CAHIERS

GUTenberg

ATTRIBUTS ET COULEURS

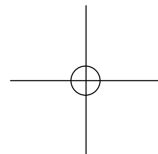
Manuel PÉGOURIÉ-GONNARD

Cahiers GUTenberg, nº 54-55 (2010), p. 57-85.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_2010___54-55_57_0>

© Association GUTenberg, 2010, tous droits réservés.

L'accès aux articles des *Cahiers GUTenberg*
(http://cahiers.gutenberg.eu.org/),
implique l'accord avec les conditions générales
d'utilisation (http://cahiers.gutenberg.eu.org/legal.html).
Toute utilisation commerciale ou impression systématique
est constitutive d'une infraction pénale. Toute copie ou impression
de ce fichier doit contenir la présente mention de copyright.



ATTRIBUTS ET COULEURS

Manuel Pégourié-Gonnard

Résumé. — Nous explorons un mécanisme d'extension de LuaTeX, les *attributs*, et leur utilisation possible dans la mise en œuvre des couleurs. Après avoir présenté le concept des attributs et les interfaces TeX et Lua permettant de les manipuler, nous rappelons les grands lignes de la mise en œuvre classique des couleurs sous LaTeX ainsi que ses problèmes connus. Nous étudions ensuite en détails une solution basée sur les attributs qui résoud ces problèmes. Celle-ci illustre certains principes généraux de l'utilisation des attributs, dont les applications ne se limitent évidemment pas aux couleurs.

Abstract. — This article presents a new tool provided by LuaTeX to extend TeX: attributes, and how they can be used to implement colors. First, we study the general concept of attributes and the TeX and Lua interfaces. Then, we recall the main points of the classical color implementation in LaTeX and its well-known limitations. Finally, a solution to these problems, using attributes, is presented, and demonstrates a few general principles in the use of attributes, which are obviously not limited to colors.

Note. — Merci à Heiko Oberdiek de m'avoir permis d'utiliser le code de son paquet luacolor pour illustrer cet article.

1. Présentation des attributs

1.1. Concept

Pour composer un texte, chaque caractère doit être pris dans une certaine police. À moins de s'astreindre à la plus grande sobriété, on voudra utiliser différentes polices à différents emplacements du texte. C'est pourquoi TeX possède une notion de « police courante » à utiliser pour composer les caractères à un instant donné.

Les changements de police sont soumis aux règles de portée de TeX et la police courante est mémorisée pour chaque caractère individuellement au moment de l'ajout de ce caractère à la liste courante, et non

pas au moment de son utilisation effective dans le document. Cette différence est essentielle sur l'exemple suivant [1]

```
\setbox0=\hbox{L} \it L\unhbox0 L
```

qui produit « *L*L*L* » : la police courante a été mémorisée avec chaque caractère de la boîte et n'est plus modifiée ensuite.

Pour composer du texte en plusieurs couleurs, il serait souhaitable de disposer d'une notion similaire de « couleur courante » qui obéirait aux mêmes règles de portée et serait mémorisée individuellement avec chaque caractère. Ceci n'a pas été prévu lors de la conception de TeX et n'avait, jusqu'à LuaTeX, pas été ajouté ultérieurement : les mécanismes de gestion de la couleur de pdfTeX ne répondent pas à ces deux critères dont nous montrerons par la suite en quoi ils sont essentiels.

En fait, conformément à ses habitudes, LuaTeX n'ajoute pas de mécanisme spécifique pour la couleur, mais propose au contraire un concept flexible généralisant la notion de police courante : les attributs. L'idée générale est simple : chaque caractère (ou plus généralement nœud d'une liste) possède à un niveau conceptuel plusieurs caractéristiques : sa police, sa couleur, son éventuel soulignement, etc. La police est particulière car elle est prise en compte directement par le moteur (par exemple quand il a besoin de connaître les dimensions d'un glyphe). Les autres caractéristiques sont mémorisées dans des *attributs* sous une forme qu'il appartient au programmeur de choisir et d'exploiter par la suite, par exemple au moyen de fonctions de rappel (*callbacks*) en Lua.

Un grand nombre d'attributs sont disponibles ; chacun d'entre eux possède à tout moment une « valeur courante » qui peut être un nombre entier ou l'état spécial « désactivé ». Les attributs sont repérés par leur numéro, par exemple on pourrait décider que l'attribut numéro 0 est désactivé la plupart du temps, prend la valeur 0 pour les passages à souligner, 1 pour les passages en soulignement double, etc. On a alors fixé une représentation concrète sous forme d'attributs de la caractéristique abstraite « niveau de soulignement », représentation qui devra être

1. Les exemples qui suivent sont basés sur Plain TeX par souci de simplicité et de neutralité ; ils sont également testés avec LaTeX et la classe article.

utilisée par une fonction Lua appelée au moment opportun afin d'être suivie d'effet.

Il importe de noter que le moteur est parfaitement indifférent à la présence des attributs et à leurs valeurs (c'est la différence majeure entre police et attributs) : son rôle se limite à garder à jour la valeur courante de chaque attribut en respectant les règles de portée (modifications locales à l'intérieur d'un groupe TEX) et à attacher aux nœuds qu'il crée la valeur courante de tous les attributs sauf [2] ceux qui ont actuellement la valeur spéciale signifiant « désactivé ».

Notons que l'ordre de création des nœuds n'est pas toujours très intuitif : par exemple un nœud de ligature sera créé bien après les nœuds des caractères qui en sont l'origine. LuaTEX incorpore un certain nombre de règles particulières pour ces cas où il est souhaitable qu'un nœud mémorise d'autres valeurs des attributs que celles en vigueur au moment de sa création. Le manuel [6, § 2.5.1] donne quelques détails à ce sujet.

1.2. Interlude : exploration de boîtes en Lua

TEX propose une primitive pour explorer le contenu des boîtes : `\showbox`. Elle nous permet en particulier d'illustrer le fait que la police est mémorisée avec chaque caractère. Par exemple, avec

```
\showboxdepth=1 \showboxbreadth=10
\tt \setbox0=\hbox{a\bf b{\it c}d}
\showbox0
```

on obtient le résultat suivant dans le log :

```
> \box0=
\hbox(6.94444+0.0)x22.62762, direction TLT
.\tentt a
.\tenbf b
.\tenit c
.\tenbf d
```

Comme LuaTEX n'étend pas cette primitive pour montrer également les attributs, il va falloir développer nos propres outils pour les examiner.

2. Ceci est une optimisation purement technique visant à limiter la taille des nœuds en mémoire.

Pour commencer, nous allons faire une version simplifiée mais personnalisable de `\showbox` :

```
\def\ShowBox#1{\directlua{
  for n in node.traverse(tex.getbox(\number#1).list) do
    local t = node.type(n.id)
    texio.write_nl(t)
    if (t == 'glyph') then
      texio.write(' ' .. unicode.utf8.char(n.char))
    end
    write_node_details(n)
  end}}
```

Cette macro récupère la boîte avec `tex.getbox` sous la forme d'un nœud de type `hbox` [6, § 8.1.2.1] qui possède un champ `list` pointant vers le contenu de la boîte. La fonction `node.traverse` [6, § 4.10.1.24] est un itérateur qui permet de parcourir les noeuds au moyen d'un `for` générique [4, § 4.3.5].

On commence par récupérer le type du nœud en tant que chaîne de caractères et l'écrire au début d'une nouvelle ligne. On s'arrête là sauf s'il s'agit d'un caractère (les glyphes et les caractères partagent le même type de nœud, [6, § 8.1.2.12]). Dans ce cas, on affiche le caractère, dont le point de code Unicode est disponible dans le champ `char` du nœud. Enfin, pour afficher les autres détails, on fait appel à une fonction spécialisée : `\directlua`.

```
\directlua{
  function write_node_details(n)
    texio.write(' font=' .. n.font)
  end}
```

C'est cette fonction que nous modifierons par la suite pour afficher les attributs plutôt que la police.

Faisons un premier test :

```
\tt \setbox0=\hbox{a\bf b{\it c}d}
\ShowBox0
```

Nous obtenons sur le terminal et dans le fichier log :

```
glyph a font=30
glyph b font=24
glyph c font=37
glyph d font=24
```

Nous sommes maintenant outillés pour examiner à la main le contenu de boîtes simples. Retournons donc à l'étude des attributs.

1.3. Interface TeX primitive

Les primitives TeX pour manipuler les attributs sont simples et très similaires à celles utilisées pour les compteurs (registres `\count`). En fait, les attributs fonctionnent exactement comme les compteurs à l'exception — essentielle — du fait suivant : la valeur courante de tous les attributs qui ne sont pas « désactivés » est attachée à tous les nœuds [3] à leur création, sauf si cette valeur est $-2^{31}+1$, soit `-"7FFFFFFF` en notation TeX-hexadécimale, qui est la plus petite valeur légale. (Avant la version 0.37 de LuaTeX, toutes les valeurs négatives étaient considérées comme équivalentes à cette valeur spéciale, de sorte que les nombres négatifs étaient inutilisables comme valeurs d'attributs.)

Les deux primitives en question sont les suivantes [4] [6, § 2.5] :

`\attribute` ⟨*nombre 16 bits*⟩ ⟨*égal*⟩ ⟨*nombre 32 bits*⟩
`\attributedef` ⟨*séquence de contrôle*⟩ ⟨*égal*⟩ ⟨*nombre 16 bits*⟩

La première sert à assigner une valeur (comprise entre $-2^{31}+1$ et 2^{31}) à un attribut désigné par son numéro, compris entre 0 et $2^{16}-1$. Il est possible de définir une séquence de contrôle comme étant équivalente à `\attribute X` pour une certaine valeur de `X` grâce à `\attributedef`.

Par ailleurs, il est possible d'utiliser un attribut partout où TeX attend un nombre ou un registre de type `\count`. En particulier, on dispose de `\the` pour obtenir la valeur d'un attribut et de `\showthe` pour l'afficher dans le log :

```
\attribute0=42
\showthe\attribute0
\attributedef\attrzero=0
\attrzero=2010
\showthe\attribute0
```

3. Sauf peut-être certains nœuds de types particuliers pour lesquels la notion d'attribut n'a pas de sens.
4. Sous LaTeX, leurs noms sont `\luatexattribute` et `\luatexattributedef`, voir l'article « Un guide pour LuaLaTeX » dans ce numéro des *Cahiers*.

```
\ifnum\attrzero>2000
  \advance\attrzero by -2000
\fi
\showthe\attrzero
```

Sur cet exemple, on voit dans le log que l'attribut numéro zéro prend successivement les valeurs 42, 2010 et 10.

Bien qu'elles soient légales, les opérations arithmétiques sur les attributs en TEX ne sont pas très intéressantes : pour ça il y a déjà les compteurs et surtout Lua. L'usage typique des attributs sera plutôt de donner des noms (et des rôles) à certains attributs dans le préambule, puis de changer leurs valeurs au cours du document. L'utilisation de ces valeurs se fera la plupart du temps en Lua.

1.4. Interfaces Lua

La partie la plus intéressante de l'interface Lua de manipulation des attributs est décrite à la section 4.9.2 du manuel [6]. La fonction essentielle est `node.has_attribute` qui permet de vérifier si un nœud possède un attribut donné et, si oui, de récupérer sa valeur. La syntaxe est :

```
v = node.has_attribute(n, a)
```

où `n` est un nœud, `a` l'attribut qui nous intéresse, et `v` devient soit `nil` si cet attribut était désactivé à la création du nœud, soit sa valeur pour ce nœud.

L'utilisation la plus classique d'un attribut consiste à lui affecter des valeurs depuis des macros TEX puis à examiner ces valeurs depuis des *callbacks* en Lua avec `node.has_attribute`. Il s'agit d'une communication unidirectionelle de TEX vers Lua.

Il arrive aussi parfois qu'on veuille modifier les attributs de certains nœuds en Lua *après* leur création, le but étant alors que la nouvelle valeur soit utilisée par une fonction Lua ultérieure (rappelons que LuaTEX lui-même ne tient pas compte des attributs, c'est à nous qu'il incombe de les examiner et d'implémenter les actions correspondantes). C'est possible avec les fonction `node.set_attribute` et `node.unset_attribute`.

Avant d'examiner le reste de l'interface Lua, reprenons notre exemple de décorticage de boîte, avec la nouvelle fonction suivante :

```
function write_node_details(n)
  for attr = 0, 1 do
      texio.write(' attr' .. attr .. '=')
```

```
        texio.write(tostring(node.has_attribute(n, attr)))
    end
  end
```

Modifions notre exemple pour manipuler des attributs plutôt que des polices :

```
\attribute0=0
\setbox0=\hbox{a\attribute1=1
  b{\attribute0=-"7FFFFFFF c}d}
\ShowBox0
```

Nous pouvons alors lire dans le log :

```
glyph a attr0=0 attr1=nil
glyph b attr0=0 attr1=1
glyph c attr0=nil attr1=1
glyph d attr0=0 attr1=1
```

Dans cet exemple, on s'intéresse spécifiquement à deux attributs, numérotés 0 et 1. C'est le cas le plus fréquent : reprenons notre hypothèse que l'attribut 0 est utilisé pour représenter le niveau de soulignement, il est évident que les fonctions Lua gérant le soulignement ne s'intéresseront qu'à cet attribut précis.

Cependant, on pourrait vouloir regarder tous les attributs d'un nœud donné, par exemple pour construire une version améliorée de `\showbox`. Il faut alors recourir à l'interface de bas niveau : les attributs sont attachés à un nœud sous la forme d'une liste de pseudo-nœuds dans le champ `attr` du nœud. On peut la parcourir avec `node.traverse`, mais il faut prendre garde que le premier pseudo-nœud de cette liste est spécial et aller chercher directement le suivant dans son champ `next` :

```
function write_node_details(n)
  for pseudo_node in node.traverse(n.attr.next) do
      texio.write(' attr', pseudo_node.number)
      texio.write('=', pseudo_node.value)
  end
end
```

Testons notre dernière version de cette fonction :

```
\attribute314=0
\setbox0=\hbox{a\attribute10000=1
  b{\attribute314=-"7FFFFFFF c}d}
\ShowBox0
```

Le résultat dans le log est le suivant :

```
glyph a attr0=0 attr314=0
glyph b attr0=0 attr314=0 attr10000=1
glyph c attr0=0 attr10000=1
glyph d attr0=0 attr314=0 attr10000=1
```

Enfin, par souci d'exhaustivité, signalons qu'il est possible de consulter et même de modifier la valeur courante des attributs au moyen du tableau `tex.attribute` et des fonctions `tex.getattribute` et `tex.setattribute` depuis Lua [6, § 4.13.4], mais ceci présente probablement peu d'intérêt : on utilisera plus généralement l'interface TEX pour cela, alors qu'en Lua on s'intéressera plus volontiers aux attributs d'un nœud donné plutôt qu'à leurs valeurs courantes.

1.5. Extensions pertinentes

Dans les exemples ci-dessus, nous avons toujours manipulé les numéros d'attribut directement et par exemple proposé d'utiliser l'attribut 0 pour coder le niveau de soulignement. Il va de soi que ceci n'est pas réaliste dans le contexte d'un paquet devant être utilisé en conjonction avec d'autres : comment décider à qui appartient l'attribut 0 ?

Le problème existe depuis longtemps pour les autres ressources que TEX met à disposition du programmeur : registres de compteurs, de longueurs, de boîtes, etc. La solution est connue depuis tout aussi longtemps, et consiste à utiliser une macro qui alloue les numéros de registre, le programmeur n'utilisant alors plus que le nom symbolique rendu équivalent à ce numéro. Plain et LATEX fournissent de telles macros (`\newcount` et associés) pour les ressources de TEX82.

Pour les nouvelles ressources de LuaTEX, il faut recourir à un paquet définissant une telle macro : c'est ce que font actuellement luatex et luatexbase-attr, qui sont également utilisables sous Plain. (Les deux sont actuellement en conflit puisqu'ils entendent gérer les mêmes ressources ; ils devraient être fusionnés prochainement, ce qui résoudra le problème.)

La commande `\newluatexattribute` permet d'allouer un attribut et de lui donner un nom, par exemple

```
\newluatexattribute\soulattr
```

réserve un attribut pour le soulignement. Peu importe le numéro effectif de l'attribut, on y accède depuis TeX avec la commande `\soulattr` :

```
\soulattr=42
\setluatexattribute\soulattr{6 * 7}
```

La première ligne utilise la syntaxe primitive, la deuxième utilise une syntaxe plus LaTeXienne, fournie par luatexbase-attr, qui accepte de plus en argument n'importe quelle expression interprétable par `\numexpr`. Finalement, la commande `\unsetluatexattribute` permet de désactiver un attribut en lui assignant une valeur spéciale adaptée à la version de LuaTeX utilisée.

Malheureusement, l'interface Lua de manipulation des attributs demande qu'on les désigne par leur numéro. Certes, on peut accéder à la valeur courante d'un attribut de façon symbolique dans la table `tex.attribute`, par exemple après l'allocation précédente, `tex.attribute.soulattr` est valide et on peut utiliser la chaîne `"soulattr"` en argument de `tex.setattribute` et `tex.getattribute`. Mais ce n'est pas (encore) vrai pour les fonction les plus importantes comme `node.has_attribute`.

Heureusement, luatexbase-attr permet d'accéder aux numéros d'attributs par leur nom dans la table `luatexbase.attributes`. Le code Lua pourra donc procéder ainsi :

```
-- au début, mais après \newluatexattribute\soulattr
local soulattr = luatexbase.attributes.soulattr)
-- plus loin
local v = node.has_attribute(n, soulattr)
```

De cette façon, le programmeur n'a jamais besoin de connaître directement le numéro effectif de l'attribut utilisé.

2. La mise en œuvre classique des couleurs et ses problèmes

Avertissement. Cette section et la suivante concernent uniquement LaTeX, seul cadre dans lequel l'auteur connaît le code existant. Nous nous concentrons dans cette section sur color, dont l'usage est supposé connu (se reporter à [3] le cas échéant). Nous commençons par rappeler son fonctionnement en se limitant aux aspects pertinents ; le lecteur désirant approfondir le sujet consultera avec profit [2] et [8] (qui ne concerne pas que color mais aussi graphics).

2.1. Le principe

Bien qu'on vante avec justesse la modularité de LuaTEX, il serait injuste d'oublier que TEX avait déjà prévu son propre mécanisme d'extension : les nœuds de type *bidule*, ou *whatsit* en anglais [5]. D'ailleurs Knuth a choisi de mettre en œuvre la primitive `\write` (entre autres) comme une extension afin de servir d'exemple (tex.web 1340). Il existe différents sous-types de bidules associés à différentes extensions (nouvelles primitives).

Dans les premiers jours de TEX, l'unique format de sortie disponible était le DVI, qui possédait son propre mécanisme d'extension, les instructions `special`. Là aussi, différents sous-types de `special` sont utilisés dans différents buts ; les choses sont compliquées par le fait que les `special` sont destinés à différents interpréteurs de DVI (dvips, dvipdfm, etc.) qui ne reconnaissent pas le même ensemble d'instructions. Du point de vue de TEX cependant les choses restent simples : les instructions `special` sont insérées litérallement dans le DVI sans chercher à les interpréter ; elle correspondent (sans surprise) à un sous-type particulier de nœud bidule.

Plus tard, Hàn Thế Thành a développé pdfTEX, une variante de TEX permettant de générer directement du PDF et d'accéder depuis TEX à de nombreuses fonctionnalités spécifiques de ce format. Là encore, la plupart des nouvelles primitives liées au format de sortie PDF sont implémentées par le biais de nœuds bidule. Par exemple, la primitive `\pdfliteral` est très proche dans l'esprit de la primitive `\special` : elle crée un nœud bidule de sous-type approprié contenant du matériel brut à insérer dans le PDF sans que TEX se soucie du sens de ce matériel.

Ainsi, le code gérant la couleur sous LATEX doit composer avec une certain variété de formats de sortie : PDF direct avec pdfTEX, et essentiellement une variante de DVI par interpréteur de DVI.

Pour corser encore un peu les choses, notons qu'il existe plusieurs façon de décrire les couleurs : monochromie (nuances d'une seule couleur, en général le noir), synthèse additive RGB (trois couleurs primaires : rouge, vert, bleu), quadrichromie CMYK (cyan, magenta, jaune, noir),

5. Il semble assez difficile de trouver une traduction consensuelle pour ce terme. Dans la version française du *TEXBook*, Jean-Côme Charpentier utilise « élément extraordinaire ». *[N.D.L.R.]*

couleurs nommées ou indexées (par exemple l'édition anglaise du LaTeX companion qui dispose d'exactement deux couleurs : le noir et un bleu donné, correspondant physiquement aux deux encres utilisée sans mélange à l'impression). Ces modèles de couleurs sont diversement supportés par les formats de sortie, et il est souhaitable de présenter à l'utilisateur un certain nombre de modèles de couleur de base utilisables avec la plupart des formats de sortie, quitte à opérer une conversion en interne.

Pour gérer ces problèmes, color introduit la notion de *pilote* (*driver*) : chaque format de sortie est géré par un pilote qui est chargé de fournir les macros spécifiques à chaque format, seules les macros indépendantes du format étant définies dans `color.sty`. Les macros relevant du pilote sont de plusieurs types :

`\color@`⟨*modèle*⟩ (par exemple `\color@rgb`) prend en argument une description de couleur dans un certain modèle en syntaxe LaTeX (par exemple trois nombres entre 0 et 1 séparés par des virgules) et la transforment en une description de la même couleur (dans le même modèle ou non) adaptée au format de sortie (exemples ci-dessous). Le résultat est stocké dans une macro au choix, en général `\current@color`.

`\set@color` produit l'instruction adaptée au format de sortie pour activer la couleur stockée dans `\current@color` (en général par un appel à une des macros précédentes). Elle programme également l'exécution de `\reset@color` pour la fin du groupe courant au moyen d'un `\aftergroup`.

`\reset@color` produit une instruction adaptée au format de sortie pour annuler l'effet de la commande `\set@color` précédente.

`\set@page@color` produit une instruction adaptée au format de sortie pour faire de la couleur contenue dans `\current@color` la couleur de fond jusqu'à ordre contraire. LuaTeX n'offrant aucune innovation intéressante à ce niveau, nous ne parlerons pas plus de cette commande ici.

`\define@color@named` produit une instruction appropriée pour définir une couleur nommée dans le fichier de sortie (DVI). Attention, ceci n'a rien à voir avec la commande `\definecolor` qui définit un nom de couleur au niveau du fichier d'entrée TeX. Cette instruction est spécifique au modèle de couleur *named* ; nous n'en parlerons pas plus ici car les points qui nous intéressent sont en fait indépendants du modèle de couleur.

Tout ceci peut paraître bien abstrait à première vue ; nous allons donc détailler les nœuds produits par `A\textcolor{red}{B}C` avec différents pilotes et signaler lesquelles des macros ci-dessus sont responsables des différents éléments. Commençons par dvips (le format de sortie est celui de `\showbox`).

```
.\T1/lmr/m/n/10 A
.\special{color push rgb 1 0 0}
.\T1/lmr/m/n/10 B
.\special{color pop}
.\T1/lmr/m/n/10 C
```

Signalons que la couleur `red` est définie indépendamment du pilote comme étant la couleur `1,0,0` dans le modèle `rgb`. C'est un appel à la macro `\color@rgb` qui a converti ceci en une description adaptée au format de sortie : `rgb 1 0 0`. Ici, il s'agit d'une pure conversion syntaxique, mais des changement plus sophistiqués auraient pu intervenir. La macro `\set@color` a produit la première ligne `special` ci-dessus en utilisant cette description de couleur, et c'est `\reset@color` qui a produit le deuxième `special`. Notons que dvips implémente un mécanisme de pile de couleurs, qui permet de restaurer automatiquement la couleur précédente.

Voyons ce qu'il en est avec PDFTEX :

```
.\T1/lmr/m/n/10 A
.\pdfcolorstack 0 push {1 0 0 rg 1 0 0 RG}
.\T1/lmr/m/n/10 B
.\pdfcolorstack 0 pop
.\T1/lmr/m/n/10 C
```

Ici, `\color@rgb` a préparé la description `1 0 0 rg 1 0 0 RG` qui est utilisée par `\set@color` pour produire le premier appel à `\pdfcolorstack`, tandis que le deuxième a été effectué par `\reset@color`.

Retenons de cette section les trois caractéristiques essentielles (et indépendantes du pilote utilisé en fin de compte) de cette mise en œuvre des couleurs, par opposition à la solution basée sur les attributs qui fera l'objet de la prochaine section :

1. La synchronisation entre les couleurs et les groupes de TEX ne survient pas de façon naturelle, mais est forcée avec un `\aftergroup`.

2. La couleur n'est pas une propriété individuelle de chaque nœud,

mais plutôt de zones.

3. Le début et la fin de ces zones dont délimités par des bidules (quel qu'en soit le sous-type, dépendant du pilote).

Nous nous proposons maintenant d'illustrer quelques conséquences plus ou moins fâcheuses de ces caractéristiques.

2.2. Les problèmes

Commençons par la synchronisation avec les groupes : à première vue, `\aftergroup` semble résoudre le problème de façon élégante. Un détail cependant : le nœud bidule signalant la fin de zone de couleur est inséré *après* la fin du groupe. Dans la plupart des cas, ce n'est pas gênant, mais lorsque le groupe en question est celui formé par une boîte, c'est crucial : si l'on n'y prend pas garde, ce nœud essentiel ne fera pas partie de la boîte. Prenons l'exemple du code suivant :

```
Noir au début.
{\color{red} Rouge.
\setbox0\hbox{\color{green} boîte}%
Avant, \usebox0, après.}
Noir à nouveau.
```

Pouvez-vous deviner les couleurs des mots « Avant » et « après », en supposant que la couleur normale (active durant la première ligne) soit le noir ? Déroulons le code : `\color{red}` produit un bidule qui fait passer la couleur courante au rouge. Ensuite, `\color{green}` produit un bidule activant le vert, qui est stocké dans le boîte 0. Après la fin du groupe (de la boîte) `\reset@color` est exécuté et produit un bidule restaurant la couleur précédente, qui n'est pas inséré dans le boîte 0 mais juste après le mode « Rouge. » ; le mot « Avant » sera donc en noir. Quand TEX insère la boîte 0, il insère en particulier le bidule faisant passer en vert, le mot « boîte » est donc bien en vert. Comme il n'y aucun bidule restaurant la couleur à la fin de la boîte, le mot « après » reste en vert. À la fin du groupe, un bidule restaurant la couleur est inséré (celui programmé par `\color{red}`) et tout redevient normal. Cet exemple est spécialement conçu pour être inoffensif, d'autres construits sur le même modèle provoquent aisément des problèmes au niveau de PDFTEX lui-même, par exemple :

```
pdfTeX warning: pdflatex: pop empty color page stack 0
```

en déséquilibrant de façon durable la pile de couleur.

Notons que dans l'exemple précédent, aux moments où TEX compose les mots « avant » et « après », la macro `\current@color` contient bien une représentation de la couleur rouge, à savoir exactement la couleur attendue. En effet, cette macro est soumise au mécanismes de portée (groupes) usuels de TEX. Le fond du problème est donc bien que le mécanisme utilisé pour gérer les couleurs n'est pas naturellement synchronisé avec les groupes de TEX.

Le problème précédent se résout aisément en ajoutant un niveau de groupe à l'intérieur de la boîte de sorte que la fin du groupe intervient suffisamment tôt pour que le bidule de restauration de couleur soit stocké dans la boîte. Les macros LATEX de manipulation de boîte (`\sbox`, `\savebox`, `lrbox`, etc.) prennent cette précaution ; le problème est donc résolu si l'on n'utilise que ces macros. Si l'on décide d'utiliser aussi les primitives, on peut entourer le contenu des boîtes de `\color@begingroup` et `\color@engroup` : ces commandes sont respectivement équivalentes à `\begingroup` et `\endgroup` quand color est chargé, et à `\relax` sinon, produisant un code légèrement plus efficace.

Cependant, l'efficacité du code n'est pas primordiale et, de l'avis de l'auteur, la vraie bonne raison pour utiliser ces commandes plutôt que les primitives ou une simple paire d'accolades est que leur nom signale clairement l'utilité de ce groupe supplémentaire, qui pourrait autrement paraître superflu à un lecteur distrait (ou non sensibilisé à ce problème). Pour résumer, le problème est résolu en remplaçant la deuxième ligne au choix par `\sbox0{\color{green} boîte}%` ou par

```
\setbox0\hbox{\color@begingroup
  \color{green} boîte%
\color@endgroup}%
```

Intéressons-nous maintenant au deuxième point : la couleur est une propriété d'une zone et non pas des nœuds. Il est bien visible dans l'exemple suivant.

```
\setbox0\hbox{noir}
\textcolor{gray}{Pas \usebox0\ du tout.}
```

Le mot « noir » sera écrit en gris : il hérite de la couleur environnante. Dans l'absolu, ceci peut peut être considéré comme souhaitable ou au contraire vu comme un problème. Ce que l'on peut en tout cas dire sans prendre parti, c'est que le comportement des couleurs diffère de celui

des fontes sur ce point. Cependant, il existe plusieurs circonstances où c'est objectivement un problème : imaginons par exemple un flottant qui serait inséré à un endroit où la couleur courante est différente de la couleur par défaut, et verrait ainsi sa couleur modifiée.

Là encore, LaTeX fournit une solution robuste pour peu qu'on pense à l'utiliser (ou à se reposer sur les macros du noyau LaTeX, qui l'utilisent) : il s'agit d'encadrer la fraction de document dont la couleur ne devra plus être modifiée par `\color@setgroup` et `\color@endgroup`. La deuxième de ces macros nous est déjà connue ; la première est définie (lorsque color est chargé) comme `\begingroup\set@color`. Elle crée ainsi une zone de couleur utilisant la couleur courante (stockée des `\current@color`, dont on a vu sur l'exemple précédent qu'elle contenait toujours la bonne valeur).

On obtient donc un comportement plus cohérent avec celui des fontes sur l'exemple précédent en replaçant sa première ligne par `\sbox0{noir}` ou bien :

```
\setbox0\hbox{\color@setgroup noir\color@endgroup}
```

Notons que ceci résout du même coup le premier problème évoqué, il n'est donc pas utile de cumuler les deux solutions : on utilisera soit `\color@begingroup` (si l'on ne veut pas geler la couleur) soit `\color@setgroup`, mais pas les deux.

Venons-en maintenant au troisième point : les bidules. Le problème est que ce sont des nœuds supplémentaires dans la liste courante, qui ne sont pas invisibles. Au moins deux types de problèmes peuvent se poser : les primitives examinant ou agissant sur le dernier item de la liste (`\unskip`, `\lastskip`, `\lastkern`, etc.) risquent de « voir » le bidule au lieu de l'espace (ou autre) qu'on souhaitait examiner. Dans un contexte LaTeXien, un exemple relativement naturel pourrait être :

```
\begin{center} \color{gray}
  Avertissement important !
\end{center}
\addvspace{1ex}\hrulefill\par
```

Rappelons que `\addvspace` essaie de compléter l'espace vertical précédent, s'il existe, pour atteindre la taille demandée, plutôt que de rajouter brutalement un espace sans tenir compte de celui qui est

éventuellement déjà présent. Comparons le résultat de ce fragment avec et sans le changement de couleur :

Avertissement important ! Avertissement important !

L'autre problème survient par exemple quand on veut fabriquer une boîte verticale et changer la couleur de tout son contenu. L'exemple suivant est totalement artificiel mais illustre bien le phénomène :

```
x\parbox[t]{1ex}{\color{gray} x}x
x\parbox[t]{1ex}{\textcolor{gray}{x}}x
```

Le résultat est éloquent : « x_xx xxx ». Le but est d'aligner les boîtes sur la base de leur première ligne (utilisation de `\vtop` en interne). Dans le premier cas, `\color` produit un bidule de changement de couleur qui est le premier élément de la liste verticale en cours (les bidules peuvent exister en mode vertical aussi bien qu'en mode horizontal). Le « x » devient donc la deuxième ligne, et c'est sur son haut (le bas de la ligne précédente ne contenant que le bidule) que l'alignement a lieu. Dans le second cas, `\textcolor`, qui n'est censé être utilisé qu'au sein d'un paragraphe, fait d'abord un `\leavevmode`, de sorte que le bidule de changement de couleur fera partie de la liste horizontale composant le paragraphe. Une fois ceci construit, se première ligne sera bien celle qui contient le « x » (précédé du bidule) et l'alignement escompté est obtenu.

Dans les deux exemples illustrant les difficultés causées par la présence de bidules, on voit qu'il est possible de corriger le tir à la main. D'ailleurs, ces problèmes sont bien connu des experts et sont documentés. Cependant, les comportements observés sont surprenants voir mystérieux pour un utilisateur non averti. De plus, aucune solution générale ne semble accessible. (Seuls les problèmes avec `\unskip` et `\last⟨...⟩` peuvent peut-être être contournés sous LuaTEX en remplaçant l'usage de ces primitives par un parcours des listes de nœuds en Lua.)

Au final, nous avons vu que la mise en œuvre usuelle des couleurs pose un certain nombre de problèmes, dont certains peuvent être résolus de façon générale par une certaine discipline de programmation, et d'autres semblent plus complexes. Nous allons maintenant étudier une mise en œuvre des couleurs basée sur les attributs, qui évite ces problèmes et fournit une solution bien plus naturelle.

3. Utilisation des attributs

Dans cette section, nous restons dans l'univers de LaTeX et étudions un paquet qui fournit une nouvelle mise en œuvre des couleurs, basée sur les attributs : luacolor. Nous choisissons de présenter du code effectivement utilisable dans toute sa complexité ; néanmoins certains aspects du code de luacolor ont été remaniés (principalement le découpage en fonctions et leur ordre, les mesures de compatibilité et la gestion du mode DVI, ainsi qu'un optimisation concernant les boîtes de `\leaders`) afin d'en faciliter l'exposé. Chaque omission ou simplification sera signalée pour que le lecteur curieux puisse se reporter [7]. Le code présenté est basé sur la version 1.6 de luacolor et disponible dans le fichier `myluacolor.lua` de l'archive que l'on peut télécharger depuis le site de la revue : `http://cahiers.gutenberg.eu.org/fitem?id=CG_2010___54-55_57_0`.

Le principe de base est le suivant : la valeur de la couleur courante est stockée dans un attribut, qui est automatiquement attaché aux nœuds par LuaTeX, et obéit naturellement aux règles de portée de TeX. Plus tard, la valeur de cet attribut sera examinée en parcourant la liste qui compose la page, et à chaque changement de la valeur, un bidule sera inséré pour rendre effectif le changement de couleur dans le fichier de sortie (PDF ou DVI). Il reste nécessaire de recourir à des bidules et à des zones de couleur, car c'est comme ça que fonctionne le format de sortie ; cependant le passage à ces objets est retardé le plus possible au profit des attributs qui se fondent plus naturellement dans le mode de fonctionnement de TeX.

Plusieurs parties de la problématique des couleurs subsistent :

- la nécessité de gérer plusieurs modèles de couleurs du point de vue de l'utilisateur, et de les convertir éventuellement en un autre modèle de couleur ;
- la diversité des formats de sortie. On peut penser à première vue que le seul format de sortie raisonnable sous LuaTeX est le PDF, car DVI ne supporte pas certaines des fonctionnalités les plus intéressantes de LuaTeX (gestion d'Unicode et d'OpenType) , cependant rien n'interdit de penser qu'à l'avenir d'autres formats de sortie puissent être disponibles, par exemple le SVG ou un successeur de PDF.

La solution apportée à ces problèmes par color reste parfaitement valable et luacolor va donc la réutiliser dans la mesure du possible.

En particulier, luacolor charge color et hérite de son mécanisme de sélection du pilote, de ses macros générales, et des macros définies par le pilote sélectionné. Penchons-nous maintenant sur les détails du code en commençant par la partie TeX.

3.1. Les macros TeX

Passons sur l'identification du paquet, la normalisation des catcodes, le chargement de paquets auxiliaires dont luatex.sty ou luatexbase-attr évoqués à la section 1.5 ainsi que color lui-même s'il n'a pas été chargé avant, la vérification du fait que LuaTeX est bien utilisé, la gestion du cas trivial où color aurait été chargé avec l'option `monochrome`.

La première action qui nous intéresse ici est de charger le code Lua qui sera utilisé :

```
\directlua{require("oberdiek.luacolor")}
```

La fonction Lua standard `require` localise et charge un module ; elle est modifiée par LuaTeX [6, § 3.2] pour rechercher les fichiers dans les arborescences TeX en plus des emplacements habituels pour les extensions Lua ; comme pour la version Lua usuelle, les points dans le nom du module sont transformés en séparateurs de répertoires pendant la recherche.

Le module lui-même est préfixé par le nom de son auteur : c'est un choix effectué pour regrouper les nombreux paquets crées par Heiko Oberdiek dans un même répertoire, mais surtout dans un même « espace de noms » (une même table) en Lua et ainsi éviter les conflits avec d'éventuels fichiers et modules homonymes. Dans le même ordre d'idée, comme c'est l'usage en LaTeX, toutes les macros internes du paquet commenceront par `\LuaCol@` afin de réduire les risques de conflits de noms avec d'autres paquets.

On réserve maintenant un attribut (nous verrons plus tard comment un seul attribut suffit) pour stocker la couleur courante, et on communique sa valeur au code Lua :

```
\newattribute\LuaCol@Attribute
\directlua{
  oberdiek.luacolor.setattribute(\number\allocationnumber)
}
```

Après l'allocation, \allocationnumber contient le numéro de l'attribut qui vient d'être alloué ; il est mémorisé dans une variable Lua par la fonction setattribute.

Il s'agit maintenant de modifier les macros de color pour utiliser cet attribut ; seules \set@color et \reset@color sont redéfinies, de la façon suivante :

```
\protected\def\set@color{%
 \setattribute\LuaCol@Attribute{%
  \LuaCol@directlua{%
    oberdiek.luacolor.get(
      "\luatexluaescapestring{\current@color}")%
  }%
 }%
}
\def\reset@color{}
```

La macro \reset@color est devenue inutile (ainsi que son activation avec \aftergroup dans \set@color) car les valeurs d'attributs obéissent naturellement aux groupes. Quant à \set@color, son seul rôle est de donner à l'attribut de couleur une valeur représentant la couleur courante, prise dans \current@color et convertie en un nombre par la fonction get dont nous présenterons les détails tout à l'heure.

Nous avons donc, mémorisé avec chaque nœud, une représentation de sa couleur, sous la forme d'un entier représentant la chaîne de caractères utilisée pour nommer cette couleur dans le format de sortie. Il faut maintenant prévoir d'utiliser cette information en appelant une fonction Lua adéquate. La macro suivante effectue cet appel pour une boîte désignée par son numéro.

```
\def\luacolorProcessBox#1{%
 \directlua{oberdiek.luacolor.process(\number#1)}%
}
```

Il faut maintenant s'assurer que cette macro (ou la fonction Lua sous-jacente) sera appelée sur la boîte représentant la page juste avant son envoi vers le fichier de sortie. Étonnamment, ceci se fait en TeX : on ajoute le code adéquat à la routine de sortie au moyen du paquet atbegshi qui fournit une commande à cet effet. LuaTeX ne propose en effet pas de fonction de rappel à cet endroit, ce qui n'est pas gênant vu qu'on y a déjà accès en TeX.

```
\RequirePackage{atbegshi}[2007/09/09]
\AtBeginShipout{%
  \luacolorProcessBox\AtBeginShipoutBox
}
```

Nous voyons que la plupart des macros TEX sont devenues de simples enrobages autour de fonctions Lua qui font le gros du travail. Avant d'étudier les détails de ces fonctions, soulignons les précautions prises ci-dessus dans le passage d'arguments de TEX vers Lua : les chaînes de caractères sont traitées par `\luatexluaescapestring` et les nombres par `\number`. Ces précautions méritent d'être employées de façon systématique.

3.2. Le code Lua

Il commence, comme souvent, par un appel à la fonction standard `module` :

```
module("oberdiek.luacolor", package.seeall)
```

Dissipons tout de suite une confusion possible : cette fonction n'est en aucun cas un analogue de la macro `\ProvidesPackage` de LATEX. Son rôle n'est pas de vérifier que le nom du module correspond à celui utilisé dans l'appel à `require` (et de fait, il n'y a pas d'obligation technique de faire coïncider les deux, même si en pratique c'est l'usage), mais de changer « l'espace de noms » (en Lua on dit « l'environnement ») courant : toutes les variables (ce qui, en Lua, inclue les fonctions) globales seront automatiquement préfixées par « `oberdiek.luacolor.` » jusqu'à la fin du fichier. Le deuxième argument `package.seeall` permet néanmoins d'accéder aux « vraies » variables globales (hors du module) comme la table `tex` par exemple.

Pour commencer, une variable locale est réservée pour stocker le numéro de l'attribut utilisé et une fonction sert à le mémoriser.

```
local attribute
function setattribute(attr)
  attribute = attr
end
```

Cette fonction sera visible dans tout le fichier sous ce nom-là ; hors du fichier, l'usage de `module` ci-dessus aura comme conséquence qu'il faudra utiliser son nom complet `oberdiek.luacolor.setattribute`. La variable `attribute` en revanche est locale : elle est totalement inaccessible

depuis l'extérieur du fichier. Un bon usage est de rendre locales toutes les fonctions et autres variable à moins qu'il ne soit utile de pouvoir les appeler depuis l'extérieur.

Abordons maintenant un point plus délicat : la valeur d'un attribut ne peut être qu'un nombre entier[6] ; or la valeur de la couleur courante, mémorisée dans `\current@color`, est plutôt une chaîne de caractères. Il faut donc trouver un moyen d'associer cette chaîne de caractères à un nombre qui permettra plus tard de retrouver la chaîne initiale. C'est le rôle de la fonction `luacolor.get` dont voici la définition.

```
local map = { n = 0 }
function get(color)
  local n = map[color]
  if not n then
    n = map.n + 1
    map.n = n
    map[n] = color
    map[color] = n
  end
  tex.write("" .. n)
end
```

Le principe de base est simple : on mémorise dans une table toutes les chaînes de couleurs, numérotées dans l'ordre de leur première utilisation. La mise en œuvre est élégante et permet d'illustrer quelques particularités des tables en Lua. En fait, la table est un peu la structure à tout faire en Lua : en particulier, elle sert à la fois de liste numérotée (par des entiers) et de tableau associatif, permettant d'associer des valeurs à des clés de tout type. Ici, les deux sont utilisés simultanément, de sorte que la table `map` permet aussi bien de retrouver une couleur par son numéro que l'inverse. Un champ particulier, `n`, mémorise le prochain numéro disponible.

Par exemple, supposons que la première couleur utilisé dans le document soit le noir, et la deuxième le rouge, et que le compilation a lieu en mode pdf. Juste après la première utilisation du rouge, la table `map` a la valeur suivante.

6. Positif si l'on veut rester compatible avec les vieilles version de LuaTEX.

```
{
  ['n'] = 2,
  [0] = '0 b 0 B',
  ['0 b 0 B'] = 1,
  [1] = '1 0 0 rg 1 0 0 RG',
  ['1 0 0 rg 1 0 0 RG'] = 2,
}
```

Pour faciliter la compréhension de cette fonction et de la table sous-jacente, rappelons que Lua propose deux syntaxes équivalentes pour les clés de type chaîne : dans la déclaration initiale, `n = ⟨valeur⟩` est équivalent à `['n'] = ⟨valeur⟩` ; pour accéder à cette valeur, `map.n` est équivalent à `map['n']`. En revanche, dans `map[n]`, le `n` est le nom d'une variable, dont la valeur sera utilisée comme indice, par exemple comme si on avait dit `map[2]`.

Notons que ce mécanisme permettant de stocker (virtuellement) des chaînes de caractères dans des attributs a une portée tout à fait générale, la seule restriction étant de ne pas utiliser plus de 2 147 483 648 valeurs différentes dans un même document…

La fonction la plus importante est celle qui parcourt une boîte et observe les valeurs de l'attribut de couleur pour insérer les bidules de changement de couleur aux endroit appropriés. Comme une boîte peut contenir d'autres boîtes, cette fonction s'appellera elle-même récursivement. Un fonction particulière est utilisée pour commencer la récursion avec une boîte désignée par son numéro.

```
function process(box)
  local color = ""
  local list = tex.getbox(box)
  traverse(list, color)
end
```

La fonction `traverse` se souvient à tout moment de la couleur du nœud précédent, pour n'insérer des bidules qu'aux moments où la couleur change ; lors de son premier appel, on commence avec la chaîne vide qui ne correspond à aucune couleur : ainsi, un bidule de couleur sera inséré au début de chaque page [7].

7. Ceci peut sembler peu efficace, mais avec la mise en œuvre standard des couleurs, au moins une instruction de couleur est également ajoutée à chaque nouvelle page, pour l'en-tête, même si celui-ci est vide.

Cette fonction `traverse` ne s'intéressera en fait pas à tous les nœuds mais seulement à ceux des types suivants :

– boîtes ou `leaders` contenant des boîtes ;
– autres nœuds visibles (par opposition aux espaces par exemple).

Pour l'instant, déclarons seulement des constantes pour désigner ces types de nœud ainsi que deux fonctions et penchons-nous directement sur le corps de `traverse` :

```
local LIST, LIST_LEADERS, COLOR = 1, 2, 3
local get_type, build_whatsit
function traverse(list, color)
  if not list then
    return color
  end
  if get_type(list) ~= LIST then
    texio.write_nl(
      "!!! Error: Wrong list type: " .. node.type(list.id)
    )
    return color
  end
```

La fonction commence par écarter d'emblée certains cas particuliers : celui où elle est appelée sur une boîte inexistante, et celui où l'argument n'est pas une boîte, qui provoque un message d'erreur.

Maintenant que nous sommes assurés de disposer d'une boîte valide, parcourons son contenu et traitons chaque nœud en fonction de son type :

```
local head = list.head
for n in node.traverse(head) do
  local t = get_type(n)
  if t == LIST then
    color = traverse(n, color)
```

Si l'on a affaire à un nœud de type liste, on le traite récursivement. Notons que lors de l'appel récursif, on passe la couleur du dernier nœud visible précédent la boîte, et on récupère celle du dernier nœud visible contenu dans la boîte, de sorte qu'on peut à tout instant comparer la couleur d'un nœud à celle de son prédécesseur.

Dans le cas d'une boîte utilisée dans des `\leaders` par contre, la boîte peut suivre soit le dernier nœud précédent les `\leaders`, soit un

précédent exemplaire de la même boîte dans le rangée courante de `\leaders` : on ne sait donc pas quelle est la couleur du nœud visible précédent, et on utilise par précaution [8] la chaîne vide.

```
local head = list.head
  elseif t == LIST_LEADERS then
    traverse(n.leader, '')
    color = ''
```

Dans le même ordre d'idées, juste après les `\leaders`, on ne sait pas quelle est la couleur du dernier élément visible, car la boîte de `\leaders` n'a peut-être pas été insérée du tout, ce qu'on n'a aucun moyen (facile) de savoir en examinant la liste de nœuds. On prend donc l'option prudente de repartir avec la couleur inexistante (chaîne vide).

Le cas des nœuds visibles est tout autre : il s'agit alors de récupérer la valeur de l'attribut, puis la chaîne de couleur correspondante, et de comparer celle-ci à la valeur précédente. Si elle est différente, on met à jour la couleur courante (qui servira de point de comparaison pour le nœud visible suivant) et on insère avant le nœud en question un bidule de changement de couleur, construit par une fonction qui sera présentée ultérieurement.

```
  elseif t == COLOR then
    local v = node.has_attribute(n, attribute)
    if v then
      local newColor = map[v]
      if newColor ~= color then
        color = newColor
        head = node.insert_before(head, n,
                         build_whatsit(color))
      end
    end
```

Les autres types de nœud ne font l'objet d'aucun traitement, c'est donc la fin de la construction `if`... `elseif`... concernant le type de nœud, puis de la boucle parcourant le contenu de la boîte.

8. En fait, le code réel de luacolor est un peu plus complexe et fait une première passe à vide pour vérifier si un changement de couleur intervient dans la boîte de `\leaders` avant de déterminer l'action adéquate. Le code présenté ici est plus simple mais parfaitement fonctionnel ; il insère cependant parfois des bidules inutiles.

```
    end
  end
  list.head = head
  return color
end
```

Il est essentiel, avant de quitter la fonction (en retournant la couleur du dernier nœud visible) de mettre à jour la valeur de `list.head`, le pointeur sur le premier élément de la boîte, pour le cas où on aurait inséré un bidule en tant que nouveau premier nœud de la boîte. C'est un fait général concernant l'insertion de nœuds dans des boîtes auquel il convient d'être toujours attentif.

Voyons maintenant comment classer les nœuds dans les catégories annoncées ci-dessus : boîtes, `\leaders` utilisant une boîte, et autres nœuds visibles. Pour cela, on utilise une table stockant l'information brute et une fonction permettant d'y accéder.

```
  local RULE = node.id("rule")
  local node_types = {
    [node.id("hlist")] = LIST,
    [node.id("vlist")] = LIST,
    [node.id("rule")]  = COLOR,
    [node.id("glyph")] = COLOR,
    [node.id("disc")]  = COLOR,
    [node.id("whatsit")]  = {
      [3] = COLOR, -- special
      [8] = COLOR, -- pdf_literal
     [14] = COLOR, -- pdf_refximage
    },
    [node.id("glue")] = function(n)
      if n.subtype >= 100 then -- leaders
        if n.leader.id == RULE then
          return COLOR
        else
          return LIST_LEADERS
        end
      end
    end,
  }
  get_type = function (n)
```

```
    local ret = node_types[n.id]
    if type(ret) == 'table' then
      ret = ret[n.subtype]
    end
    if type(ret) == 'function' then
      ret = ret(n)
    end
    return ret
  end
```

En premier lieu, on essaie de classer les nœuds par leur champ `id` (qui contient un nombre représentant leur type, ce nombre étant obtenu à partir d'une description lisible par les humains — par exemple `hlist` — grâce à la fonction `node.id`). Certains types sont trop généraux, et utilisent alors une sous-table pour les classer par leur sous-types. Par exemple, les bidules sont souvent invisibles (comme ceux utilisés par `\write`) mais certains peuvent être visibles : c'est le cas des `\special` et `\pdfliteral` qui produisent du matériel arbitraire. Notons que contrairement aux types, on est obligé de désigner les sous-types directement par leur valeur numérique lue dans le manuel au paragraphe du type correspondant.

Enfin, certains nœuds sont tellement complexes que leur type et leur sous-type ne suffisent pas à décider dans quelle catégorie les classer ; on utilise alors une fonction spécialisée (observons au passage l'usage facile de fonctions anonymes en Lua). C'est le cas des nœuds de type *glue*, qui sont en général invisibles, sauf s'il s'agit de `\leaders`, ce que l'on détecte par leur sous-type. Mais il y a encore plusieurs variétés de `\leaders` : ceux utilisant une réglure sont des nœuds visibles normaux, les autres utilisent une boîte qu'il convient d'explorer récursivement comme on l'a vu ci-dessus.

À ce stade le lecteur peut se demander d'où provient le contenu de la table `node_types`. Il n'y a pas de réponse simple, la meilleure approximation de la vérité étant sans doute un mélange de connaissance générale de TEX, de lecture du manuel LuaTEX et d'expérimentation. Par exemple, c'est en compilant avec luacolor un des exemples utilisés dans cet article que l'auteur a découvert que certains nœuds de type *glue* étaient visibles, puis en explorant le manuel et en relisant son *TEXbook* et son *TEX by topic* qu'il a réalisé qu'il fallait même parfois traiter

récursivement la boîte utilisée par certains \leaders, avant d'être en mesure de soumettre un correctif à l'auteur de luacolor, appuyé par un fichier de test indispensable à sa santé mentale.

Il ne nous reste plus qu'à définir la fonction servant à construire un bidule de changement de couleur. Cette fonction dépend du format de sortie utilisé.

```
local WHATSIT = node.id("whatsit")
if tex.pdfoutput > 0 then
  local PDFLITERAL = 8
  local mode = 2 -- direct
  function build_whatsit(color)
    w = node.new(WHATSIT, PDFLITERAL)
    w.mode = mode
    w.data = color
    return w
  end
end
```

Comme d'habitude nous définissons quelques constantes plutôt de d'utiliser directement des valeurs numériques (ce que les informaticiens appellent des « nombres magiques » puisqu'en lisant le code on a l'impression que leur valeur a été déterminée par magie). L'identifiant du type de nœud bidule est donné comme avant par node.id, celui du sous-type pdfliteral par le manuel, et celui du mode direct par une prochaine version du manuel (probablement la version 0.70). Au final, cette fonction construit le même nœud que \pdfliteral direct {⟨*color*⟩}. Ce mot clé direct sert à insérer le code PDF directement sans prendre de précautions particulières quant à son contexte [1, § 7.2].

On peut être surpris à première vue de l'usage de \pdfliteral plutôt que de \pdfcolorstack : en fait, le maintien d'une pile spécifique aux couleurs est devenu inutile, son rôle étant maintenant assumé par les groupes de TeX auxquels sont soumis les valeurs d'attributs ; ce changement est donc cohérent avec un des buts poursuivis : lier intimement les couleurs aux groupes de TeX.

Afin de ne pas allonger davantage cette section déjà longue et technique, nous laisserons de côté le cas du mode DVI. Mentionnons toutefois son principe, élégant et astucieux : il s'agit, avant d'avoir redéfini

la gestion des couleurs, de créer (en TEX) une boîte contenant un changement de couleur, puis de l'examiner en Lua pour voir comment color aurait fait le changement de couleur, et d'en déduire comment créer le contenu des bidules `\special` qui joueront le même rôle que les bidules `\pdfliteral` ci-dessus.

Au final, luacolor parvient à réutiliser la majeure partie du mécanisme classique de gestion des couleurs tout en utilisant un attribut pour mémoriser la couleur et en retardant le plus possible l'insertion des bidules, ce qui évite les problèmes signalés précédemment. La mise en œuvre est propre et robuste (à l'opposé des *dirty tricks* si courants auparavant) mais reste technique et exige une connaissance assez approfondie de TEX.

4. Conclusion

TEX dispose d'un mécanisme d'extension puissant et qui a été largement utilisé notamment par PDFTEX : les bidules. LuaTEX propose un nouveau mécanisme d'extension permettant d'attacher à chaque nœud un certain nombre de caractéristiques logiques obéissant aux règles de portée de TEX et pouvant être exploitées tout à fait librement en Lua : les attributs.

Nous espérons avoir montré en quoi les bidules ne sont pas parfaitement adaptés dans le cas particulier des couleurs et comment les attributs en autorisent une mise en œuvre plus naturelle et robuste. Bien sûr, l'usage des attributs ne se limite pas aux couleurs ; on trouvera d'autres exemples de leur utilisation dans [5].

À titre anecdotique, signalons que la gestion des langues a connu, en passant de PDFTEX à LuaTEX, un glissement similaire à celui présenté ici pour les couleur, passant de l'usage de bidules au stockage individuel avec chaque nœud. Cependant, comme les fontes, les langues ne peuvent être mise en œuvre au moyen d'attributs, car elles sont utilisées directement par le moteur (pour calculer les césures) alors que les attributs ne sont utilisés que par le programmeur.

C'est d'ailleurs une grande souplesse des attributs : il s'agit d'un mécanisme d'extension accessible à l'utilisateur du moteur. Par opposition, les bidules ne sont utilisables que par le concepteur d'un moteur étendu (comme l'est PDFTEX par rapport à TEX3). En fait, pour faire justice aux

bidules, il convient de signaler que LuaTEX a créé un type de bidule à définir par l'utilisateur, les bidules `user_defined`, ce qui leur confère une souplesse d'utilisation comparable à celle des attributs.

La morale de l'histoire pourrait donc être : toujours plus de choix et d'extensibilité, à tous les niveaux. Qui s'en plaindra ?

Bibliographie

1. Hàn Thế Thành, S. Rahtz, H. Hagen, H. Henkel, P. Jackowski & M. Schröder, *The pdfTEX user manual*, 2007, disponible en ligne sur CTAN : `systems/pdftex/manual/pdftex-a.pdf`.

2. D. P. Carlisle, *The color package*, disponible en ligne sur CTAN : `macros/latex/required/graphics/`.

3. D. P. Carlisle, *Packages in the "graphics" bundle*, The LATEX3 Project, disponible en ligne sur CTAN : `macros/latex/required/graphics/grfguide.pdf`.

4. R. Ierusalimschy, *Programming in Lua*, 1re éd., Lua.org, 2003, disponible en ligne : `http://lua.org/pil/`.

5. P. Isambert, « Three things you can do with LuaTEXthat would be extremely painful otherwise », *TUGboat* **31** (2010), no 3, p. 184-190, disponible en ligne : `http://www.tug.org//Contents/contents31-3.html`.

6. LuaTEX development team, *LuaTEX reference manual*, version 0.65.0, disponible en ligne : `http://luatex.org/svn/tags/beta-0.65.0/manual/luatexref-t.pdf`.

7. H. Oberdiek, *The luacolor package*, disponible en ligne sur CTAN : `macros/latex/contrib/oberdiek/luacolor.pdf`.

8. S. Rahtz & D. Carlisle, *Graphics drivers for LATEX2ε*, disponible en ligne sur CTAN : `macros/latex/required/graphics/`.

Manuel Pégourié-Gonnard
Institut de mathématiques de Jussieu
mpg@elzevir.fr