

Faire bonne figure avec MLPOST

R. Bardou¹ & J. Kanig¹ & J.-C. Filliâtre¹ & S. Lescuyer¹

1: ProVal / INRIA Saclay – Île-de-France

91893 Orsay Cedex, France

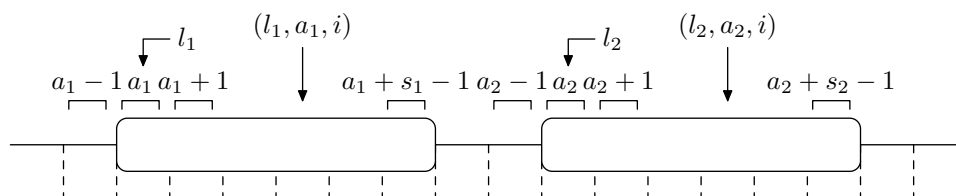
LRI / CNRS – Université Paris Sud

91405 Orsay Cedex, France

`{bardou,kanig,filliatr,lescuier}@lri.fr`

Résumé

1. Introduction



Lors de la rédaction de documents à nature scientifique (articles, cours, livres, etc), il est très souvent nécessaire de réaliser des figures. Ces figures permettent d’agrémenter le texte en illustrant aussi bien les objets dont il est question dans le document que les liens qui existent entre eux et facilitent ainsi leur compréhension. Elles sont donc un composant fondamental au caractère didactique de tels documents, mais leur réalisation est souvent fastidieuse. En particulier, il est souvent nécessaire d’y inclure des éléments mis en forme par \LaTeX (formules, etc), ce que bon nombre de logiciels de dessin ne permettent pas.

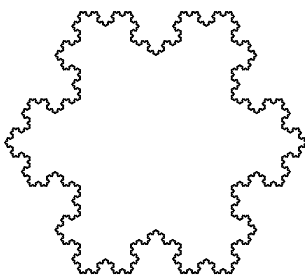
Il existe plusieurs familles d’outils pour réaliser des figures à intégrer dans un document \LaTeX :

- des interfaces graphiques disposant d’une sortie \LaTeX , telles que Dia [?] ou XFIG [?];
- des bibliothèques \LaTeX , telles que PSTricks [?] ou encore Tikz [?];
- des outils externes en ligne de commande, tel METAPOST [?].

Chaque famille a ses avantages et ses inconvénients. Les interfaces graphiques sont les plus accessibles, notamment pour un placement rapide et intuitif des différents éléments de la figure, mais l’intégration de texte mis en forme par \LaTeX est délicate. Dans le cas de XFIG et de Dia, la taille des éléments \LaTeX n’est pas connue lors de l’édition de la figure ; en outre, dans le cas de Dia, l’intégration de \LaTeX dans une figure nécessite de l’exporter sous forme de macros Tikz et d’éditer le résultat.

Les bibliothèques \LaTeX telles que PSTricks ou Tikz offrent l’intégration la plus naturelle avec \LaTeX . En particulier, elles permettent de combiner arbitrairement éléments graphiques et textes \LaTeX . En revanche, elles demandent d’apprendre un certain nombre de macros et de notations et souffrent surtout des défauts inhérents à \LaTeX :

- des erreurs détectées uniquement à l’interprétation, peu claires et parfois mal localisées ;
- un langage *de programmation* peu commode (syntaxe obscure, absence de typage, code difficile à structurer).



```

vardef koch(expr A,B,n) =
  save C; pair C; C = A rotatedaround(1/3[A,B], 120);
  if n>0:
    koch( A,          1/3[A,B], n-1);
    koch( 1/3[A,B], C,      n-1);
    koch( C,          2/3[A,B], n-1);
    koch( 2/3[A,B], B,      n-1);
  else:
    draw A--1/3[A,B]--C--2/3[A,B]--B;
  fi;
enddef;
z0=(4cm,0); z1=z0 rotated 120; z2=z1 rotated 120;
koch( z0, z1, 4 ); koch( z1, z2, 4 ); koch( z2, z0, 4 );

```

FIG. 1 – Exemple de figure METAPOST

Ces inconvénients sont notamment un frein au développement de bibliothèques de haut niveau au dessus de ces langages ainsi qu'à la réutilisation de figures.

METAPOST se présente comme une alternative à ces bibliothèques \LaTeX , en proposant un langage de programmation à part entière spécialisé dans la construction de figures contenant des éléments \LaTeX . Il permet notamment de manipuler symboliquement la taille et la position de ces éléments et de les relier de manière implicite par des équations. En revanche, le langage de METAPOST s'inspire de celui de METAFONT [?] et présente, à l'exception de la syntaxe, les défauts soulevés ci-dessus. La figure 1 donne un exemple de programme/figure réalisé avec METAPOST. Dans un souci d'exhaustivité, nous devons aussi mentionner Asymptote[?], également un langage dédié à la création de figures. Nous n'allons pas rentrer dans les détails de cet outil car nous ne considérons plus les langages dédiés par la suite.

Une alternative séduisante aux solutions précédentes consiste à utiliser un langage de programmation existant. Ainsi l'utilisateur n'a pas à apprendre un langage spécialisé et il bénéficie d'autre part de tous les avantages d'un langage de programmation moderne : erreurs détectées à la compilation, types de données complexes, structuration, etc. Toute la difficulté réside alors dans la manipulation des éléments \LaTeX , notamment la prise en compte de leur taille dans l'élaboration de la figure. Si on considère la famille des langages fonctionnels, on peut citer au moins deux exemples de telle intégration :

- $\text{mlP}\text{\LaTeX}$ [?] est¹ un ensemble de macros \LaTeX permettant d'inclure du code Caml Light arbitraire dans un document \LaTeX . Ce code s'appuie sur une bibliothèque de dessin PostScript et peut faire référence à des éléments \LaTeX , ainsi qu'à leur taille.
- *functional* METAPOST [?] est une bibliothèque Haskell produisant du code METAPOST. C'est une approche légère qui réutilise les capacités graphiques de METAPOST et ne fait que changer le langage de programmation.

C'est cette dernière approche qu'adopte MLPOST.

MLPOST est librement distribué à l'adresse <http://mlpost.lri.fr>. Toutes les figures de cet article ont été faites avec MLPOST, à l'exception des exemples pour METAPOST et Tikz.

Cet article est organisé de la manière suivante. La section 2 ... La section 3 ... Enfin, la section 4 ...

¹À notre connaissance, $\text{mlP}\text{\LaTeX}$ n'est plus distribué.

2. Principes et exemples

2.1. Principes

Boîtes. Les briques de base de MLPOST sont les *boîtes* : une boîte est un moyen d'encapsuler n'importe quel élément de dessin au sein d'un contour, qui peut être effectivement tracé ou non. On peut construire la boîte vide, des boîtes avec du \LaTeX arbitraire, etc. Ces boîtes peuvent ensuite être manipulées : imbrication arbitraire, placement à une position précise, alignement de plusieurs boîtes, flèches reliant plusieurs boîtes entre elles, création de tableaux, etc. Plusieurs boîtes peuvent aussi être regroupées au sein d'une seule afin de pouvoir les déplacer ensemble. L'exemple suivant montre deux boîtes simples, la deuxième déplacée un peu à droite, en utilisant la fonction `shift`.

\LaTeX	○	OCaml [draw (tex "LaTeX") ; draw (shift (Point.pt (cm 1., zero)) (circle (empty ())))]
-----------------	---	--

Placement relatif. Un principe que nous avons suivi lors de la conception de MLPOST est de favoriser un placement relatif des objets plutôt qu'absolu. Ceci permet d'obtenir des figures plus robustes. En effet, imaginons que vous vouliez placer une boîte *A* à *droite* d'une boîte *B*. Une première possibilité serait de spécifier les positions approximativement, par exemple en donnant les abscisses *0cm* pour *A* et *2cm* pour *B*. Cependant, si vous changez d'avis sur le contenu de *A* et que la taille de cette boîte change, *A* risque alors de se superposer à *B*. Il faut alors replacer toutes les boîtes de votre figure manuellement. Pour éviter ça, MLPOST propose diverses méthodes pour placer les boîtes *les unes par rapport aux autres*. On gagne alors du temps lors de la création et lors des modifications de la figure. L'exemple suivant utilise l'alignement horizontal `hbox`, où l'argument optionnel `padding` permet de spécifier l'espacement horizontal entre deux boîtes :

\LaTeX	○	OCaml [draw (hbox padding :(cm 1.) [tex "LaTeX" ; circle (empty ())])]
-----------------	---	--

Nous revenons plus en détail sur les boîtes et leur implémentation dans la section 3.2.

Persistance. Un autre choix que nous avons fait est celui de la persistance [?] : lorsqu'un attribut d'une boîte (positionnement, couleur, etc.) est modifié, on obtient une nouvelle boîte, identique à la première sauf en l'attribut changé. En faisant le choix de structures de données persistantes, nous permettons à l'ancienne boîte, avec ses attributs inchangés, d'être préservée et encore accessible. Ainsi, on peut réutiliser plus facilement une boîte à plusieurs endroits du dessin, avec des attributs différents. Dans l'exemple suivant, on a utilisé trois instances de la même boîte `b`, dont le contour est tracé pour la deuxième.

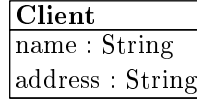
\LaTeX	○	OCaml let b = hbox padding :(cm 1.) [tex "LaTeX" ; circle (empty ())] in [draw (vbox [b ; set _s strokeColor.blackb; b])]
-----------------	---	---

2.2. Exemples

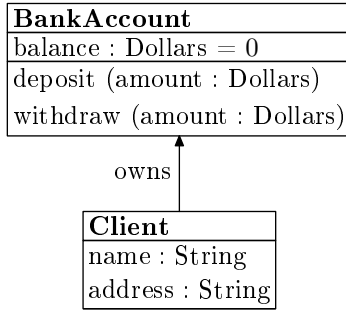
Dans cette section, nous montrons quelques applications immédiates des boîtes de MLPOST.

Représentation de la mémoire. De manière générale, les fonctions `Box.tex` et `Box.box`, ainsi que toutes les autres fonctions de création de boîtes, disposent d'arguments optionnels `dx` et `dy` permettant de spécifier les marges séparant le contenu du contour de la boîte. Elles prennent également un argument `name` permettant de nommer les boîtes pour y accéder plus tard.

Diagrammes de classes. Avec la fonction `Box.vblock` introduite ci-dessus, il est facile de dessiner des diagrammes UML. Supposons que l'on veuille dessiner des schémas de classes tels que :



Pour cela, introduisons une fonction `classblock` qui attend le nom de la classe ainsi que la liste des attributs et des méthodes. OCaml `let classblock name attrlistmethodlist = let vbox = Box.vbox pos : 'Left in Box.vblock pos : 'Left name [tex("bf" ^ name)]; vbox(List.map tex attrlist); vbox(List.map tex methodlist)` Ici, le nom `name` de la classe est utilisé à la fois pour désigner le schéma dans le diagramme (l'argument labelisé `~name` de `Box.vblock`) et comme titre du schéma créé. Les attributs et les méthodes sont alignés verticalement indépendamment, puis on aligne le titre et les deux nouvelles boîtes obtenues en les encadrant. On peut maintenant s'en servir pour dessiner un petit diagramme de classes :



```
OCaml let a = classblock "BankAccount" [ "balance : Dollars = 0" ] [ "deposit (amount : Dollars)"; "withdraw (amount : Dollars)" ] in let b = classblock "Client" [ "name : String"; "address : String" ] [] in let diag = Box.vbox padding : (cm 1.) [a;b] in [ Box.draw diag; box_label_arrow pos : 'Left(Picture.tex "owns")(get "Client" diag)(get "BankAccount" diag)]
```

Ici, on a d'abord créé deux schémas de classe avec la fonction `classblock`. Ces schémas sont ensuite alignés verticalement, et une flèche avec une étiquette est dessinée entre ces deux classes avec `box_label_arrow`. Le code pour cette figure est conceptuellement très simple, ne contient aucun placement absolu et ne dépasse pas les 15 lignes de code.

Automates. L'un des principes de MLPOST est la possibilité d'écrire facilement des bibliothèques. Si on étudie la théorie des langages, on est rapidement amené à dessiner des automates. Illustrons une façon d'utiliser MLPOST dans ce but. On va définir les fonctions suivantes :

- `state` pour créer un état ;
- `initial` pour transformer un état en un état initial ;
- `final` pour transformer un état en un état final ;
- `transition` pour dessiner une transition d'un état à un autre ;
- `loop` pour dessiner une transition d'un état vers lui-même.

On choisit de représenter les états par des boîtes MLPOST. La fonction `state` renvoie simplement une boîte au contour circulaire. OCaml `let state = Box.tex style : Circle stroke : (Some Color.black)` Le paramètre `stroke` permet de spécifier si le contour doit être tracé et, le cas échéant, dans quelle couleur. La fonction `final` consiste à rajouter un deuxième cercle autour d'un état. C'est une fonction qui prend une boîte et qui renvoie une boîte. OCaml `let final = Box.box style : Circle`

On peut déjà placer des états, finaux ou non, et les dessiner. Pour le placement, on utilise les fonctions d'alignement horizontal et vertical `Box.hbox` et `Box.vbox`. On suit donc le principe consistant à placer les objets de façon relative, les uns par rapport aux autres.

Ⓐ Ⓑ OCaml let states = Box.vbox padding : (cm 0.8) [Box.hbox padding : (cm 1.4) [state name : "alpha" "alpha"; state "beta"; final (state "gamma")] in [Box.draw states]

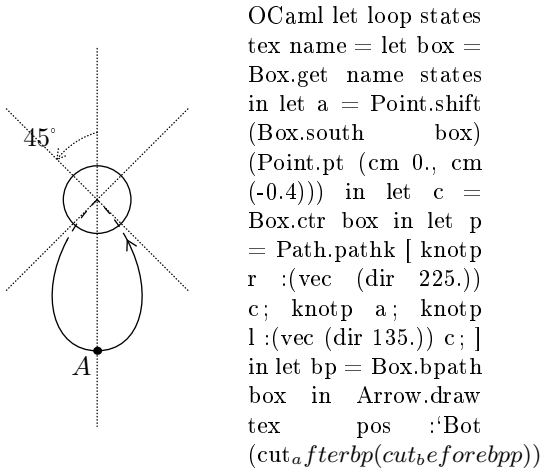
Ⓒ

On note que l'ensemble des états est lui-même une boîte, **states**, contenant les états comme autant de sous-boîtes nommées.

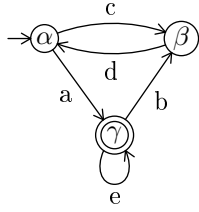
La fonction **initial** appose une flèche entrante à un état. Il s'agit donc d'une fonction qui prend un état q et qui renvoie une commande dessinant une flèche vers q . On pourrait aussi renvoyer une boîte sans contour contenant q et la flèche, ce qui permettrait d'utiliser **initial** de la même façon que **final**. Cependant, la boîte obtenue n'aurait pas la même taille et la même forme que q , ce qui poserait des problèmes pour placer q ou pour dessiner des transitions vers ou à partir de q . OCaml let initial states name = let b = Box.get name states in let p = Box.west b in Arrow.draw (Path.pathp [Point.shift p (Point.pt (cm (-0.3), zero)); p]) La fonction accède à la boîte **b** par son nom **name** dans la boîte **states** et détermine le point d'arrivée de la flèche avec **Box.west**. On pourrait généraliser cette fonction pour spécifier la position de la flèche.

La fonction **transition** dessine une flèche d'un état à un autre. Cette fonction prend deux arguments optionnels **outd** et **ind** pour spécifier, en degrés, la direction sortante et la direction entrante de la flèche. On doit les convertir en vecteurs directeurs pour les passer à **cpath**, qui calcule un chemin allant d'un bord d'une boîte au bord d'une autre boîte. Ce chemin est ensuite donné à la fonction **Arrow.draw** qui trace la flèche en plaçant une étiquette **tex** à la position **pos**. OCaml let transition states tex pos?outd?ind x_name y_name = let x = Box.get x_name states and y = Box.get y_name states in let outd = match outd with None -> None | Some a -> Some (vec (dir a)) in let ind = match ind with None -> None | Some a -> Some (vec (dir a)) in Arrow.draw tex pos (cpath?outd?ind x y)

La fonction **loop** est similaire à la fonction **transition**, mais elle doit calculer un chemin plus complexe. En effet, **cpath** appliqué à deux boîtes identiques renvoie un chemin vide et on ne peut donc pas l'utiliser. À la place, on calcule un point A suffisamment éloigné de la boîte et on trace un chemin qui part du centre, qui passe par A puis qui revient au centre.



On peut maintenant dessiner facilement des automates en utilisant cette bibliothèque.



```

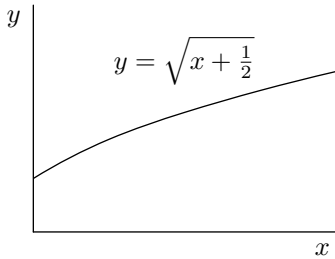
OCaml let automate = let states
= Box.vbox padding :(cm 0.8) [
Box.hbox padding :(cm 1.4) [ state
name : "alpha" "
alpha"; state
name : "beta" "
beta" ]; final
name : "gamma"
(state "
gamma") ] in [
Box.draw states;
transition states "a"
'Lowleft "alpha"
"gamma"; transition
states "b" 'Lowright
"gamma" "beta";
transition states
"c" 'Top outd :25.
ind :335. "alpha"
"beta"; transition
states "d" 'Bot
outd :205. ind :155.
"beta" "alpha"; loop
states "e" "gamma";
initial states "alpha"
]

```

2.3. Exemples utilisant des calculs en OCaml

Cette section illustre l'un des avantages de MLPOST : la capacité de dessiner directement un objet que l'on a calculé/programmé en OCaml.

Graphe de fonction. Un exemple simple de dessin résultant d'un calcul est celui du graphe d'une fonction. MLPOST fournit un module `Plot` à cet effet. La figure suivante montre un exemple basique d'utilisation de cette extension :

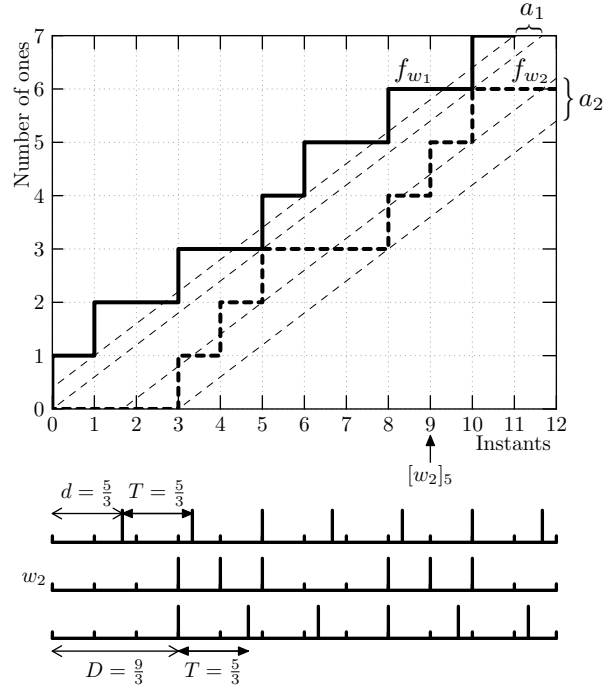


```

OCaml let u = cm 1. in let sk = Plot.mk_skeleton43uwinletlabel =
Picture.tex"y=
sqrtx+
frac12", 'Upleft, 3inletfx = sqrt(floatx + .05)inletgraph =
Plot.draw_func label f skin[graph; Plot.draw_simpla_xes"x"y"sk]

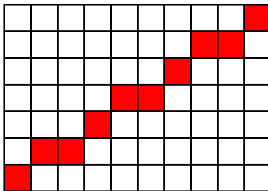
```

La fonction `mk_skeleton` permet de construire un canevas de 4 unités sur 3, qui est l'objet de base de l'extension `Plot`. Il est alors possible de dessiner un graphe de fonction et des axes au sein de ce canevas, comme illustré ci-dessus. L'extension dispose de beaucoup d'options (tracé de la grille, affichage des abscisses et ordonnées, différents types de graphes de fonctions) qui permettent de réaliser des figures plus complexes, telle que celle décrite dans le paragraphe suivant.

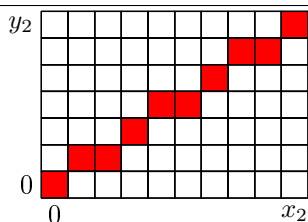


Abstractions d’horloges dans un système synchrone flot-de-données. L’exemple en haut de la page 7, réalisé par Florence Plateau, provient d’un problème réel [?] et illustre un certain nombre des possibilités de l’extension Plot. Ainsi les fonctions illustrées sur cette figure ont été codées comme des fonctions OCaml standard. De plus, la ligne intermédiaire dans la partie située sous le graphe principal, et dénotée par w_2 , représente les discontinuités de la fonction f_{w_2} du graphe principal. Cette ligne est calculée *directement* à partir de la fonction f_{w_2} ; si l’on décide de changer la fonction f_{w_2} , la ligne w_2 sera mise à jour automatiquement. Cela est également vrai pour certaines étiquettes de la figure, comme l’abscisse $[w_2]_5$. Ceci offre une flexibilité très intéressante lors de la phase de développement d’une telle figure.

Bresenham. À titre de dernier exemple, supposons que l’on veuille illustrer l’algorithme de tracé de segment de Bresenham [?], par exemple sur le segment reliant le point $(x_1, y_1) = (0, 0)$ au point $(x_2, y_2) = (9, 6)$. Pour cela, on commence par stocker le résultat de l’algorithme dans un tableau `bresenham_data`, tel que `bresenham_data(x)` donne l’ordonnée du point d’abscisse x . OCaml `let x2 = 9 let y2 = 6 let bresenham_data = leta = Array.create(x2 + 1)0in...remplissagedutableauaavecl'algorithmedeBresenham...a` On peut alors réaliser la figure très facilement, à l’aide de la fonction `Box.gridi` fournie par MLPOST, qui construit une matrice de boîtes alignées à partir d’une largeur, d’une hauteur et d’une fonction construisant la boîte (i, j) (d’une manière analogue à `Array.create_matrix`).



```
OCaml let width = bp 6. and height = bp 6. in let g = Box.gridi
(x2+1) (y2+1) (fun i j -> let fill = if bresenham_data.(i) = y2 -
j then SomeColor.red else None in Box.rect?fill(Box.empty width height())) in [Box.draw
```



3. Architecture logicielle

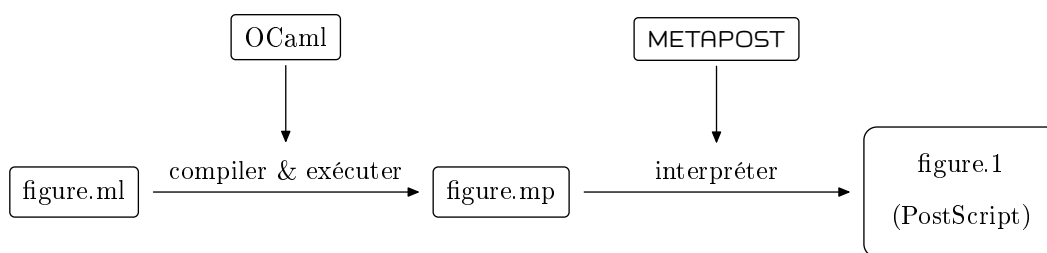


FIG. 2 – Architecture de MLPOST

La figure 2 montre le fonctionnement de MLPOST. Tout d’abord, MLPOST est un outil de génération de fichier METAPOST sous forme de bibliothèque OCaml. À l’aide de cette bibliothèque, l’utilisateur écrit un programme qui, à l’exécution, construit un arbre de syntaxe abstraite METAPOST. Cet arbre est imprimé dans un fichier `figure.mp` qui est lu par METAPOST pour générer un ou plusieurs fichiers PostScript². Pour inclure ces figures dans un document L^AT_EX, il suffit d’utiliser la commande `\includegraphics` du package `graphicx`.

Il est important de noter que le fichier METAPOST de sortie n’est pas obtenu par *compilation* du code source OCaml, mais par une *exécution* du programme qui construit un arbre de syntaxe abstraite METAPOST. Cette méthode a l’inconvénient qu’une boucle ou itération dans le programme de départ sera traduite par une suite de commandes obtenues par le déroulement de la boucle, et non par une construction de boucle du langage cible. Ceci étant dit, dans notre cas, le coût supplémentaire est faible, puisque METAPOST déroule également les boucles dans les fichiers PostScript de sortie.

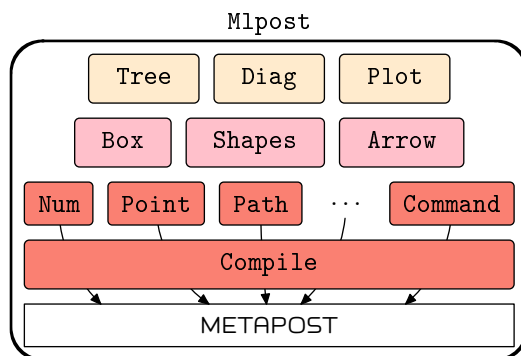


FIG. 3 – Architecture de MLPOST

²Ces fichiers n’ont pas le suffixe `.ps` car il leur manque l’en-tête.

L'architecture générale de MLPOST est schématisée en figure 3. Au niveau le plus bas se trouvent les interfaces correspondant aux types primitifs de METAPOST. Ce sont ces objets qui sont *in fine* traduits en du code METAPOST, et nous les décrivons de manière plus détaillée dans la section 3.1. La couche intermédiaire contient des éléments que nous estimons être de bas niveau mais qui ne sont pas présents dans METAPOST : ils sont propres à MLPOST et ont été construits à partir de la couche inférieure. Les sections 3.2 et 3.3 reviennent plus en détail sur deux de ces modules, respectivement `Box` et `Arrow`. Enfin, on trouve au plus haut niveau des modules tels que `Tree` ou `Plot` présentés dans la section précédente. L'intégralité des modules de MLPOST est empaquetée dans un module `Mlpost`, afin de ne pas polluer l'espace de noms d'OCaml.

3.1. Types primitifs de METAPOST

La couche de bas niveau de MLPOST est une interface fidèle à METAPOST. Elle comporte tous les types de base de METAPOST :

Le type numérique (module `Num`) représente des longueurs. En première approximation, ce type pourrait être assimilé au type `float` d'OCaml, mais certaines valeurs, telle que la taille d'un élément \LaTeX , ne sont connues qu'à l'interprétation du fichier METAPOST. La plupart des calculs sont donc effectués de manière symbolique et le type `Num.t` doit donc être abstrait.

Le type point (module `Point`) représente des points dans l'espace à deux dimensions. Pour les mêmes raisons que les numériques, les points ne sont pas simplement des paires de flottants, mais doivent être représentés de manière symbolique. Le type `Point.t` est également utilisé pour représenter les vecteurs.

Les chemins (module `Path`) sont des lignes représentées par des courbes de Bézier. Ils sont à la base de tout dessin METAPOST. Toutes les possibilités de construction de chemin dans METAPOST ont été interfacées. On peut dessiner des lignes droites ou des lignes courbes en précisant les points de contrôle, la tension de la courbe, etc.

Les transformations (module `Transform`) permettent d'appliquer une transformation linéaire à un objet quelconque. Il est ainsi possible de déplacer des objets, les redimensionner, les faire pivoter ou encore combiner toutes ces transformations.

Les plumes (module `Pen`) permettent de choisir l'épaisseur et la forme du stylo utilisé pour dessiner les chemins.

Les figures (module `Picture`) permettent de rassembler plusieurs éléments graphiques en un seul, qu'il s'agisse d'éléments \LaTeX ou de commandes de dessin arbitraires. Le type `Picture.t` permet de traiter une figure arbitrairement complexe comme un objet de base que l'on peut copier, transformer, etc. Le module `Picture` permet également de découper une figure à l'aide d'une surface décrite par un chemin clos (*clipping*).

Les autres types METAPOST (chaînes de caractères, booléens, couleurs) sont facilement représentés en OCaml. L'interface de MLPOST contient aussi un module `Command` qui définit le type des commandes METAPOST : commandes de dessin, de remplissage, itérations, séquences, etc.

Dépendances circulaires. La réalisation de ces modules de bas niveau présente quelques difficultés. Premièrement, la plupart des modules présentés sont *a priori* mutuellement récursifs : par exemple, les transformations s'appliquent à tous les autres objets, donc chaque module contient une fonction OCaml `val transform : Transform.t -> t -> t` où le type `t` représente le type principal du module en question. D'un autre côté, les transformations sont elles-mêmes construites à l'aide de numériques et de points : OCaml `val shifted : Point.t -> t` où `t` est le type des transformations. Des dépendances circulaires existent aussi entre types et sont aggravées par la représentation symbolique des objets

(par exemple, les projections `xpart` et `ypart` du module `Point` doivent retourner des numériques et non des flottants).

Nous souhaitons réaliser ces différents modules dans des fichiers différents mais OCaml ne permet pas de dépendances circulaires entre des fichiers. Notre solution consiste à définir tous les *types* dans un seul fichier `types.mli`. Chaque module fait maintenant référence à ce fichier. Par exemple, dans le fichier `path.ml`, qui fournit l'implémentation du module `Path`, on trouvera OCaml type `t = Types.path`. La dépendance circulaire entre les modules est ainsi cassée de manière très classique. En revanche, on souhaite cacher l'existence du module `Types` pour les deux raisons suivantes :

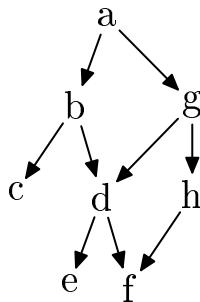
- la clarté des messages d'erreur ;
- la clarté de la documentation générée par `ocamldoc`.

On souhaite donc rétablir la circularité entre les modules au sein de l'interface du module `Mlpost`. Pour cela, on écrit un unique fichier `mlpost.mli` qui contient les définitions des signatures des modules à exporter : OCaml module rec Num : sig type t ... end and Point : sig type t val xpart : t -> Num.t ... end and Path : sig type t ... end ... Toutes ces signatures sont déclarées de manière mutuellement récursive. La signature de `Point` peut ainsi faire référence à `Path` et inversement. En revanche, les implémentations de ces modules sont contenues dans des fichiers indépendants `num.ml`, `point.ml`, etc., qui sont compilés avec l'option `-for-pack Mlpost`. Cet agencement est également très pratique pour la documentation de l'API : c'est uniquement le fichier `mlpost.mli` qui sert d'entrée à l'outil `ocamldoc`.

Hashconsing et traduction vers METAPOST. Le choix d'une représentation symbolique de la plupart des objets impose des efforts supplémentaires pour minimiser l'utilisation de la mémoire et la taille des fichiers METAPOST de sortie. En effet, l'arbre de syntaxe abstraite (AST) contient beaucoup de nœuds identiques mais construits de manière différente, qui prennent donc inutilement de la place aussi bien en mémoire que dans le fichier METAPOST généré. C'est d'autant plus gênant que METAPOST devra lire ce fichier et passera donc davantage de temps sur des calculs répétés.

Pour y remédier, nous utilisons la technique du *hash-consing* [?] appliquée à l'arbre de syntaxe abstraite. Cette technique permet de partager des valeurs structurellement égales. Elle utilise une table de hachage globale qui stocke toutes les valeurs déjà créées. Avant de créer un nouvel objet, on regarde dans cette table si un objet structurellement égal existe déjà. Pour que le calcul de la valeur de hachage ainsi que la comparaison des objets soit efficace, chaque (sous-)terme vient avec sa valeur de hachage. Ainsi, on n'obtient pas seulement le partage de tous les sous-termes communs, mais également un test d'égalité très efficace, utile pour toute structure de données contenant ces valeurs : il suffit de comparer physiquement ces objets.

Cette technique diminue l'utilisation de la mémoire, mais ne change rien *a priori* à la taille des fichiers générés. La structure hash-consée *réalise* le partage, mais elle ne sait pas quels sont les nœuds effectivement utilisés au moins deux fois. Pour cela, il suffit de traverser la structure, en comptant les occurrences de chaque nœud. Quand on visite un nœud déjà rencontré, on ne considère pas ses sous-nœuds. Il est néanmoins possible de rencontrer une nouvelle fois un sous-nœud d'une structure, comme le montre l'exemple ci-dessous :



Dans cette configuration, le nœud f est réellement utilisé deux fois, alors que le nœud e n'est utilisé qu'une seule fois, par le nœud d , même si celui-ci est utilisé deux fois à son tour. Autrement dit, on comptabilise pour chaque nœud le nombre de flèches incidentes.

Après cette analyse, la génération du fichier METAPOST devient très simple : il suffit de traverser de nouveau l'arbre de syntaxe abstraite et, quand on visite un nœud qui est utilisé au moins deux fois, on construit une définition METAPOST pour cet objet. Il faut néanmoins prendre en compte les particularités syntaxiques de METAPOST telles que la précedence inhabituelle des opérateurs arithmétiques et la restriction de l'application de certaines constructions à des variables. De cette façon, on arrive à avoir du code METAPOST relativement petit³, malgré la délégation des calculs à METAPOST.

3.2. Boîtes

- principe récursif : boîte = picture ou ensemble de boîtes + contour + fond
- boîtes primitives - réunion, alignements

Application : arbres.

3.3. Flèches

METAPOST ne propose qu'un seul type de flèche. Une flèche METAPOST suit un chemin arbitraire mais son tracé est limité aux différents styles de trait (plume et pointillés) et la tête de flèche est toujours la même :




Avec MLPOST, l'utilisateur peut créer ses propres catégories de flèches à l'aide du module **Arrow**. Celui-ci propose deux types :

- Le type **head** décrit comment dessiner une tête de flèche. Les éléments de type **head** sont des fonctions prenant en argument la position et la direction de la tête de flèche et renvoyant une commande dessinant la tête de flèche.
- Le type abstrait **kind** décrit une catégorie de flèche. Une catégorie décrit les différents éléments dans le dessin d'une flèche, les têtes de flèche pouvant en réalité être placées n'importe où le long de la flèche. Pour construire une nouvelle catégorie, on part de la catégorie vide et on ajoute des traits et des têtes.

La fonction **draw** permet de dessiner une flèche d'une catégorie donnée en suivant un chemin donné. Les flèches sont alors dessinées en utilisant les primitives de METAPOST. Ceci a l'inconvénient d'utiliser plus de ressources mais permet d'imaginer de nombreuses catégories de flèches.

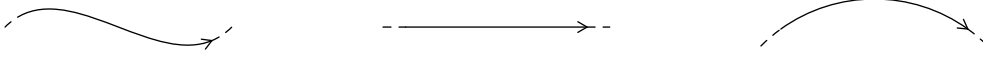
On peut en particulier retrouver les flèches de METAPOST. On part d'un corps vide et on lui ajoute un trait normal sur toute la longueur. On ajoute enfin une tête triangulaire remplie, et on obtient :



```
OCaml let kind =
  Arrow.addhhead head :
  Arrow.headttriangleffull(Arrow.addilineArrow.empty)in[Arrow.draw kind(...path...)]
```

³En l'absence de boucles **for** et de macros dans le code METAPOST, nous avons observé, sans avoir fait de tests très exhaustifs, une taille du code généré du même ordre de grandeur que du code METAPOST écrit à la main.

La section 2.2 contient une figure décrivant le fonctionnement de la fonction `loop`. Pour les besoins de cette figure, on a créé un type de flèche spécial, composé d'un début et d'une fin en pointillés et avec une tête placée différemment :



Le code permettant d'obtenir cette catégorie de flèche est le suivant : `OCaml let kind = Arrow.add_elt point : 0.9(Arrow.add_line dashed : Dash.evenly to_point : 0.1(Arrow.add_line dashed : Dash.evenly from_point : 0.9(Arrow.add_line from_point : 0.1 to_point : 0.9 Arrow.empty)))`

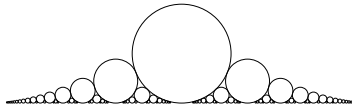
4. Conclusion

Nous avons présenté MLPOST, une bibliothèque OCaml au dessus de METAPOST. Nous espérons avoir convaincu le lecteur des avantages que la présentation sous forme de bibliothèque apporte : familiarité avec le langage pour les programmeurs OCaml, typage fort, des constructions de programmation de haut niveau, des dessins résultants de calculs arbitraires, etc. MLPOST fournit volontairement un nombre de primitives restreint, car l'utilisateur peut aisément construire des extensions au dessus de MLPOST. En cela, MLPOST diffère de bibliothèques L^AT_EX telles que Tikz ou PSTricks, où de très nombreuses fonctionnalités sont fournies mais où il est très difficile d'en ajouter pour qui ne maîtrise pas T_EX.

L'une des forces de MLPOST est de proposer un style déclaratif, là où la majorité des bibliothèques graphiques propose un style impératif. Ceci permet en particulier un *partage* immédiat de sous-éléments à travers une ou plusieurs figures. Une autre force de MLPOST est le typage statique directement hérité d'OCaml. On évite ainsi l'immense majorité des erreurs à l'exécution de METAPOST, souvent cryptiques. Il reste néanmoins les erreurs éventuellement contenues dans les extraits de L^AT_EX ou les erreurs de nature géométrique telles que le remplissage d'un chemin en forme de 8. Nous pourrions envisager d'utiliser des types OCaml plus précis, par exemple pour distinguer les chemins clos et non clos ou encore les points et les vecteurs.

Il reste une fonctionnalité intéressante de METAPOST qui n'est pas interfacée dans MLPOST : la résolution d'équations linéaires. Il y a deux raisons à cela. D'une part, les équations servent souvent au placement implicite, et MLPOST fournit une alternative sous la forme de fonctions d'alignement de boîtes. D'autre part, la résolution d'équations de METAPOST procède de manière impérative et il n'est pas simple de l'intégrer dans le contexte déclaratif qui est le nôtre.

Enfin, il est important de noter que MLPOST n'est pas lié à METAPOST de manière intrinsèque. On pourrait facilement ajouter une sortie Tikz, ou même directement une sortie PostScript à condition d'utiliser une technique similaire à celle de METAPOST pour l'inclusion de L^AT_EX.



Remerciements

Les auteurs tiennent à remercier Florence Plateau, Yannick Moy et Claude Marché pour leur contribution à MLPOST et Sylvie Boldo pour la suggestion du titre de l'article.

