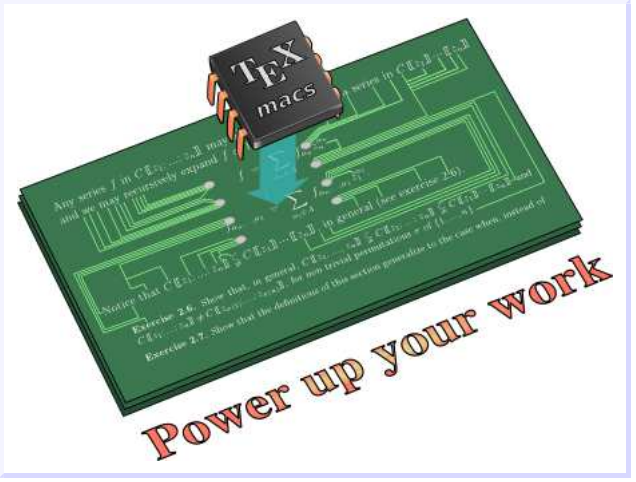
Vers un compilateur pour Mathemagix



Gecko 2007

http://www.mathemagix.org

http://www.TEXMACS.org

Motivations et objectifs

- Langage de programmation orienté vers les mathématiques
  - Langage fonctionnel fortement typé avec généricité.
  - Manipulation simple et intégrée d'objets symboliques.
  - Possibilité de compilation $\rightsquigarrow$ bonnes performances.
- Historique
  - Lisp, Maxima.
  - Axiom, Aldor.
  - Mathemagix.
    1. Compilateur sur le modèle d'Aldor.
    2. Interpréteur, mmx-shell, système de types avec surcharge.
    3. Interpréteur abstrait, mmx-light, type symbolique.
    4. Compilateur.

Langages existants

	C++	Aldor	Lisp	Ocaml
Surcharge	oui	oui	clos	non
Conversions implicites	ouais	non	non	non
Templates (modèles)	ouais	non	macro	oui
Généricité	non	oui	non	ouais
Types symboliques	non	non	oui	oui
Fonctionnel	non	oui	oui	oui
Efficacité	oui	ouais	ouais	ouais
Standard	oui	non	oui	non

- Langages sans surcharge : Ocaml, etc.

$$f(x) \equiv x + 1$$

Typage automatique $f : \mathrm{Int} \longrightarrow \mathrm{Int}$

$$f(x) \equiv x +_{\mathrm{Float}} 1.0$$

- Langages avec surcharge : C++, Aldor, Mathemagix, etc.

$$\begin{array}{rcl} + & : & \mathrm{Int}, \mathrm{Int} \longrightarrow \mathrm{Int} \\ + & : & \mathrm{Float}, \mathrm{Float} \longrightarrow \mathrm{Float} \end{array}$$

Déclarations typées :

$$f(x : \mathrm{Int}) : \mathrm{Int} \equiv x + 1$$

Conversions implicites

- Transitivité

  Caster. Pas de transitivité.

  Upgrader. À droite : $A \xrightarrow{\text{up}} B, B \longrightarrow C \Rightarrow A \longrightarrow C$ constructeur

  Downgrader. À gauche : $A \longrightarrow B, B \xrightarrow{\text{down}} C \Rightarrow A \longrightarrow C$ héritage

  Converter. Les deux.

- Priorité

  Inclusions. $\text{Integer} \longrightarrow \text{Rational}$

  Homomorphismes. $\text{Integer} \longrightarrow \text{Modular\_integer}(p)$

  Caster. $\text{Integer} \longrightarrow \text{Float}$

  Abstraction. $\text{Integer} \longrightarrow \text{Generic}$

  $\forall T, T \longrightarrow \text{Generic}$ et $\forall T, T \longrightarrow \text{Vector}(T)$

- Décidabilité

$$A_1 \longrightarrow \cdots \longrightarrow A_n \longrightarrow B_n \longrightarrow \cdots \longrightarrow B_0$$

- Templates sans vérification : C++, etc.

```
template<typename C> vector<C>
operator + (const vector<C>& v, const vector<C>& w) {
  int i, n= min (dimension (v), dimension (w));
  vector<C> r (n)
  for (i=0; i<n; i++) r[i]= v[i] + w[i];
  return r;
}
```

- Catégories : Aldor, etc.

```
Ring: Category == with {
  +: (%, %) -> %;
  ...
}

Vector (R: Ring): Ring == add {
  (x: %) + (y: %): % == {
    import R;
    ...
  }
  ...
```

```
}
```

- Desavantages
  - Importation implicite disagréable.
  - Demande structuration au préalable en catégories.
  - Routines doivent être attachées à des catégories.

- « Paradigme Mathemagix » : déclarations sous hypothèses

```
Ring: Category == category {
  +: (%, %) -> %;
  ...
}

forall (R: Ring)
infix + (v: Vector R, w: Vector R): Vector R == {
  ...
}
```

Différences dans l'implantation

- Templates C++
  - Macros. Exemple : `complexify<C>` $\rightarrow$ `complexify<double>`.
  - Problème : `modular<p>` quand `p` est variable.
  - Solution : variable globale avec `p`, mais multi-threading...
  - Problème : `polynomial<C>` avec `C` variable.
- Généricité
  - Passage de paramètre : `mpz_mul (limb* d, limb* a, limb* b, int i, int j);`
  - Ou encore : `ring_el* square (ring_el* x, ring R);`
  - Problème : passage du paramètre admet un coût.
- Unification
  - Même code dans les deux cas.
  - Exemple : $\forall R\colon \mathrm{Ring}, \forall i\colon \mathbb{N}, \forall j\colon \mathbb{N}, \forall k\colon \mathbb{N}, \mathrm{Mat}(R,i,j) \times \mathrm{Mat}(R,j,k) \rightarrow \mathrm{Mat}(R,i,k)$.
  - Utilisation générique et spécialisation pour $\mathrm{Mat}(\mathrm{Double}, 2, 2)$.

Spécialisations

- Préférence des types les plus spécialisés

```
forall (T: Type) f (x: T): Void == ...;
f (x: Rational): Void == ...;
f (x: Integer): Void == ...;

f(3);
```

- Priorités

```
(v: Vector T) [i: Integer]: T == ...;
(v: Alias Vector T) [i: Integer]: Alias T == ...;

v: Vector T := ...;
mmout << v[3] << "\n";
v[4] := x;
```

Note : priorités $\leftrightsquigarrow$ relation d'ordre abstrait $\leftrightsquigarrow$ conversions implicites atomiques

Typer les ambiguïtés

- Surcharge $\wedge$

  $+ : (\text{Int}, \text{Int}) \longrightarrow \text{Int} \ \wedge \ (\text{Float}, \text{Float}) \longrightarrow \text{Float}$

- Unions $\vee$

  $\text{Failable}(T) \equiv T \vee \{\text{false}\}$

- Modèles $\forall$

  $\text{print} : \forall T, (\text{Printer}, T) \longrightarrow \text{Void}$

- Généricité $\exists$

  $\text{Generic} \equiv \exists T, T$

- Types conditionnels $\Rightarrow$

  $\times : \forall T, T : \text{Ring} \Rightarrow (\text{Vector}\, T, \text{Vector}\, T) \rightarrow \text{Vector}\, T$

- Conversions implicites, hypothèses

  $\forall T, T \longrightarrow \text{Vector}\, T$

- Compatibilité entre Generic et l'environnement

```
x: Generic == 2;
f (x: Integer): Integer == ...;
f(x);
```

- Dispatching automatique

```
(x: dispatch Generic) + (y: dispatch Generic): Generic;
(x: Integer) + (y: Integer): Integer;
(x: Rational) + (y: Rational): Rational;
```

Problèmes de scope ?

- Downgrade automatique

```
2 * ((3 :> Generic) / 2)
```

Spécialisation

```
cast: Rational -> Generic
```