

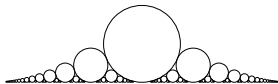
# Faire bonne figure avec MLPOST

R. Bardou & J.-C. Filliâtre & J. Kanig & S. Lescuyer

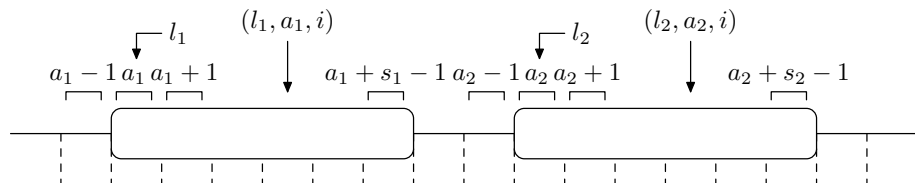
Équipe ProVal

INRIA Saclay – Île-de-France  
CNRS / LRI Université Paris-Sud

JFLA — 3 février 2009

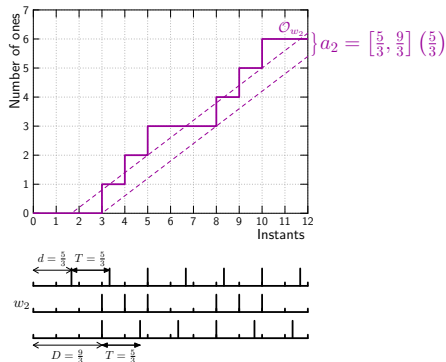
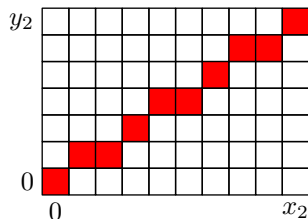


comment réaliser de belles figures contenant des éléments  $\text{\LaTeX}$  ?



# Motivations 2/2

comment réaliser des figures utilisant des calculs ?

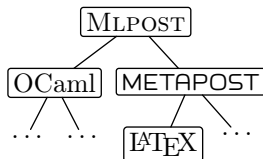


- interfaces graphiques : Dia, Xfig
  - + simple à utiliser (CQVVECQVO)
  - intégration d'éléments  $\text{\LaTeX}$  difficile
  - pas d'automatisation possible
- bibliothèques  $\text{\LaTeX}$  : PSTricks, TikZ/PGF
  - + intégration avec  $\text{\LaTeX}$  optimale
  - programmation difficile
- outils externes : METAPOST
  - + très bonne intégration avec  $\text{\LaTeX}$
  - + langage spécifique
  - langage spécifique, avec de nombreux défauts

# Notre solution : MLPOST

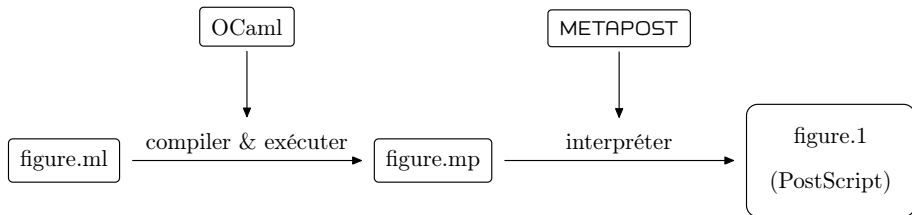
## Une bibliothèque OCaml basée sur METAPOST

- programmer sa figure en OCaml
  - possibilité de développer des extensions de haut niveau
  - se prémunir contre un certain nombre d'erreurs grâce au typage
- bonne intégration avec  $\text{\LaTeX}$  grâce à METAPOST



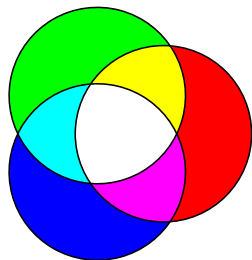
un projet similaire utilisant Haskell : *functional* METAPOST

# Utilisation de MLPOST



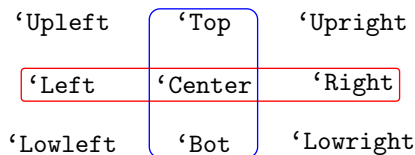
```
open Mlpost
let figure = ...
emit (draw figure)
```

```
beginfig(1)
draw ...;
...
endfig;
```



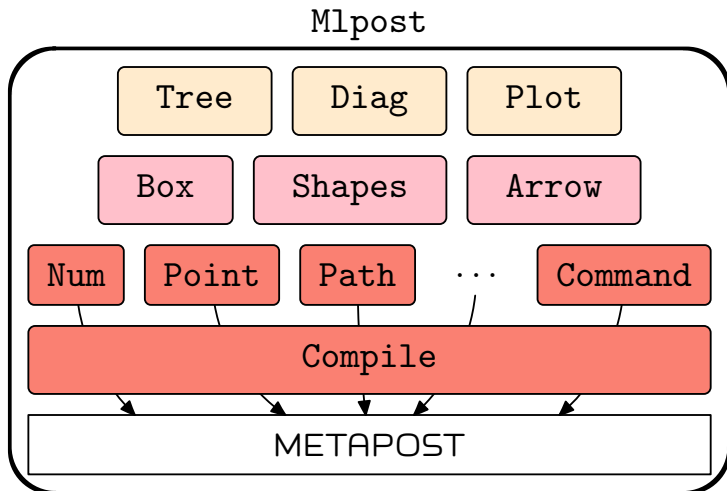
démo !

- **persistance**  
permet de réutiliser les éléments d'une figure
- **arguments optionnels**  
évite la pollution de l'espace de noms  
exemple : `Box.empty`
- une utilisation de **variants polymorphes**



- **minimalité de l'API**  
effort limité de notre part / API cependant complète



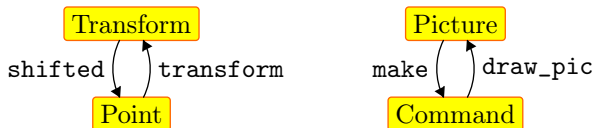


# Problèmes techniques

idéalement, on souhaite interfacer les 7 types de METAPOST dans 7 modules OCaml différents

problème : il y a **circularité**

```
Transform.shifted : Point.t -> Transform.t
Point.transform   : Transform.t -> Point.t -> Point.t
...
Picture.make     : Command.t -> Picture.t
Command.draw_pic : Picture.t -> Command.t
...
```



# Solution

- solution classique : tous les types dans un même module (`types.mli`)
- tous les modules regroupés dans un module `Mlpost` avec `-pack`
- une interface `mlpost.mli` réalisée avec module `rec`

```
mlpost.mli
```

```
module rec Num : sig ... end  
and Point : ...
```

```
Mlpost
```

```
num.ml      color.ml      box.ml  
type t = ... type t = ... type t = ...  
let cm = ...
```

```
types.mli
```

```
type num = ...  
and point = ...  
and box = ...
```

```
graph TD; num_ml[num.ml] --> types_mli[types.mli]; color_ml[color.ml] --> types_mli; box_ml[box.ml] --> types_mli; mlpost_mli[mlpost.mli] --> types_mli;
```

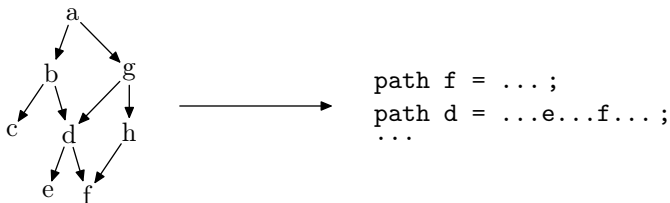
# Traduction vers METAPOST

## limitations de METAPOST

- profondeur de l'arbre de syntaxe
- longueur des lignes
- nombre d'éléments d'un même type

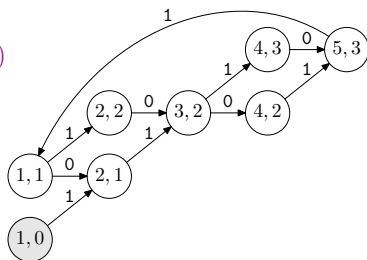
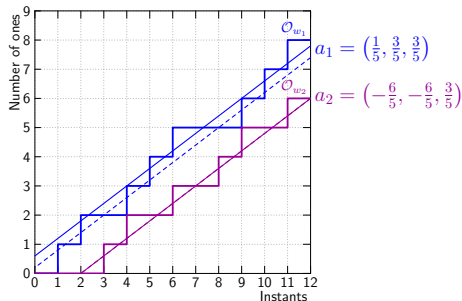
imposent une **compilation** des expressions METAPOST

- *hash-consing* pour factoriser les sous-expressions communes
- introduction de variables intermédiaires

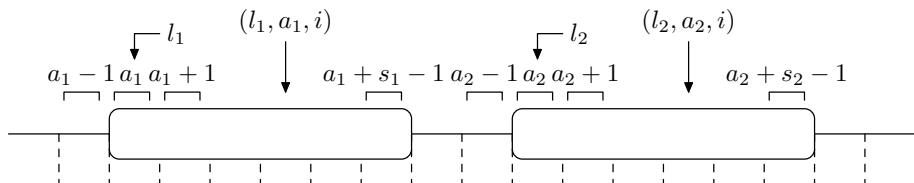


MLPOST est déjà utilisé

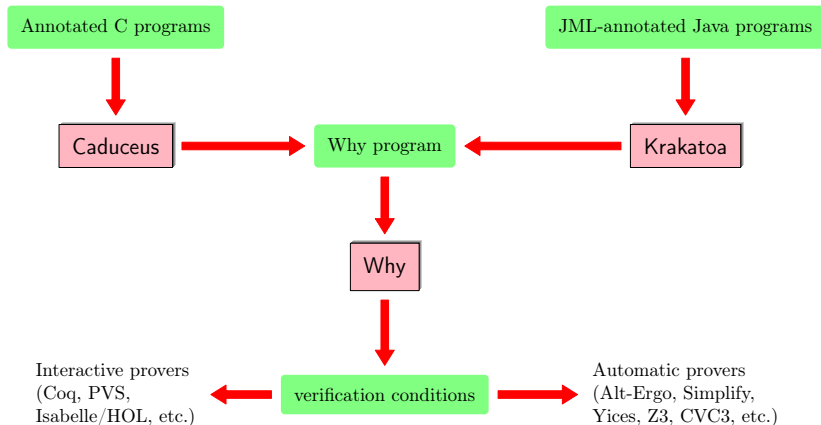
- pour des articles et des présentations (hier ici-même)



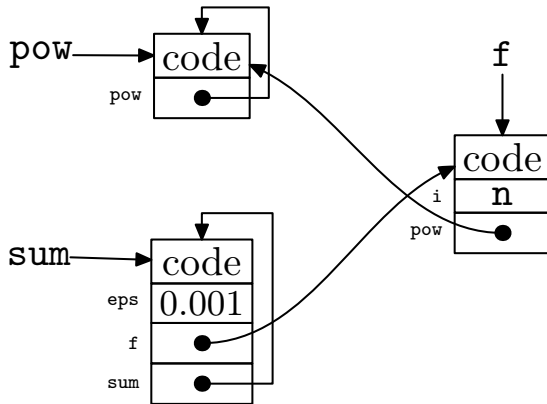
- pour deux thèses (une soutenue, une en cours)



- pour plusieurs posters ProVal

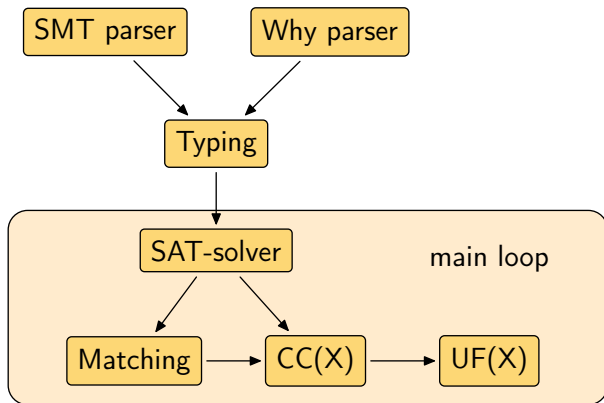


- pour deux cours (U Paris Sud, ENS)





- pour le site web <http://alt-ergo.lri.fr>



- limitations héritées de METAPOST
  - exemple : nombre de nœuds par chemin limité
- limitations de l'interprétation par `latex`
- calculs symboliques
  - difficulté pour écrire des branchements, des boucles, etc.
- résolution d'équations linéaires de METAPOST non interfacée
  - en partie remplacé par le placement relatif des boîtes

**une solution** : reprogrammer l'équivalent de METAPOST

- interface avec  $\text{\LaTeX}$
- calcul des courbes de Bézier

`http://mlpost.lri.fr/`

utilisez-le !

contribuez !