Faire bonne figure avec Mlpost

R. Bardou[1] & J.-C. Filliâtre[1] & J. Kanig[1] & S. Lescuyer[1]

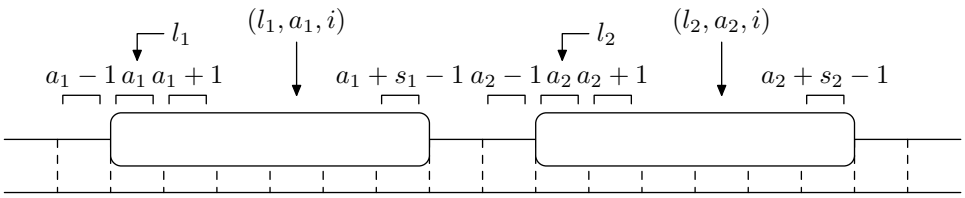
1: ProVal / INRIA Saclay – Île-de-France
91893 Orsay Cedex, France
LRI / CNRS – Université Paris Sud
91405 Orsay Cedex, France
{bardou,filliatr,kanig,lescuyer}@lri.fr

Résumé

Cet article présente Mlpost, une bibliothèque OCaml de dessin scientifique. Elle s'appuie sur METAPOST, qui permet notamment d'inclure des fragments LaTeX dans les figures. OCaml offre une alternative séduisante aux langages de macros LaTeX, aux langages spécialisés ou même aux outils graphiques. En particulier, l'utilisateur de Mlpost bénéficie de toute l'expressivité d'OCaml et de son typage statique. Enfin Mlpost propose un style déclaratif qui diffère de celui, souvent impératif, des outils existants.

1. Introduction

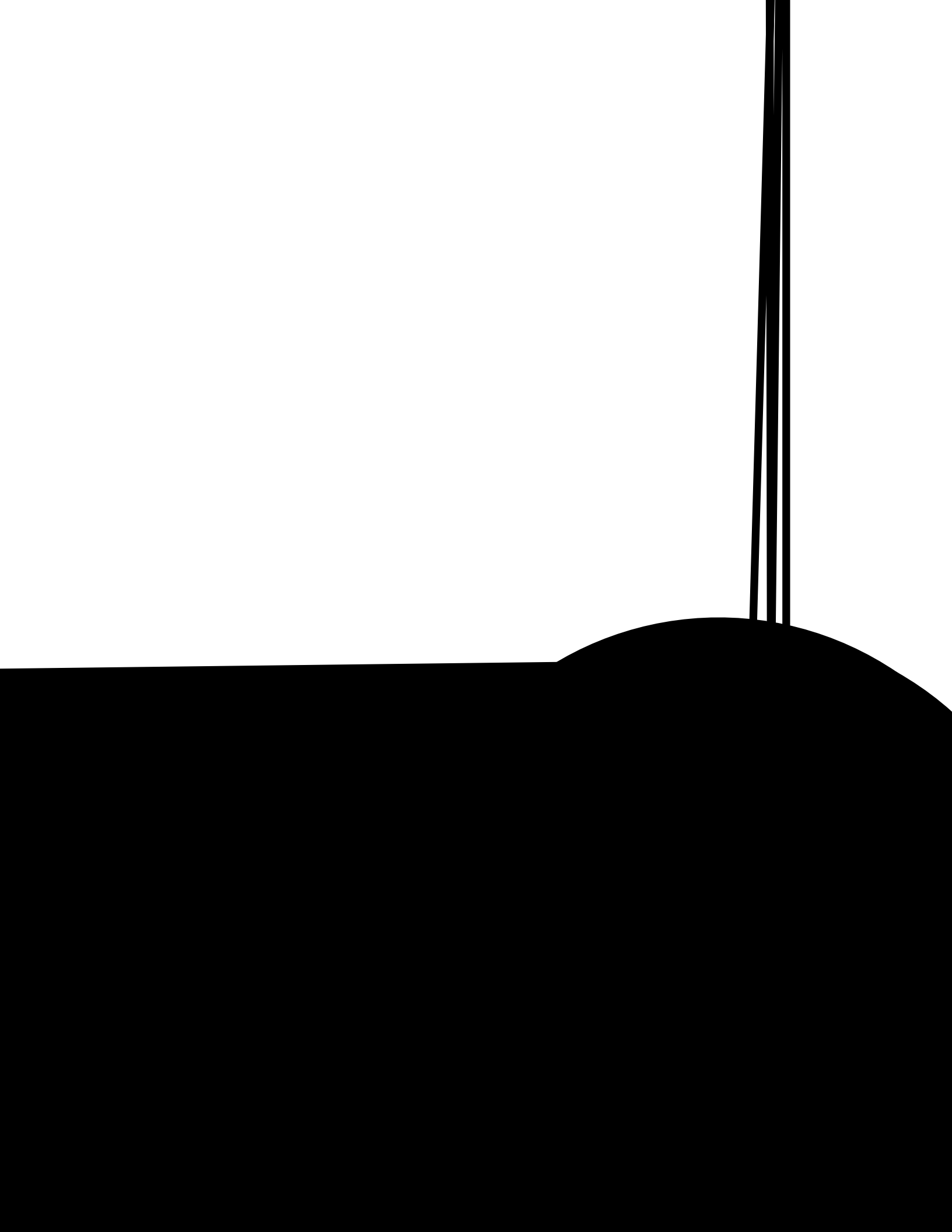
Lors de la rédaction de documents à nature scientifique (articles, cours, livres, etc.), il est très souvent nécessaire de réaliser des figures. Ces figures permettent d'agrémenter le texte en illustrant aussi bien les objets dont il est question dans le document que les liens qui existent entre eux 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. Ainsi, on peut souhaiter réaliser un schéma tel que celui-ci



en attachant de l'importance au fait que des expressions comme $a_1 + s_1 - 1$ apparaissent exactement comme dans le corps du document. 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 [12] ou Xfig [14] ;
– des bibliothèques LaTeX, telles que PSTricks [8] ou encore TikZ [15] ;
– des outils externes en ligne de commande, tel que METAPOST [10].

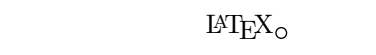
Chaque famille a ses avantages et ses inconvénients. Les interfaces graphiques sont les plus accessibles, notamment pour un placement rapide et intuitif des différents éléments de la figure, mais l'intégration de texte mis en forme par LaTeX est délicate. Dans le cas de Xfig et de Dia, la taille des 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.



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 étant déplacée un centimètre vers la droite en utilisant la fonction `shift`.



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

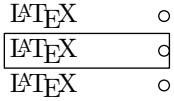
Une figure MLpost est simplement une liste de commandes de dessin. Ci-dessus, elle est réduite à deux occurrences de `Box.draw`, la commande qui dessine une boîte.

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 l'on veuille 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 0 cm pour A et 2 cm pour B. Cependant, si on change 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 la 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 :

```
[ Box.draw (Box.hbox ~padding:(cm 1.)
                     [Box.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 [7] : 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.

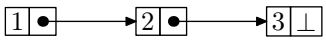


```
let b =
  Box.hbox ~padding:(cm 1.) [Box.tex "\\LaTeX"; circle (empty ())] in
[ Box.draw (Box.vbox [b; set_stroke Color.black b; b]) ]
```

2.2. Exemples

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

Représentation de la mémoire. Un besoin récurrent lorsque l'on enseigne l'algorithmique ou les concepts liés à un langage de programmation consiste à illustrer la structure des données en mémoire par des schémas de la forme



Deux éléments sont nécessaires : le dessin des blocs d'une part et le dessin des pointeurs d'autre part. Pour réaliser les blocs, on utilise la fonction `Box.hblock` qui aligne des boîtes horizontalement, leur donne une hauteur commune et trace leur contour. Voici un exemple :

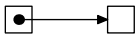

```
let b = Box.hblock ~pos:‘Bot [Box.tex "a"; Box.tex "b"; Box.tex "C"] in
[ Box.draw b ]
```

À l'aide de `Box.hblock`, on peut facilement écrire une fonction `cons` qui construit un bloc de taille 2, dont le premier élément contient un texte LaTeX arbitraire `hd` et le second est soit vide, soit le symbole $\bot$, selon la valeur du booléen `tl` :

```
let cons hd tl =
  let p1 = Box.tex ~name:"hd" hd in
  let p2 = Box.tex ~name:"tl" (if tl then "" else "\\ensuremath{\\bot}") in
  Box.hblock [p1; p2]
```

L'argument optionnel `~name` permet de nommer les sous-boîtes, de manière à pouvoir y accéder facilement par la suite.

Écrivons maintenant une fonction `pointer_arrow` pour matérialiser les pointeurs. Il s'agit de tracer une flèche entre deux boîtes données `a` et `b`. Pour cela, on commence par construire un chemin `p` reliant les centres de `a` et `b`, que l'on tronque à l'endroit où il intersecte le bord de la boîte `b` (avec la fonction `Path.cut_after`). Puis on trace l'origine de la flèche à l'aide d'un chemin réduit à un point (le centre de `a`) et la flèche proprement dite à l'aide du chemin `p`.



```
let pointer_arrow a b =
  let p = pathp [Box.ctr a; Box.ctr b] in
  let p = Path.cut_after (Box.bpath b) p in
  let pen = Pen.scale (bp 4.) Pen.circle in
  Command.draw ~pen (pathp [Box.ctr a]) ++ draw_arrow p
```

On utilise ici le symbole infixe `++` qui permet de concaténer des commandes de dessin. On utilise d'autre part `bp`, qui est une unité propre à METAPOST, proche du point PostScript.

Nous avons maintenant tous les éléments nécessaires pour écrire une fonction `draw_list` qui prend en argument une liste OCaml de fragments LaTeX et réalise l'illustration correspondante. On commence par construire les différents blocs constituant la liste :

```
let draw_list l =
  let rec make = function
    | [] → []
    | [x] → [cons x false]
    | x :: l → cons x true :: make l in
```

Puis on aligne ces blocs avec `Box.hbox`, en insérant de l'espace horizontal avec l'option `padding` :

```
  let l = hbox ~padding:(bp 30.) (make l) in
```

Pour dessiner les pointeurs, il suffit de parcourir la liste des boîtes (qui ont été placées) et d'utiliser la fonction `pointer_arrow` précédente sur chaque paire de boîtes consécutives :

```
  let rec arrows = function
    | [] | [_] → nop
    | b1 :: (b2 :: _ as l) →
        pointer_arrow (Box.get "tl" b1) (Box.get "hd" b2) ++ arrows l in
```

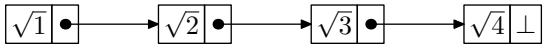
On utilise ici la fonction `Box.get` qui permet de récupérer une sous-boîte par son nom. On accède ainsi aux boîtes nommées respectivement `"hd"` et `"tl"` qui ont été créées par la fonction `cons` puis encapsulées dans d'autres boîtes par les fonctions d'alignement. Enfin, on dessine les boîtes avec `Box.draw`, puis les flèches avec la fonction `arrows` :

```
[ Box.draw l; arrows (Array.to_list (Box.elts l)) ]
```

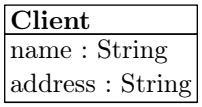
On peut tester avec

```
draw_list (List.map (fun n → Printf.sprintf "$\\sqrt{%d}$" n) [1;2;3;4])
```

qui donne bien le résultat attendu :



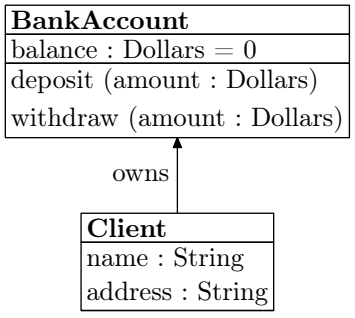
Diagrammes de classes. Avec la fonction `Box.vblock`, analogue pour l'alignement vertical de la fonction `Box.hblock` introduite ci-dessus, il est facile de dessiner des diagrammes UML. Supposons que l'on veuille dessiner des schémas de classes tels que :



Pour cela, introduisons une fonction `classblock` qui attend le nom de la classe ainsi que la liste des attributs et des méthodes.

```
let classblock name attr_list method_list =
  let vbox = Box.vbox ~pos:‘Left in
  Box.vblock ~pos:‘Left ~name
  [ tex ("{\\bf " ^ name ^ "}");
    vbox (List.map tex attr_list); vbox (List.map tex method_list) ]
```

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[2]. On peut maintenant s'en servir pour dessiner un petit diagramme de classes :




```
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.

[2]Notez qu'au contraire de `hbox` et `vbox`, les fonctions `hblock` et `vblock` tracent par défaut le contour de leurs sous-boîtes.

Automates. La théorie des langages est un domaine où l'on a rapidement besoin de dessiner des automates. Illustrons une façon d'utiliser MLpost dans ce but. Nous allons définir les fonctions suivantes :

- `state` pour créer un état ;
- `final` pour transformer un état en un état final ;
- `initial` pour dessiner une flèche entrante sur un état initial ;
- `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` est juste une spécialisation de la fonction `Box.tex` à un contour circulaire, définie par l'application partielle suivante :

```
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. De manière similaire, la fonction `final` est une spécialisation de la fonction `Box.box` dont le rôle consiste à rajouter un cercle autour d'une boîte :

```
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.

```
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 ]
```

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 le nom d'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`.

```
let initial (states : Box.t) (name : string) : Command.t =
  let q = Box.get name states in
  let p = Box.west q in
  Arrow.draw (Path.pathp [Point.shift p (Point.pt (cm (-0.3), zero)); p])
```

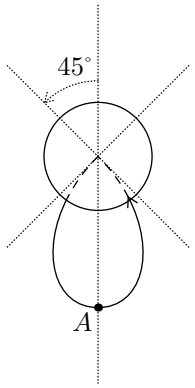
La fonction accède à la boîte `q` par son nom `name` dans la boîte `states` et détermine le point d'arrivée de la flèche avec `Box.west`. On pourrait généraliser cette fonction pour spécifier la position de la flèche.

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[3] à `cpath`, qui calcule un chemin allant du 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`.

[3]Nous utilisons ici une fonctionnalité d'OCaml qui permet d'accéder à des arguments optionnels sans spécifier leur valeur par défaut, sous la forme d'un type `option`. Nous passons ensuite directement ces valeurs de type `option`, en tant qu'arguments optionnels, à la fonction `cpath` par la syntaxe `cpath ?outd ?ind x y`.

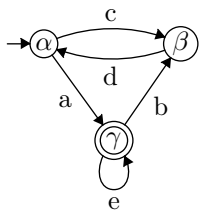
```
let transition states tex pos ?outd ?ind x_name y_name =
  let x = Box.get x_name states and y = Box.get y_name states in
  let outd = match outd with None → None | Some a → Some (vec (dir a)) in
  let ind = match ind with None → None | Some a → Some (vec (dir a)) in
  Arrow.draw ~tex ~pos (cpath ?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 utilise au passage la fonction `knotp` qui permet de spécifier un point avec une tangente, et `pathk` qui transforme une liste de tels points en un chemin.




```
let loop states tex name =
  let box = Box.get name states in
  let a = Point.shift (Box.south box) (Point.pt (cm 0., cm (-0.4))) in
  let c = Box.ctr box in
  let p = Path.pathk [
    knotp ~r:(vec (dir 225.)) c;
    knotp a;
    knotp ~l:(vec (dir 135.)) c;
  ] in
  let bp = Box.bpath box in
  Arrow.draw ~tex ~pos:‘Bot (cut_after bp (cut_before bp p))
```

Ici encore, on pourrait généraliser cette fonction pour spécifier la position de la flèche. On peut maintenant dessiner facilement des automates en utilisant cette bibliothèque.

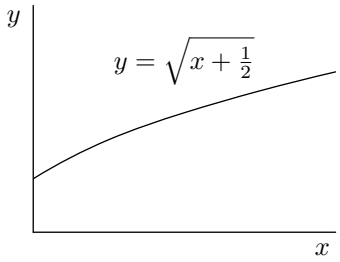



```
let automate =
  let states = ... in
  [ Box.draw states;
    transition states "a" ‘Lowleft "alpha" "gamma";
    transition states "b" ‘Lowright "gamma" "beta";
    transition states "c" ‘Top ~outd:25. ~ind:335. "alpha" "beta";
    transition states "d" ‘Bot ~outd:205. ~ind:155. "beta" "alpha";
    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.

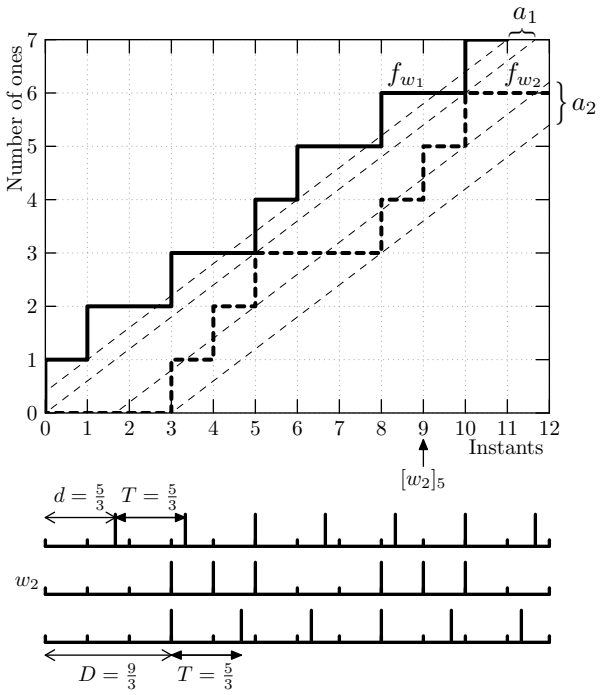
Graphe de fonction. Un exemple simple de dessin résultant d'un calcul est celui du graphe d'une fonction. MLpost fournit un module `Plot` à cet effet. La figure suivante montre un exemple basique d'utilisation de cette extension :




```
let u = cm 1. in
let sk = Plot.mk_skeleton 4 3 u u in
let label = Picture.tex "$y=\\sqrt{x+\\frac{1}{2}}$",
            ‘Upleft, 3 in
let f x = sqrt (float x +. 0.5) in
let graph = Plot.draw_func ~label f sk in
[ graph; Plot.draw_simple_axes "$x$" "$y$" sk ]
```

La fonction `mk_skeleton` permet de construire un cannevas 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 cannevas, 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 ci-dessous, réalisé par Florence Plateau, provient d'un problème réel [5] 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 flexiblité très intéressante lors de la phase de développement d'une telle figure.

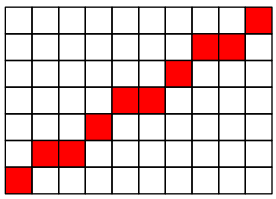


Bresenham. À titre de dernier exemple, supposons que l'on veuille illustrer l'algorithme de tracé de segment de Bresenham [3], par exemple sur le segment reliant le point $(x_1, y_1) = (0, 0)$ au point $(x_2, y_2) = (9, 6)$. Pour cela, on commence par stocker le résultat de l'algorithme dans un tableau `bresenham_data`, tel que `bresenham_data.(`x`)` donne l'ordonnée du point d'abscisse x.

```
let x2 = 9 and y2 = 6
let bresenham_data = Array.create (x2+1) 0
let () = (* remplissage du tableau a avec l'algorithme de Bresenham *) ...
```

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

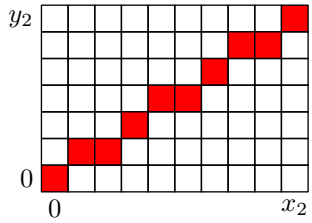
construisant la boîte (i, j), d'une manière analogue à `Array.create_matrix`.



```
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 Some Color.red else None in
    Box.rect ?fill (Box.empty ~width ~height ())) in
[ Box.draw g ]
```

Les boîtes sont des boîtes vides de 6 points de côté, créées avec `Box.empty`. Pour les boîtes correspondant à des points dessinés par l'algorithme de Bresenham, on indique que la boîte doit être remplie en rouge, à l'aide de l'argument optionnel `fill`.

Pour parachever la figure, on va ajouter des étiquettes indiquant les coordonnées des deux extrémités du segment. Pour placer une étiquette à côté de la case (i, j), on récupère la boîte correspondante à l'aide de `Box.nth`, puis on récupère un point particulier de cette boîte (par exemple le point au milieu en bas avec `Box.south`), puis enfin on trace l'étiquette avec `Command.label`.




```
let bresenham =
  let width = bp 6. and height = bp 6. in
  let g = ... in
  let get i j = Box.nth i (Box.nth (y2-j) g) in
  let label pos s point i j =
    Command.label ~pos (Picture.tex s) (point (get i j)) in
  [ Box.draw g;
    label ‘Bot "0" Box.south 0 0; label ‘Bot "$x_2$" Box.south x2 0;
    label ‘Left "0" Box.west 0 0; label ‘Left "$y_2$" Box.west 0 y2 ]
```

3. Architecture logicielle

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 PostScript[4]. L'inclusion de ces figures dans un document LaTeX se fait simplement en utilisant la commande `\includegraphics` du package graphicx. Pour compiler le document LaTeX avec `pdflatex`, il suffit de changer l'extension des figures générées, ce qu'une option de l'outil MLpost permet de faire facilement.

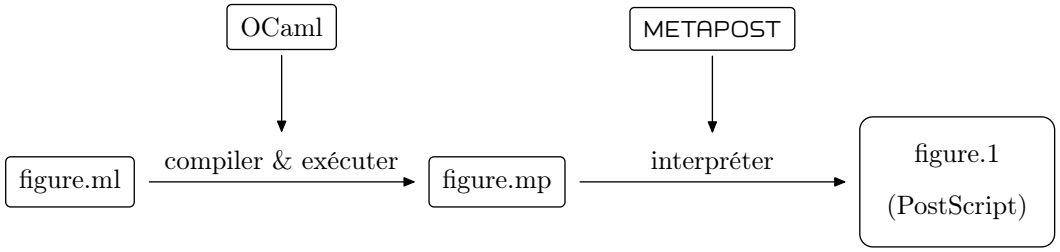


Fig. 2 – Architecture de MLpost

[4]Ces fichiers n'ont pas le suffixe `.ps` car il leur manque l'en-tête.

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.

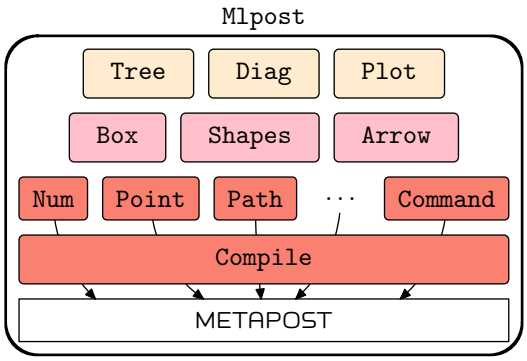


Fig. 3 – Architecture de Mlpost

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 le module `Plot` présenté 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 directement 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

```
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 :

```
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

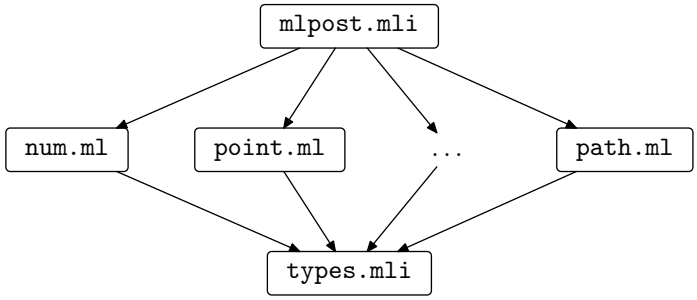
```
type t = Types.path
```

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 :

```
module rec Num : sig
  type t
  ...
end
and Point : sig
  type t
  val xpart : t → Num.t
  ...
end
and Path : sig
  type t
  ...
end
...
```

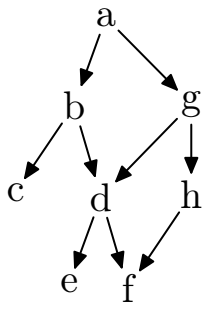


Toutes ces signatures sont déclarées de manière mutuellement récursive. La signature de `Point` peut ainsi faire référence à `Path` et inversement. On notera que les types tels que `Point.t` sont maintenant abstraits dans l'interface, rendant invisible l'existence du module `Types`. Par ailleurs, les 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 en outre très pratique pour la documentation de l'API : c'est uniquement le fichier `mlpost.mli` qui sert d'entrée à l'outil `ocamldoc`.

Hash-consing 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* [6] 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 soit efficace, chaque (sous-)terme vient avec sa valeur de hachage. Le partage réalisé est maximal, ce qui permet de substituer l'égalité physique (`==`) à l'égalité structurelle (`=`).

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, on réalise un simple parcours en profondeur de la structure, en comptant les occurrences de chaque nœud. 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 précédence 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 petit[5], malgré la délégation des calculs à METAPOST.

[5]En 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 celle du code METAPOST écrit à la main.

3.2. Boîtes

Comme nous l'avons illustré déjà maintes fois, les boîtes de MLpost peuvent être réduites à de simples objets LaTeX ou bien être constituées d'un ensemble d'autres boîtes. Le type des boîtes est donc un type récursif de la forme suivante :

```
type t =
  { name : string option;
    width : Num.t;  height : Num.t; pen : Pen.t option; ...
    desc : desc; }

and desc =
  | Emp
  | Pic of Picture.t
  | Grp of t array × t Smap.t
```

Chaque boîte est éventuellement nommée (champ `name`), possède un certain nombre d'attributs (position, taille, couleur, bordure, remplissage, etc.) et sa nature est donnée par le champ `desc`. Ce dernier indique s'il s'agit d'une boîte vide, d'une image ou bien d'une boîte composite. Dans ce dernier cas, les sous-boîtes sont contenues dans un tableau, qui est accompagné d'une table (réalisée par le module `Smap`) permettant un accès plus rapide à une sous-boîte par son nom.

Le dessin d'une boîte est immédiat. On trace d'une part son contour et d'autre part son contenu. Ce dernier est soit une boîte atomique directement dessinée à l'aide de `Command.draw_pic`, soit une boîte composite dont le dessin est tout simplement obtenu en dessinant récursivement chaque sous-boîte.

Pour réaliser les diverses fonctions de placement, on commence par écrire une fonction de translation d'une boîte par un vecteur donné :

```
Box.shift : Point.t → Box.t → Box.t
```

Cette fonction est naturellement récursive sur la structure de la boîte. Il est important de noter que cette fonction renvoie une *nouvelle* boîte, sans altérer son argument (les boîtes sont persistantes). Une fois cette fonction donnée, il est aisé de réaliser les fonctions `Box.hbox`, `Box.hblock`, etc.

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 :


```
let kind =
  Arrow.add_head ~head:Arrow.head_triangle_full
    (Arrow.add_line Arrow.empty) in
[ Arrow.draw ~kind (...path...) ]
```

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 :

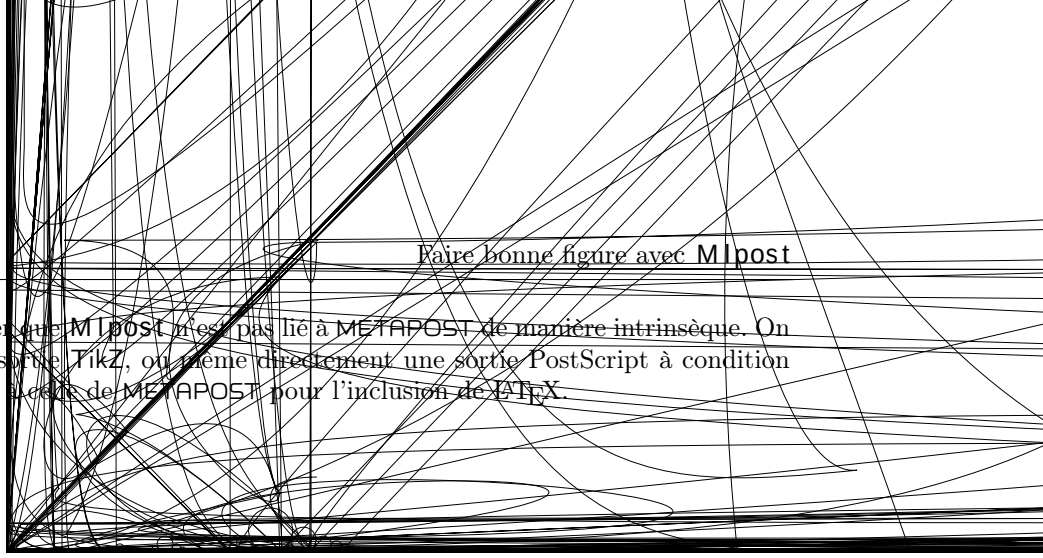
```
let kind =
  Arrow.add_belt ~point:0.9
    (Arrow.add_line ~dashed:Dash.evenly ~to_point:0.1
       (Arrow.add_line ~dashed:Dash.evenly ~from_point:0.9
          (Arrow.add_line ~from_point:0.1 ~to_point:0.9
             Arrow.empty)))
```

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 restreint de primitives, car l'utilisateur peut aisément construire des extensions au dessus de MLpost. En cela, MLpost diffère de bibliothèques LaTeX 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 TeX.

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 dans 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 LaTeX 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. Ceci étant dit, il serait intéressant d'explorer des méthodes de placement plus automatiques que celles que nous proposons, par exemple inspirées de la manière dont TeX mets en page lignes, pages et paragraphes.



Enfin, il est important de noter que MIpost 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 LaTeX.