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

<table>
<thead>
<tr>
<th>Guiding principle.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prototype</td>
</tr>
<tr>
<td>Implementation</td>
</tr>
</tbody>
</table>
forall (R: Ring) square (x: R) == x * x;
Example

\[\text{forall } (R: \text{ Ring}) \text{ square } (x: R) == x * x;\]

Mathemagix

\[
\text{category Ring == \{} \n\quad \text{convert: Int -> This; } \\
\quad \text{prefix -: This -> This; } \\
\quad \text{infix +: (This, This) -> This; } \\
\quad \text{infix -: (This, This) -> This; } \\
\quad \text{infix *: (This, This) -> This; } \\
\text{\}}
\]

Note. No mathematical ring properties required
forall (R: Ring) square (x: R) == x * x;

C++

```cpp
template<typename R>
square (const R& x) {
    return x * x;
}
```
forall (R: Ring) square (x: R) == x * x;

C++

```cpp
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**

```plaintext
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

\[
\text{forall } (R: \text{Ring}) \text{ square} \ (x: R) = x \times x;
\]

Ocaml

```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;`
# 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
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++

```cpp
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.

\[
\text{postfix .}x:\text{ Point} \rightarrow \text{ Double} == \text{ get}_x;
\]

Data conversions.

\[
\text{upgrade: Integer} \rightarrow \text{ Rational} == \text{ keyword}\text{ constructor};
\]

\[
\text{downgrade: Colored_Point} \rightarrow \text{ Point} == \text{ as_point};
\]

Tuples.

\[
\text{num_vector: Tuple Double} \rightarrow \text{ Num\_Vector} == \text{ keyword}\text{ constructor};
\]

Generators.

\[
\text{downgrade: Num\_Vector} \rightarrow \text{ Generator Double} == \text{ explode};
\]

\[
v: \text{ Num\_Vector} == \ldots;
\]
\[
\text{for } x: \text{ Double } \text{ in } v \text{ do } \ldots;
\]
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;
}
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;
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*
c: \texttt{Int} \texttt{==} \texttt{cube 3};

struct \texttt{Int\_Ring\_rep}: \texttt{public Ring\_rep} {
\dots
\texttt{generic}
\texttt{mul (const generic\& x, const generic\& y) const \{ return as\_generic<int> (from\_generic<int> (x) * from\_generic<int> (y)); \}}
\texttt{\dots}
};

\texttt{Monoid Int\_Ring= new Int\_Ring\_rep (); \texttt{int c= from\_generic<int> (cube (Int\_Ring, as\_generic<int> (3)))};}
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

```cpp
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 $\iff$ Pairs (value, type)

**Approach 1.** Directly store pairs (value, 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 ($\iff$ more work for support of multi-threading)
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));\}
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;
}
Optimizations.

\[
p: \text{Pol Integer} \equiv \ldots; \\
q: \text{Pol Integer} \equiv p \ast p;
\]

\[
p: \text{polynomial<integer>} \equiv \ldots; \\
q: \text{polynomial<integer>} \equiv p \ast p;
\]

Need for the *Nr* parameter.

\[
\text{forall} (R1: \text{Ring}, R2: \text{Ring}) \\
\text{map} (f: R1 \rightarrow R2, v: \text{Vector} R1): \text{Vector} R2 \equiv \text{map}; \\
// R1 \rightarrow \text{instance<Ring,1>}
// R2 \rightarrow \text{instance<Ring,2>}
\]

Dependent types.

Automagically works