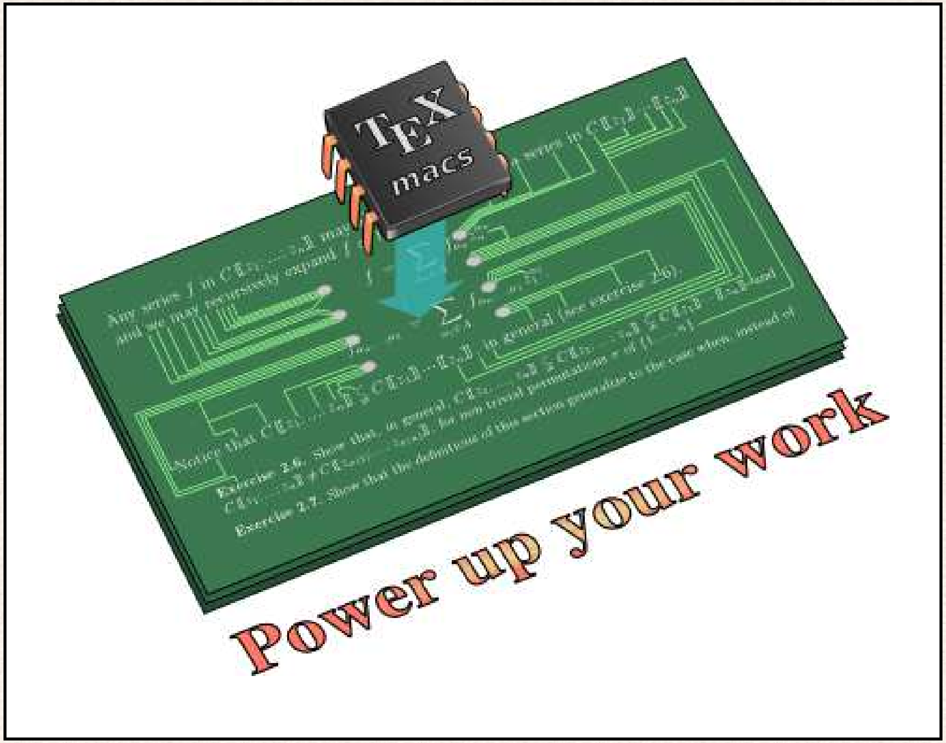
Interfacing MATHEMAGIX and C++



Joris van der Hoeven, ISSAC 2013
http://www.TeXmacs.org

Main design goals of Mathemagix

- Freely available from `http://www.mathemagix.org` (including documentation)

Main design goals of Mathemagix

- Freely available from `http://www.mathemagix.org` (including documentation)
- Strongly typed functional language for mathematical programming

Main design goals of Mathemagix

- Freely available from `http://www.mathemagix.org` (including documentation)
- Strongly typed functional language for mathematical programming
- Efficient compiler

Main design goals of Mathemagix

- Freely available from `http://www.mathemagix.org` (including documentation)
- Strongly typed functional language for mathematical programming
- Efficient compiler
- Access to low level details and encapsulation

Main design goals of Mathemagix

- Freely available from `http://www.mathemagix.org` (including documentation)
- Strongly typed functional language for mathematical programming
- Efficient compiler
- Access to low level details and encapsulation
- Inter-operability with C/C++ and other languages

Main design goals of Mathemagix

- Freely available from `http://www.mathemagix.org` (including documentation)
- Strongly typed functional language for mathematical programming
- Efficient compiler
- Access to low level details and encapsulation
- Inter-operability with C/C++ and other languages
- Large scale programming *via* intuitive, strongly local writing style

Main design goals of Mathemagix

- Freely available from `http://www.mathemagix.org` (including documentation)
- Strongly typed functional language for mathematical programming
- Efficient compiler
- 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	$\leftrightsquigarrow$	*Mathematical theorem*
Implementation	$\leftrightsquigarrow$	*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;
}
```

Note. No mathematical ring properties required

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
```

Full genericity

```
do_something (K: Field, n: Int): Void == {
  ...
  if n>0 then do_something (Polynomial K, n-1);
}

for p: Int in 2..100000 do
  if prime? p then {
    F_p: Field == Modular (Int, p);
    do_something (F_p, p);
  }
```

Preparing for imports from C++

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

Importing basic types and functions

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

  class Integer == integer;
  literal_integer: Literal -> Integer == make_literal_integer;

  prefix -: Integer -> Integer == prefix -;
  infix +: (Integer, Integer) -> Integer == infix +;
  infix -: (Integer, Integer) -> Integer == infix -;
  infix *: (Integer, Integer) -> Integer == infix *;
  ...
}
```

Special constructor for literal integers such as 12345678987654321:

```
integer make_literal_integer (const literal&);
```

Syntactic sugar

Data access.

```
postfix .x: Point -> Double == get_x;
```

Data conversions.

```
upgrade: Integer -> Rational == keyword constructor;
```

```
downgrade: Colored_Point -> Point == as_point;
```

Tuples.

```
num_vector: Tuple Double -> Num_Vector == keyword constructor;
```

Generators.

```
downgrade: Num_Vector -> Generator Double == explode;
```

```
v: Num_Vector == ...;
for x: Double in v do ...;
```

Exporting to C++

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

Implementing genericity: categories

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

```
class Monoid_rep: public rep_struct {
  inline Monoid_rep ();
  virtual inline ~Monoid_rep ();
  virtual generic mul (const generic&, const generic&) const = 0;
  ...
};

typedef managed_pointer<Monoid_rep> Monoid;
```

Implementing genericity: function templates

```
forall (M: Monoid) cube (x: M): M == x*x*x;
```

```
generic
cube (const Monoid& M, const generic& x) {
  // x is assumed to contain an object "of type M"
  return M->mul (x, M->mul (x, x));
}
```

Note. `generic` ⇜⇝ managed `void*`

Implementing genericity: instantiations

```
c: Int == cube 3;
```

```
struct Int_Ring_rep: public Ring_rep {
  ...
  generic
  mul (const generic& x, const generic& y) const {
    return as_generic<int> (from_generic<int> (x) *
                            from_generic<int> (y));
  }
  ...
};
```

```
Monoid Int_Ring= new Int_Ring_rep ();
int c= from_generic<int> (cube (Int_Ring, as_generic<int> (3)));
```

Importing C++ templates ...

```
foreign cpp import {
  ...
  class Pol (R: Ring) == polynomial R;

  forall (R: Ring) {
    pol: Tuple R -> Pol R == keyword constructor;
    upgrade: R -> Pol R == keyword constructor;

    deg: Pol R -> Int == deg;
    postfix []: (Pol R, Int) -> R == postfix [];

    prefix -: Pol R -> Pol R == prefix -;
    infix +: (Pol R, Pol R) -> Pol R == infix +;
    infix -: (Pol R, Pol R) -> Pol R == infix -;
    infix *: (Pol R, Pol R) -> Pol R == infix *;
    ...
  }
}
```

... requires exportation of categories

```
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 *;
}
```

Two approaches to genericity

Generic function arguments $\leftrightsquigarrow$ Pairs $(\text{value}, \text{type})$

Approach 1. Directly store pairs $(\text{value}, \text{type})$

+ Allows for functional programming style
– Needs tweaking of C++ code:
  what is the type of the sum the elements in an empty vector?

Approach 2. Store the type in a global variable which is dynamically changed

+ No need to tweak the C++ code
– Less functional ($\rightsquigarrow$ more work for support of multi-threading)

Generic instance classes

```
template<typename Cat, int Nr>
class instance {
public:
  generic rep;
  static Cat Cur;
  inline instance (const instance& prg2): rep (prg2.rep) {}
  inline instance (const generic& prg): rep (prg) {}
  instance ();
  template<typename C1> instance (const C1& c1);
  ...
};
```

```
// export of infix *: (This, This) -> This == infix * from Ring
template<int Nr> inline instance<Ring,Nr>
operator * (const instance<Ring,Nr> &a1,
            const instance<Ring,Nr> &a2) {
  typedef instance<Ring,Nr> Inst;
  return Inst (Inst::Cur->mul (a1.rep, a2.rep));
}
```

Importing C++ templates

```
forall (R: Ring)
infix *: (Pol R, Pol R) -> Pol R;
```

```
polynomial<generic>
mul (const Ring &R,
     const polynomial<generic> &p1,
     const polynomial<generic> &p2)
{
  typedef instance<Ring,1> Inst;
  Ring old_R= Inst::Cur;
  Inst::Cur= R;
  polynomial<Inst> P1= as<polynomial<Inst> > (p1);
  polynomial<Inst> P2= as<polynomial<Inst> > (p2);
  polynomial<Inst> R = P1 * P2;
  polynomial<generic> r= as<polynomial<generic> > (R);
  Inst::Cur= old_R;
  return r;
}
```

Final notes

Optimizations.

```
p: Pol Integer == ...;
q: Pol Integer == p * p;
```

```
polynomial<integer> p= ...;
polynomial<integer> q= p * p;
```

Need for the `Nr` parameter.

```
forall (R1: Ring, R2: Ring)
map (f: R1 -> R2, v: Vector R1): Vector R2 == map;
// R1 -> instance<Ring,1>
// R2 -> instance<Ring,2>
```

Dependent types.

Automagically works