FONTES INTELLIGENTES, TEXTÈMES ET TYPOGRAPHIE DYNAMIQUE

Gábor Bella et Yannis Haralambous

Lavoisier | *Document numérique*

2006/3 - Vol. 9
pages 167 à 216

ISSN 1279-5127

Fontes intelligentes, textèmes et typographie dynamique

Gábor BELLA* — **Yannis HARALAMBOUS***

* *Département Informatique*
ENST Bretagne
CS 83 818
29238 Brest cedex 3
France
{gabor.bella, yannis.haralambous}@enst-bretagne.fr

RÉSUMÉ. *Le modèle numérique de représentation textuelle adopté aujourd'hui par l'industrie est le modèle véhiculé par le standard Unicode. Ce modèle repose sur les notions de* caractère *et de* glyphe *et devient également la base théorique pour les technologies de fonte dites* intelligentes *tels qu'AAT, OpenType ou Graphite. Par une formalisation de ce modèle de texte, nous comparerons les trois formats de fontes mentionnés, notamment concernant la préservation des correspondances entre caractères et glyphes — contenu et présentation — mais aussi concernant l'égalisation de paragraphes. Dans la seconde partie de l'article, nous proposerons une nouvelle approche à ces deux problèmes à travers un modèle de texte alternatif basé sur les concepts de* textème *et de* typographie dynamique.

ABSTRACT. *The digital text representation model used by the industry today is the model proposed by the Unicode standard. This model, based on the concepts of* character *and* glyph*, provides the foundations for the so-called* smart font technologies *such as AAT, OpenType and Graphite. Using a formal description of Unicode's text model, the authors compare the three said font formats, especially regarding paragraph breaking and how mapping information is preserved between characters and glyphs, in other words, between content and presentation. In the second part of the paper, the authors propose a novel approach to these two problems through an alternative text model based on the* texteme *concept and* dynamic typography.

MOTS-CLÉS : *Unicode, fontes intelligentes, modèle de texte, caractère, glyphe, paragraphage, textème, typographie dynamique, OpenType, AAT, Graphite*

KEYWORDS: *Unicode, smart fonts, text model, character, glyph, paragraph breaking, texteme, dynamic typography, OpenType, AAT, Graphite*

DN – 9/2006. Fontes numériques, pages 167 à 216

1. Introduction

Depuis les années 1980, il y a eu de nombreux efforts afin d'étendre l'outil informatique de la fonte numérique qui jusque-là consistait en un répertoire d'images, un ensemble de données métriques et une simple table de correspondances entre codes de caractère et images associées. Puis, l'apparition du standard de codage Unicode et du modèle de texte qu'il proposait ont mis en évidence les graves limitations des technologies de fonte de l'époque[1] et ont incité les ingénieurs à adopter une nouvelle vision du traitement de texte numérique. Un des résultats de ce changement d'attitude a été l'apparition des formats de fonte dits *intelligents* dont les premiers ont été conçus par les grands industriels du marché des systèmes d'exploitation : Microsoft, Adobe et Apple.

Les nouveaux systèmes ont dû murir pendant au moins une décennie avant d'être réellement déployés sur le marché à partir du début des années 2000. Entretemps, les expériences venant de la découverte des faiblesses (théoriques ou implémentationnelles) des nouveaux formats de fonte et des moteurs de rendu proposés par les industriels — pour ne pas mentionner le mécontentement dû à la nature propriétaire et parfois fermée de leur développement — ont incité la naissance de plusieurs projets de recherche « indépendants » qui visaient à proposer des solutions alternatives, souvent mieux conçues[2]. Cependant, à la base de toutes ces technologies, les principes restent les mêmes : le modèle de texte d'Unicode, le rapport entre contenu (fond) et présentation (surface) ou encore entre texte et moteur de rendu, la nature et l'ordre des opérations à effectuer, les méthodes classiques de paragraphage et ainsi de suite.

Le but du présent article n'est pas d'être une introduction à ces principes, technologies et modèles, encore moins de les décrire en détail : ceci a été fait dans de nombreux articles ou ouvrages tels que (Andries, 2007) ou (Haralambous, 2004a). On tentera plutôt de les analyser avec un esprit critique afin de voir plus clairement les conséquences — positives et négatives — du modèle Unicode sur le fonctionnement des fontes intelligentes. Cela nous amènera à des réflexions sur des modèles de texte alternatifs à travers lesquels on arrive à mieux exploiter les richesses de ces formats de fonte.

Nous nous intéresserons particulièrement à deux points précis : la correspondance entre caractères et glyphes et la relation entre opération de paragraphage et fontes intelligentes.

---

1. À part les pays occidentaux, le reste du monde avait bien évidemment déjà pris conscience de ces limitations depuis longtemps, puisque les systèmes électroniques de composition développés en Occident n'étaient guère capables d'adresser les besoins de leurs écritures non latines.
2. On trouvera diverses communications sur ce sujet dans les actes des conférences *Raster Imaging and Digital Typography*.

2. À la base : Unicode et son modèle de texte

Dans les systèmes informatiques, la notion de base de la modélisation du texte a toujours été le *caractère* : il s'agit de relier un élément d'écriture (lettre, signe de ponctuation ou autre symbole) à une valeur numérique — un entier naturel — qui représentera celui-ci dans l'ordinateur. Cette relation est établie par la *table de codage* qui définit ainsi l'ensemble de caractères disponibles pour un ordinateur donné. Pendant longtemps, ce modèle n'a pas eu besoin d'évoluer, l'usage unique du caractère étant le stockage des données textuelles brutes — la PAO n'existait pas encore. Lorsque l'ordinateur est devenu capable d'afficher du texte par l'intermédiaire d'une imprimante ou d'un écran, un troisième élément est entré dans l'équation : une image concrète associée au caractère, son représentant visuel que l'on appelle aujourd'hui *glyphe*. En effet, l'association entre un caractère et son image était si forte et directe que l'image en est devenue le porteur, véhicule primaire de la relation entre élément d'écriture et code numérique. On observe donc trois concepts différents — l'élément d'écriture, le code numérique et l'image — qui, par leur liaison étroite, semblent former une unité indivisible.

Tout a changé lorsque les premiers systèmes de composition de texte sont apparus et ont apporté de la variété visuelle au texte en supplantant l'unique *fonte du système* par un choix de fontes numériques. Ces fontes contenaient cependant souvent des glyphes pour lesquels il n'existait pas de caractère correspondant dans la table de caractères standard de la plate-forme informatique utilisée : lettres propres à une langue ou à une écriture, pictogrammes, signes typographiques, etc. Parmi ces glyphes « orphelins », certains (ligatures, signes typographiques et de ponctuation) apparaissaient régulièrement dans les fontes. Afin d'éviter une polysémie *ad hoc* des caractères qui aurait eu comme résultat qu'un texte ne serait lisible que par l'intermédiaire d'une fonte particulière, il était important d'élargir le nombre total de codes disponibles dans la table de codage. Cet élargissement étant limité par la représentation uni-octet des caractères, un grand nombre de nouvelles tables de codage sont apparues, chacune adaptée à un système d'écriture ou de notation (ou à une langue ou bien à une famille de langues) particulier. Ainsi, le gain n'a été que partiel : *au lieu de textes dépendants de fontes données, on a produit des textes dépendants de codages donnés*.

En même temps, on observait des incohérences internes aux tables de codage : le manque de certains caractères (comme le *œ* du français dans le codage « occidental » ISO-8859-1) et la présence d'autres comme le caractère « ... » (points de suspension) et les ligatures *fi* et *fl*. Ces trois derniers glyphes peuvent pourtant très bien être associés aux chaînes de caractères « `...` » (trois points), « `fi` » et « `fl` ». La raison qui a motivé malgré tout leur inclusion dans les tables de codage d'Apple et de Microsoft était une faiblesse du modèle de texte utilisé : l'obligation d'établir une relation bijective entre caractère et glyphe. Dés lors, il est devenu possible de représenter le même contenu textuel par plusieurs chaînes de caractères différentes.

Les effets néfastes de ces phénomènes sont devenus vite apparents et ont été une des raisons de la conception d'Unicode, le système de codage *uni*que et *uni*versel. À

part l'extension drastique et évidemment nécessaire de l'espace de codage, le but de l'introduction d'Unicode était de redéfinir, une fois de plus mais cette fois-ci de manière cohérente, la notion de caractère. Le principe directeur a été l'idée classique de séparer contenu et présentation : d'un côté les *caractères*, une classe restreinte d'entités abstraites qui sont eux seuls censés être porteurs du contenu textuel, de l'autre côté les *glyphes*, images visuelles pouvant être associées aux premiers dont le standard Unicode ne se soucie guère (si ce n'est en fournissant un glyphe indicateur sans valeur normative). Cette approche a permis d'uniformiser les techniques divergentes de représentation du même texte : un pas incontournable dans l'ère du Web vers l'échange efficace de documents électroniques.

Cependant, le syndrome bipolaire d'Unicode qui n'admet l'existence que de caractères et de glyphes s'est avéré être simplificateur. Plusieurs articles ont déjà traité de ce sujet, comme par exemple : (André *et al.*, 3-4/2002, passim), (Haralambous, 3-4/2002) et (Haralambous *et al.*, 2003). Il en ressort, notamment, qu'Unicode échoue dans sa tentative de donner des définitions précises aux termes *caractère* et *glyphe*, bien que ceux-ci soient les objets de son intérêt principal. C'est peut-être aussi pour cette raison que l'on trouve de nombreuses lettres et de nombreux symboles d'écritures — et même des écritures entières comme le gaélique — qui ont du mal à trouver leur place entre ces deux catégories.

Pour donner un exemple où la classification binaire entre caractères et glyphes est inadaptée, prenons le cas de l'écriture arabe. L'alphabet consonantique (abjad) arabe est composé de 29 lettres. Cependant, la lettre arabe est un concept abstrait dans le sens où sa représentation visuelle dépend toujours de sa position dans le mot (initiale, médiane, finale ou isolée), l'écriture étant connectée par nature. Ces quatres formes dites *contextuelles* (dans le jargon informatique) sont indépendantes de tout style ou fonte, il s'agit bien donc d'une propriété intrinsèque de l'écriture. L'arabe connaît également plusieurs *styles* traditionnels, chacun associé à une valeur sémantique plus ou moins précise : le style *naskh* est le plus courant, le *koufique*, plus géométrique, est plutôt réservé aux titres et aux inscriptions monumentales, etc. Le choix de fonte complète le passage depuis l'abjad phonétique jusqu'au rendu graphique final.

Entre un texte abstrait (tel qu'il est représenté par les 29 lettres arabes) et sa manifestation graphique concrète on observe donc au moins quatre strates d'informations grammaticales, sémantiques et visuelles (pas forcément dans cet ordre), voir figure 1. Unicode, conformément à ses principes déclarés, s'intéresse uniquement à la représentation abstraite de l'écriture : en effet, le plan arabe d'Unicode se restreint essentiellement aux 29 consonnes (ainsi qu'aux voyelles, aux diacritiques et à quelques autres symboles)[3]. Selon Unicode, ni les formes contextuelles ni les formes stylistiques ne doivent être modélisées par le concept de caractère. Et pourtant, un modèle de texte qui ne distingue que caractères et glyphes n'est pas suffisamment riche pour décrire un texte arabe de manière précise.

3. Même si, pour des raisons historiques de compatibilité avec des codages pré-existants, toutes les formes contextuelles arabes, y compris un grand nombre de ligatures, ont été incluses dans les tableaux *Formes de présentation* d'Unicode.

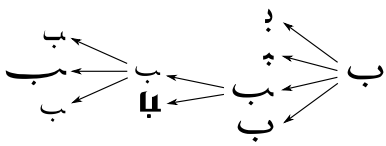


Figure 1. *Quatre strates de l'écriture arabe : à droite, la lettre abstraite* bâ *représentant le phonème « b », dans le modèle Unicode elle correspond au caractère. Puis ses quatre formes contextuelles, deux styles (naskh et koufique) et finalement à gauche trois variantes graphiques venant de trois fontes.*

Unicode est peut-être encore plus mal adapté aux écritures idéo(phono)graphiques de l'Extrême-Orient telles que le chinois, le japonais ou le coréen. Ici, le choix de faire correspondre *un caractère chinois unique* à *un caractère Unicode unique* est une représentation inadaptée à la nature constructive de ces écritures où l'invention de nouveaux caractères[4] par composition d'éléments de base est trop fréquente pour qu'un standard tel qu'Unicode ne puisse suivre ce phénomène de manière efficace.

Malgré la pauvreté descriptive d'Unicode vis-à-vis de la structure et de la sémantique très riches des graphèmes de certaines écritures, d'un point de vue purement pratique, et surtout en comparaison avec les codages pré-Unicode, sa manière de modéliser les écritures orientales s'est avérée être plutôt un succès, et Unicode a été adopté comme le modèle de base pour la représentation de texte à tous les niveaux, y compris ceux des systèmes d'exploitation, des applications de composition typographique, et *last but not least* des fontes numériques.

3. Les modèles et systèmes de composition « classiques »

Dans les ordinateurs personnels d'aujourd'hui, la composition de texte de haute qualité est devenue un service omniprésent au lieu d'être le rôle d'un certain nombre de logiciels typographiques spécialisés, souvent au travers de bibliothèques au niveau du système d'exploitation. Dans le monde Microsoft, c'est la technologie de fonte *OpenType* et la bibliothèque *Uniscribe* correspondante qui s'occupent de la composition de texte, aussi bien pour l'affichage des interfaces utilisateur que pour les applications telles que Word ou OpenOffice.org. L'équivalent sous MacOS X est le service ATSUI d'Apple, lui aussi accompagné de son propre format de fonte, AAT. Les systèmes Linux divergent : Gnome utilise *Pango* qui à son tour fait appel à la bibliothèque *m17n* (*multilingualization*), KDE a son propre moteur typographique dans *Qt*, OpenOffice.org se base sur les bibliothèques *ICU* d'IBM. Un effort indépendant mais fort intéressant du point de vue technique est *Graphite* (Hosken *et al.*, 2000) qui a récemment commencé à être intégré dans Pango.

4. Ce qui est similaire au phénomène de la naissance de nouveaux mots dans le cas des écritures alphabétiques.

Cependant, les outils typographiques de plus haute qualité tels que les logiciels professionnels d'Adobe (InDesign, Illustrator, etc.), QuarkXPress, ou TEX et ses successeurs utilisent leurs propres moteurs typographiques plus complexes car moins soumis aux contraintes de vitesse et de consommation de mémoire que les bibliothèques des systèmes d'exploitation. Néanmoins, la tendance semble être la convergence des deux mondes : avec les ordinateurs devenant de plus en plus puissants, les services du niveau système deviennent capables d'effectuer des opérations typographiques jusqu'alors privilèges des applications professionnelles.

La base commune à la totalité de ces logiciels, services et formats de fonte (à l'exception de TEX), est leur adoption du modèle de texte d'Unicode et de la séparation claire entre caractère et glyphe. Dans la suite, nous nous intéresserons à trois formats de fonte intelligents et aux systèmes de traitement correspondants : OpenType et son moteur Uniscribe, AAT et ATSUI, et finalement Graphite. Nous examinerons les similarités et les différences entre eux, à partir de trois points de vue majeurs : la représentation de texte, la puissance des transformations et la gestion du paragraphage.

3.1. *Préliminaire : une formalisation du concept de modèle de texte*

Avant même de commencer la description des différents formats de fonte intelligents, nous allons tenter d'éclaircir ce que nous entendons par *modèle de texte*. Le mot *texte* ayant un large champ sémantique, il peut désigner une gamme de concepts en fonction de son support (oral, écrit, stocké par l'ordinateur), de ses aspects visuels (manuscrit, typographié, etc.), de sa longueur (un mot, un ouvrage de mille pages) et ainsi de suite. Au lieu de chercher un modèle général qui engloberait toutes les manifestations de ce que l'on appelle texte, restons dans les cadres énoncés de l'article présent. Ainsi, nous allons nous intéresser uniquement au texte écrit, et plus précisément à sa reproduction informatique ; il s'agit donc de trouver des modèles qui *reproduisent* les propriétés principales de l'écrit que nous trouvons autour de nous. De plus, les fontes numériques étant toujours le sujet principal de cet article et de ce numéro, nous allons restreindre notre étude aux éléments constitutifs du texte et à leurs interactions. En termes typographiques, la portée de nos modèles ne dépassera jamais la longueur d'un paragraphe.

Un aspect important n'a pas encore été pris en compte ici : l'*usage* du texte. Un usage important de tout texte écrit est la transmission d'information dans l'espace et dans le temps. De nombreuses propriétés visuelles, structurelles, de support, etc. du texte transmis dépendent de son émetteur, de son récepteur et aussi du médium de transmission. Il en est de même pour l'ordinateur : un *email* interne d'une entreprise qui invite un collègue au déjeuner habituel n'aura probablement pas la même représentation informatique qu'une carte d'invitation au Festival de Cannes.

Une réponse possible au problème de représenter des textes à usages différents est d'associer à chaque usage un *modèle de texte* qui lui est propre. Une des raisons pour lesquelles les codages historiques décrits dans l'introduction ont été des échecs était

justement le fait qu'un modèle simpliste et unique, le modèle de caractère en tant que simple valeur numérique, a été censé répondre à de multiples usages : cette valeur désignait à la fois une lettre de l'alphabet et une image graphique. C'est pour résoudre ce problème qu'Unicode suggère la création d'au moins deux modèles de texte : d'une part un seul et unique modèle de *représentation interne* du contenu textuel, sujet du standard Unicode, d'autre part un ou pluiseurs autres modèles de *description présentationnelle*, basés sur le concept de *glyphe*, dont Unicode évoque le besoin sans les définir.

3.1.1. *Le modèle formel du texte Unicode*

Le *modèle du contenu textuel* d'Unicode (UTR, n.d.) définit le texte comme une série de *caractères*, un caractère étant la correspondance entre un entier naturel et une sémantique. Plus formellement, soit $\mathcal{S}_\mathcal{G}$ un ensemble d'entités sémantiques qui décrivent des *graphèmes* des écritures du monde (comme par exemple, *lettre latine majuscule A*). Soit $\mathcal{S}_{\mathcal{NG}}$ un autre ensemble d'entités sémantiques décrivant des fonctions textuelles *non graphémiques* (comme par exemple, *retour-chariot*). $\mathcal{S}_\mathcal{G} \cup \mathcal{S}_{\mathcal{NG}}$ est un ensemble que le standard Unicode appelle *RCA (répertoire de caractères abstraits, Abstract Character Repertoire)* et qui représente la couche inférieure dans la transition à partir de données abstraites vers une représentation matérielle adaptée à une plate-forme informatique précise. Ce dernier aspect étant sans intérêt pour notre modèle, nous nous intéresserons uniquement aux deux couches inférieures du modèle Unicode. Alors que la couche RCA représente $\mathcal{S}_\mathcal{G} \cup \mathcal{S}_{\mathcal{NG}}$, la couche immédiatement supérieure, nommée *JCC (jeu de caractères codés, Coded Character Set)*, associe aux concepts « abstraits » des valeurs uniques.

Soit N_C le nombre de *points de code* disponibles, un entier naturel, défini par la couche JCC du standard Unicode (au moment de l'écriture de l'article, $N_C = 2^{21}$). On définit alors le *codage Unicode* comme une relation surjective $\mathcal{R}_{\text{uni}}$ entre $\mathcal{S}_\mathcal{G} \cup \mathcal{S}_{\mathcal{NG}}$ et les entiers entre 0 et N_C :

$$\mathcal{R}_{\text{uni}} \subset (\mathcal{L} \cup \mathcal{S}) \times \{0, 1, 2, ..., N_C\}$$

Un *caractère Unicode* c est défini comme un élément de $\mathcal{R}_{\text{uni}}$:

$$c \in \mathcal{R}_{\text{uni}}$$

Unicode associe à chaque graphème ou fonction textuelle non-graphémique un entier naturel — un *point de code* ou tout simplement *code* — qui lui est propre (d'où la surjectivité) ; ainsi, $\mathcal{R}_{\text{uni}}$ est bien ordonné. Par conséquent, tout caractère c peut être identifié par son code uniquement, fait qui nous permettra dans la suite — conformément au procédé des systèmes informatiques actuels — de référencer les caractères par leurs codes. Nous allons suivre la même approche en définissant le terme *texte Unicode* en tant que série de codes au lieu de série de caractères : soit $\Sigma_{\text{uni}} = \{0, 1, 2, ..., N_C\}$, l'ensemble de points de code. Un *texte Unicode* est alors une chaîne de caractères Unicode, élément de Σ^*_{uni} :

$$T_{\text{uni}} \in \Sigma^*_{\text{uni}}, \text{ autrement dit, } T_{\text{uni}} = c_1 c_2 c_3 ... c_n \text{ où } c_i \in \Sigma_{\text{uni}}$$

Il est connu que le codage Unicode contient des « trous », des points de code qui ne réfèrent à aucune entité sémantique. Notre modèle formel exclut ces points de code « vides » de l'ensemble de caractères mais, et ceci peut paraître paradoxal, en même temps permet leur inclusion dans les textes Unicode. Pourtant, cette contradiction reflète bien la réalité car nous allons voir qu'il est tout à fait possible en pratique de créer des textes Unicode qui contiennent de tels points de code vides — par exemple, des points de code de la zone *ZUP (zone à usage privé, Private Use Area en anglais)*, ensemble de codes réservés à l'usage privé — mais qui se visualisent et s'impriment parfaitement grâce à des fontes spécialement adaptées.

3.1.2. *Le modèle typographique employé par les fontes intelligentes*

Unicode se restreint à une modélisation du contenu textuel, tout en prévoyant l'existence d'autres modèles éventuels pour usages présentationnels, sans pour autant les définir. Le *modèle typographique* que nous allons présenter est un tel modèle présentationnel qu'ont adopté les formats de fonte intelligents ainsi que de nombreux systèmes typographiques. Il est important de noter que le modèle présenté est loin d'être le seul modèle possible. Ainsi, dans la section 4.1 nous proposerons un modèle présentationnel alternatif.

Une fonte F contient (entre autres) un ensemble $\mathcal{I}_F$ d'*images*, $N_F := |\mathcal{I}_F|$. Par analogie au modèle en couches d'Unicode, on peut considérer cet ensemble d'images comme la couche inférieure de notre modèle qui représente les entités à modéliser. Toujours de manière très similaire au modèle d'Unicode, chaque image est identifiée dans la fonte par un entier naturel, de manière unique. Un *glyphe* g est alors le couple formé d'une image et de son identifiant :

$$g = (n, i) \text{ où } n \in \{0, 1, 2, ..., N_F\}, i \in \mathcal{I}_F$$

Soit Σ_F l'ensemble d'identifiants des glyphes présents dans la fonte F. Similairement aux chaînes de caractères, une chaîne de glyphes est définie comme la chaîne de leurs identifiants uniques :

$$T_F \in \Sigma_F^*, \text{ autrement dit, } T_F = g_1 g_2 g_3 ... g_n \text{ où } g_i \in \Sigma_F$$

D'un point de vue algébrique, Σ_F^*, tout comme Σ_{uni}^*, sont des *monoïdes libres* avec comme opération la concaténation, comme élément neutre la chaîne vide et comme générateurs libres[5] Σ_F et Σ_{uni} respectivement. Ces deux derniers n'ayant en pratique jamais le même nombre d'éléments (il serait presque impossible de réaliser une fonte qui contienne suffisamment de glyphes pour couvrir la totalité des caractères, voire des points de code d'Unicode), les deux structures ne sont pas isomorphes.

Un *texte formaté* est plus complexe qu'une simple suite d'identifiants de glyphes $g_1 g_2 ... g_n$ car de nombreuses propriétés présentationnelles en font également partie :

5. Le générateur libre est souvent appelé *alphabet*, terme plus graphique mais en même temps trompeur dans un contexte multi-écritures ; pour cette raison, nous allons l'éviter.

les coordonnées (x, y) des glyphes, leur couleur, leur taille etc. Définissons pour le moment le concept de *propriété* de manière simple, comme fonction ϕ d'un glyphe en tant qu'élément de la chaîne, autrement dit, fonction de l'indice (et non de l'identifiant) de glyphe, renvoyant vers un ensemble quelconque de taille finie :

$$\phi_{\text{couleur}}(i) \text{ est la propriété } \textit{couleur} \text{ du glyphe } g_i$$

Remarquons que cette définition est suffisamment générale pour également être appliquée aux chaînes de caractères, ainsi, $\phi_{\text{couleur}}(i)$ peut aussi jouer le rôle d'une propriété liée au caractère de position i dans la chaîne. (Une propriété de couleur attachée à un caractère ferait en réalité référence à la couleur du glyphe associée à ce caractère.)

Dans la suite, pour des raisons de simplicité, nous allons parler de *chaîne de caractères* et *chaîne de glyphes* au lieu de *chaîne de codes de caractères* et *chaîne d'identifiants de glyphes* respectivement.

Notre première tentative pour la définition du *texte formaté* (T_{form}) consistera en l'ensemble d'une chaîne de glyphes et de toutes ses propriétés :

$$T_F = g_1 g_2 ... g_n, \Phi_{T_F} = \left\{ \bigcup_{i=1}^{n} \{\cup_{\forall j} \phi_j(i)\} \right\}, T_{\text{form}} = \{T_F, \Phi_{T_F}\}$$

Dans les systèmes informatiques de composition de texte, le texte formaté est en général obtenu par la génération d'une chaîne de glyphes à partir d'une chaîne T_{uni} de caractères d'entrée, de propriétés Φ_{uni} associées à cette chaîne et d'une fonte F. La fonction qui exécute cette génération s'appellera *fonction de formatage* :

$$T_{\text{form}} = f_{\text{form}\,F}(T_{\text{uni}}, \Phi_{\text{uni}})$$

Il existe cependant un aspect de la composition électronique de texte qui échappe au modèle que nous venons d'établir : il s'agit du maintien des correspondances entre éléments de la chaîne de caractères d'entrée (T_{uni}) et éléments de la chaîne de glyphes de sortie (T_F). Pour plusieurs raisons, il est primordial pour tout système de composition de texte de suivre les correspondances entre caractères du texte d'entrée (T_{uni}) et glyphes du texte composé (T_F). D'une part, pendant le processus de composition même, il arrive un moment où le texte d'un paragraphe déjà composé doit être coupé en lignes. Cependant, la césure est une opération qui s'effectue sur des mots, c'est-à-dire sur des chaînes de *caractères* alors que le paragraphe composé est essentiellement une suite de glyphes et d'autres informations visuelles. Il s'ensuit que le système de composition doit être capable de retrouver les caractères d'origine à partir desquels les glyphes du paragraphe ont été calculés, et ceci n'est possible qu'en connaissance des correspondances entre les deux textes. D'autre part, ces correspondances demeurent pertinentes après le processus de formatage car elles sont essentielles pour assurer l'interactivité du document généré (l'extraction du contenu, la recherche, l'indexation).

Pour la facilité d'expression, introduisons encore une définition : deux textes, T et T', sont *en bijection ordonnée* si

1) les longueurs de T et de T' sont identiques ;

2) pour tout élément t de T et t' de T', l'image de t_i est t'_i.

Par « image de t_i est t'_i », on entend que la sémantique véhiculée dans T par t_i est représentée dans T' par t'_i. La plupart du temps, nous allons référer à cette notion de représentation par le terme *correspondance* : t'_i correspond à t_i.

Il est facile de voir que les correspondances sont triviales lorsque T_{uni} et T_F sont en bijection ordonnée. Cependant, certaines transformations peuvent rompre la bijection : par exemple, le remplacement de deux glyphes *f* et *i* par une ligature *fi* diminue la longueur de la chaîne de glyphes. En général, si f_{form} ne réalise pas une bijection ordonnée, il devient indispensable de maintenir en plus des chaînes de caractères et de glyphes un troisième type d'informations, notamment les correspondances entre les deux chaînes.

La solution adoptée par les systèmes comme ATSUI, Uniscribe et Graphite consiste en la création d'un *graphe de correspondances* entre caractères d'entrée et glyphes de sortie. Ce graphe, maintenu par la fonction de formatage, permet de retrouver le ou les caractère(s) correspondant(s) à chaque glyphe et vice versa. Nous allons donc étendre la définition du texte formaté, T_{form} :

$$T_{\text{form}} = \{T_{\text{uni}}, T_F, \Phi_{T_F}, C\}$$

où C est un graphe non-orienté dont les nœuds correspondent aux caractères de T_{uni} et aux glyphes de T_F. Une arête est présente entre un nœud de caractère et un nœud de glyphe lorsque ce dernier « correspond au » (« représente » ou « fait partie de la représentation du ») premier :

$$\text{Soit } T_{\text{uni}} = c_1 c_2 ... c_n, T_F = g_1 g_2 ... g_m,$$
$$\text{alors } C = (V, E); V = \{c_1, c_2, ..., c_n\} \cup \{g_1, g_2, ..., g_m\};$$
$$E = \{\cup(c_i, g_j)\} \text{ pour tout } i, j \text{ tels que } g_j \text{ « correspond à » } c_i$$

3.2. *Bases communes d'AAT, d'OpenType et de Graphite*

Ces trois formats intelligents se basent tous sur la structure en tables du format TrueType, sans doute parce qu'elle facilite l'extensibilité et garantit la compatibilité ascendante. La plupart des nouvelles informations qui rendent ces formats « intelligents » par rapport à leurs prédécesseurs ont été ajoutées sous forme de nouvelles tables et sont ensuite organisées dans une structure hiérarchique, d'abord par écriture, puis par langue. À chaque langue correspond un ensemble de *fonctionnalités (features)* disponibles. Une fonctionnalité représente un sous-ensemble fonctionnel des opérations typographiques offertes par la fonte telles que ligaturage, positionnement de marques, crénage, variantes stylistiques, etc. Les fonctionnalités constituent également une interface entre la fonte, l'application et l'utilisateur : c'est à travers elles que le choix de ressources à partir de la fonte peut être contrôlé avec finesse. Finalement, la fonctionnalité regroupe un ensemble d'actions concrètes qui s'effectuent

sur le texte. Dans la suite, nous appellerons comme terme général ces actions *règles de transformation*.

La notion de fonctionnalité s'intègre dans notre modèle formel comme propriété ϕ_{fonct} associée initialement à la chaîne de caractères d'entrée mais qui en réalité contrôle les opérations effectuées sur les glyphes qui lui correspondent.

3.3. *Les actions principales, leur ordre et les données fournies par les fontes*

Nous avons introduit dans la section précédente la fonction de formatage afin de modéliser le passage du texte d'entrée au texte formaté. Le rôle de la fonte intelligente est de fournir une grande partie des informations nécessaires pour effectuer le formatage. Malgré les implémentations divergentes, les trois formats de fonte (AAT, OpenType et Graphite) ont une propriété en commun : ils divisent les opérations en une succession d'étapes majeures exécutées dans un ordre précis. La fonction de formatage peut donc être considérée comme la composition d'une série de fonctions correspondant aux étapes (qui, à leur tour, seront de nouveau composées d'autres fonctions). Voici les étapes principales et leurs fonctions correspondantes :

Étape 1)
La chaîne de caractères d'entrée est convertie en glyphes par une table CMAP (character mapping). Dans le cas des formats plus anciens comme TrueType ou PostScript de type 1, c'est cette étape unique qui générait la chaîne de glyphes. Pour AAT, OpenType et Graphite, il ne s'agit que de la première étape d'un long processus de transformations qui sert à passer du domaine de la représentation textuelle interne au domaine de la représentation graphique : la majorité (OpenType) si non la totalité (AAT, Graphite) des opérations de formatage sont effectuées sur des chaînes de glyphes. Cette étape peut être considérée comme une fonction f^{CMAP} de la fonte F dont le domaine est celui des points de code Unicode et dont l'image est un sous-ensemble des glyphes présents dans F :

$$\Sigma_F^{\text{CMAP}} \subseteq \Sigma_F ; f_F^{\text{CMAP}} : \Sigma_{\text{uni}} \mapsto \Sigma_F^{\text{CMAP}} ; f_F^{\text{CMAP}}(c_i) = g_i$$

c_i étant élément de la chaîne de caractères d'entrée et g_i glyphe de la chaîne de glyphes résultante au même indice.

On constate à partir de cette définition que f^{CMAP} exécute une transformation un-à-un, ainsi la longueur de la chaîne de glyphes résultante est égale à celle de la chaîne de caractères d'entrée. En pratique, aucune fonte ne contient autant de glyphes qu'il existe de points de code Unicode : f^{CMAP} renvoie ces caractères « orphelins » vers un glyphe spécial nommé *non défini* (souvent appelé *notdef* d'après son nom PostScript) qui fait partie de toute fonte. Par conséquent, f^{CMAP} n'est pas bijective et n'a pas d'inverse. (D'ailleurs, même en restreignant le domaine de f^{CMAP} aux caractères associés à des glyphes autres que *notdef*, on ne peut pas être certain de sa bijectivité : par exemple, rien n'empêche d'associer le même glyphe aux caractères *latin majuscule A*, *grec majuscule Alpha* et *cyrillique majuscule A*.) Et même si T_{uni} et

$T_F = f_F^{\text{CMAP}}(T_{\text{uni}})$ restent en bijection ordonnée, il devient indispensable que T_{uni} fasse partie du texte formaté afin de ne pas perdre le contenu d'origine.

Étape 2)
La chaîne de glyphes obtenue par CMAP subit des transformations suivant les règles définies dans la fonte. Les types de transformation sont : *substitution de glyphes*, *insertion*, *suppression* et *réordonnancement*. Chacune des transformations τ (à l'exception du réordonnancement d'OpenType que nous allons considérer séparément) prend comme entrée un texte formaté et produit comme sortie un autre texte formaté dont la chaîne de glyphes a subi des modifications :

$$\tau_F(T_{\text{form}}) = T'_{\text{form}}$$

Étape 3)
Les positions des glyphes sont calculées à l'aide des métriques et des règles de positionnement de la fonte. De nouveau, il s'agit de transformations de forme $\tau_F(T_{\text{form}}) = T'_{\text{form}}$ mais cette fois-ci la seule intervention effectuée est l'ajout ou la modification des propriétés $\phi_{\Delta x}(i)$ et $\phi_{\Delta y}(i)$, c'est-à-dire des décalages bidimensionnels pour chaque glyphe g_i.

Étape 4)
Lors du paragraphage, certaines opérations des deux dernières étapes peuvent éventuellement être réappliquées au cas où la division des paragraphes en lignes aurait interféré avec les résultats des calculs précédents, les ayant rendus invalides.

Si l'on ignore les modifications éventuelles liées au paragraphage, le fonction de formatage peut alors s'écrire de la manière suivante :

$$T_{\text{form}} = f_{\text{form}}(T_{\text{uni}}, \Phi_{\text{uni}}) = \tau_F^{\text{pos}}(\tau_F^{\text{sub}}(f_F^{\text{CMAP}}(T_{\text{uni}}, \Phi_{\text{uni}})))$$

où τ^{sub} est formé de la composition de fonctions élémentaires effectuées sur les glyphes en étape 2 et où τ^{pos} est la composition des positionnements de l'étape 3.

L'ordre des opérations donné ci-dessus n'est pas toujours respecté car chacun des trois systèmes et formats de fonte a sa propre manière de procéder. Ainsi, la bibliothèque Uniscribe effectue un réordonnancement sur la chaîne de caractères avant la conversion de celle-ci selon la table CMAP, alors que Graphite et AAT le font plus tard, lors de l'étape 2. Derrière ce détail se cache une différence profonde entre l'approche du format OpenType et celle des deux autres. Alors que les fontes Graphite et AAT sont auto-suffisantes dans le sens où elles décrivent la totalité des opérations nécessaires pour arriver à partir de la chaîne de caractères jusqu'à la chaîne de glyphes, OpenType considère qu'il est nécessaire de distinguer entre, d'une part, les opérations indépendantes de fonte et exécutées invariablement et, d'autre part, les opérations typographiques qui peuvent être contrôlées par la fonte. Les opérations du premier type sont principalement de nature linguistique, c'est-à-dire particulières aux écritures et aux langues : par exemple, le réordonnancement des écritures indiennes, l'analyse contextuelle de l'arabe, l'algorithme bidirectionnel, etc. Ainsi, à la différence d'AAT et de Graphite, les informations nécessaires pour l'exécution de ces opérations

se trouvent non pas dans la fonte OpenType mais au niveau du système d'exploitation, plus précisément dans la bibliothèque Uniscribe. Il s'agit d'une approche centralisatrice qui simplifie le processus de création de fontes en déplaçant la responsabilité du support des écritures du monde vers Uniscribe. Pour la même raison, les opérations effectuées sur la chaîne de glyphes dans le cas d'AAT et de Graphite se font pour OpenType sur la chaîne de caractères par Uniscribe avant la conversion par la table CMAP. Est-ce une différence purement implémentationnelle ou bien une différence de modèle ? Pour trouver réponse à cette question, nous devrons d'abord examiner les modèles de transformation de chaque format séparément, y compris la gestion de correspondances caractère-glyphe.

Dans la liste d'actions que nous venons de présenter, on observe une restriction délibérée de l'ordre des opérations, un point commun pour les trois systèmes : aucune modification n'est faite aux caractères une fois que le passage aux glyphes a eu lieu, et similairement, aucune modification ne peut être faite sur la chaîne de glyphes après être passé à l'étape de positionnement. Cette restriction se justifie par le fait qu'en remplaçant un seul caractère on remplace également le glyphe correspondant, et ce changement dans la chaîne de glyphes pourra éventuellement annuler les résultats déjà obtenus par l'application des règles de l'étape 2. De la même manière, si l'on remplace un glyphe après avoir appliqué les règles de positionnement, les positions doivent être recalculées non seulement pour le glyphe en question mais aussi pour les autres affectés par le glyphe remplacé.

Il existe cependant certains comportements typographiques qui ne peuvent être implémentés suivant l'ordre où les substitutions précèdent toujours les positionnements : dans l'écriture arabe diacritée, le calligraphe a une grande liberté dans le placement de différentes marques autour les lettres. Parfois, suivant le texte, ces marques s'accumuleraient de manière très dense, ce qui nuirait à la lisibilité et à la beauté de la calligraphie. Le calligraphe peut alors choisir une forme de lettre de base différente (disons, allongée) afin de mieux espacer les marques et ainsi de rendre le texte plus aéré. Dans le modèle informatique propre aux formats AAT, OpenType et Graphite, la même procédure ne peut être suivie puisque l'on ne permet pas la modification *a posteriori* des glyphes, c'est-à-dire, après l'étape de positionnement.

3.4. *Les transformations élémentaires*

Nous allons maintenant comparer les façons dont les règles de transformation des étapes 2, 3 et 4 sont décrites par les trois formats de fonte. Il s'agit des actions suivantes :

– substitution ;

– réordonnancement ;

– insertion

– suppression ;

– positionnement.

3.4.1. *Substitution*

La substitutiton est une fonction τ^{sub} qui remplace une sous-chaîne d'un chaîne de glyphes par un autre sous-chaîne :

$$\tau^{\text{sub}}(g_1 g_2 ... \underbrace{g_i g_{i+1} ... g_{i+k-1}} ... g_m) = g_1 g_2 ... \underbrace{g'_i g'_{i+1} ... g'_{i+l-1}} ... g_{m-k+l}$$

où une sous-chaîne de longueur k, $g_i ... g_{i+k-1}$, est remplacée par une autre de longueur l, $g'_i g'_{i+1} ... g'_{i+l-1}$. Quelques exemples de substitutions : remplacement de glyphes par défaut par des variantes stylistiques, composition de ligatures, remplacement de glyphes arabes par leurs formes contextuelles (initiale, médiane, finale, isolée) appropriées, etc. Lors de l'introduction du concept de graphe de correspondance, nous avons vu qu'*en général* il n'était pas facile de déduire, en connaissant les chaînes d'entrée et de sortie uniquement, quels glyphes exactement ont été remplacés par quels autres. Pour cette raison, la manière dont les opérations de substitution sont définies et implémentées doit permettre que le système puisse toujours suivre quel glyphe a été remplacé par quel autre et utiliser cette information pour la maintenance du graphe C.

3.4.2. *Réordonnancement*

Une propriété de certaines écritures de l'Asie du Sud est la différence entre *ordre logique* des graphèmes (ceci correspond à l'ordre de prononciation) et *ordre visuel* (de l'écriture). Puisque l'ordre des caractères Unicode, par principe, suit l'ordre logique[6] alors que l'affichage des glyphes se fait de façon linéaire, une étape de réordonnancement est nécessaire lors de la composition typographique. Le besoin de réordonner est donc une conséquence de la particularité d'Unicode de s'aligner à l'ordre logique.

De point de vue formel, le réordonnancement peut être considéré comme un cas spécial de la substitution : lorsque l'on remplace une chaîne par une permutation de

6. Unicode admet des exceptions à son principe, comme remarque Patrick Andries (correspondance personnelle) : tel est l'exemple du thaï où certaines voyelles précèdent la consonne dans la chaîne de caractères alors que la voyelle est prononcée après la consonne.

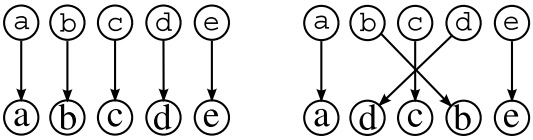


Figure 2. *Le réordonnancement modifie le graphe de correspondances entre chaîne d'entrée et chaîne de sortie.*

ses éléments, le changement d'ordre doit être pris en compte dans la gestion de correspondances entre chaîne d'entrée et chaîne de sortie. Le graphe de correspondances C doit donc être mis à jour comme indiqué dans la figure 2.

3.4.3. *Insertion et suppression*

L'insertion d'un glyphe dans une chaîne peut se faire d'au moins trois manières qui donnent trois résultats différents.

– Comme un cas spécial de la substitution, par exemple, $\tau^{\text{sub}}(ab) = acb$. Ici, le glyphe *c* pourra être associé aux caractères correspondants à *a* et *b*. L'inconvénient de cette solution est que les glyphes autour doivent être précisés explicitement.

– Insertion dans une position spécifique en associant le glyphe à un autre glyphe existant, et à travers lui à un caractère existant. Par exemple, une *voyelle scindée*, phénomène des écritures de l'Asie du Sud comme le bengali, dérive d'un caractère Unicode unique qui se décompose en deux glyphes se plaçant des deux côtés d'une consonne. Cette décomposition peut être implémentée par l'insertion d'un nouveau glyphe qui s'associera au même caractère de voyelle.

– On peut imaginer qu'un glyphe est inséré sans qu'un caractère lui soit associé. Il deviendrait alors nœud à degré zéro (« orphelin ») dans le graphe de correspondances C. Cette approche a ses propres avantages et inconvénients. D'une part, certains ajouts visuels comme le trait de césure ou le *kachidé* arabe, « rallonge » insérée entre deux glyphes pour raisons de justification peuvent être modélisés de manière logique par l'insertion de glyphes orphelins car ils n'appartiennent pas vraiment au contenu textuel et ont été ajoutés pour des raisons liées à une mise en page précise. D'autre part, le logiciel de rendu (tel qu'*Adobe Reader*, les traitements de texte interactifs, etc.) doivent être capables de gérer du texte contenant des glyphes sans caractères, notamment en ce qui concerne l'interactivité : que faire, par exemple, si un tel glyphe est sélectionné individuellement, puis copié-collé ?

La problématique de suppression de glyphe est « symétrique » à celle de l'insertion : les caractères d'entrée dont les glyphes ont été supprimés peuvent se retrouver sans glyphe correspondant dans le texte composé, devenant des nœuds à degré zéro dans le graphe de correspondances C. Ce caractère pourra éventuellement être associé, de manière explicite ou implicite, à un des glyphes restants de la chaîne (le jouxtant ou

non). Dans le cas échéant, la non-association du caractère orphelin aux glyphes de la chaîne résulterait en une perte de contenu textuel lors d'un copier-coller, par exemple.

3.4.4. *Positionnement*

Cette opération diffère de celles présentées plus haut en ce qu'elle ne modifie pas la chaîne de glyphes proprement dite : le positionnement est un ajout de propriétés $\phi_{\Delta x}$ et $\phi_{\Delta y}$ associées aux instances de glyphes (en tant qu'éléments de la chaîne) :

$$g_1 g_2 ... g_m \mapsto \{\phi_{\Delta x}(g_1), \phi_{\Delta y}(g_1)\}, \{\phi_{\Delta x}(g_2), \phi_{\Delta y}(g_2)\}, ..., \{\phi_{\Delta x}(g_m), \phi_{\Delta y}(g_m)\}$$

3.5. *La description des transformations par les trois formats de fonte intelligents*

3.5.1. *AAT et ATSUI*

Le principe de fonctionnement d'AAT et de sa bibliothèque de support ATSUI est que la totalité des opérations, depuis la chaîne de caractères d'entrée jusqu'au texte composé, est décrite par la fonte. Autrement dit, le traitement n'est pas partagé, comme c'est le cas avec OpenType, entre tâches « linguistiques » entreprises par le système d'exploitation et tâches « typographiques » propres à une fonte particulière : c'est cette dernière qui prend en charge les deux tâches. Ainsi, le créateur de la fonte a plus de contrôle mais aussi plus de responsabilités lorsqu'il définit le comportement de sa fonte vis-à-vis d'une écriture.

Dans le modèle de texte du format AAT, l'élément atomique est le glyphe : les transformations d'AAT s'effectuent sans exception sur les chaînes de glyphes. Par conséquent, la toute première opération entreprise par ATSUI est de convertir la chaîne de caractères d'entrée à l'aide de la table CMAP. Ceci permet d'une part une certaine indépendance du codage de caractères utilisé (puisque, après le CMAP, la chaîne de caractères n'est plus exploitée), d'autre part, puisque les fontes AAT se chargent de l'intégralité des opérations de traitement, y compris le traitement propres aux écritures, certaines propriétés liées aux caractères (telles que les propriétés Unicode) doivent forcément être reproduites dans la fonte, associées aux glyphes.

De plus, AAT se distingue de Graphite et d'OpenType par le fait qu'il décrit les règles de transformation (ou de *métamorphose* dans le jargon d'Apple) à l'aide d'automates (ou *transducteurs* pour être précis). Les actions prises en charge par les automates sont le réordonnancement, la substitution et l'insertion de glyphes, la suppression n'étant possible qu'indirectement, en tant qu'effet d'une substitution telle qu'un remplacement de plusieurs glyphes par une ligature. Dans ce dernier cas, les glyphes qui ne sont pas remplacés par le glyphe de ligature deviennent des *glyphes vides* : ce sont des *pseudo-glyphes* dont l'identifiant n'est associé à aucune image réelle. Parmi les actions de positionnement de glyphes, les automates ne gèrent que le crénage. Finalement, les automates interviennent une dernière fois lors de la justification pour classifier les glyphes (nous parlerons de cela dans la section 3.6). Dans la suite, nous ne nous intéresserons pas aux substitutions non conditionnelles qui ne nécessitent pas d'automates, ni au crénage contextuel qui, en revanche, en utilise.

Il faut noter que la puissance transformationnelle des automates d'AAT diffère en pratique de celle des transducteurs finis déterministes « classiques » de la linguistique computationnelle, malgré une similarité superficielle entre les deux concepts. D'un point de vue purement théorique, le modèle de transducteur à états finis suffirait toujours pour décrire tous les types d'automates d'AAT tout simplement parce que le texte traité n'est jamais de longueur infinie : tout texte fini peut être codé par une séquence de transitions entre états du transducteur.

Avant d'aller plus loin, précisons les transformations que les automates d'AAT sont capables d'effectuer sur les chaînes de glyphes (une seule opération par automate parmi les suivantes) :

– *réordonnancement* suivant un nombre restreint de motifs ;

– au plus deux *substitutions contextuelles* $g \mapsto g'$ (un-à-un) par transition entre états : substitution du *glyphe courant*, du *glyphe marqué*, ou des deux à la fois ;

– la substitutition contextuelle d'au plus 16 glyphes par au plus le même nombre d'autres glyphes (pour composer des *ligatures*, entre autres) ;

– l'*insertion* de glyphes, au plus deux par transition (avant ou après le glyphe courant et/ou le glyphe marqué).

Le *marqueur*, utilisé dans le réordonnancement et aussi dans la substitution, sert à marquer un glyphe qui pourra ensuite être remplacé même s'il a déjà été dépassé dans la lecture et si l'automate a depuis traversé de nombreux autres états. La possibilité d'ainsi marquer des glyphes afin d'y revenir ensuite augmente la puissance des automates d'AAT au-delà de celle des transducteurs finis : par exemple, la substitution $a\gamma c \mapsto A\gamma C$ (où γ correspond à une chaîne de glyphes de longueur arbitraire) nécessite que l'automate puisse mémoriser γ jusqu'à après avoir lu c ; cependant, les automates finis ne possèdent pas de mémoire. Il en est de même pour les motifs de réordonnancement tels que $a\gamma c \mapsto c\gamma a$.

Les automates de substitution un-à-un de type $a\gamma c \mapsto A\gamma C$ ne peuvent cependant pas effectuer de substitutions plus complexes telles que $abc \mapsto ABC$ car ils ne sont pas capables de remplacer à la fois a par A, b par B et c par C. Une solution pourrait être d'insérer deux nouveaux glyphes A et B après avoir lu c. Le grand inconvénient de cette approche est qu'elle modifie les correspondances entre glyphes d'entrée et de sortie : au lieu de $a \mapsto A$, $b \mapsto B$, $c \mapsto C$, l'automate ne spécifie qu'une correspondance globale $abc \mapsto ABC$, d'une finesse réduite. Une meilleure solution est de décomposer la règle en plusieurs automates : la paire de règles

$$abc \mapsto XbC$$

$$Xb \mapsto AB$$

réalise en deux étapes la même substitution $abc \mapsto ABC$ tout en gardant les correspondances entre glyphes d'entrée et de sortie. Le prix de la décomposition est, d'une part, la multiplication du nombre d'automates nécessaires et, d'autre part, l'introduction d'un glyphe « magique » pseudo-glyphe que nous avons indiqué par X dans les règles. Ce pseudo-glyphe est un simple indice sans image correspondante (ni d'autres

données associées telles que métriques) qui fait apparition uniquement dans les deux règles indiquées, afin d'établir une connexion entre deux règles et ainsi entre deux automates spécifiques.

La *substitution de ligature*, évoquée en troisième place parmi les quatre type d'automate, a été inventée pour composer des ligatures contextuelles qui s'assemblent à partir de trois glyphes ou plus, avec d'autres glyphes pouvant intervenir entre les composantes de ligature (comme par exemple les voyelles arabes). En réalité, d'après les spécifications disponibles sur le Web (AAT, n.d.), ce type de substitution semble être capable de substituer au plus 16 glyphes en même temps ; ainsi, la règle $abc \mapsto ABC$ devient réalisable par une seule substitution[7]. En général, une substitution de ligature peut être décrite dans la forme

$$\gamma_0 g_1 \gamma_1 g_2 \gamma_2 \cdots g_n \gamma_n \mapsto \gamma_0 g'_1 \gamma_1 g'_2 \gamma_2 \cdots g'_n \gamma_n$$

où $n \leq 16$ et où les γ_i sont des chaînes de glyphes de longueurs arbitraires qui ne seront pas modifiées par la substitution. Le nombre de glyphes g_i est identique au nombre des g'_i ; pourtant, ceci ne devrait pas être le cas lors d'une substitution de ligature où plusieurs glyphes d'entrée sont convertis en un seul glyphe de sortie. Dans ce genre de cas, chaque glyphes superflu est remplacé par un *glyphe supprimé* ; il s'agit de nouveau d'un pseudo-glyphe. Par exemple, une substitution de ligature ffi conforme à la forme de règle ci-dessus s'écrit de la manière suivante :

$$ffi \mapsto ffi\_\_$$

où le symbole « _ » dénote le pseudo-glyphe de suppression.

Le quatrième type d'opération entreprise par les automates est celle de l'insertion de glyphes. L'insertion peut avoir lieu avant ou après le glyphe courant et avant ou après le glyphe marqué. Ainsi, les deux règles

$$\gamma_0 g \gamma_1 \mapsto \gamma_0 \gamma' g \gamma_1$$
$$\gamma_0 g \gamma_1 \mapsto \gamma_0 g \gamma' \gamma_1$$

sont réalisables (γ' est la chaîne de glyphe insérée). L'insertion se fait avant ou après g qui représente le glyphe courant lorsque γ_1 est vide ou le glyphe marqué lorsque γ_1 n'est pas vide. Les nouveaux glyphes de γ' pourront être associés soit à g soit au glyphe courant, c'est-à-dire au dernier glyphe de γ_1 lorsque celui-ci n'est pas vide. Il n'est donc pas possible de définir arbitrairement les correspondances entre les glyphes insérés et les glyphes lus par l'automate.

7. Bien que ce type d'opération s'appelle officiellement *substitution de ligature* dans les spécifications et que les exemples se limitent aux ligatures uniquement (autrement dit, aux substitutions multiple-à-un), les auteurs trouvent que rien n'empêche d'effectuer avec la même opération des substitutions multiple-à-multiple.

Notons enfin que les deux types de substitution (contextuelle et de ligature) effectuent en général des bijections ordonnées (puisque leurs remplacements élémentaires sont toujours un-à-un), sauf dans le cas où des glyphes sont supprimés. Le réordonnancement et l'insertion, par contre, ne respectent pas cette critère.

Après avoir analysé en détail les automates d'AAT, intéressons-nous à la table *Zapf* de ce format qui a pour but de faciliter l'identification et la description des glyphes contenues dans la fonte. Avoir une table dédiée à la description des correspondances entre caractères et glyphes rend AAT unique parmi les technologies de fonte intelligentes. Parmi les informations contenues dans cette table, celles susceptibles de nous intéresser sont :

– la définition des noms de glyphes suivant plusieurs systèmes de nommage en même temps ;

– la chaîne de caractères Unicode qui peut produire le glyphe (il n'est pas possible de définir plusieurs chaînes, le fait que le même glyphe est associé à la fois aux caractères *A latin* et *A cyrillique* ne peut pas être exprimé) ;

– la description textuelle du glyphe, y compris son usage, son histoire, etc. ;

– les fonctionnalités et le contexte qui ont servi à l'obtention du glyphe.

Concernant ce dernier, même si le contexte n'est décrit ici qu'approximativement, il est nettement plus pratique de consulter la table *Zapf* pour obtenir cette information que de devoir la déduire à partir des descriptions des machines à états et des autres règles. (D'autre part, il faut avouer que cette déduction serait un peu moins compliquée à partir de règles déclaratives comme dans OpenType.)

3.5.2. *OpenType et Uniscribe*

Devant la solution très orientée « ingénieur » des machines à états d'Apple, OpenType adopte une méthode de description plus facile à maîtriser par les graphistes et les non-informaticiens en général. Dans OpenType, une transformation est décrite par un ensemble de *règles déclaratives* relativement simples. Deux types d'opérations sont concernées : la substitution et le positionnement de glyphes. Par rapport à AAT, il manque donc le support du réordonnancement (ceci est fait au niveau des caractères par Uniscribe) et de l'insertion-suppression ; en revanche, OpenType est capable d'exprimer le positionnement contextuel de glyphes alors que dans AAT la contextualité est limitée au crénage.

OpenType distingue plusieurs types de *motifs (lookup)* de substitution : substitution un-à-un, un-à-multiple, substitution de ligature, substitution contextuelle et même *contextuelle chaînée*. Cette dernière, la plus générale et puissante (les autres types de règles en forment des sous-ensembles fonctionnels), est de la forme :

$$\tau_F^{\text{sub}}(T_{\text{pre}\,F} \cdot T_F \cdot T_{\text{post}\,F}) = T_{\text{pre}\,F} \cdot T'_F \cdot T_{\text{post}\,F}$$
$$\text{où } T_F = \underbrace{g_{1_1}...g_{1_{N_1}}}_{C_1}\underbrace{g_{2_1}...g_{2_{N_2}}}_{C_2}\underbrace{g_{n_1}...g_{n_{N_n}}}_{C_n},$$
$$T'_F = \underbrace{g_{1_1}...g_{1_{M_1}}}_{C'_1}\underbrace{g_{2_1}...g_{2_{M_2}}}_{C'_2}\underbrace{g_{n_1}...g_{n_{M_n}}}_{C'_n}$$

Cette règle signifie que lorsqu'une série de chaînes $C_1...C_n$ est précédée par un *pré-contexte* T_{pre} et suivie par le *post-contexte* T_{post}, alors elle sera remplacée par une autre série contenant le même nombre de chaînes, $C'_1...C'_n$. Bien évidemment, T_{pre} et T_{post} peuvent être vides, et on obtient une règle de type *contextuel simple* (non chaîné). Par contre, ni les chaînes C_i ni les chaînes C'_i ne peuvent être vides ($N_i > 0, M_i > 0$), il en est de même pour le nombre de séquences ($n > 0$). Par le choix approprié de ces valeurs, on peut obtenir les autres types de règles également : lorsque $(n, N_1, M_1) = (1, 1, 1)$, alors on obtient une substitution simple (un-à-un), avec $(1, > 1, 1)$ une substitution de ligature, avec $(1, 1, > 1)$ une substitution multiple et ainsi de suite. Par contre, puisque aucune de ces valeurs ne peut être zéro, les opérations d'insertion ($N_i = 0$) et de suppression ($M_i = 0$) ne sont pas prises en charge.

Pour quelle raison OpenType partage-t-il le texte T_F en une série de chaînes au lieu de le traiter comme une chaîne de glyphes unique ? Le découpage sert à expliciter la correspondance de glyphes entre T_F et T'_F : par exemple, une substitution

$$g_1g_2g_3 \mapsto g_4g_5g_6$$

peut impliquer de nombreuses correspondances possibles, telles que

$$g_1 \mapsto g_4, g_2 \mapsto g_5, g_3 \mapsto g_6 \text{ ou } g_1 \mapsto g_4g_5, g_2g_3 \mapsto g_6$$

Par contre, le découpage

$$C_1 = g_1, C_2 = g_2g_3, C'_1 = g_4g_5, C'_2 = g_6$$

arrive à indiquer les correspondances avec précision. En revanche, l'ordre des chaînes C_i et C'_i étant toujours identique, les règles OpenType ne sont pas capables d'exprimer le réordonnancement. Les correspondances restent statiques entre C_i et C'_i. Pour cette raison, il revient à la bibliothèque Uniscribe ou à la bibliothèque de support alternative d'effectuer le réordonnancement, indépendamment de la fonte, sur la chaîne de caractères.

Contrairement à AAT où le positionnement contextuel est limité au crénage, les règles de positionnement d'OpenType sont toujours contextualisables. En général, une règle de positionnement contextuel peut être décrite par une règle

$$\tau_F^{\text{pos}}(T_{\text{pre}\,F} \cdot g_1...g_n \cdot T_{\text{post}\,F}) = [\phi_{\Delta x}(1), \phi_{\Delta y}(1)]...[\phi_{\Delta x}(n), \phi_{\Delta y}(n)]$$

On associe donc à chaque glyphe des décalages $(\Delta x, \Delta y)$. OpenType propose également des *ancres d'attache*, une méthode de positionnement plus conviviale, notamment pour positionner les marques sur les lettres de base ou positionner les lettres d'une écriture cursive entre elles.

Le standard OpenType en soi ne propose aucun langage de description, les règles présentées ci-dessus étant en réalité décrites dans un format tabulaire à la TrueType, peu conviviale à comprendre et encore moins à produire. Pour palier cette faiblesse, Adobe commercialise une plate-forme de développement (SDK) qui permet la description de règles dans un langage simple pour les convertir ensuite au format tabulaire exigé par la fonte.

3.5.3. *Graphite*

Alors que le modèle de texte d'OpenType est un modèle mixte où certains traitements s'effectuent sur les caractères et d'autres sur les glyphes, dans le système Graphite, similairement à AAT, l'intégralité du processus de formatage se passe dans le domaine des glyphes auxquels les caractères sont convertis dès la toute première étape, au travers du CMAP.

Dû à ce fait, Graphite a également besoin d'élargir l'interprétation de la notion de glyphe : par le mécanisme de *pseudo-glyphe* (que l'on a déjà vu chez AAT) on crée une entité liée en général à un caractère Unicode mais qui n'a pas forcément d'image graphique associée, ou bien son image ne lui est pas unique. Ainsi, un caractère *liant sans chasse (ZWJ, Zero Width Joiner)*, bien qu'invisible, aura un glyphe associé, et les caractères *A latin* et *A cyrillique* deviendront deux glyphes différents même s'ils correspondent à la même image. Il existe même des pseudo-glyphes sans caractère Unicode associé, tel que le glyphe de début et fin de ligne.

Graphite étend également la notion de propriété : celle-ci (appelée *attribut* dans le jargon de Graphite) a deux interprétations différentes : l'*attribut de boîte* ou *d'instance de glyphe* qui correspond à notre définition de propriété $\phi_F(i)$ liée à une position dans la chaîne et l'*attribut de glyphe* qui est fonction du glyphe même et non pas de son instance : $\phi_F^G(g_i)$. Un exemple d'attribut de glyphe est la chasse, un d'attribut de boîte est le décalage de position. Quant à sa représentation informatique, l'attribut est représenté dans Graphite par une structure de données arborescente.

Le concept d'attribut de boîte est d'une très grande importance car en réalité il élargit le modèle de texte d'Unicode en repoussant les frontières qui définissent les éléments atomiques du texte. Même si ce n'est que dans la phase de composition, les éléments textuels deviennent des structures de données riches, au-delà de simples identifiants d'images avec leurs emplacements sur un système cartésien. Graphite permet aux auteurs de fontes la création de nouveaux attributs de boîte qui peuvent également participer aux transformations, ce qui ouvre la porte vers l'extensibilité du système.

Il faut comprendre que, même si *en interne* Uniscribe et ATSUI permettent eux aussi l'association d'attributs et de structures de données aux glyphes, ce qui dis-

tingue Graphite est l'ouverture du modèle de texte depuis le moteur typographique vers l'extérieur, c'est-à-dire vers les règles de transformation dans les fontes Graphite, désormais capables d'accéder aux attributs de glyphe et de boîte d'une manière claire et transparente. Nous allons voir en section 4.1 que cette ouverture du modèle peut être étendue même à l'extérieur des fontes, jusqu'au texte d'entrée et de sortie, par le concept de *textèmes*.

Le contenu d'une fonte Graphite, y compris les règles de transformation, est décrit au travers d'un *langage de description de glyphes* (*GDL*). Les transformations comme le réordonnancement, la substitution ou le positionnement de glyphes sont exprimées par une notation proche de la syntaxe traditionnellement utilisée en linguistique. GDL suit l'idée des règles déclaratives d'OpenType, sauf que cette fois-ci il s'agit d'un vrai langage doté de variables, de listes etc. Aussi, la puissance des règles de Graphite dépasse celles d'OpenType : alors que les règles OpenType sont exprimées par l'intermédiaire d'un répertoire limité de motifs (*lookup*), chacun correspondant à une fonctionnalité de base (crénage, substitution de ligatures, etc.), Graphite à la fois formalise et étend la description de règles à travers son langage GDL. Celui-ci permet de créer des règles qui prennent en compte non seulement des identificateurs de glyphes mais aussi les attributs de glyphe et de boîte, ces derniers étant même modifiables : un vrai gain de puissance par rapport à OpenType où la dépendance d'une action de son contexte est exprimée uniquement par une succession d'identificateurs de glyphes.

Appelons *élément textuel* e_i, l'union d'un indice de glyphe g_i et d'un sous-ensemble de paires clé-valeur $\phi_j(i) = v_j$ où ϕ_j est l'attribut de boîte j du glyphe g_i et sa valeur est v_j.

$$e_i = g_i \cup \{\cup_{\forall j}(\phi_j(i) = v_j)\}$$

Soit e'_i un élément textuel obtenu comme résultat de l'application de la règle, identique en structure à e_i avec une différence près : les $\phi'_j(i) = v_j$ de ce dernier dénotent des affectations au lieu de conditions. Alors une règle Graphite s'écrit de la manière suivante :

$$e_1 ... e_n \mapsto e'_1 ... e'_n \,/\, e^c_1 ... e^c_m$$

où $e_1 ... e_n$ est une chaîne d'éléments textuels remplacés par $e'_1 ... e'_n$ pourvu qu'ils apparaissent dans la chaîne de contexte décrite par $e^c_1 ... e^c_m$, $m \geq n$. Cette forme unique de règle peut servir à décrire tout type de transformations, y compris substitutions, insertions, suppressions, réordonnancements et positionnements. Étant donné que le nombre de glyphes transformés, n, doit être égal dans les motifs d'origine et de remplacement, le symbole « _ » a été introduit pour servir de point d'insertion lorsqu'il apparaît à gauche de la flèche et de suppression lorsqu'il apparaît à droite. Dans la chaîne de contexte, il joue un troisième rôle : il sert à indiquer la position des glyphes modifiés. Ainsi, par exemple, la règle

$$f\,i \mapsto fi\_ \,/\, a\ \_\ \_\ b$$

décrit une substitution $a\,f\,i\,b \mapsto a\,fi\,b$. Le réordonnancement nécessite l'introduction d'un nouvel élément de langage, la *référence* (« @ ») : la règle $a\,b\,c \mapsto$ *@3 @2 @1* invertit l'ordre des trois glyphes et à la fois indique la correspondance entre glyphes

d'entrée et glyphes de sortie. Graphite utilise le terme *association* pour référer à ce concept. Dans la plupart des cas, les associations s'expriment implicitement, comme dans le cas de substitutions simples, de ligature ou de réordonnancement simple comme dans l'exemple donné. Elles peuvent cependant aussi être définies explicitement lorsque ceci est nécessaire, comme dans le cas des *voyelles scindées* dont le fragment inséré en tant que nouveau glyphe devant la consonne doit être associé au caractère correspondant à la voyelle qui se trouve derrière. Il est à la charge du moteur Graphite de prendre en compte et de gérer les deux types d'associations par la maintenance d'un graphe de correspondances. C'est donc la gestion plus puissante des associations qui permet à Graphite de réordonner alors qu'OpenType n'en est pas capable.

3.5.4. *Comparaison de puissance descriptive entre règles de transformation*

Après avoir décrit rapidement les transformations que les trois formats de fonte sont capables d'effectuer, une question se pose naturellement : est-ce qu'un format parmi les trois est plus puissant que les autres ? Existe-t-il des transformations supportées par certains formats et pas par d'autres ?

Nous proposons (page 190) un tableau récapitulatif des différences majeures entre les trois formats. Globalement, c'est Graphite qui possède la gamme transformationnelle la plus large : grâce à la possibilité de gérer manuellement les associations entre glyphes avant et après transformation, Graphite arrive à permettre réordonnancement, insertion et suppression de glyphes sans limitations importantes. De plus, par rapport aux actions permises par AAT ou par OpenType, les règles de transformation de Graphite sont capables de prendre en compte mais aussi de modifier les propriétés (attributs) associées aux glyphes et à leurs instances. AAT, à son tour, est capable d'accepter à l'aide de ses automates des répétitions arbitraires de motifs de glyphes, par exemple, ab^*c (un glyphe a suivi par un nombre quelconque de b puis par un c) : OpenType et Graphite auraient besoin de préciser autant de règles distinctes que de répétitions possibles du glyphe b. AAT permet également de définir l'ordre du parcours de la chaîne de glyphes (ordre logique ou son inverse) séparément pour chacun de ses automates. Quant à OpenType, il a été conçu avec l'idée de limiter la capacité transformationnelle (en évitant entre autres les problèmes dus au maintien des correspondances entre caractères et glyphes), en partie afin de faciliter la création de règles, en partie pour déplacer une part de l'intelligence depuis la fonte vers une bibliothèque centralisée.

3.6. *Le paragraphage*

Les trois formats de fonte vont assister le moteur typographique lors de la justification de paragraphes : des paramètres qui gouvernent le changement de l'espacement inter-mots et inter-lettres, des glyphes extenseurs (kachidé arabe), des variantes de glyphes à chasse variable, etc. Il appartient au client (au service typographique ou à l'application) d'exploiter les informations fournies.

	AAT	OpenType	Graphite
Réordonnancement	Oui, suivant un ensemble restreint de motifs	Non	Oui, sans limitation
Insertion	Oui, en ou hors contexte. Le glyphe inséré peut être associé à un glyphe voisin ou éloigné, avec certaines limitations.	Non (uniquement en tant que substitution multiple)	Oui, le glyphe inséré peut être associé à tout autre glyphe sans limitation
Suppression	Oui	Non (uniquement en tant que substitution)	Oui
Substitution	Un-à-un et multiple-à-un uniquement	Tous types (multiple-à-multiple)	Tous types (multiple-à-multiple)
Motifs contextuels de longueurs arbitraires	Oui	Non	Non
Positionnement contextuel	Uniquement pour le crénage (horizontal ou vertical)	Oui	Oui
Prise en compte et modification des attributs par les règles de transformation	Non	Non	Oui
Gestion de correspondances entre caractères et glyphes	Implicite pour la plupart des règles ; explicite pour l'insertion ; indications globales par la table *Zapf*	Implicite (les règles sont limitées afin d'éliminer toute ambiguïté)	Implicite ou contrôle manuel très fin
Ordre de parcours de la chaîne de glyphes	Choisi librement	Toujours en ordre logique (sauf substitution contextuelle chaînée inverse, dont l'utilité est en pratique restreinte à quelques écritures précises comme le *nastaliq*)	Substitutions en ordre logique, positionnements en ordre graphique

Tableau 1. *Comparaison d'AAT+ATSUI, d'OpenType+Uniscribe et de Graphite.*

En même temps, paradoxalement, l'« intelligence » incorporée dans les nouveaux formats de fonte peut parfois augmenter considérablement la complexité du paragraphage. Plus concrètement, on pense ici aux effets secondaires dus à la coupure de la chaîne de glyphes en fin de ligne : une coupure peut introduire de nouveaux glyphes (tel qu'un trait de césure, voire de nouvelles lettres comme dans le cas du mot hongrois *lánnyal*↦*lány-nyal*) ou en modifier (lorsque la ligature *ffi* est brisée en un glyphe *f* et une ligature *fi* dans le mot *dif-ficile*).

Le problème n'est pas celui de la réalisation technique des phénomènes présentés (césure non-standard, décomposition et refonte des ligatures) : ceux-ci ont été reconnus et résolus depuis un quart de siècle, notamment dans le système TEX. La véritable nouveauté apportée par les fontes intelligentes est le potentiel de formuler des transformations contextuelles : chaque glyphe et ses propriétés peuvent dépendre de leur entourage, voire d'autres glyphes plus éloignés. Théoriquement, la longueur d'un contexte n'est limitée que par les sauts de paragraphe et de ligne. Les transformations contextuelles sont d'abord calculées sur des chaînes de glyphes ininterrompues qui correspondent à des paragraphes entiers et la coupure de celles-ci en lignes n'arrive qu'une fois les transformations terminées. Or, la modification de la chaîne de glyphes par une coupure de ligne introduite peut altérer le contexte qui à son tour peut entraîner une régression des calculs effectués jusque-là. Dans un cas extrême, par un effet de réaction en chaîne, les changements provoqués par les coupures peuvent traverser le paragraphe entier.

Ce que l'on appelle souvent dans le jargon des développeurs de fontes intelligentes *règle* ou *transformation contextuelle* peut être modélé mathématiquement par une fonction τ qui n'est pas un homomorphisme : si le transformé du texte T_1 dépend du texte T_2 qui le suit, alors forcément

$$\tau(T_1 \cdot T_2) \neq \tau(T_1) \cdot \tau(T_2)$$

Bien évidemment, la même affirmation peut être faite lorsque c'est T_2 qui dépend de T_1. (Il faut noter que suivant cette définition la composition d'une ligature est également une transformation contextuelle car $\tau(\mathrm{fi}) \neq \tau(\mathrm{f}) \cdot \tau(\mathrm{i})$.)

Pour revenir au problème de paragraphage, remarquons que la coupure d'un paragraphe en lignes correspond à une division d'une chaîne T en sous-chaînes $T_1, ..., T_n$ (de longueurs supérieures à zéro). Puisque la fonction de formatage f_{form} n'est en général pas un homomorphisme à cause de règles contextuelles, on ne peut plus supposer que

$$f_{\text{form}}(T) = f_{\text{form}}(T_1) \cdots f_{\text{form}}(T_n)$$

ni que la longueur de l'image de la sous-chaîne T_i dans $f_{\text{form}}(T)$ est égale à la longueur de $f_{\text{form}}(T_i)$. La conséquence peut être une régression des calculs car les lignes obtenues par la découpage de $f_{\text{form}}(T)$ ne seront pas toujours cohérentes avec les règles de la fonte.

La solution brute à ce problème serait de réappliquer toutes les règles de transformation (substitutions, positionnements) à chaque paragraphe candidat. C'est le seul

moyen — du moins dans le cadre des modèles classiques de texte — d'assurer la cohérence du résultat de la mise en page par rapport aux règles définies par les fontes. Les effets d'un tel algorithme sur la performance du système de mise en page seraient cependant dévastateurs : déjà, pour une justification simple (égalisation par ligne) de type *first* ou *best fit*, la complexité calculatoire augmente de $\mathcal{O}(n)$ à $\mathcal{O}(n^2)$.

Il existe cependant des approches plus prudentes dont le choix peut être fait selon la difficulté d'implémentation et les contraintes de temps de calcul :

1) Une solution primitive mais efficace est celle des systèmes de traitement de texte WYSIWYG tels que Microsoft Word ou OpenOffice.org qui ont préféré renoncer complètement à l'idée d'employer des règles contextuelles voire même des ligatures pour les écritures latine-grecque-cyrillique. Sans contextualité ni ligatures, c'est-à-dire en se limitant aux homomorphismes comme transformations, le problème ne se pose plus.

2) Cette simplification brutale n'est cependant pas admissible pour certaines autres écritures où les ligatures et la contextualité ne peuvent pas être négligées sous prétexte de n'être que des concepts purement esthétiques. La solution proposée par certains systèmes de plus haute qualité est d'appliquer dans un premier temps toutes les opérations sur le paragraphe entier, puis de trouver les points de coupure, et finalement de refaire une seconde fois tous les calculs sur chaque ligne séparément. Cette méthode peut être appliquée aussi bien avec un algorithme d'égalisation par ligne qu'avec l'égalisation sur paragraphe entier. L'imperfection de l'approche est qu'il n'y a pas garantie que la longueur de la ligne composée individuellement soit égale à la longueur du morceau de texte correspondant lorsqu'il n'y a pas de coupures.

3) On montrera qu'il est possible d'éliminer l'imperfection de l'approche précédente sans devoir recalculer le paragraphe entier pour chaque coupure potentielle, à l'aide du concept de *lien*. La solution sera présentée dans la section 4.2.2.

4) Finalement, la *composition dynamique* proposée en section 4.4 est une approche alternative qui évite le problème complètement par la prise en compte préalable de toutes les coupures potentielles.

3.6.1. *La méthode de paragraphage d'AAT*

ATSUI implémente son propre algorithme de paragraphage. Il s'agit d'une simple justification de lignes, l'égalisation de paragraphes entiers n'étant pas prévue. La fonte fournit à l'algorithme les informations suivantes (utilisées dans cet ordre) :

1) une *classification* des glyphes de la ligne à égaliser, exécutée par une machine à états — la classe peut donc dépendre du contexte tel que la position dans la ligne ;

2) les *facteurs* de dilatation-compression associés aux espaces sur les deux côtés de chaque glyphe ;

3) pour chaque glyphe, sa priorité lors de la participation à la justification (« kachidé », « espaces inter-mots », « interlettrage », « aucune modification », dans cet ordre) ;

4) au cas où les compressions ou dilatations ainsi définies ne seraient pas suffisantes, dans l'étape de *post-composition*, la modification de la chaîne de glyphes est

un dernier recours : il s'agit d'insérer des nouveaux glyphes, de déformer leur géométrie, de les répéter...

Cette approche offre de nombreuses possibilités d'adapter les lignes aux coupures mais en même temps elle laisse très peu de liberté par rapport aux priorités établies entre les différentes stratégies. Ainsi, il est supposé que dans la typographie arabe le kachidé a toujours priorité sur le changement des espaces inter-mots et que la modification de la chaîne de glyphes (décomposition des ligatures, etc.) n'est qu'un dernier recours, appliqué uniquement si les autres opérations sont insuffisantes. En d'autres termes, la *stratégie* de justification est pré-établie par ATSUI, la fonte ne fait qu'y fournir les données nécessaires.

3.6.2. *La table de justification d'OpenType*

OpenType a prévu une table nommée JSTF *(justification)* pour assister le moteur de paragraphage en lui proposant des alternatives dans la ligne à justifier. Comme dans le cas d'AAT, il est implicitement supposé que l'on traite une seule ligne à la fois (algorithme *best-fit*), l'égalisation sur paragraphe entier n'est pas prévue, même si en réalité rien ne nous empêche d'exploiter la table JSTF à ce but. Contrairement à AAT, Uniscribe n'offre aucun support algorithmique pour la justification, il appartient entièrement à l'application d'implémenter cette fonctionnalité.

Fidèle à la nature déclarative d'OpenType, la table JSTF contient une liste prioritisée de *suggestions* de modifications à la ligne en cours de traitement. Les actions possibles ne sont qu'un sous-ensemble des règles définies dans les tables GSUB et GPOS telles que décomposition de ligatures, remplacement de glyphes par des variantes, crénage (ou son absence), élargissement d'espaces. La grande différence par rapport à AAT est que ce dernier n'applique les substitutions de glyphe que dans la phase finale de post-composition, uniquement si les interventions plus « standard » ne se sont pas avérées suffisantes. OpenType, par contre, peut choisir les priorités avec une liberté totale ; ceci dit, la prise en compte ou non de ces suggestions sera toujours fonction des capacités et des décisions du moteur typographique.

3.6.3. *Graphite*

Au moment de la rédaction de l'article, Graphite n'offrait pas encore de support pour la justification digne de ce nom. Les développeurs de Graphite semblent être en train de travailler sur cet aspect.

3.7. *Conclusion*

Nous avons présenté le modèle de texte adopté par les services typographiques de la majorité des systèmes d'exploitation et des logiciels PAO. Au travers d'une séparation entre contenu (caractères) et présentation (glyphes), ce modèle arrive à représenter les particularités de la grande majorité des écritures du monde avec une précision respectable. Nous avons néanmoins observé que certains aspects de l'écrit

ne trouvaient pas facilement leur place entre ces deux couches d'informations. Quant aux fontes intelligentes, elles sont désormais capables de décrire des correspondances complexes (contextuelles) entre caractères et glyphes. Ces correspondances, souvent dues aux particularités de certaines écritures et conventions typographiques, peuvent rompre la bijection ordonnée entre la chaîne de caractères et sa chaîne de glyphes correspondante. Par conséquent, le modèle de texte doit être élargi par la gestion de correspondances entre les deux couches d'informations, un aspect qui devient crucial aussi bien pour les usages du texte formaté faisant appel à l'interactivité avec l'utilisateur que pour une exécution correcte de certaines opérations de formatage telles que le paragraphage. Jusqu'ici, avec les techniques actuelles, il s'est avéré particulièrement difficile d'effectuer cette dernière opération tout en maintenant la cohérence du texte avec les règles typographiques définies par les fontes intelligentes.

Dans la quatrième, dernière section de l'article, nous allons proposer un nouveau modèle de texte ainsi qu'une nouvelle approche, plus générale, à l'opération de paragraphage.

4. Un nouveau modèle de texte et une nouvelle approche au paragraphage

Notre but est de trouver une meilleure solution aux problèmes évoqués dans la section précédente — liés aux faiblesses du modèle d'Unicode et à la relation entre fontes intelligentes et paragraphage — mais aussi de mieux exploiter les informations fournies par les fontes intelligentes. Nous allons procéder par l'intermédiaire de trois nouveaux concepts :

1) le *textème*, introduit par les auteurs dans les articles (Haralambous *et al.*, 2005a) et (Haralambous *et al.*, 2005b) (sous le nom de *signe* dans ce dernier) ;

2) l'extensibilité du processeur typographique : il s'agit de réaliser le processus de composition par l'exécution consécutive de *modules* où chaque module implémente une fonctionnalité indépendante et de nouveaux modules peuvent être introduits librement dans le processeur ; voir les articles (Haralambous *et al.*, 2005a) et (Haralambous *et al.*, 2006) ;

3) la *typographie dynamique*, un concept nouveau qui consiste à composer non pas une seule mais plusieurs variantes du même paragraphe dont l'optimale pourra être choisie suivant un ensemble de critères bien plus riche que le simple choix de l'espacement inter-mots.

4.1. *Une courte introduction à la notion de textème*

Le textème est, dans notre modèle, la nouvelle unité atomique de texte, censée remplacer dans ce rôle à la fois le caractère et le glyphe. Un textème consiste en un ensemble de *propriétés*. Une propriété est une paire *clé-valeur*. Les deux propriétés principales dans le textème sont la propriété *caractère* et la propriété *glyphe*. On retrouve donc ces deux informations également dans le nouveau modèle proposé, désor-

mais associées dans une seule unité atomique textuelle. D'autres propriétés peuvent être ajoutées dans un textème : propriétés linguistiques (césure potentielle ou effective, forme contextuelle, racine sémitique dans les mots arabes ou hébraïques, composants des caractères chinois, etc.), propriétés typographiques (position du glyphe, interlettrage, chasse, insécabilité, etc.) ou autres. Il est important à noter que l'ensemble des propriétés potentielles est ouvert.

Formalisons : en reprenant la définition de *propriété* de la section 3.1, le textème t est défini comme suit :

$$t = \left\{ \bigcup_j \phi_j \right\}$$

où deux propriétés d'intérêt particulier sont la propriété de caractère ϕ_C dont la valeur est un élément d'un codage de caractères tel que $\mathcal{R}_{\text{uni}}$ et la propriété de glyphe ϕ_G dont la valeur identifie un glyphe donné d'une fonte donnée.

Les textèmes rapprochent les mondes de *texte brut* (tel que le code HTML) et de *texte formaté* (tel qu'un document PDF) : la différence entre les deux, du moins au niveau micro-typographique, est dans le nombre de propriétés incluses dans les textèmes. Le texte brut consistera de textèmes ne contenant que très peu de propriétés autres que le caractère alors que le texte formaté contiendra des propriétés riches. Le processus de composition de texte n'est donc que l'enrichissement de textèmes par propriétés.

En d'autres termes, nous introduisons un nouveau modèle de texte *unique*, basé sur les textèmes, où aucune distinction n'est faite entre texte composé de caractères Unicode (T_{uni}), texte composé de glyphes (T_F) et texte formaté (T_{form}). Les trois concepts peuvent être modélisés par la simple définition de *texte* :

$$T = t_1 t_2 ... t_n \text{ où } t_i \text{ est un textème.}$$

Ce modèle simple parvient donc à représenter à la fois le contenu textuel et le texte formaté ainsi que les correspondances caractère-glyphe lorsque les deux chaînes sont en bijection ordonnée. Le cas contraire sera couvert par l'extension du modèle de textème par la notion de *dépendance* (section 4.2.2).

Le manque de force descriptive d'Unicode et son incohérence, évoqués à la section 2, peuvent être également remédiés par la possibilité d'associer au code de caractère des propriétés qui restent orthogonales entre elles et qui n'interfèrent pas avec la chaîne de texte. Cette manière d'enrichir le texte est souvent préférable au balisage (qui n'est pas dans toutes les situations séparable du contenu) et à la création de caractères spéciaux dans Unicode. Un simple exemple est le concept d'*espace* pour lequel Unicode contient au moins une vingtaine de caractères différents (espace « ordinaire », espace insécable, espace étroite, espace insécable et étroite, espace mathématique, espace idéographique, une douzaine d'espaces typographiques, etc.). Il est évident que plusieurs de ces paramètres correspondent à des sémantiques indépendantes : chasse typographique, propriété linguistique, sécabilité, etc. Puisque ces sémantiques peuvent se combiner (espace étroite insécable), il est plus naturel d'attacher ces informations

aux caractères en tant que propriétés de textème que de créer un nouveau caractère Unicode pour chaque combinaison.

Le lecteur a certainement remarqué la similarité des textèmes aux *boîtes* de Graphite, et des propriétés de textème aux *attributs de boîte*. Il s'agit effectivement de concepts analogues : permettre l'enrichissement de texte par une quantité de données de types différents. Les deux modèles permettent également la définition de nouvelles attributs-propriétés, ainsi qu'une structuration hiérarchique. Il existe pourtant une différence fondamentale entre les deux : les attributs de boîte sont des structures de données utilisées *en interne* par Graphite, dans la description des règles et dans les calculs. Les textèmes ont un cycle de vie plus long, ils sont les unités atomiques de texte en général, aussi bien en entrée qu'en sortie. Une autre différence théorique est qu'une chaîne de textèmes contient à la fois les caractères d'origine et les glyphes composés alors que les boîtes de Graphite se réfèrent uniquement aux glyphes, même si leurs attributs peuvent également contenir des informations liées aux caractères.

Pour plus d'informations sur la définition et sur l'intérêt de la notion de textème, voir (Haralambous *et al.*, 2005b) et (Haralambous *et al.*, 2005a).

4.2. *Textèmes et fontes intelligentes*

Nous avons présenté comment le processus de composition peut être interprété comme l'enrichissement de textèmes par propriétés. Par conséquent, notre modèle doit être capable de s'adapter aux transformations décrites dans les fontes intelligentes ou à celles effectuées par les services typographiques comme Uniscribe. Notre intérêt n'est bien évidemment pas de créer un nouveau format de fonte ni un nouveau système de description de transformations, cette fois-ci basé sur les textèmes. Au contraire, ce qui nous intéresse est l'interaction du texte — et par conséquent du textème qui en est l'élément constitutif — avec les transformations que définissent les fontes intelligentes.

4.2.1. *Substitution et positionnement de glyphes*

Le défi est le suivant : il faut inclure dans notre chaîne de textèmes, représentant le texte d'un paragraphe, les informations provenant de la fonte intelligente : glyphes, leurs positions, d'autres propriétés mais aussi les correspondances entre glyphes et leurs caractères associés.

Pour les informations de positionnement, la solution est simple et élégante : nous allons définir des propriétés de textème Δx et Δy (décalages horizontal et vertical) associées au glyphe qui représentent sa position relative à celle par défaut. Les règles de positionnement définies par les fontes n'auront qu'à modifier ces valeurs.

La substitution, réordonnancement, insertion et suppression de glyphes est un problème plus difficile car ces opérations peuvent rompre la bijection ordonnée entre chaîne de caractères et chaîne de glyphes alors que le modèle de textème tel qu'il a

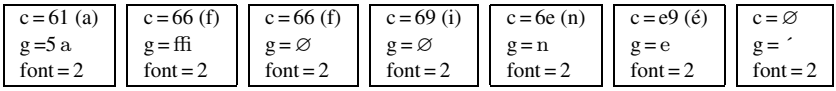


Figure 3. *Substitutions multiple-à-une et une-à-multiple implémentées sur le mot* affiné*, représenté par une chaîne de textèmes.*

été présenté ci-dessus semble être basé sur une supposition implicite de cette particularité. Il faut donc adapter notre modèle aux transformations qui ne réalisent pas une bijection ordonnée, telles que réordonnancement, insertion, suppression et substitutions un-à-multiple, multiple-à-un et multiple-à-multiple.

Pour ce faire, nous allons d'abord introduire le concept de *propriété vide* : il s'agit tout simplement d'une propriété dont la valeur est une valeur spéciale « vide » que l'on dénotera par le symbole « ∅ ». Un *textème sans glyphe* est un textème dont la propriété glyphe est vide ($\phi_G(t) = \varnothing$) mais la propriété caractère ne l'est pas. Ce type de textème peut être considéré comme un bon modèle de l'élément constitutif du texte brut. On définit également le *textème sans caractère* dont la propriété caractère est vide ($\phi_C(t) = \varnothing$) mais la propriété glyphe ne l'est pas. Un exemple intuitif d'un textème sans caractère est le *trait de césure* : il s'agit d'un symbole qui apparaît uniquement dans le document formaté et son rôle est purement présentationnel. Le trait de césure ne fait pas partie du contenu textuel et par conséquent, le fait qu'un mot est coupé ne devrait généralement pas interférer avec les opérations de recherche ou de copier-coller. Il est donc logique que ce glyphe n'ait pas de caractère correspondant.

La solution proposée est illustrée en figure 3. Nous allons reprendre l'idée de textème sans glyphe et textème sans caractère. L'exemple qui illustrera nos propos dans le reste de l'article est très simple : il s'agit du mot *affiné*. Dans ce mot, les trois glyphes *f f* et *i* sont remplacés par une seule ligature (substitution multiple-à-un), le glyphe de ligature sera stockée dans le textème qui contient le premier caractère correspondant alors que les textèmes suivants contiendront des glyphes vides. L'opération inverse est illustrée sur le caractère *é*, représenté par la paire de glyphes *e* + *accent aigu* (substitution un-à-multiple) : ici, on introduit un textème sans caractère. Dans cette figure, « $c =$ » signifie propriété de caractère, ϕ_C, et « $g =$ » signifie propriété de glyphe, ϕ_G.

La solution donnée ci-dessus arrive à représenter les substitutions un-à-multiple et multiple-à-un ; cependant, rien n'indique que désormais une relation plus forte existe entre certains textèmes. En effet, le glyphe introduit dans le deuxième textème représente à la fois les deuxième, troisième et quatrième caractères, et le caractère de l'avant-dernier textème correspond à la fois aux glyphes avant-dernier et dernier. En d'autres termes, la perte de la bijection ordonnée crée des *dépendances* entre textèmes, plus précisément, entre valeurs de certaines propriétés de textèmes. Ainsi, on dit que le glyphe de ligature *ffi dépend* des deux glyphes vides qui le suivent (et vice versa) car désormais ces trois valeurs ne peuvent être modifiées indépendamment. Pour des raisons de simplicité, nous dirons que *deux textèmes sont en dépendance* au lieu de dire qu'une dépendance existe entre deux propriétés de deux textèmes.

Des relations de dépendance (que nous allons prochainement définir de manière plus précise) apparaissent plus souvent que l'on ne pourrait le croire :

– chaque glyphe est en dépendance de son caractère car le changement de ce dernier implique le changement du premier ;

– une propriété de crénage dépendra forcément des deux glyphes voisins ;

– un glyphe dépend d'autres glyphes avoisinants lorsqu'il a été défini par une règle contextuelle d'une fonte intelligente ;

– l'existence d'une (propriété de) césure dépend des caractères du mot entier ;

– et ainsi de suite.

On voit ainsi que les correspondances entre caractères et glyphes peuvent aussi être considérées comme des dépendances car les glyphes dépendent des caractères. On voit également que l'existence de dépendances entre éléments d'information constituant un texte est donc un phénomène bien fréquent qui n'est pas particulier au modèle de textème : les exemples cités ci-dessus sont valables pour tout texte formaté, quel que soit le modèle.

L'existence de dépendances a une grande importance pratique due au fait que le processus de formatage f_{form} se divise en plusieurs étapes (voir section 3.3) qui à leur tour se décomposent en de nombreuses transformations élémentaires. Ces transformations doivent néanmoins respecter la contrainte de produire des résultats cohérents avec l'intégralité des informations associées au texte. Par exemple, après avoir calculé le crénage entre deux glyphes, la modification ultérieure d'un de ces glyphes entraînera l'incohérence des données qui ensuite ne pourra être corrigée que par recalcul du crénage.

Les modèles de texte d'AAT et ATSUI, d'OpenType et Uniscribe et celui de Graphite prennent une approche préventive vis-à-vis de la gestion de dépendances. D'une part, ils imposent des limitations par rapport à l'ordre des opérations admises (les substitutions doivent toujours précéder les positionnements) afin de réduire le nombre de dépendances pendant les calculs. D'autre part, les transformations elles-mêmes sont limitées à quelques cas simples afin de permettre une gestion automatique des correspondances entre caractères et glyphes (qui sont donc aussi une forme de dépendance) derrière les coulisses. Ainsi, les fontes AAT restreignent les actions de leurs automates à quelques transformations bien précises et OpenType ne permet ni réordonnancement, ni insertion, ni supression. Le système Graphite impose relativement peu de restrictions mais en revanche requiert parfois une résolution explicite des correspondances entre glyphes d'entrée et glyphes de sortie par le mécanisme d'associations.

4.2.2. *Les dépendances et leur gestion dans le modèle de textème*

Il est temps de donner une définition plus précise de la notion de dépendance : nous allons dire que la valeur v_A de la propriété $\phi_A(i)$ du textème t_i dépend de la valeur v_B de la propriété $\phi_B(j)$ du textème t_j ($i = j$ est possible) si l'égalité $\phi_B(j) = v_B$ a

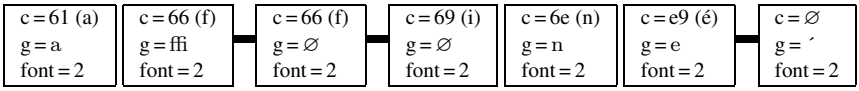


Figure 4. *Les dépendances (ligature et décomposition) dans le mot* affiné *sont représentées par des liens.*

contribué au calcul de $\phi_A(i) = v_A$. En d'autres termes, $\phi_A(i)$ *est fonction de* $\phi_B(j)$. Nous allons exprimer cette relation de dépendance par :

$$\{\phi_B(j) = v_B\} \rightarrow \{\phi_A(j) = v_A\}$$

La relation « $\rightarrow$ » est associative et non commutative, même si souvent les dépendances sont bidirectionnelles, comme entre la ligature *ffi* et les glyphes vides.

Lorsqu'une valeur de propriété dépend d'une autre, la modification de celle-ci doit forcément entraîner la modification de la valeur dépendante. Cependant, la modification de la valeur de propriété n'est pas le seul phénomène qui peut entraîner une réévaluation ; en voici quelques autres :

1) la suppression d'une propriété (voire d'un textème entier) dont dépend une autre propriété ;

2) un saut de ligne qui brise la chaîne entre deux textèmes en dépendance ;

3) l'introduction d'un nouveau textème entre deux textèmes en dépendance.

Nous allons proposer deux approches différentes à la représentation des dépendances dans le cadre du modèle de textèmes. La première établira des *liens* entre textèmes, la seconde décrira les dépendances au travers de formules logiques simples. Chacune des deux solutions a sa propre utilité : la première est plus adaptée au paragraphage classique (que l'on appelle *statique* dans cet article) alors que la seconde est souhaitable dans le cas de la *typographie dynamique*, présentée dans la section 4.4. Les deux approches ont cependant un intérêt et peuvent être implémentées pour les modèles de texte autres que celui des textèmes, y compris les modèles classiques présentés à la section 3.

L'approche par liens est illustré en figure 4. Il s'agit du même mot *affiné*, sauf que cette fois-ci les textèmes dépendants sont *connectés*. La règle est simple : s'il existe une dépendance entre deux valeurs de propriétés de deux textèmes, alors ceux-ci sont connectés par un lien. Si ces deux textèmes ne sont pas voisins, alors tous les textèmes entre les deux seront connectés. Il est important à noter que le lien existe au niveau des textèmes et non pas au niveau des propriétés : ceci permet une simplification considérable du modèle qui reste néanmoins fonctionnel. En effet, les liens servent à *indiquer l'existence* d'une dépendance sans la préciser. Ainsi, une modification de glyphe dans un textème contribuant à un lien indiquera au moteur typographique une régression potentielle qui pourrait entraîner le recalcul des glyphes en question. Nous arrivons ainsi à une solution au problème de paragraphage dont nous avons extensivement parlé dans section 3.6 : la présence d'un lien entre deux textèmes indiquera qu'une coupure de

ligne entre les deux entraîne le recalcul des glyphes et de leurs propriétés sur les deux côtés. La longueur du lien servira à limiter ce calcul aux textèmes réellement affectés au lieu de recalculer le paragraphe entier.

Les liens peuvent être représentés par deux propriétés de textème de valeur binaire : lien vers la gauche et vers la droite. Il n'est donc pas nécessaire de modifier la structure du textème. Cette solution permet d'ailleurs de raffiner notre modèle de liens afin de prendre en compte la non-commutativité de la relation de dépendance. Lorsque le textème t_1 définit un lien vers le textème t_2 mais le contraire n'est pas vrai, alors il s'agit d'un *lien orienté*. Imaginons maintenant que dans la chaîne $t_1 \leftarrow t_2 \rightarrow t_3$ il existe deux liens orientés comme indiqués par les flèches. Ceci signifie que t_1 et t_3 dépendent de t_2. On peut déduire à partir des orientations données qu'il n'y a pas de dépendance entre t_1 et t_3 : ainsi, ni la modification de t_1 ni celle de t_3 n'entraînera un recalcul superflu des textèmes connectés.

La seconde approche à la représentation des dépendances nécessite une légère extension du modèle de textèmes. On introduit un paramètre de *condition* pour chaque propriété qui participe à une dépendance. Cette condition servira à décrire cette dépendance à l'aide d'une formule logique. Dans le plus simple des cas (qui est aussi le plus courant), la formule consistera en une seule variable logique dont le nom servira comme identifiant unique de la dépendance et sa valeur servira comme *sélecteur* : lorsqu'elle est fausse, la propriété correspondante est désactivée dans le textème (comme si elle n'était pas là). Dans le reste de l'article, nous allons représenter les variables — que nous appellerons *variables de condition* — par des α_i où l'unicité est assurée par un indice i strictement croissant.

Comment représenter une dépendance à l'aide de variables de condition ? Prenons toujours le même exemple du mot *affiné*. Dans la chaîne de textèmes

c = 66 (f) font = 2 g = f g_2 = ffi [$\neg\alpha_1$]	c = ∅ font = 2 g = - g_2 = ∅ [$\neg\alpha_1$] brk = 1 [α_1]	c = 66 (f) font = 2 g = f g_2 = ∅ [$\neg\alpha_1$]	c = 69 (i) font = 2 g = i g_2 = ∅ [$\neg\alpha_1$]

nous avons inséré un textème de césure entre les deux caractères *f*. Celui-là a bel et bien une propriété de caractère vide et non pas une seule mais deux propriétés *glyphe* : premièrement, le trait de césure, et deuxièmement, la valeur *vide* comme glyphe alternatif, utilisé si la césure est désactivée à cet endroit. La possibilité d'inclure deux glyphes dans ce textème nous permet de prendre en compte une future césure éventuelle tout en laissant la décision de couper ou non à une étape de traitement ultérieure.

Dans l'exemple, les formules conditionnelles sont indiquées par des crochets. Commençons par la propriété *brk = 1* : elle signifie *coupure de ligne activée*, cependant, cette coupure est condition de la variable α_1. Si α_1 est fausse, alors la ligne n'est pas coupée à cet endroit, il n'y a donc pas besoin de trait de césure et le glyphe vide g_2 sera choisi puisque sa formule conditionnelle est vraie. Si par contre α_1 est vrai,

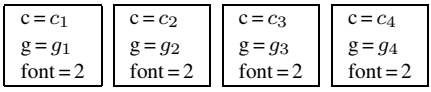
alors la coupure aura bien lieu et le glyphe g sera choisi, faute d'autres possibilités. Le lecteur a certainement remarqué que la variable α_1 apparaît également dans les autres textèmes : là, elle exprime que la ligature *ffi* dépend de l'absence de la césure, sinon elle ne sera pas composée.

La grande différence par rapport à la représentation de dépendances par des liens entre textèmes est qu'ici on contrôle la dépendance au niveau des propriétés au lieu du textème entier, ce qui permet une finesse descriptive ainsi qu'une généralisation potentielle du modèle, au prix de l'augmentation de sa complexité. Cette manière de décrire les dépendances sera utile pour implémenter la *typographie dynamique*, présentée en détail dans section 4.4.

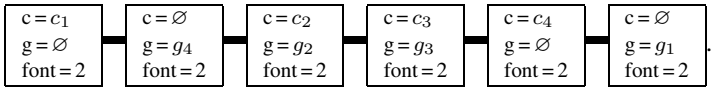
4.2.3. *Réordonnancement*

Le réordonnancement est une opération effectuée principalement sur les écritures de l'Inde telle que la *dévanagari*. Puisque l'ordre dit *logique* des caractères — suivant, par convention, l'ordre dans lequel les phonèmes de la langue sont prononcés — et l'ordre *graphique* des glyphes ne correspondent pas toujours, à un certain moment dans le processus de composition les glyphes doivent être réordonnés. Cela peut être fait au niveau des caractères (comme dans Uniscribe) mais également au niveau des glyphes (comme dans AAT). L'ordre de la chaîne de caractères d'origine étant important à préserver, dans notre modèle de texte constitué de textèmes, celui-ci ne changera pas.

Prenons un cas abstrait de quatre textèmes :



Nous souhaitons échanger les glyphes g_1 et g_4. Nous allons introduire deux nouveaux textèmes à caractère vide de la manière suivante :



Bien évidemment, à cause de l'insertion de nouveaux textèmes, les dépendances devront également être représentées, ce que l'on a fait ici à l'aide de liens. Notons que les dépendances sont en réalité une conséquence de la nature de l'écriture donnée et non pas de la manière de réordonner les glyphes. Si une indication plus exacte des correspondances entre les glyphes réordonnés et leurs caractères est nécessaire, alors une représentation des dépendances par formules logiques pourra remplacer les liens.

4.3. *Extensibilité*

Nous avons déjà vu que le processus de composition typographique se découpe facilement en étapes comme les tâches linguistiques (réordonnancement du dévanagari, analyse contextuelle de l'arabe, césure du latin, segmentation en mots du chinois), la substitution de glyphes, le positionnement de glyphes et le paragraphage. Puis, chacune de ces étapes se partage à son tour en transformations élémentaires. Cette organisation est donc une réalité pratique que reflète également la structure des fontes intelligentes. En même temps, les services de composition tels qu'Uniscribe ou Pango sont construits de manière monolithique et leur fonctionnement est peu modifiable. Cependant, un processeur typographique *modulaire* et *extensible* présente de nombreux avantages. Par *modulaire*, on entend une division (partielle) du processus typographique en entités (modules) de traitement indépendantes. Par *extensibilité*, on entend la création et l'introduction libre de nouveaux modules dans le processeur où l'ordre d'exécution de ces modules est modifiable.

Les opérations exactes à effectuer dépendront d'une part de l'écriture et de la langue en question, d'autre part des potentiels informatiques présents. Ainsi, jusqu'à nos jours la segmentation du texte thaï en mots reste une opération très difficile. Néanmoins, une fois le problème résolu au point qu'une implémentation informatique devient envisageable, cette implémentation pourra également devenir une étape du processus typographique. Il est donc en général souhaitable de pouvoir étendre le processeur par de nouveaux modules.

Un autre intérêt de l'extensibilité est la possibilité de rectifier certaines opérations. Ainsi, il existe de nombreuses fontes arabes et hébraïques qui ne proposent qu'un positionnement partiel et restreint des diverses marques (voyelles, diacritiques, etc.) de ces écritures. En réalité, la complexité du positionnement de marques pour ces deux écritures dépasse dans la plupart des cas les capacités théoriques des fontes intelligentes. Avec un module de positionnement automatique évolué qui intervient *directement après* le module de positionnement de glyphe de la fonte intelligente, il devient possible d'annuler les positions incorrectes introduites par celle-ci et de les rectifier. On dépasse alors les limitations des fontes intelligentes et de leurs modèles de texte classiques.

La combinaison du modèle de textème et la modularité du processeur typographique ouvre une nouvelle dimension d'extensibilité (Haralambous *et al.*, 2005a) : il devient possible d'inventer de nouvelles propriétés de textèmes, de les introduire dans son texte, et puis d'effectuer des traitements sur elles par des modules spécialement créés dans ce but. Ainsi, l'inventeur de l'analyseur syntaxique-sémantique du thaï pourra instruire son module de introduire des propriétés « fin-de-mot » à la fin de chaque mot trouvé. Puis, un autre module pourra prendre en compte cette information afin de couper les lignes du texte aux endroits légaux.

L'utilité pratique du concept de modularité a déjà été prouvée par le logiciel typographique « Ω_2 », développé par les auteurs[8]. Il s'agit d'une extension du système TEX qui adopte le modèle de textèmes et qui est extensible par trois types de modules externes : modules qui affectent uniquement la chaîne de caractères (opérations linguistiques et transcodage), modules qui affectent les glyphes et les autres propriétés de textème (opérations typographiques telles que substitutions et positionnements de glyphes), et finalement modules qui interviennent au moment du paragraphage, sur des lignes (crénage optique, optimisation de l'égalisation).

4.4. *Typographie dynamique, une nouvelle approche au paragraphage*

Dans la typographie actuelle des écritures latine-grecque-cyrillique, la technique d'égalisation des lignes des paragraphes part du principe que les lettres et les espaces inter-lettres sont fixes : l'unique manière d'étirer ou de comprimer une ligne est par le jeu des espaces inter-mots ou bien par la césure.[9] Et pourtant, Gutenberg savait mieux faire : il avait à sa disposition des centaines de ligatures qu'il employait abondamment lorsqu'une ligne était trop longue et au contraire, il les évitait lorsque sa ligne était trop courte (Wild, 1995). Dans les traditions calligraphiques ou typographiques d'autres écritures on trouve de nombreuses autres solutions pour altérer la longueur des lignes. L'exemple flagrant est l'écriture arabe où l'espace inter-mots a un rôle nettement moins prioritaire dans l'égalisation des lignes par rapport aux techniques comme les *kachidé*, la présence ou absence de ligatures ou de variantes de lettres plus ou moins larges. (L'article (Jamal *et al.*, 2006) est un traité exhaustif de ce sujet.)

Nous avons vu que les formats de fonte intelligents comme AAT ou OpenType sont capables de définir des actions similaires aux techniques que nous venons d'évoquer : AAT est capable de modifier les ligatures dans l'étape de post-composition comme dernier recours, et OpenType définit dans sa table JSTF des actions de substitution de glyphe (y compris les ligatures) avec des priorités associées. Ces approches impliquent qu'il appartient au créateur de la fonte de prévoir ces éventualités et de désigner les glyphes destinés à être modifiés lors du paragraphage. D'une part, il faut admettre que le graphiste est certainement la première personne à prendre des décisions esthétiques lorsqu'il s'agit de choisir une ligature à briser ou d'une variante stylistique à utiliser à un endroit précis. D'autre part, cependant, les préférences sont parfois davantage fonction des traditions de l'écriture donnée que de la fonte, notamment dans le cas de l'arabe (Jamal *et al.*, 2006). Aussi, les créateurs de fontes s'occupent rarement de ce genre de problèmes car traditionnellement c'est le rôle du typographe de les résoudre. Les typographes d'aujourd'hui sont cependant les logiciels de composition, et jus-

8. Le lecteur intéressé par le système Ω_2 trouvera plus d'informations dans (Haralambous *et al.*, 2006).

9. Nous n'avons pas mentionné la technique douteuse qui consiste à modifier l'interlettrage pour des raisons de justification, un recours rare dans la typographie de haute qualité. Il faut cependant distinguer ce genre d'interlettrage « sauvage » et interlettrage sémantique utilisé en écritures gothique, cyrillique, grecque, etc.

qu'à présent, à la connaissance des auteurs, actuellement aucun logiciel n'est capable de prendre en compte les règles de justification établies dans les fontes intelligentes, fait qui n'incite bien évidemment pas les fonderies à investir de l'énergie et de l'argent dans le développement de telles fonctionnalités. Afin de résoudre cette situation d'attente mutuelle, les auteurs proposent une approche complètement différente qu'ils appellent *typographie dynamique*.

Un algorithme de paragraphage, qu'il s'agisse de simple justification par lignes ou d'égalisation au niveau du paragraphe entier, dispose d'un nombre d'emplacements dans le texte qui peuvent être modifiés afin d'obtenir une ligne de longueur plus optimale. Dans un contexte latin-grec-cyrillique, ces points sont les espaces inter-mots et les points de césure. L'aide à la justification, si elle est présente dans les fontes intelligentes, peut ajouter quelques emplacements supplémentaires *explicitement définis*. Dans la typographie dynamique, par contre, on considère que toute ligature et toute variante de glyphe devient un point d'intervention potentiel pour modifier la longueur d'une ligne, que cet usage des variantes et des ligatures soit explicitée dans la fonte ou non. L'autre différence majeure de cette approche est que ces points d'intervention sont décidés non pas lors du paragraphage, dans la connaissance des lignes candidates, mais *en amont*, lors de l'étape de substitutions sur le texte jusque-là ininterrompu. Nous avons une bonne raison pour faire ainsi : lorsqu'un glyphe est modifié *a posteriori*, c'est-à-dire *après* que l'étape des substitutions a eu lieu — c'est le cas des méthodes d'AAT et d'OpenType —, il n'y a pas de garantie que cette modification ne brise une dépendance existante entre ce glyphe et une autre propriété quelconque. Exemple : on tente d'élargir une ligne trop étroite par le remplacement d'un glyphe par une variante plus large qui, à son tour, causera une régression dans la forme d'un dépassement de ligne. Toute modification *a posteriori* peut donc entraîner des modifications nécessaires ailleurs dans le paragraphe, et éventuellement un effet d'avalanche de modifications, notamment lorsque la fonte définit de nombreuses relations contextuelles entre glyphes. Même si ce genre de situation est relativement rare car les fontes actuelles ne contiennent que peu de règles contextuelles et de variantes, le moteur typographique doit être prêt à les gérer de manière correcte et cohérente. Alors que les spécifications d'OpenType et d'AAT ne mentionnent pas ces effets secondaires potentiels lorsqu'elles décrivent leurs méthodes de justification, la typographie dynamique permet de les prendre en compte correctement puisqu'elle s'y prépare en faisant les calculs nécessaires en amont.

4.4.1. *Première étape : le calcul en amont des paragraphes candidats*

Le principe général de la typographie dynamique est de calculer non pas un seul paragraphe composé mais *tous* les paragraphes potentiels. *A priori*, toutes les césures potentielles sont susceptibles d'être appliquées, toutes les ligatures brisées, toutes les variantes insérées, et on peut trouver d'autres possibilités encore, selon l'écriture donnée. Les propriétés de textèmes nous seront utiles pour stocker les résultats des calculs. On procède de la manière suivante :

1) tout d'abord, on trouve tous les points de césure potentiels dans le paragraphe et on les marque par une propriété de textème ;

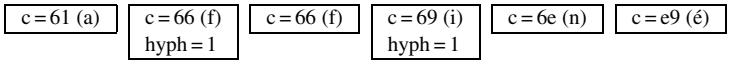


Figure 5. *Les points de césure potentiels ont été marqués dans les textèmes.*

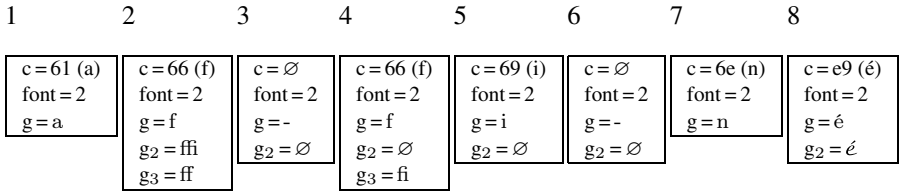


Figure 6. *Tous les glyphes candidats ont été calculés.*

2) ces propriétés de césure potentielle deviennent de nouveaux textèmes qui représentent le trait de césure : des glyphes sans caractère ;

3) lors de l'application des règles de substitution définies dans les fontes intelligentes, on prend en compte les éventualités suivantes :

- tout espace et tout point de césure peut devenir point de coupure de ligne et ainsi affecter les ligatures et les règles contextuelles,

- toute ligature pourra être brisée et toute variante pourra être utilisée afin de modifier la longueur d'une ligne.

En conséquence, au lieu d'obtenir un seul texte formaté, on en obtient autant que de variations possibles ;

4) toutes les versions du texte obtenues seront stockées dans les textèmes en tant que *glyphes candidats* : ainsi, il pourra y avoir plusieurs glyphes potentiels dans le même textème.

Prenons toujours comme exemple le mot *affiné*. Ce mot contient deux points de césure potentiels que nous indiquerons par des propriétés de textème *hyph*, comme en figure 5. La figure suivante, 6, représente l'étape après la substitution de glyphes. On observe que des textèmes de césure ont été ajoutés, ainsi que plusieurs glyphes candidats (g, g_2, g_3, etc.). La chaîne de caractères *ffi* peut donc devenir :

– une ligature *ffi*, dans ce cas-là, les glyphes des textèmes 3, 4 et 5 seront vides (pas de césure) ;

– une ligature *ff* et un glyphe *i*, dans ce cas-là, les glyphes des textèmes 3 et 4 seront vides (pas de césure) ;

– un glyphe *f* et une ligature *fi*, dans ce cas-là, le glyphe du textème 4 sera vide (césure appliquée) ;

– les glyphes *f*, *f* et *i* séparés (aucun glyphe vide, césure appliquée).

En plus de toutes ces possibilités, le dernier textème contient deux variantes de glyphe pour le caractère *é*. Désormais, les textèmes contiennent toutes les versions possibles du mot *affiné*.

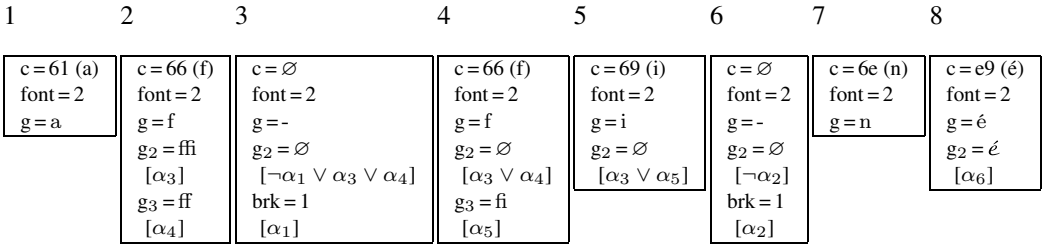


Figure 7. *Les dépendances entre glyphes candidats ont été précisées.*

Cependant, le fait d'inclure plusieurs glyphes candidats dans les textèmes soulève un nouveau problème : quelles sont les combinaisons permises ? Car il est évident que g_2 du textème n° 2 (ligature *ffi*) peut être combiné uniquement avec g_2 du textème n° 3 (glyphe vide) et non pas avec le trait de césure : la césure doit empêcher la composition de la ligature. De telles contraintes sont applicables aux autres glyphes candidats également.

Remarquons que, une fois de plus, nous nous retrouvons en face du problème des *dépendances*. Parmi les deux solutions suggérées dans la section 4.2.2, cette fois-ci nous allons appliquer celle basée sur les formules logiques. À chaque glyphe candidat on associe une formule logique, sa *condition* (voir figure 7). Suivant l'ordre des opérations établi ci-dessus, dans un premier temps, on associe des variables conditionnelles aux textèmes de césure : α_1 pour *af-fi* et α_2 pour *fi-né*. Lors de l'étape de substitution de glyphes, α_3, α_4 et α_5 sont introduits, exprimant les ligatures *ffi*, *ff* et *fi* respectivement. Pourquoi a-t-on besoin de trois nouvelles variables alors que la composition de la ligature *ffi* dépend déjà de α_1, comme dans l'exemple de la section 4.2.2 ? N'oublions pas que dans la typographie dynamique nous sommes à la recherche de toutes les variantes possibles du paragraphe, et cela comprend même les cas qui ne devraient normalement pas être composés du tout. Ainsi, nous allons briser la ligature *ffi* de toutes manières possibles (*f f i, ff i, f fi*), même si aucune césure n'aura lieu. Similairement, nous introduisons la variable α_6 pour identifier la variante stylistique du glyphe *é*.

Le lecteur attentif a certainement remarqué l'absence de conditions pour le premier glyphe de chaque textème. Il s'agit d'une omission délibérée : si aucune condition n'est associée à un glyphe, cela signifie le *cas par défaut* dont la condition implicite est la non-application des autres glyphes du même textème. Ainsi, dans le textème n° 2 de notre exemple, le glyphe g = *f* sera choisi si et seulement si les conditions $[\alpha_3]$ et $[\alpha_4]$ sont toutes les deux fausses.

4.4.2. *Deuxième étape : la construction du graphe étendu*

Bien évidemment, nous n'avons pas introduit le mécanisme des textèmes et des dépendances logiques pour nous contenter plus tard d'une simple justification de pa-

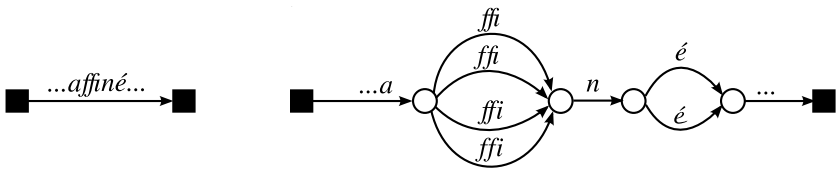


Figure 8. *L'extension d'une arête du graphe de paragraphage par les nœuds de variante. À gauche, l'arête d'origine représente une ligne qui contient le mot* affiné *au milieu. Les carrés noirs aux extrémités de l'arête correspondent aux coupures faisables en début et en fin de ligne. À droite, l'arête a été étendue par toutes les variantes du mot* affiné*. Les cercles sont les nœuds de variante.*

ragraphe ligne par ligne. Nous allons voir comment le paragraphe, avec toutes ses versions pré-déterminées, sera traité par l'algorithme d'égalisation au niveau du paragraphe entier.

TEX a été le premier, et pendant une vingtaine d'années, le seul logiciel capable d'une égalisation optimale. L'algorithme inventé par Knuth et Plass tourne en pratique dans un temps borné en $\mathcal{O}(n)$, cf. (Haralambous, 2004b), (Knuth *et al.*, 1981).

Dans le graphe qui décrit les coupures potentielles d'un paragraphe, les nœuds représentent les coupures *faisables* (en même temps légales et probables), les arêtes dirigées entre nœuds représentent les lignes candidates entre deux coupures, et le poids numérique associé à chaque arête est le *démérite* (l'inverse du mérite) de la ligne correspondante. Dans le modèle proposé dans (Knuth *et al.*, 1981), seuls les espaces inter-mots et les points de césure peuvent devenir points de coupure, et donc nœuds du graphe. La recherche du paragraphe optimal consiste à trouver le chemin le plus court dans ce graphe.

Nous allons étendre ce graphe et notre modèle de paragraphage en y introduisant le *nœud de variante*. Un tel nœud ne représente pas une coupure de ligne mais plutôt un endroit où un glyphe peut être remplacé par une variante ou une ligature peut être brisée. Pour chaque glyphe ou séquence de glyphes qui peut varier, deux nœuds de variante seront introduits, connectés par autant d'arêtes que de variantes possibles. La figure 8 montre comment une arête du graphe d'origine contenant le mot *affiné* est étendue par les nœuds de variante.

À chaque nouvelle arête qui connecte les nœuds de variante correspond une chaîne de glyphes ayant sa propre chasse. Lorsque le moteur de paragraphage calcule la longueur d'une ligne, il obtiendra non pas une valeur fixe (qu'il pourra ensuite comprimer ou étendre par la modification des espaces inter-mots) mais un ensemble de longueurs différentes : sur la figure 8, par exemple, au plus huit longueurs sont possibles. Selon l'écriture et les règles typographiques en question, le moteur pourra choisir la chaîne la mieux adaptée à la longueur souhaitée et ainsi réduire les modifications effectuées sur l'espacement.

L'exemple que nous avons utilisé jusqu'à maintenant, le mot *affiné*, n'est peut-être pas idéal pour montrer l'intérêt de la typographie dynamique car en typographie latine, à part les textes de Gutenberg, la non-application de ligatures pourrait souvent causer une détérioration de la qualité du texte (collision des lettres composantes) qui dépasserait l'avantage gagné par une meilleure égalisation du paragraphe. En allemand, où la présence d'une ligature dépend de critères morphologiques (pas de ligatures à la frontière de deux composantes de mot), ce genre d'opération comporte des risques aux niveaux esthétique et sémantique. D'un autre point de vue cependant, avec la création de nouvelles fontes mieux adaptées à la typographie dynamique, cette technique pourrait redevenir courante et contribuer à la redécouverte d'une esthétique typographique proche de celle de l'arabe et des textes gutenbergiens. Car dans l'écriture arabe, comme nous l'avons déjà mentionné, l'élargissement des espaces n'est qu'une technique de recours qui est précédée en priorité par l'utilisation des kachidés et des lettres allongées ainsi que par la décomposition des ligatures.

Après avoir vu le principe de l'extension du graphe, il faut encore répondre à une question : comment extraire les différentes versions du paragraphe à partir de notre chaîne de textèmes ? C'est à ce moment-ci que les variables conditionnelles montrent leur puissance : une version d'un paragraphe peut être facilement sélectionnée en choisissant une combinaison de valeurs légale pour toutes les variables y contenues. Le mot *légal* signifie ici que, en introduisant ces valeurs dans les formules logiques, on ne tombe pas sur une contradiction logique, autrement dit, parmi un nombre d'alternatives dans le même textème, une et une seule choix est satisfaite. Dans l'exemple de la figure 7, la combinaison $(\alpha_3 =\uparrow, \alpha_4 =\uparrow)$ est contradictoire parce qu'elle sélectionne à la fois g_2 et g_3. Le cas où aucune alternative n'est sélectionnée ne peut jamais arriver dans notre exemple puisque les glyphes n'ayant pas de condition explicitement spécifiée seront sélectionnés par défaut.

Les combinaisons possibles pour l'exemple de la figure 7 sont énumérées en figure 9. Les textèmes du mot contiennent six variables de condition, $\alpha_1 ... \alpha_6$, qui peuvent être combinées librement, ce qui signifie $2^6 = 64$ combinaisons. C'est un nombre énorme, même s'il s'agit d'un mot plutôt spécial (une triple ligature, deux césures et une variante pour seulement six caractères) : dans un paragraphe on pourra trouver très probablement des dizaines, voir une centaine de variables conditionnelles, ce qui signifie que le nombre de combinaisons s'élève à 2^{100}...

Ceci étant dit, 40 parmi les 64 cas évoqués sont des combinaisons contradictoires et peuvent être éliminées par une analyse des formules conditionnelles contenues dans les textèmes. Rappelons le lecteur que la condition qu'une combinaison de valeurs de variables soit contradictoire est qu'il existe un textème où le nombre de formules logiques satisfaites par ces valeurs soit différent à 1*Si un* glyphe par défaut *est présent, il peut être choisi lorsque le nombre de formules satisfaites est égal à zéro ; ainsi on évite la contradiction.*. Dans notre exemple, si α_4 et α_5 sont vrais en même temps, deux glyphes, g_2 et g_3 seront choisis dans textème n° 4, ce qui relève d'une contradiction.

α_1	α_2	α_3	α_4	α_5	α_6	Texte sélectionné
0	0	0	0	0	0	*a f f i n é*
0	0	0	0	0	1	*a f f i n é*
0	0	0	0	1	0	*a f fi n é*
0	0	0	0	1	1	*a f fi n é*
0	0	0	1	0	0	*a ff i n é*
0	0	0	1	0	1	*a ff i n é*
0	0	0	1	1	0	*contradiction*
0	0	0	1	1	1	*contradiction*
0	0	1	0	0	0	*a ffi n é*
0	0	1	0	0	1	*a ffi n é*
0	0	1	0	1	0	*contradiction*
						⋮
0	0	1	1	1	1	*contradiction*
0	1	0	0	0	0	*a f f i - n é*
0	1	0	0	0	1	*a f f i - n é*
0	1	0	0	1	0	*a f fi - n é*
						⋮
1	0	0	0	0	0	*a f - f i n é*
1	0	0	0	0	1	*a f - f i n é*
1	0	0	0	1	0	*a f - fi n é*
1	0	0	0	1	1	*a f - fi n é*
1	0	0	1	0	0	*contradiction*
						⋮
1	0	1	1	1	1	*contradiction*
1	1	0	0	0	0	*a f - f i - n é*
1	1	0	0	0	1	*a f - f i - n é*
1	1	0	0	1	0	*a f - fi - n é*
1	1	0	0	1	1	*a f - fi - n é*
1	1	0	1	0	0	*contradiction*
						⋮
1	1	1	1	1	1	*contradiction*

Figure 9. *Toutes les versions du mot* affiné*, y compris les césures potentielles. Parmi les* $2^6 = 64$ *combinaisons des six variables de condition* $\alpha_1 ... \alpha_6$*, 40 sont contradictoires.*

Parmi les 24 cas dans notre liste qui ne sont pas contradictoires, quatre correspondent à la double césure *af-fi-né*, un cas pratiquement impossible. De plus, les 20 options restantes ne sont pas toutes possibles en même temps : lorsque le mot se trouve au milieu de la ligne, les variables α_1 et α_2 associées aux points de césure ne seront pas activées, et le nombre de versions se réduit à huit. En outre, notre but est de ne trouver que des variantes qui diffèrent réellement dans leur chasse, alors que parmi les huit candidats mentionnés plusieurs ont des chasses presque identiques. De ce point de vue, il n'y a pratiquement pas de différence entre les chaînes *f fi* et *ff i*, par conséquent, rien ne justifie l'inclusion de toutes les deux parmi les candidats potentiels.

Nous avons donc besoin d'une manière de restreindre l'ensemble des candidats possibles afin d'optimiser le temps de calcul nécessaire. La méthode utilisée dépendra bien évidemment de plusieurs paramètres : l'écriture, le genre d'ouvrage composé, le contexte typographique, les préférences de l'utilisateur, etc. Bref, le moteur de paragraphage doit être capable de permettre plusieurs méthodes ; dans la suite, nous appellerons ces méthodes *stratégies de sélection*.

4.4.3. *Les stratégies de sélection de variantes*

Une stratégie, dans le sens général, sert de guide parmi une série de choix à partir d'un ensemble d'actions potentielles. En typographie dynamique, la stratégie consiste à extraire à partir du vaste ensemble de variantes possibles un nombre réduit de candidats qui ne sont ni trop nombreux — ce qui pourrait ralentir considérablement l'algorithme d'égalisation —, ni trop peu nombreux — ce qui pourrait empêcher une réelle amélioration dans la qualité de la composition.

Les stratégies les plus évidentes ont déjà été évoquées : le rejet des combinaisons illégales et inappropriées. Le premier peut être fait assez simplement puisque les contradictions se restreignent aux variables qui coexistent dans les mêmes textèmes et la vérification de ceci est relativement facile. Savoir si un choix est inapproprié est une question moins évidente, mais il y a des cas précis que l'on peut évoquer :

– deux points de césure à une distance bien inférieure à la longueur d'une ligne ;

– un point de césure au milieu d'une ligne ;

– le choix de variantes à chasses inférieures alors que la ligne a besoin de s'étendre (et vice versa).

Afin d'accélerer le filtrage de combinaisons non réalistes et toutes les opérations liées aux variables de condition en général, il est utile de construire — en même temps que le remplissage des textèmes par les variantes de glyphes — un tableau au niveau du paragraphe courant qui associe à chaque variable de condition une sémantique (ligature, composante de ligature, variante stylistique, césure), une chasse, si possible (même approximative), et la position du (des) textème(s) où elle se trouve. À l'aide d'une telle structure de données, les combinaisons illégales et inappropriées peuvent être discernées et exclues très rapidement.

Une stratégie doit cependant servir à bien plus qu'à l'exclusion des erreurs les plus flagrantes. Elle peut établir un ordre de priorité entre différentes opérations (exemple :

les ligatures latines, souhaitables même malgré des paragraphes sous-optimaux, ne doivent être brisées que comme dernier recours), elle peut donner des indications glyphe par glyphe (il ne faut jamais briser la ligature arabe *lam-alif*) ou des contraintes contextuelles (n'utiliser les formes allongées des lettres qu'en position de fin de ligne). Comme nous l'avons mentionné, des stratégies différentes s'appliqueront à chaque écriture, à chaque genre de document (texte coranique, article de journal), à chaque usage. D'autre part, les glyphes présents dans les fontes utilisées limiteront également l'efficacité de l'approche dynamique.

Si le lecteur voit une parallèle avec les tables JSTF d'OpenType et `just` d'AAT, ceci n'est pas un hasard : elles définissent, elles aussi, des priorités, elles classifient les glyphes selon leur comportement vis-à-vis de la justification, elles prescrivent des substitutions de glyphes ; bref, elles représentent également des stratégies. La grande différence de la typographie dynamique par rapport à OpenType ou AAT est que les stratégies sont implémentées au niveau du moteur typographique qui peut ainsi fonctionner avec des fontes privées de tables de justification, ce qui signifie la quasi-totalité des fontes existantes. Les créateurs de fontes n'ont en général pas l'habitude de générer ce genre d'informations, surtout qu'il n'existe ni outil de développement, ni plate-forme où ils pourraient tester le comportement de leurs fontes.

Pour clore la présente section et l'article, nous allons donner un exemple concret et détaillé d'une implémentation pratique et optimisée de la typographie dynamique, telle que nous envisageons de la réaliser dans le système Ω_2.

4.4.4. *Étude de cas : une stratégie de typographie dynamique pour la justification arabe*

Nous avons déjà évoqué les différences entre les méthodes traditionnelles de justification des écritures alphabétiques et celles de l'écriture arabe. Ne pouvant donner un traité exhaustif de la justification arabe dans le cadre du présent article, nous nous contenterons d'un bref résumé de ses propriétés principales et de nous référer à l'article (Jamal *et al.*, 2006).

Dans l'écriture arabe, lorsqu'une ligne est trop étroite, elle peut être étirée des manières suivantes :

– par l'utilisation de variantes de glyphes à chasses plus larges ;

– par la décomposition de ligatures ;

– par l'insertion de *kachidé*, « allonges » curvilignes (de préférence) entre glyphes ;

– par l'élargissement des espaces inter-mots.

La compression de la ligne ne peut être réalisée que de deux manières : par l'emploi de (davantage de) ligatures ou par des espaces inter-mots plus étroites (les variantes plus étroites étant rares en pratique). La typographie dynamique nous permet d'appliquer en même temps toutes les méthodes mentionnées ci-dessus. Notons toutefois que les vrais *kachidé* curvilignes, sujets de recherches actuelles (Lazrek, 2003) à cause des difficultés techniques qu'ils présentent, sont pratiquement inexistants dans les fontes arabes actuels et seuls quelques systèmes typographiques expérimentaux sont capables

d'en générer (par exemple, en forme de fontes dynamiques en format PostScript de type 3). Dû à des contraintes techniques et contrairement à la nature inhérente de l'écriture arabe, le *kachidé* rectiligne, formé par l'accumulation de caractères au plomb en forme de petits segments droits, était très répandu dans la typographie arabe du XIX^e^ et du XX^e^ siècle.

Nous avons vu les difficultés rencontrées lors de la recherche d'une solution générale par la typographie dynamique pour l'optimisation d'un paragraphe : en théorie, toutes les combinaisons de glyphes variantes et de ligatures sont possibles. (En revanche, fort heureusement, l'absence de la césure dans l'écriture arabe[10] réduit le nombre de combinaisons possibles.) Dans la suite, nous allons proposer une variante simplifiée de la typographie dynamique qui permet toujours un gain considérable en qualité de mise en page sans l'augmentation exponentielle du temps de calcul. Encore une fois, nous allons supposer que l'algorithme de paragraphage en question est l'algorithme à optimalité globale de Knuth et de Plass afin de démontrer que les avantages de la typographie dynamique peuvent être combinés avec la puissance de cet algorithme sans que la complexité de celui-ci ne change : nous garderons la complexité de l'algorithme au-deçà de $\mathcal{O}(n)$.

La méthode proposée peut être divisée en trois étapes : avant, pendant et après le paragraphage.

Étape 1a)
Cette étape de *classification de glyphes* est exécutée une seule fois par fonte utilisée ; bien évidemment, on ne s'intéresse qu'à la (ou aux) fonte(s) courante(s) (à l'opposition des fontes mathématiques et d'autres fontes à usages spéciaux). Dans un premier temps, il s'agit de rechercher tous les glyphes qui possèdent de variantes stylistiques ou de justification dans la fonte, ainsi que toutes les ligatures. Par exemple, dans une fonte OpenType, les variantes et ligatures peuvent très facilement être retrouvées à l'aide des propriétés comme *jalt (justification alternates)*, *liga*, etc. La classification se fait selon la chasse du glyphe, plus précisément, selon la différence de chasse par rapport au glyphe « standard » (choisi par défaut), ou, dans le cas des ligatures, par rapport à la somme des chasses de leurs composantes individuelles. Dans la grande majorité des cas, cette différence sera positive pour les variantes et négative pour les ligatures. On suppose qu'un nombre restreint de différences de chasse a été pré-défini : ce seront nos classes. Typiquement, un nombre de classes inférieur à une dizaine sera suffisant, mais ce nombre, ainsi que la distribution exacte des classes, sera toujours fonction de la qualité de mise en page souhaitée. Cette étape de classification se fera une fois par fonte, et le résultat pourra même être stocké sur le disque dur afin de ne pas devoir la refaire pour chaque document utilisant la même fonte.

Étape 1b)
Toujours avant le paragraphage, pour chaque glyphe du paragraphe ayant des variantes ou participant à une ligature, on insère dans le textème correspondant les identifiants

10. La seule exception à cette règle est la langue ouïghour, dans laquelle, par contre, on n'utilise pas de ligatures.

de classe pour chacune de ses variantes ou ligatures. Ceci nous permettra d'évaluer rapidement lors du paragraphage les possibilités d'étirement et de compression pour chaque ligne candidate.

Étape 2)
Lors de l'algorithme de paragraphage, chaque ligne candidate présentera une différence de longueur par rapport à la longeur idéale. Sans typographie dynamique, cette différence est comblée (ou réduite) par la modification de l'espacement des mots ou alors par l'ajout de *kachidé*. Dans notre cas, le manque ou l'excès de longueur pourra être corrigé par *le choix judicieux d'une classe de différence de chasse.* Pour ce faire, il faut parcourir la ligne candidate à la recherche d'identifiants de classe (insérés en étape 1b) et en choisir une dont la chasse correspond de manière optimale à la correction souhaitée. Si la ligne candidate contient plusieurs variantes ou ligatures, leurs classes pourront éventuellement être combinées, si une correction suffisante ne peut être atteinte par le choix de la classe d'un seul glyphe. Il ne faut cependant pas oublier que nous sommes au plein milieu de l'algorithme de paragraphage, c'est-à-dire à l'endroit le plus sensible vis-à-vis de la complexité de nos calculs. Par conséquent, on ne cherchera pas à trouver la meilleure combinaison possible de classes que la ligne candidate est capable d'offrir : dans la plupart des cas, il suffira de trouver une approximation, une classe *suffisamment proche* de la correction recherchée. La différence de chasse correspondant à la classe choisie contribuera à la réduction de la démérite de la ligne candidate dans le calcul du paragraphe optimal. C'est ainsi, par ce choix préliminaire restreint, que la recherche exhaustive de toutes les combinaisons de variantes et de ligatures — et donc l'explosion combinatoire des calculs — est évitée.

Étape 3)
Dans cette dernière étape de *post-paragraphage*, on finit par peaufiner le choix de variantes et de ligatures pour les lignes qui font partie du paragraphe optimal. Désormais, on est sorti de l'algorithme de paragraphage, les lignes optimales ont déjà été choisies, rien n'empêche donc d'effectuer une analyse plus rigoureuse des possibilités d'intervention. Lorsqu'une ligne présente plusieurs moyens d'atteindre la correction de longueur choisie en étape 2, on a la possibilité d'en sélectionner le plus précis, mais aussi de prendre en compte d'éventuelles priorités entre variantes et ligatures (certaines ligatures étant plus souhaitables que d'autres, il en est de même pour les variantes : la combinaison de deux variantes légèrement allongées est une intervention plus subtile que l'insertion d'une seule variante très large). Il devient possible même de prendre en compte des *contraintes contextuelles entre lignes*, afin par exemple d'éviter les variantes allongées qui se retrouvent alignées verticalement, ainsi formant les équivalents arabes des « rivières ». Finalement, comme dernière opération, la différence entre la longueur de la ligne corrigée et la longueur idéale est réduite à zéro par l'ajustement des espaces inter-mots ou bien par des *kachidé*. Cet ajustement sera cependant bien inférieur à ce qui aurait été appliquée sans la méthode de typographie dynamique.

La méthode décrite ci-dessus est un exemple particulier de *stratégie de sélection* de glyphes en typographie dynamique. Il est d'un intérêt pratique, réalisable dans les

systèmes typographiques actuels à l'aide de fontes intelligents contenant des ligatures et des variantes de glyphes. Les stratégies de sélection en général représentent un sujet de recherche en cours.

5. Conclusions

Le modèle de textème a déjà été implémenté dans le système de composition extensible Ω_2. Ce modèle permet une représentation de texte plus riche au niveau de ses éléments constitutifs, tout en internalisant les notions de caractère et de glyphe. Il permet également l'unification du texte brut et du texte formaté dans un seul modèle de texte. Alors que dans le modèle véhiculé par Unicode la problématique des correspondances entre caractères et glyphes se manifeste par la nécessité de maintenir un graphe entre deux chaînes de texte, dans le modèle de textème, elle s'est présentée par la notion plus générale de *dépendance*. Deux solutions ont été conçues afin de gérer les dépendances : les liens entre textèmes et les formules logiques associées aux propriétés. Ces dernières se sont avérées particulièrement utiles dans la *typographie dynamique*, une méthode pour calculer en amont toutes les versions potentielles d'un paragraphe afin de permettre un paragraphage plus puissant mais aussi d'éviter les problèmes de l'interaction entre paragraphage et règles typographiques contenues dans les fontes intelligentes.

Dans l'ensemble, bien qu'en théorie les formats de fonte intelligents puissent fournir de l'aide en support des méthodes avancées de paragraphage, celles-ci se tardent à être déployées dans les bibliothèques typographiques et dans les logiciels de PAO, sans doute à cause de l'augmentation de calculs qu'elles entraînent. La typographie dynamique risque également d'être gourmande en calculs, mais les auteurs espèrent bientôt pouvoir en implémenter un prototype dans le logiciel Ω_2 afin d'évaluer le gain en qualité de composition par rapport à la complexité relative de la méthode, notamment pour l'écriture arabe. Il est fort probable que davantage de recherches seront nécessaires afin de trouver des bons compromis entre puissance et rapidité d'exécution des méthodes de paragraphage.

6. Bibliographie

AAT, « TrueType Reference Manual, AAT Tables », n.d.
`http://developer.apple.com/textfonts/TTRefMan/`.

Andries P., « Unicode et les polices : deux mondes », *Actes de l'atelier « La typographie entre les domaines de l'art et de l'informatique », Publications de l'Institut Royal de la Culture Amazighe, Rabat*, 2007.
`http://hapax.qc.ca/pdf/deux-mondes.pdf`.

André J., Hudrisier H. (eds), *Document Numérique*, vol. 6 (« Unicode, écriture du monde »), 3-4/2002.

GX, « How GX Does Justification by Apple Computer », n.d.
`http://developer.apple.com/textfonts/WhitePapers/GXJustification.html`.

Haralambous T., Haralambous Y., « Characters, Glyphs and Beyond »,
Actes du Glyph and Typesetting Workshop*, Kyoto, Japon*, 2003.

Haralambous Y., *Fontes et codages*, O'Reilly France, avril, 2004a. ISBN 2-84177-273-X.

Haralambous Y., « Voyage au centre de TEX: composition, paragraphage, césure », *Cahiers GUTenberg*, 2004b.

Haralambous Y., « Infrastructure for Typesetting High-Quality Arabic », *TUGboat, 27(2)*, 2006. Actes de la conférence TUG 2006, Marrakech, Maroc.

Haralambous Y., « Unicode et typographie : un amour impossible », *Document Numérique*, vol. 6 (« Unicode, écriture du monde »), 3-4/2002.

Haralambous Y., Bella G., « Injecting Information into Atomic Units of Text »,
ACM Symposium on Document Engineering, 2005a.

Haralambous Y., Bella G., « Omega Becomes a Sign Processor », *Actes de la conférence EuroTEX 2005, Pont-à-Mousson, France*, 2005b.
(Version mise à jour : *Omega Becomes a Texteme Processor*.
`http://omega.enstb.org/yannis/pdf/eurotex05.pdf`).

Haralambous Y., Bella G., « Open-Belly Surgery upon Ω_2 », *TUGboat 26(1)*, 2006.
Actes de la conférence EuroTEX 2006, Debrecen, Hongrie.

Hosken M., Hallissy B., Cleveland W., Correll S., Ward A., « Graphite Description Language, version 2.001 », 2000.
`http://scripts.sil.org`.

Jamal M., Benatia E., Elyaakoubi M., Lazrek A., « Arabic Text Justification », *TUGboat 27(2)*, 2006. Actes de la conférence TUG 2006, Marrakech, Maroc.

Knuth D. E., Plass M. F., « Breaking Paragraphs into Lines », *Software—Practice and Experience*, vol. 11, 1981.

Lazrek A., « CurExt, Typesetting Variable-sized Curved Symbols », *TUGboat 24(3)*, 2003. Actes de la conférence EuroTEX 2003, Brest, France.
`http://www.ucam.ac.ma/fssm/rydarab/doc/communic/curext.pdf`.

Trager E. H., « The Penguin and Unicode: The State of Unicode and Internationalization in Linux », 2005. Présentation à la 27e Conférence Unicode.

UFr, « Le standard Unicode, Introduction », n.d. Traduction française par P. Andries.
`http://hapax.qc.ca`.

Uni, « The Unicode Standard, version 5.0 », n.d. `http://www.unicode.org`.

UTR, « Unicode Technical Report #17 : Character Encoding Model », n.d. `http://unicode.org/reports/tr17/`.

Wild A., « La typographie de la Bible de Gutenberg », *Cahiers GUTenberg*, vol. 22 (« Numéro spécial : ligatures & caractères contextuels »), 1995.