, FLINT) painful for compound objects

  ⇝ we need a mathematically expressive language

- More and more complex architectures (SIMD, multicore, web)

  Non standard but efficient numeric types

  ⇝ general efficient algorithms cannot be designed by hand

- Existing computer algebra systems lack sound semantics

  Difficult to connect different systems in a sound way

  ⇝ we need mathematically clean interfaces

Main design goals

- Strongly typed functional language
- Access to low level details and encapsulation
- Inter-operability with C/C++ and other languages
- Large scale programming *via* intuitive, strongly local writing style

Guiding principle.

Prototype	$\leftsquigarrow\!\!\rightsquigarrow$	*Mathematical theorem*
Implementation	$\leftsquigarrow\!\!\rightsquigarrow$	*Formal proof*

Example

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

Example

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

Mathemagix

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

Example

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

C++

```
template<typename R>
square (const R& x) {
  return x * x;
}
```

Example

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

C++

```
concept Ring<typename R> {
  R::R (int);
  R::R (const R&);
  R operator - (const R&);
  R operator + (const R&, const R&);
  R operator - (const R&, const R&);
  R operator * (const R&, const R&);
}

template<typename R>
requires Ring<R>
operator * (const R& x) {
  return x * x;
}
```

Example

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

Axiom, Aldor

```
define Ring: Category == with {
  0: %;
  1: %;
  -: % -> %;
  +: (%, %) -> %;
  -: (%, %) -> %;
  *: (%, %) -> %;
}

Square (R: Ring): with {
  square: R -> R;
} == add {
  square (x: R): R == x * x;
}

import from Square (Integer);
```

Example

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

Ocaml

```
# let square x = x * x;;
val square: int -> int = <fun>

# let square_float x = x *. x;;
val square_float: float -> float = <fun>
```

Example

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

Ocaml

```
# module type RING =
    sig
      type t
      val cst : int -> t
      val neg : t -> t
      val add : t -> t -> t
      val sub : t -> t -> t
      val mul : t -> t -> t
    end;;

# module Squarer =
  functor (El: RING) ->
    struct
      let square x = El.mul x x
    end;;

# module IntRing =
    struct
      type t = int
      let cst x = x
      let neg x = - x
      let add x y = x + y
      let sub x y = x - y
      let mul x y = x * y
    end;;

# module IntSquarer = Squarer(IntRing);;
```

Language overview: functional programming

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

v: Vector Int == map (shift 123, [ 1 to 100 ]);

test (i: Int): (Int -> Int) == {
  f (): (Int -> Int) == g;
  g (j: Int): Int == i * j;
  return f ();
}
```

Language overview: overloading

```
category Type == {}

forall (T: Type) f (x: T): T == x;
f (x: Int): Int == x * x;
f (x: Double): Double == x * x * x * x;

mmout << f ("Hallo") << "\n";
mmout << f (11111) << "\n";
mmout << f (1.1) << "\n";
```

```
Castafiore:basic vdhoeven$ ./overload_test
Hallo
123454321
1.4641
Castafiore:basic vdhoeven$
```

Language overview: classes

```
class Point == {
  mutable x: Int;
  mutable y: Int;

  constructor point (a: Int, b: Int) == {
    x == a; y == b; }

  mutable method translate (dx: Int, dy: Int): Void == {
    x := x + dx; y := y + dy; }
}

flatten (p: Point): Syntactic ==
  'point (flatten f.x, flatten f.y);

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

Language overview: implicit conversions

```
convert (x: Double): Floating == mpfr_as_floating x;

forall (R: Ring)
upgrade (x: R): Complex R == complex (x, 0);
// allows for conversion Double --> Complex Floating

convert (p: Point): Vector Int == [ p.x, p.y ];
downgrade (p: Colored_point): Point == point (p.x, p.y);
// allows for conversion Colored_point --> Vector Int
// abstract way to implement class inheritance
```

Language overview: categories

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

category Module (R: Ring) == {
  prefix -: This -> This;
  infix +: (This, This) -> This;
  infix -: (This, This) -> This;
  infix *: (R, This) -> This;
}

forall (R: Ring, M: Module R)
square_multiply (x: R, y: M): M == (x * x) * y;

mmout << square_multiply (3, 4) << "\n";
```

Language overview: foreign imports

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

foreign cpp import {
  cpp_flags   "`numerix-config --cppflags`";
  cpp_libs    "`numerix-config --libs`";
  cpp_include "numerix/complex.hpp";

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

  forall (R: Ring) {
    complex: R -> Complex R == keyword constructor;
    complex: (R, R) -> Complex R == keyword constructor;
    upgrade: R -> Complex R == keyword constructor;
    Re: Complex R -> R == Re;
    Im: Complex R -> R == Im;

    prefix -: Complex R -> Complex R == prefix -;
    square: Complex R -> Complex R == square;
    infix +: (Complex R, Complex R) -> Complex R == infix +;
    infix -: (Complex R, Complex R) -> Complex R == infix -;
    infix *: (Complex R, Complex R) -> Complex R == infix *;
  }

  forall (R: Field) {
    infix /: (Complex R, Complex R) -> Complex R == infix /;
    infix /: (R, Complex R) -> Complex R == infix /;
    infix /: (Complex R, R) -> Complex R == infix /;
```

Language overview: foreign exports

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

Language overview: value parameters

```
class Vec (R: Ring, n: Int) == {
  private mutable rep: Vector R;

  constructor vec (v: Vector R) == {
    rep == v; }
  constructor vec (c: R) == {
    rep == [ c | i: Int in 0..n ]; }
}

forall (R: Ring, n: Int) {
  flatten (v: Vec (R, n)): Syntactic == flatten v.rep;
  postfix [] (v: Vec (R, n), i: Int): R == v.rep[i];
  postfix [] (v: Alias Vec (R, n), i: Int): Alias R == v.rep[i];
  infix + (v1: Vec (R, n), v2: Vec (R, n)): Vec (R, n) ==
    vec ([ v1[i] + v2[i] | i: Int in 0..n ]);

  assume (R: Ordered)
  infix <= (v1: Vec (R, n), v2: Vec (R, n)): Boolean ==
    big_and (v1[i] <= v2[i] | i: Int in 0..n);
}
```

Type system: logical and penalty types

Overloading. Explicit types for overloaded objects

```
forall (T: Type) f (x: T): T == x;
f (x: Int): Int == x * x;
```

Type of `f`: `And (Forall (T: Type, T -> T), Int -> Int)`

Logical types: $f : \mathrm{And}(T, U) \Longleftrightarrow f : T \wedge f : U$

Penalties for overloading and conversions.

```
penalty (access)
postfix []: (Alias Vector C, Int) -> Alias C == write_access;
postfix []: (Vector C, Int) -> Vector C == read_access;
```

When **reading** an entry of a **mutable** vector, the second method is preferred.

Penalty types: `Penalty (access, (Alias Vector C, Int) -> Alias C)`

Type system: ambiguities

Partial ordering on (synthetic compound) penalties.

```
p: Polynomial C == ...;
z: Complex C == ...;
mmout << p + z << "\n";
// ERROR: Polynomial Complex C or Complex Polynomial C?
```

Penalties $(\mathrm{convert}, \mathrm{none})$ and $(\mathrm{none}, \mathrm{convert})$ are incomparable.

Apply best first.

```
v: Vector Integer == [ 1 to 10 ];
w: Vector Rational == map (square, v);
// use square on Integer or Rational entries?
```

Solution: we perform `map (square :> (Integer -> Integer), v)`

Prefer $\mathrm{none}; \mathrm{convert}$ to $\mathrm{convert}; \mathrm{none}$.

Efficiency: no garbage collection

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

```
static function_1<int, const int& > LAMBDA_NEW_pGU1 (const int &x_1);

function_1<int, const int& >
shift_GU1 (const int &x_1) {
  return LAMBDA_NEW_pGU1 (x_1);
}

struct LAMBDA_NEW_pGU1_rep: public function_1_rep<int, const int& > {
  int x_1;

  int
  apply (const int &y_1) {
    return x_1 + y_1;
  }

  inline LAMBDA_NEW_pGU1_rep (const int &x_1_bis): x_1 (x_1_bis) {}
};

static function_1<int, const int& >
LAMBDA_NEW_pGU1 (const int &x_1) {
  return function_1<int, const int& > (new LAMBDA_NEW_pGU1_rep (x_1));
}
```

Efficiency: low level access

```
foreign cpp import {
  cpp_preamble "#define ptr(x,n) (x)*";

  class Array (T: Type, n: Int) == (cpp_macro ptr) (T, n);

  forall (T: Type, n: Int) {
    postfix [] (v: Array (T, n), i: Int): T == postfix [];
    postfix [] (v: Alias Array (T, n), i: Int): Alias T == postfix [];
  }
}

forall (R: Ring, n: Int) {
  inplace prefix - (d: Alias Array (T, n), s: Array (T, n)) ==
    for i: Int in 0..n do d[i] := -s[i];
}
```

Efficiency: implementation of categories I

```
category Magma == {
  infix *: (This, This) -> This;
}
```

Virtual representation base class for magmas

```
typedef generic Magma_EL_1;

struct Magma_1_rep: public rep_struct {
  virtual Magma_EL_1 sample_1 () const;
  virtual Magma_EL_1 TT_1 (const Magma_EL_1 &TT_ARG1_5,
                           const Magma_EL_1 &TT_ARG2_5) const;
  inline Magma_1_rep ();
  virtual inline ~Magma_1_rep ();
};

Magma_EL_1
Magma_1_rep::sample_1 () const {
  generic ();
}

Magma_EL_1
Magma_1_rep::TT_1 (const Magma_EL_1 &TT_ARG1_5,
                   const Magma_EL_1 &TT_ARG2_5) const {
  throw string ("not implemented");
}

inline Magma_1_rep::Magma_1_rep () {}
inline Magma_1_rep::~Magma_1_rep () {}
```

Efficiency: implementation of categories I

```
category Magma == {
  infix *: (This, This) -> This;
}
```

Public magma class as pointer to representation class

```
struct Magma_1 {
  Magma_1_rep *rep;
  inline Magma_1 (): rep (new Magma_1_rep ()) {}

  inline ~Magma_1 () {
    DEC_COUNT (rep);
  }

  inline Magma_1 (const Magma_1 &x): rep (x.rep) {
    INC_COUNT (rep);
  }

  inline Magma_1
  &operator = (const Magma_1 &x) {
    INC_COUNT (x.rep);
    DEC_COUNT (rep);
    rep = x.rep;
    return *this;
  }

  inline Magma_1 (const Magma_1_rep *x):
    rep (const_cast<Magma_1_rep* > (x))
  {
    INC_COUNT (rep);
  }

  inline const Magma_1_rep
  *operator -> () const {
    return rep;
```

Efficiency: implementation of categories II

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

Concrete magma representation class

```
struct Int_Magma_pGY1_rep: public Magma_1_rep {
  Magma_EL_1
  sample_1 () const {
    return concrete_to_abstract<int, Magma_EL_1 > (int ());
  }

  Magma_EL_1
  TT_1 (const Magma_EL_1 &ARGA1_1, const Magma_EL_1 &ARGA2_1) const {
    return concrete_to_abstract<int, Magma_EL_1 >
      (abstract_to_concrete<Magma_EL_1, int > (ARGA1_1) *
       abstract_to_concrete<Magma_EL_1, int > (ARGA2_1));
  }

  inline Int_Magma_pGY1_rep () {}
};

Magma_1
Int_Magma_pGY1 () {
  return Magma_1 (new Int_Magma_pGY1_rep ());
}
```

Efficiency: implementation of categories II

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

The generic cube function

```
struct LAMBDA_NEW_pGY1_rep:
  public function_1_rep<Magma_EL_1, const Magma_EL_1& >
{
  Magma_1 R_1;

  Magma_EL_1
  apply (const Magma_EL_1 &x_1) {
    return R_1->TT_1 (R_1->TT_1 (x_1, x_1), x_1);
  }

  inline LAMBDA_NEW_pGY1_rep (const Magma_1 &R_1_bis): R_1 (R_1_bis) {}
};

function_1<Magma_EL_1, const Magma_EL_1& >
cube_pGY1 (const Magma_1 &R_1) {
  return function_1<Magma_EL_1, const Magma_EL_1& >
    (new LAMBDA_NEW_pGY1_rep (R_1));
}

void
mmx_initialize_GY () {
  ...
  mmout << as<int > (cube_GY1 (Int_Magma_pGY1 ()) (as<Magma_EL_1 > (111)))
        << string ("\n");
}
```

Generic instantiation of foreign templates

```
foreign cpp import {
  forall (M: Magma) cube: (M, M) -> M == cube;
}
```

```
struct Magma_OBJ_1_rep: public rep_struct {
  Magma_EL_1 val;
  Magma_1 tp;
  inline Magma_OBJ_1_rep () {}
  inline Magma_OBJ_1_rep (const Magma_EL_1 &val_bis,
                          const Magma_1 &tp_bis):
    val (val_bis), tp (tp_bis) {}
};

struct Magma_OBJ_1 {
  Magma_OBJ_1_rep *rep;
  ...
};

inline Magma_OBJ_1
operator * (const Magma_OBJ_1 &a1, const Magma_OBJ_1 &a2) {
  return Magma_OBJ_1 (a1->tp->TT_1 (a1->val, a2->val), a1->tp);
}
```

Programming in the large

- No makefiles
  - ⇝ automatically determine dependencies from include instructions
- No object files
  - ⇝ maintain a local cache with object files
- No main function
  - ⇝ any file can be the main entry point
- Little and only global compilation options (`--debug`, `--optimize`)
  - ⇝ maintain separate caches as a function of options
- Fine grained parallel building
  - ⇝ independent processes can be spawn at the middle of compilation

Compiling the compiler

```
Shell] cd $HOME/mathemagix/mmx/
Shell] source ./set-devel-paths
Shell] cd mmcompiler/compiler
Shell] rm -rf $HOME/.mathemagix/mmc
Shell] mmc --verbose mmc.mmx
Shell] mmc --optimize --verbose mmc.mmx
Shell]
```