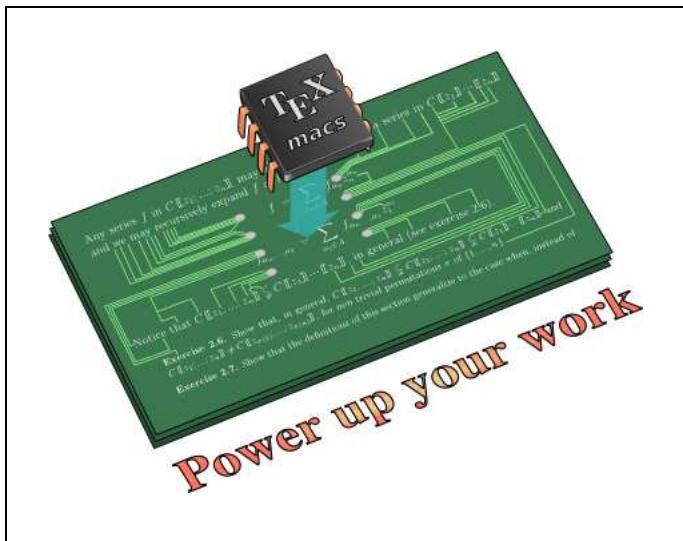


## The MATHEMAGIX type system



Power up your work

Joris van der Hoeven, ASCM 2012

<http://www.TEXMACS.org>

## 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)
  - ~~> general efficient algorithms cannot be designed by hand
- 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
- 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*      ↪↔      *Mathematical theorem*  
*Implementation*    ↪↔      *Formal proof*

## 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;
}
```

## C++

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

```

## 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);

```

## Ocaml

```

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

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

```

## 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);;

# IntSquarer.square 11111;;
- : int = 123454321

```

## 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 ();  
}
```

## 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 p.x, flatten p.y);

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

## 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$
```

## 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";
```

## Implicit conversions

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

forall (R: Ring) {
    infix * (v: Vector R, w: Vector R): Vector R == [
    ... ];
    forall (K: To R)
        infix * (c : K, v: Vector R): Vector R ==
            [ (c :> R) * x | x: R in v ];
    infix * (v: Vector R, c :> R): Vector R ==
        [ x*c | x: R in v ];
}

forall (R: Ring)
convert (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
```

## Value parameters for containers

```
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);
}
```

## Abstract data types

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

l1: List Int == cons (1, cons (2, null ()));
l2: List Int == cons (1, cons (2, cons (3, null
())));

forall (T: Type)
prefix # (l: List T): Int ==
    if null? l then 0 else #l.tail + 1;
```

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

l1: List Int == cons (1, cons (2, null ()));
l2: List Int == cons (1, cons (2, cons (3, null
())));

forall (T: Type)
prefix # (l: List T): Int ==
    match l with {
        case null () do return 0;
        case cons (_, l: List T) do return #l + 1;
    }
```

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

l1: List Int == cons (1, cons (2, null ()));
l2: List Int == cons (1, cons (2, cons (3, null
())));

forall (T: Type) {
    prefix # (l: List T): Int := 0;
    prefix # (cons (_, t: List T)): Int := #t + 1;
}
```

## Symbolic types

```
structure Symbolic := {
    sym_literal (literal: Literal);
    sym_compound (compound: Compound);
}

infix + (x: Symbolic, y: Symbolic): Symbolic :=
    sym_compound ('+ (x :> Generic, y :> Generic));
```

```
structure Symbolic := {
    sym_literal (literal: Literal);
    sym_compound (compound: Compound);
}

infix + (x: Symbolic, y: Symbolic): Symbolic :=
    sym_compound ('+ (x :> Generic, y :> Generic));

structure Symbolic += {
    sym_int (int: Int);
    sym_double (double: Double);
}

infix + (sym_double (x: Double),
          sym_double (y: Double)): Symbolic :=
    sym_double (x + y);
```

```

structure Symbolic := {
    sym_literal (literal: Literal);
    sym_compound (compound: Compound);
}

infix + (x: Symbolic, y: Symbolic): Symbolic :=
    sym_compound ('+ (x :> Generic, y :> Generic));

structure Symbolic += {
    sym_int (int: Int);
    sym_double (double: Double);
}

pattern sym_as_double (as_double: Double): Symbolic
:= {
    case sym_double (x: Double) do as_double == x;
    case sym_int (i: Int) do as_double == i;
}

infix + (sym_as_double (x: Double),
          sym_as_double (y: Double)): Symbolic :=
    sym_double (x + y);

```

## Type system: logical 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:  $f : \text{And}(T, U) \iff f : T \wedge f : U$

**Preferences in case of ambiguities.**

```
infix +: (Int, Int): Int;  
infix +: (Int, Integer): Integer;  
infix +: (Integer, Integer): Integer;  
  
prefer infix + :> (Int, Int) -> Int  
to     infix + :> (Int, Integer) -> Integer;
```

## Formal theory and compilation

**Level 1.** Source language with syntax constructs for ambiguous notations

$$\text{square: } (\forall T^{\text{Ring}} \rightarrow T) \wedge \text{String} \rightarrow \text{String}$$

**Level 2.** Intermediate unambiguous language with additional constructs for disambiguating the ambiguous notations

$$\text{square} \xrightarrow{\text{valid interpretation}} \pi_1(\text{square}) \# \text{Int: Int} \rightarrow \text{Int}$$

Compilation: transform source program in intermediate program.

**Level 3.** Interpretation in traditional  $\lambda$ -calculus

$$\text{square} \equiv \text{pair}(\lambda T. \lambda x. \text{get}_x(T)(x, x), \lambda x. \text{concat}(x, x))$$

Backend: transform intermediate program in object program