THÈSE

de Doctorat

présentée à

L'UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS

École doctorale des Sciences Pour l'Ingénieur

pour obtenir le titre de

DOCTEUR

Spécialité

INFORMATIQUE

par

Olivier ARSAC

Sujet de la thèse :

Interfaces homme machine pour le calcul formel

Soutenue le 1er juillet 1997
devant un Jury composé de :

Michel Rueher	Président
Marc Gaëtano	Directeur
Arjeh Cohen	Rapporteur
Joëlle Coutaz	Rapporteur
Daniel Lazard	Rapporteur
Stephane Dalmas	Examinateur
Colas Nahaboo	Examinateur

“There is no portable software,
there are only softwares that have been ported.”

Jim Fulton

Présentation du serveur Emath 61

Chapitre IV

Implémentation d'Emath 93

Chapitre V

Introduction

Durant les vingt dernières années, le calcul formel s'est progressivement imposé comme un outil incontournable pour le scientifique, qu'il soit mathématicien ou ingénieur. À vocation généraliste, des outils comme *Maple* ou *Mathematica* jouent aujourd'hui un rôle déterminant dans des domaines aussi variés que les mathématiques, la chimie, la physique, l'économie, etc. Les systèmes de calcul formel se présentent, pour la plupart, comme des programmes interactifs permettant d'effectuer des calculs sur des objets mathématiques symboliques (des variables). À l'aide de ces systèmes, il est possible de calculer les solutions formelles de l'équation du second degré $ax^2+bx+c=0$ en fonction des paramètres a, b et c. Ce calcul est dit 'formel' car les valeurs des paramètres a, b et c ne sont pas précisées. Depuis leur apparition dans les années soixante et au fil des versions, les systèmes de calcul formel ont beaucoup gagné en puissance et en efficacité. La nette progression de la qualité des algorithmes employés et de leur implémentation permet d'augmenter la taille des problèmes traités tout en réduisant le temps nécessaire à leur résolution. Cependant, dans ces différents systèmes, l'interface a peu évolué, n'exploitant que faiblement les possibilités graphiques des environnements actuels. Ainsi, les systèmes précédemment cités restent conformes à leur structure initiale. L'interaction avec l'utilisateur se limite à une suite linéaire de requêtes-résultats, ce qui, en terme d'ergonomie, correspond probablement au modèle le plus élémentaire et le moins adapté à la résolution d'un problème de calcul formel. En matière d'édition de formules, seules les toutes dernières versions de *Maple* et de *Mathematica* proposent une édition bi-dimensionnelle incrémentale. Dans les autres systèmes, la saisie des expressions s'effectue toujours linéairement dans une syntaxe souvent lourde et sans rapport avec le sens mathématique des objets décrits. En revanche, depuis les toutes premières versions, les systèmes sont capables d'afficher les résultats en deux dimensions, se rapprochant ainsi de la notation mathématique naturelle des formules. Conséquence directe de la différence de format entre la requête entrée par l'utilisateur et le résultat affiché par le système, entrées et sorties ne peuvent pas se combiner simplement. Même si la typographie des résultats a sensiblement progressé, passant d'un affichage à l'aide de caractères alpha-numériques à un affichage sur écran matriciel (bitmap), aucune solution satisfaisante n'a été apportée aux problèmes de présentation des grandes expressions. Des techniques comme la césure ou l'abréviation de sous-expressions sont toujours employées de façon naïve. Plus généralement, la manipulation interactive d'une expression mathématique se limite à quelques actions simples qui s'expriment souvent à l'aide d'opérateurs à la sémantique obscure et à la syntaxe peu intuitive. D'un point de vue plus global, les systèmes de calcul formel existants n'offrent aucune fonctionnalité pour aider l'utilisateur à organiser les nombreux calculs intermédiaires qui conduisent à la solution de son problème. Les *notebooks* de *Mathematica* offrent un outil de présentation des calculs qui mélange texte et expressions mathématiques, mais dont la structure reste linéaire et les fonctionnalités limitées à l'édition du document.

L'interface homme-machine, composant crucial commun à tous les systèmes de calcul formel, est aussi celui qui est le moins développé et qui présente encore aujourd'hui d'importantes limitations. Au niveau des expressions, l'édition et la manipulation interactive d'expressions mathématiques ne sont traitées de façon satisfaisante par aucun système de calcul formel, ni même par aucun éditeur de formules. Au niveau des calculs, les systèmes de calcul formel ne disposent d'aucun outil pour gérer l'ensemble d'un calcul et pour guider l'utilisateur dans la résolution d'un problème. Enfin, à un niveau plus global, chaque système, qu'il soit général ou spécialisé, dispose de sa propre interface, et il n'existe pas à l'heure actuelle d'interface homme-machine qui soit utilisée pour accéder à plus d'un système de calcul formel.

Nos buts initiaux

L'objectif principal de ce travail est l'étude et la conception d'un modèle d'interface pour les systèmes de calcul formel doté de fonctionnalités originales destinées à améliorer significativement l'ergonomie par rapport aux interfaces existantes. Les problèmes relatifs à la conception et à la réalisation d'interfaces homme-machine sont difficiles en général, et nécessitent beaucoup d'expérimentations afin d'être résolus. Une bonne ergonomie pour une interface est un objectif difficile à atteindre, surtout quand les fonctionnalités de cette interface ne sont pas clairement spécifiées a priori, ce qui est le cas d'une interface pour les systèmes de calcul formel. Communiquer avec un tel système pose des problèmes spécifiques ardus qui ne se retrouvent pas dans le cadre des logiciels informatiques traditionnels. Pour atteindre notre objectif, nous avons mené au préalable un travail de réflexion réparti sur trois axes:

Etudes des interfaces existantes

Nous avons étudié et expérimenté les interfaces de la plupart des systèmes de calcul formel. Cette étude s'est limitée aux aspects liés à l'édition des formules mathématiques proprement dite, toute autre fonctionnalité (manipulation des expressions) étant en général absente. Nous avons également étudié et expérimenté la partie édition de formules mathématiques de la plupart des éditeurs de textes disponibles. Cette étude a montré que les fonctionnalités proposées par les interfaces ou les éditeurs restent minimales, et ne répondent pas, en général, aux attentes de l'utilisateur. Par exemple, les déplacements à l'intérieur d'une formule sont, dans la plupart des systèmes, syntaxiques, c'est à dire dépendants de la structure interne de la formule, et par conséquent peu intuitifs. Nous montrerons qu'un autre type de déplacement, les déplacements spatiaux, sont beaucoup mieux adaptés au contexte mathématique.

Etude de l'activité de l'utilisateur

En collaboration avec de 'vrais' utilisateurs de systèmes de calcul formel, nous avons étudié les problèmes relatifs à l'interface, rencontrés lors de la résolution complète d'un problème. Les difficultés liées à l'interface sont nombreuses et pénalisent significativement l'utilisateur durant son travail d'exploration et de calcul.

Réflexion sur l'ergonomie

En collaboration avec des mathématiciens et des ingénieurs, nous avons mené une réflexion plus générale sur les qualités, en terme d'ergonomie, qu'une interface devrait rassembler pour augmenter la productivité de l'utilisateur d'un système de calcul formel.

À l'issue de cette étude préalable, nous avons acquis la conviction qu'il était impossible de définir a priori les fonctionnalités d'une telle interface, et qu'il fallait plutôt envisager de fournir un prototype suffisamment souple pour permettre d'expérimenter de nouvelles idées, et d'implémenter le plus simplement et le plus rapidement possible plusieurs versions des fonctionnalités envisagées. À terme, une interface devant s'adapter aux exigences et aux caractéristiques d'un utilisateur ou d'un domaine particulier, la flexibilité et la paramétrabilité nous sont apparues comme les deux qualités essentielles que devait posséder un tel prototype. Pour réaliser ces objectifs, nous avons opté pour une architecture originale basée sur un composant d'édition paramétrable et facilement intégrable dans une application. Lorsque nous avons commencé cette thèse, un tel composant n'existait pas et nous avons pensé qu'il pouvait être utile. Aujourd'hui, des composants d'édition de formules ont été développés et sont disponibles sur différentes plate-formes, comme par exemple *MathEdit*, un éditeur de formules mathématiques pour *Windows* vendu par la société *K-Talk*, *Leibniz*, une interface au système *Mathematica* disponible sur *MacIntosh* ou encore *EqnWriter*, une applet *Java* distribuée par la société *Poliplus* qui permet, par exemple, l'édition de formules mathématiques dans une page web. Aucun de ces composants ne peut être intégré dans une application car tous sont des logiciels fermés, non extensibles et destinés à un usage unique. Ils montrent cependant que l'idée d'un composant spécialisé pour l'édition de formules mathématiques a séduit d'autres développeurs.

Les résultats obtenus

Un composant logiciel paramétrable assurant l'édition de formules mathématiques, baptisé *Emath*, a été développé et expérimenté au sein de deux applications: un éditeur de formules et une interface pour les systèmes de calcul formel intégrant des fonctionnalités nouvelles et originales. C'est cette interface qui nous a servi de plate-forme d'expérimentation pour valider nos idées en matière d'ergonomie. En outre, cette interface permet l'accès à plusieurs systèmes de calcul formel via le protocole *OpenMath*.

Le composant Emath

La conception et la réalisation d'*Emath* constituent l'essentiel de notre travail. *Emath* est un composant logiciel paramétrable pour l'édition de formules mathématiques. Il permet d'éditer incrémentalement en deux dimensions des formules mathématiques quelconques, et dispose d'un affichage soigné. Il permet la gestion de plusieurs fenêtres qui partagent ou non les mêmes formules, et offre des mécanismes élaborés pour la navigation dans une formule: déplacements syntaxiques et spatiaux, pointage et sélection multiple, etc. *Emath* n'est pas une interface mais un composant général pouvant être employé dans toute application nécessitant l'affichage ou l'édition de formules mathématiques. C'est un processus indépendant et il est utilisé comme un serveur par l'application qui l'intègre. *Emath* embarque un interprète Lisp via lequel il est possible d'ajouter, de modifier ou d'étendre n'importe quelle fonctionnalité (un peu à la manière du célèbre éditeur *Emacs*). *Emath* intègre plusieurs techniques sophistiquées destinées à fournir une édition de grande qualité, comme par exemple, l'utilisation de processus légers qui garantit un maximum de réactivité. Les deux principales qualités d'*Emath* sont son intégrabilité et sa paramétrabilité:

- il peut être intégré dans une application qui lui délègue une ou plusieurs fenêtres destinées à l'édition de formules. Un exemple d'une telle application est une interface pour un système de calcul formel.

- il est entièrement paramétrable. Ses comportements sont décrits par un automate modifiable, et peuvent donc être adaptés aux besoins dictés par le contexte d'utilisation.

Nous avons utilisé *Emath* pour construire un prototype d'interface plus ergonomique pour les systèmes de calcul formel, mais ce composant est destiné à être utilisé dans d'autres contextes, comme par exemple, l'édition de documents mathématiques ou l'affichage de formules mathématiques dans un browser *Web*.

Applications

Pour valider *Emath*, nous avons développé deux applications basées sur ce composant, un éditeur de formules et une interface pour les systèmes de calcul formel.

EdMa, un éditeur de formules

EdMa, est un Editeur Mathématique permettant d'éditer des expressions mathématiques et de générer le code *LaTex*, *HTML* ou *OpenMath* correspondant. Une formule encodée dans un de ces trois formats peut être lue et chargée dans l'éditeur, modifiée interactivement, puis transformée à nouveau dans un des formats disponibles. *EdMa* bénéficie de toute la puissance d'édition et de manipulation d'*Emath*: édition incrémentale en deux dimensions, déplacements spatiaux, sélection multiple, etc. En outre, *EdMa* offre des fonctionnalités originales comme le copier-coller inter-application qui permet par exemple de sélectionner une formule dans la fenêtre d'édition puis d'en coller la forme *LaTeX* par un simple clic souris dans une fenêtre de l'éditeur *Emacs*. Ce genre de fonctionnalité n'est pas intégrée a priori dans *Emath* mais peut être ajoutée et définie par l'application.

MathBoard, une interface tableau noir

Aboutissement de notre étude sur l'utilisation des systèmes de calcul formel, nous avons réalisé un prototype d'interface implémentant la plupart des fonctionnalités résultant de notre étude sur les interfaces. Ce prototype, appelé *MathBoard*, n'est pas une interface figée, mais constitue à notre connaissance la première plate-forme d'expérimentation en matière d'édition et de manipulation de formules dans le cadre du calcul formel. Toutes les fonctionnalités existantes sont modifiables et un grand nombre de nouvelles peuvent être ajoutées sans grande difficulté. Par exemple, *MathBoard* permet d'effectuer sur certaines formules des déplacements sémantiques, c'est à dire des déplacements qui tiennent compte de la sémantique de la formule. Si on déplace la variable b dans l'équation $a + b = 0$ du membre de gauche vers le membre de droite, la formule devient $a = -b$. Ce genre de comportement n'est pas intégré au cœur du composant d'édition, mais il est programmé dans l'application *MathBoard* qui exploite la grande paramétrabilité d'*Emath*. L'interface possède plusieurs fonctionnalités originales comme la sélection multiple ou la gestion de feuilles de calcul et peut accéder de façon concurrente à plusieurs systèmes de calcul formel.

Plan de lecture

Ce manuscrit se compose de cinq chapitres suivis d'une conclusion, d'appendices et d'une bibliographie.

Après une étude des interfaces existantes, nous présentons dans le chapitre I, page 13 un certain nombre d'idées en matière d'interface pour le calcul formel. Nous donnons également, dans ce chapitre, des éléments pour la réalisation d'une telle interface.

Le chapitre II, page 61, présente rapidement les principales caractéristiques du composant *Emath*. Cet aperçu peut servir de base pour la lecture des chapitres suivants.

L'architecture retenue pour la construction d'*Emath* est décrite dans le chapitre III, page 71. Cette section propose, par exemple, quelques réflexions sur l'utilisation d'un interprète embarqué.

Les points les plus originaux de l'implémentation d'*Emath* sont discutés au chapitre IV, page 93. Le lecteur y trouvera par exemple des commentaires sur les problèmes liés à l'utilisation du multi-threading.

Enfin, le chapitre V, page 129, présente les applications envisageables pour *Emath* et les interfaces effectivement réalisées.

Conventions typographiques

- Les caractères italiques indiquent les *noms propres*.
- Les caractères dans la fonte Palatine désignent les termes anglais: english terms.
- Les caractères de machine à écrire désignent du `code informatique` ou des `identificateurs`.

Chapitre I
Calcul symbolique et interfaces

La spectaculaire progression de la puissance des ordinateurs et les développements de la micro informatique permettent à un nombre de plus en plus important d'utilisateurs de bénéficier des possibilités du calcul symbolique. Les systèmes de calcul formel[1] sont des outils de plus en plus répandus. Beaucoup de calculatrices scientifiques intègrent maintenant un moteur de calcul formel. Ainsi, après le monde de la recherche, ce sont entre autres, les ingénieurs, les étudiants ou les économistes qui réalisent l'intérêt de ces outils. Ces nouveaux utilisateurs sont le plus souvent occasionnels. Le faible temps qu'ils consacrent à l'emploi de ces systèmes ne leur permet pas d'investir beaucoup d'énergie dans l'apprentissage des interfaces. Elles doivent donc être les plus intuitives possibles. Pour le moment, la qualité générale des interfaces aux systèmes de calcul symbolique n’est pas satisfaisante (le tableau 1, page 20, propose un comparatif rapide des principales interfaces). Les efforts ont en effet longtemps porté sur l'amélioration des moteurs de calcul ou l'enrichissement des bibliothèques de fonctions plus que sur l’interface. Aujourd’hui, il convient de reconsidérer ce point pour proposer des interfaces qui soient à la fois simple d’emploi et directement accessibles aux débutants, mais aussi riches et assurant aux utilisateurs expérimentés le maximum de fonctionnalités.

[1] Dans la suite de ce document, Système de Calcul Formel sera souvent abrégé en SCF.

Ce chapitre contient un bref aperçu historique de l'évolution des systèmes de calcul formel et de leur interface. Puis, une étude d'exemples nous permet de proposer quelques améliorations envisageables pour les interfaces aux systèmes de calcul formel. Enfin, nous présentons des considérations d'ordre général sur la conception des interfaces homme-ordinateur[1]. Ces réflexions sont le fruit de l'étude des interfaces existantes et de la lecture d'ouvrages sur ce thème comme [Coutaz90], [KS94] ou [Quin89].

1 Historique

Les premiers systèmes de calcul formel ont fait leur apparition dans les années 60 et se limitaient en général à des domaines d'applications spécifiques (*Formac* de *Bond* et *Tobey* en 1964, *Alpak* de *Brown* également en 1964). Les systèmes généraux tels que nous les connaissons aujourd'hui datent de la fin des années 60. Parmi les plus connus, on peut citer *Reduce* [Hearn85] et *Macsyma* [MG83], dont les premières versions ont vu le jour respectivement en 1968 et 1970. Basés sur le langage *Lisp*, ces systèmes n'ont que peu évolué depuis leur création, cédant le devant de la scène à *Maple* [CGGM88] et *Mathematica* [Wolf96] créés respectivement en 1980 et 1988 et tous deux écrits en *C*. Un peu en marge de ces logiciels, le système *Axiom* [JS92], successeur de *Scratchpad* [GJ71], est distribué depuis 1991 par la société *NAG*. Ce logiciel diffère fondamentalement dans sa conception des autres systèmes de calcul formel: avant tout un langage de programmation basé sur un système de type original, *Axiom* est particulièrement adapté à l'implémentation d'algorithmes mathématiques. Ces différents systèmes de calcul formel possèdent des fonctionnalités variées associées à diverses parties des mathématiques. Ils savent manipuler des nombres entiers ou flottants en précision arbitraire. Ils savent développer ou factoriser des polynômes ou des fractions rationnelles à nombre de variables quelconque et à coefficients quelconques. Ils peuvent dériver, intégrer des expressions contenant des fonctions élémentaires. Ils savent également résoudre des équations, des équations différentielles, calculer des développements limités et manipuler des matrices de taille et coefficients quelconques. En complément des systèmes de calcul formel commercialisés, il existe aujourd'hui un grand nombre de logiciels spécialisés dédiés à des domaines particuliers des mathématiques. Ces logiciels ne disposent pas de toutes les fonctionnalités d'un système général mais offrent des outils très efficaces spécifiques au domaine donné. Parmi les plus connus, on peut citer *Pari* et *Kant* pour la théorie des nombres, *Magma* et *Gap* pour la théorie des groupes, *Macaulay* pour les manipulations d'idéaux de polynômes et *Gb* pour les calculs de base de *Gröbner*.

[1] Aussi dénommées interfaces homme-machine ou IHM.

L'intérêt des chercheurs envers les interfaces homme-machine évoluées pour la manipulation des expressions mathématiques est ancien, et même antérieur à l'apparition des premiers systèmes de calcul formel. En 1963, *Marvin Minsky* propose un système pour la manipulation de formules mathématiques sur un écran d'ordinateur, le projet *Mathscope*, projet expérimental qui n'influencera pas la première génération de systèmes de calcul formel. Les systèmes *Macsyma*, *Reduce*, *Scratchpad* et *Maple*, dont les premières versions apparaissent dans les années soixante-dix, sont tous dotés d'interfaces "textuelles", interfaces adaptées aux terminaux alphanumériques. Le dialogue avec le système est basé sur une suite linéaire de requêtes-réponses. La requête est entrée par l'utilisateur dans une syntaxe linéaire et la réponse du système est affichée en deux dimensions sur plusieurs lignes. Les symboles spéciaux, comme le symbole d'intégrale $\int$ ou le symbole de sommation $\sum$, sont formés par l'assemblage de plusieurs caractères alphanumériques. Les expressions de taille importante sont affichées sur plusieurs lignes en utilisant césure et abréviations.

Il faut attendre 1986 pour voir l'apparition de *MathScribe* [SS86], la première interface graphique pour un système de calcul formel, le système *Reduce*. Cette interface est le résultat d'un travail de recherche sur les interfaces développé dans sa thèse par *Neil Soiffer* [Soif91]. *MathScribe* offre à l'utilisateur un environnement multi-fenêtres dans lesquelles il peut manipuler des expressions mathématiques en deux dimensions. L'édition des formules est particulièrement soignée et peut s'effectuer en deux dimensions incrémentalement, ou encore à l'aide de schémas syntaxiques disponibles dans des menus accessibles à la souris. On peut également effectuer des opérations de copier-coller sur des formules et utiliser des abréviations. Aucune de ces fonctionnalités n'est cependant paramétrable et *MathScribe* n'autorise la modification que d'un certain nombre d'attributs graphiques comme la couleur, les polices de caractères et la taille des fenêtres.

Un autre logiciel du même type que *MathScribe* apparaît en 1987 : c'est *GI/S* [YW87], une interface graphique pour le système *Macsyma*. Dans *GI/S*, l'édition des formules s'effectue linéairement mais, en revanche, le mécanisme de sélection est plus puissant que celui disponible en *MathScribe*. Par exemple, seule une partie d'une formule (termes consécutifs d'une somme ou d'un produit, sous-bloc d'une matrice) peut être sélectionnée. Au niveau plus global de la gestion des calculs, *GI/S* possède des fenêtres spéciales, appelées fenêtres scratch, qui permettent à l'utilisateur d'effectuer des calculs dans un contexte temporaire dans lequel des calculs et variables précédents ne sont pas visibles.

A la même époque apparaissent des petits systèmes de calcul formel pour *MacIntosh*, *PowerMath* [DR86] qui intègre comme *GI/S* la notion de contexte de calcul, et plus tard *Milo* [PI88] et *Theorist* [BW89] qui permettent de résoudre des problèmes mathématiques simples. L'interface de *Milo* est basée sur la notion de document, intégrant au sein d'une même fenêtre du texte, des formules et des tracés de courbes. Dans ces mini-systèmes de calcul formel, l'accent est mis sur l'édition et plusieurs fonctionnalités originales sont expérimentées, comme les déplacements sémantiques. L'éditeur *Milo* sera par la suite intégré à l'éditeur *FrameMaker* [FTC88]. *Theorist*, un assistant mathématique pour *MacIntosh*, propose un éditeur assez élaboré et possède un mode de déplacement sémantique. *Theorist* possède en outre des fonctionnalités d'édition puissantes comme la sélection multiple qui permet d'appliquer une opération à différentes sous-expressions sélectionnées.

Il faut mentionner ici l'expérience menée avec *CaminoReal* [ABIW88] à *Xerox PARC* dès 1988. *CaminoReal* est une interface permettant l'édition et la manipulation symbolique de formules. Cette interface permet d'utiliser les systèmes de calcul formel *Reduce*, *SMP* et *SAC-2* dans une architecture distribuée. C'est vraisemblablement le premier prototype d'interface graphique donnant accès à plusieurs systèmes de calcul formel. Par la suite, d'autres prototypes du même genre ont été développés comme le système *CAS/Pi* [Kajl92] basé sur *Centaur* [BCDI88].

2 Comparatif des interfaces actuelles

Dans les paragraphes précédents, nous avons dressé une bref historique des interfaces pour le calcul formel. Voici de façon plus détaillée, un comparatif des interfaces existantes.

2.1 Liste des interfaces

Cette section n'a pas pour but de d'établir une liste exhaustive des interfaces pour le calcul formel. Nous avons eu l'occasion de tester, au moins dans des versions de démonstration, un certain nombre de logiciels permettant la manipulation de formules. Nous ne nous sommes pas cantonnés aux interfaces spécifiquement dédiées au calcul formel, car des solutions intéressantes sont apparues notamment dans le domaine de l'édition de documents mathématiques. Pour rappel, les principaux systèmes existant pour le calcul formel sont *Axiom* [JS92], [GJ71], *Macsyma* [MG83], *Maple* [CGGM88], *Mathematica* [Wolf96]. Des systèmes plus modestes mais possédant une interface intéressante ont été développés sur MacIntosh comme *Theorist* [BW89] et *PowerMath* [DR86]. Parmi les interfaces, on peut citer *Mathscribe* [SS86], [Tekt88] et *MathStation* [MI89]. Des environnements graphiques expérimentaux pour le calcul formel ont été développés, comme *GI/S* [YW87] ou *Cas-Pi* [Kajl92]. Citons également *Inform* [EHV89].
Les logiciels pour l'édition de documents mathématiques les plus significatifs sont: *ScientificWorkplace* [TCI96], *LaTeX* [Knut84], *FrameMaker* [FTC88] qui inclut *Milo* [PI88], *MathType* [DS96], *MathWriter* [CS86] et *Expressionist* [Bona87]. Des logiciels plus

expérimentaux pour l'édition de documents mathématiques ont été également développés comme *Grif* [QVAR91] basé sur *Edimath* et *Figue* [Hasc91].

Lors du comparatif final, seuls les logiciels les plus intéressants, ou ceux faisant office de référence seront détaillés. Nous avons défini un ensemble de critères permettant leur évaluation.

2.2 Liste des critères

Réactivité.

Cette section comprend tous les points qui définissent une bonne réactivité de l'interface.

- Interruptibilité (IN): un calcul trop long peut-il être interrompu simplement? Il n'est pas facile pour l'utilisateur de prévoir la durée d'un calcul et il peut souhaiter stopper un calcul qui s'avère trop coûteux en ressources pour essayer de le reformuler.

- Parallèle (PL): est-il possible de mener plusieurs calculs en parallèle? Il est souhaitable que l'utilisateur puisse mener plusieurs raisonnements en parallèle. Certaines interfaces ne le permettent pas car elles interdisent l'édition tant que le calcul en cours n'est pas achevé. Il faut donc assurer que les calculs ne bloquent pas l'interface.

- Vitesse (V): La vitesse d'affichage est-elle suffisante? Ce critère est plus difficile à définir. Selon la plate-forme sur laquelle s'exécute l'interface, les résultats peuvent varier. Dans ce comparatif, elles ont toutes (sauf mention contraire) été testées sur la même machine mais parfois sous des systèmes d'exploitation différents. Une interface est jugée trop lente si la frappe d'un utilisateur expérimenté peut être limitée par la vitesse d'affichage. La vitesse d'exécution des calculs n'est bien sûr pas prise en compte car elle ne découle pas des performances de l'interface mais de celles du SCF sous-jacent.

Typographie

Les critères suivants sont regroupés du fait de leur impact sur la qualité de la typographie lors de l'affichage.

- Grande taille (GT): Les caractères de grande taille sont-ils adaptés à leur contexte? Les dimensions de certains symboles (comme les parenthèses ou l'intégrale) dépendent de leur environnement. Ces règles typographiques doivent être respectées si l'on souhaite avoir un affichage esthétique et lisible.

- Finesse de l'affichage (FA): Les utilisateurs sont habitués à lire des documents imprimés avec une résolution élevée (300dpi au moins). Les technologies actuelles d'écrans ne permettent pas d'obtenir la même finesse pour une interface (100dpi au plus). Tout effort pour tenter d'améliorer la qualité de l'affichage dans ce sens doit donc être noté. Des techniques basées sur l'emploi de dégradés de couleurs permettent, par exemple, de simuler une résolution plus forte [AP92].

Ergonomie

Cette section regroupe les critères qui permettent de juger la simplicité et l'efficacité de l'emploi de l'interface.

- Frappe intuitive (FI): L'entrée d'une formule est-elle facile à réussir pour un utilisateur novice? Cette mesure est assez subjective mais elle nous semble importante à mentionner. Il faut noter que, pour certains systèmes, la plupart des symboles ne peuvent être choisis que par l'intermédiaire d'un panneau de boutons (portant des schémas syntaxiques, voir figure 1). Ce choix est inacceptable car il induit un surcoût important dû au déplacement des mains.

- Puissance d'expression (PE): Des opérations complexes peuvent-elles être réalisées rapidement? Il s'agit de juger si un utilisateur expérimenté peut accélérer certaines manipulations complexes. Par exemple, la sélection multiple peut être employée comme raccourci pour exprimer certaines manipulations à plusieurs paramètres. Il est aussi agréable de pouvoir définir de nouvelles séquences de commandes (des macros). Ces critères sont plus difficiles à évaluer de manière quantitative. C'est donc plutôt une impression d'ensemble qui est notée.

- Déplacement intuitif (DI): Est-il rapide et intuitif de se déplacer d'un point à un autre de la formule grâce au clavier? Cette question est détaillée au paragraphe 4.5, page 56. Ici, nous avons noté le nombre d'opérations à effectuer pour aller d'un point à un autre et surtout le temps de réflexion nécessaire pour y parvenir.

- Multi-vue (MV): Est-il possible d'éditer plusieurs zones du même document au même instant? Ce point est crucial lorsque l'utilisateur souhaite avoir accès à des fragments du document trop éloignés pour être affichés sur la même page.

- Multi-style (MS): Est-il possible de voir le même document sous des aspects différents au même instant? Par exemple, il peut être utile de voir une grande matrice à la fois à son échelle normale et à une échelle réduite où l'on repère mieux sa structure globale.

- Correction d'erreurs (CE): Les mécanismes de correction d'erreurs sont-ils efficaces? Par exemple, est-il possible de revenir à plusieurs états antérieurs du document?

Figure 1 : Exemple de palette pour la saisie des opérateurs dans *Scientific Workplace.*

Évolutivité

Les critères suivants notent la possibilité de faire évoluer l'interface pour l'adapter aux besoins des utilisateurs et aux caractéristiques de différents SCF.

- Ajout de notations (AN): Il n'est pas possible de fournir une interface capable, à l'origine, d'afficher tous les symboles et de gérer toutes les notations possibles. La variété est telle qu'il est difficile d'être exhaustif. De plus, de nouvelles notations peuvent apparaître. Le plus sûr est donc d'autoriser leur ajout par les utilisateurs.

- Réutilisabilité (RE): Comme décrit au paragraphe 4, page 48, il est intéressant de pouvoir utiliser la même interface ou du moins les mêmes fonctions de base pour plusieurs SCF. Est-ce possible avec cette interface?

2.3 Protocole de comparaison

Les formules qui servent d'échantillon de base sont tirées de la littérature et sont les plus variées possibles (cf paragraphe 3, page 133). Les personnes qui ont participé au test sont au nombre de six et peuvent être classées de novice complet à expert confirmé dans la manipulation d'interfaces pour le calcul formel. Il n'a pas été possible de réunir au moins un habitué pour chaque interface du comparatif. Nous espérons que les résultats obtenus sont tout de même suffisamment significatifs.

2.4 Tableau comparatif et conclusion

Comme il l'a déjà été mentionné dans le paragraphe précédent, nous avons tenté de réaliser un test le plus significatif possible. Les logiciels sont exécutés dans les mêmes conditions[1]. Dans le tableau comparatif les * donnent les qualités relatives des différentes interfaces au regard des critères déjà mentionnés. L'absence d'étoile signifie que le critère n'est pas vérifié ou n'a pas de sens pour cette interface. Quand le critère est quantitatif, le nombre d'étoiles donne une idée du niveau atteint par l'interface.

	IN	PL	V	GT	FA	FI	PE	DI	MV	MS	CE	AN	RE
Mathematica 3.0	*		**	*	*	*	**	*		*	*	*	*
Maple V.4	*		**	*	*		*			*	*		
CasPi	*	*	*	*	*		**		*	***	*	**	*
FrameMaker 5	*		*	*	*	*	*			*	*		
ScientificWP 2.0	*		*	*	**			*			*		

Table 1: Comparatif des interfaces manipulant des formules mathématiques.

Ce tableau ne peut pas être présenté comme un test exhaustif des caractéristiques d'une interface pour les systèmes de calcul formel. Nous espérons toutefois que les éléments de comparaison qu'il présente démontrent que les interfaces actuelles ne proposent pas toutes les fonctionnalités souhaitables. La frappe ou les déplacements intuitifs sont très peu répandus. Le multi-vue n'est proposé que dans une interface expérimentale (*CasPi*). De même pour la notion de feuille de travail permettant de mener des calculs en parallèle. Par contre la qualité globale de l'affichage est bonne, que ce soit en terme de vitesse ou de typographie.

[1] Sur un Sun Sparc20 sous SunOS4 ou un PC Pentium100 sous Linux2.0.28 ou Windows95 tous deux dotés de 64 mega-octets de mémoire vive. Ces deux machines sont relativement comparables sur le plan des performances globales avec toutefois un léger avantage au PC pour la vitesse d'affichage. Durant les tests, toute surcharge de la machine a été évitée.

Paradoxalement, les logiciels les plus distribués ne bénéficient pas forcément des meilleures interfaces. Il faut noter les efforts récents de *Mathematica* et *Maple* pour offrir des interfaces plus complètes (bien qu'un peu complexes). Les progrès réalisés en matière d'édition et de déplacements bidimensionnels prouvent que la communauté des utilisateurs est sensible à ce problème. S

3 *Interfaces et calcul formel*

Dans cette section, nous allons commencer par nous intéresser aux besoins en interface des systèmes de calcul formel et plus généralement aux besoins des utilisateurs des mathématiques (ingénieurs, physiciens, mathématiciens...). Nous en déduirons quelques principes pour la conception d'une interface, principes qui seront mis en pratique dans le reste de cette section pour présenter un ensemble de solutions possibles pour les différents problèmes qui se posent.

3.1 *Les besoins*

L'utilisateur d'un système de calcul formel peut se livrer à deux activités assez distinctes. La première est une activité que nous qualifierons d'exploration mathématique, la seconde est une activité de programmation. Dans l'activité exploratoire, l'utilisateur essaye de résoudre un problème en enchaînant une série de commandes dans l'interface du système. Dans l'activité de programmation, l'utilisateur écrit, met au point, un programme écrit dans le langage du système. Bien entendu, l'écriture d'un programme peut être souvent une étape nécessaire à une activité exploratoire.

Dans la suite de ce travail, nous nous concentrerons essentiellement sur cette activité exploratoire, ceci pour deux raisons principales. La première est que l'activité de programmation relève d'une problématique plus générale et relativement éloignée des mathématiques: environnements de programmation, outils de débogage, *etc*. La seconde est que l'activité exploratoire est celle qui est la plus intéressante à étudier du point de vue des interfaces mais aussi celle où le plus de progrès intéressants pour l'avenir du calcul formel nous semble pouvoir être réalisés dans un futur proche.

Nous n'essayerons cependant pas d'occulter l'importance de la programmation dans les systèmes ni le fait que la séparation de ces deux activités n'est pas sans poser quelques problèmes d'interaction; les deux activités collaborent souvent pour la résolution d'un problème. Il nous semble cependant plus clair de séparer conceptuellement l'exploration de la programmation. Comme un bon langage de programmation n'est pas, en général, un bon langage de commande, un bon environnement de programmation n'est certainement pas un bon environnement d'exploration.

Il est important de noter ici que cette activité d'exploration peut utiliser des outils qui ne relèvent pas strictement du calcul formel, comme des traceurs de courbes et de surface ou des routines numériques. *In fine*, le résultat peut être soit de générer du code dans un langage conventionnel comme *C* ou *Fortran*, soit de produire un document dans un format natif au système utilisé (comme les “worksheets” de *Maple* ou les “notebooks” de *Mathematica*), ou encore dans un format plus universel comme *LATEX* ou *html* (dans un futur proche, lorsque ce langage supportera nativement les formules mathématiques).

Si les premiers systèmes avaient essentiellement pour but de faire réaliser des calculs qu'un être humain ne pouvait pas mener à bien dans un temps raisonnable, un objectif aujourd'hui (notre objectif) est bien de viser la résolution complète et interactive d'un problème à travers une interface qui puisse faire abandonner le crayon et le papier.

Dans le paragraphe suivant, nous allons en venir à l'essence du problème en essayant d'isoler les besoins de l'utilisateur des mathématiques indépendamment (ou presque) d'un système de calcul formel particulier.

3.2 Calcul formel, une critique des interfaces actuelles

Demandons-nous tout d'abord s'il existe une différence fondamentale entre la façon de calculer à la main et dans un système de calcul formel. Les systèmes actuels et leurs interfaces sont loin de permettre un travail aussi naturel que sur une feuille de papier. Nous ne nous intéressons évidemment pas aux questions ergonomiques de base, liées à l'utilisation des dispositifs d'entrée (comme le clavier et la souris) et d'un écran, mais à la façon différente d'organiser les calculs, de les enchaîner aux opérations de base et à la forme variée des résultats que l'on obtient. Parmi les problèmes immédiatement perceptibles par un néophyte utilisant un système pour faire ses mathématiques, on trouve:

- La difficulté de changer la forme d'une expression (réordonner les termes, faire certaines simplifications sur des morceaux).
- La difficulté de travailler sur des sous-expressions (appliquer une simplification ou une identité particulière sur un sous-terme).
- La difficulté de contrôler la forme d'un résultat: là ou l'utilisateur n'aurait calculé qu'un certain nombre limité de termes, ou pas du tout si une estimation avait montré un résultat trop complexe.

En règle générale, on peut dire que les systèmes actuels se concentrent sur les opérations algorithmiquement coûteuses, non pas sur les opérations élémentaires dont sont composés les calculs à la main. Leur philosophie est dans l'automatisation: plutôt un algorithme assez général qui "marche tout seul" que plusieurs petites fonctions qui nécessitent le choix de l'utilisateur.

Cette philosophie s'explique en partie par une certaine quête historique de l'automatisation absolue dont le but ultime serait une seule fonction `simplify` qui donnerait la "meilleure forme". Même si cette quête est bien sûr vaine, on peut aussi y voir l'héritage du caractère "batch" des premiers calculs, ainsi qu'une certaine envie de montrer la puissance du calcul formel, mais également une conséquence de la pauvreté des interfaces qui rendrait très frustrante une utilisation dans un autre mode.

Pour les lecteurs les moins familiers du domaine, signalons que le champ de prédilection des systèmes de calcul formel généraux actuels peut être qualifié simplement comme "les mathématiques de l'ingénieur", même si des systèmes comme *Maple* ou *Mathematica* intègrent des algorithmes en théorie des groupes, (par exemple) dont les "clients" sont différents. Ces "mathématiques de l'ingénieur" consistent essentiellement en la manipulation d'expressions contenant des fonctions transcendantes élémentaires, simplification, dérivation, intégration, développements en séries ou asymptotiques, résolution d'équations, évaluation numérique en précision arbitraire, algèbre linéaire "formelle", etc. Les connaissances de ces systèmes sur les fonctions spéciales ne sont cependant pas toujours ce qu'elles devraient être.

Si on accepte le système comme une base de lemmes ou un oracle, il n'existe pas de différences fondamentales entre preuves (ou morceaux de preuves) et sessions d'un système, mis à part le fait que l'on n'a pas forcément un but précis (un théorème à démontrer) mais que l'on explore. Cependant, l'accent est mis sur les calculs plus que sur la déduction. C'est pourquoi on ne peut pas espérer effectuer la preuve d'un théorème d'analyse dans les interfaces qui nous intéressent. Citons néanmoins l'intéressant travail autour de *Analytica*, un module *Mathematica* capable d'aider à la démonstration de théorèmes non triviaux [CZ93].

Il existe cependant une différence importante entre ces deux mondes, celle de la justesse. Tous les systèmes de calcul formel aujourd'hui n'effectuent que des calculs pouvant être qualifiés de génériquement justes au sens où le résultat d'un calcul faisant intervenir des paramètres peut être faux pour certaines valeurs des paramètres. Un exemple typique est:

```
>A:=matrix(2,2,[1,1,x,1]);

                               [ 1   1 ]
                         A := [        ]
                               [ x   1 ]
>rank(A);
                               2
```

Alors que:

```
>rank(subs(x=1,matrix(2,2,[1,1,x,1])));

                               1
```

Il n'y a dans le résultat rendu aucune trace de sa condition de validité.

Un exemple

Pour illustrer les problèmes éventuels rencontrés en mimant les calculs réalisés à la main dans un système de calcul formel, nous allons prendre un exemple classique, pouvant être considéré comme relativement typique. Cet exemple est l'étude d'une famille d'équations différentielles (non linéaires), dont la plupart des fonctions dites spéciales (fonctions de *Bessel*, fonctions hypergéométriques...) sont des solutions.

Nous avons utilisé le système que nous maîtrisions le mieux: *Maple*, en espérant en faire une utilisation assez optimale. Pour simplifier la présentation, nous utilisons l'interface "terminal" de *Maple*, version V.4.

Prenons l'équation différentielle:

où les $\sigma(z)$ sont des polynômes de degré au plus 2 et $\tau(z)$ est un polynôme de degré au plus 1.

En *Maple*, on écrit l'équation en donnant explicitement les dépendances par rapport à la variable z:

```
> Eq1 := diff(u(z), z, z) + p1(z)/q(z) * diff(u(z), z) + p2(z)/q(z)^2 *u(z)
= 0;
                                          /d        \
                        /          \   p1(z) |-- u(z)|
                        |d         |          \dz      /   p2(z) u(z)
              Eq1 :=    |--- u(z)| + --------------- + ---------- = 0
                        |  2       |        q(z)                2
                        \dz        /                         q(z)
```

Comme on le voit, la présentation du système est assez différente de la présentation humaine, où l'équation est représentée comme un polynôme en la fonction u et ses dérivées.

La méthode classique pour étudier cette équation est de regarder son comportement au changement de variable $u = \phi(z)y$. Ceci se traduit par:

```
> u(z) = phi(z) * y(z);
                                  u(z) = phi(z) y(z)
> subs(", "");
                                     /d              \
        / 2               \    p1(z) |-- phi(z) y(z)|
        |d                |          \dz             /   p2(z) phi(z) y(z)
        |--- phi(z) y(z)| + --------------------- + ----------------- = 0
        |  2              |            q(z)                        2
        \dz               /                                    q(z)
```

Les symboles spéciaux `"` et `""` représentent respectivement la dernière et l'avant-dernière expression calculée. Comme on le voit, la substitution a détruit toute la structure du résultat. On va donc faire de l'expression obtenue une équation différentielle en $y(z)$. On doit d'abord faire un `expand` avant de pouvoir faire un `collect` qui va effectivement regrouper les termes.

```
> expand(");

/ 2         \                                           / 2       \
|d          |           /d         \ /d        \         |d         |
|--- phi(z)| y(z) + 2 |-- phi(z)| |-- y(z)| + phi(z) |--- y(z)|
|  2        |           \dz        / \dz       /         |  2       |
\dz         /                                           \dz       /
              /d         \                       /d       \
       p1(z) |-- phi(z)| y(z)    p1(z) phi(z) |-- y(z)|
             \dz        /                     \dz      /   p2(z) phi(z) y(z)
     + --------------------- + --------------------- + -----------------
                 q(z)                   q(z)                       2
                                                               q(z)
= 0

> Eq11 := ":
```

On remarque que `expand` ne fait pas qu'opérer la distribution des produits sur les sommes, mais implémente aussi la règle de différentiation des produits. C'est bien ce que nous voulions, mais dans le cas où nous n'aurions souhaité que l'application de la règle sur la différentiation des produits, cela aurait été beaucoup plus compliqué. Pour regrouper les termes d'une manière plus classique, deux utilisations de `collect` sont nécessaires.

```
> collect(Eq11, diff(y(z), z));

                                               / 2          \
/p1(z) phi(z)      /d         \\ /d       \    |d           |
|------------ + 2 |-- phi(z)|| |-- y(z)| + |--- phi(z)| y(z) +
\    q(z)          \dz        // \dz      /    |  2         |
                                               \dz          /


                                                  /d         \
                          / 2       \    p1(z) |-- phi(z)| y(z)
p2(z) phi(z) y(z)         |d        |          \dz        /
----------------- + phi(z) |--- y(z)| + ---------------------- = 0
          2               |  2      |             q(z)
      q(z)                \dz       /

> collect(", y(z));

/                                        /d         \\
|/ 2        \                      p1(z) |-- phi(z)||
||d         |   p2(z) phi(z)             \dz        /|
||--- phi(z)| + ------------ + -----------------| y(z)
||  2       |          2                q(z)         |
\\dz        /      q(z)                              /

                                                           / 2       \
       /p1(z) phi(z)      /d         \\ /d       \          |d        |
     + |------------ + 2 |-- phi(z)|| |-- y(z)| + phi(z) |--- y(z)| = 0
       \    q(z)          \dz        // \dz      /          |  2      |
                                                           \dz       /

> Eq12 := ":
```

Ce qui donne une forme beaucoup plus raisonnable. En fait, on aurait pu abréger cela en `collect(Eq11, [diff(y(z), z), y(z)]);`. On remarque la nécessité de nommer les résultats intermédiaires au cas où une manipulation n'aurait pas l'effet escompté. Par exemple, on aurait pu vouloir utiliser l'option `diff` de `collect` sur `Eq11`, ce qui semble naturel en lisant la page de manuel. Malheureusement, on obtient l'expression suivante:

```
> collect(Eq11, diff);

                                               / 2          \
/p1(z) phi(z)      /d         \\ /d       \    |d           |
|------------ + 2 |-- phi(z)|| |-- y(z)| + |--- phi(z)| y(z) +
\    q(z)          \dz        // \dz      /    |  2         |
                                               \dz          /
```

```
                                                     /d         \
                              / 2        \   p1(z) |-- phi(z)| y(z)
p2(z) phi(z) y(z)            |d          |         \dz       /
----------------- + phi(z) |--- y(z)| + ---------------------- = 0
          2                  |  2       |            q(z)
      q(z)                  \dz        /
```

ce qui s'explique bien sûr par le fait qu'il n'y a pas moyen de dire de quelle fonction on souhaite “collecter” les dérivées. Malheureusement, notre expression contient les dérivées de `phi` et `y`.

L'idée de base maintenant consiste à transformer l'équation obtenue en une équation de la même forme que l'équation de départ, en faisant, bien entendu, les hypothèses nécessaires sur le changement de variable. On commence donc par diviser par `phi(z)` pour retrouver un coefficient unité pour la dérivée seconde:

```
> Eq12/phi(z);
//                                      /d         \\
||/ 2         \                   p1(z) |-- phi(z)||
|||d          |   p2(z) phi(z)          \dz       /|
|||--- phi(z)| + ------------ + -----------------| y(z)
|||  2        |         2               q(z)       |
\\\dz         /      q(z)                          /
                                                                \
                                                     / 2       \|
       /p1(z) phi(z)     /d        \\ /d       \      |d        ||
     + |------------ + 2 |-- phi(z)|| |-- y(z)| + phi(z) |--- y(z)||/phi(z)
       \    q(z)         \dz       // \dz      /      |  2      ||
                                                     \dz       //
= 0
```

Malheureusement, pour rendre la division effective, nous devons faire un `expand` qui va casser notre belle forme. Nous devons donc refaire les `collect` successifs:

```
> expand("):  collect(", [diff(y(z), z), y(z)]);

/ 2                                    \
|d                                     |
|--- phi(z)                  /d       \|
|  2                   p1(z) |-- phi(z)||
|dz            p2(z)         \dz       /|
|---------- + ----- + -----------------| y(z) +
|  phi(z)         2       phi(z) q(z)   |
\              q(z)                    /


/          d        \
|          -- phi(z)|             / 2       \
|p1(z)     dz       | /d       \  |d         |
|----- + 2 ---------| |-- y(z)| + |--- y(z)| = 0
\q(z)       phi(z)  / \dz      /  |  2       |
                                  \dz       /
> Eq13 := ":
```

Cette fois, nous avons atteint notre but. Il aurait évidemment été plus simple de diviser avant de mettre sous la forme “normale”, mais ce n’était pas directement naturel alors.

Pour retrouver une équation similaire à celle du départ, commençons par nous intéresser au coefficient de la dérivée première. L’extraire n’est pas très simple. Une tentative comme:

```
> coeff(", diff(y(z), z));
Error, unable to compute coeff
```

échoue. La fonction `coeff` n’admet pas un tel argument (elle ne semble admettre qu’une variable ou une variable à une certaine puissance). Nous avons donc deux solutions. La première consiste à aller récupérer le coefficient par une suite de `op` (`lhs` récupère le “left hand side” d’une équation):

```
> op(1, (op(2, lhs(Eq13))));
                                          d
                                          -- phi(z)
                               p1(z)      dz
                               ----- + 2 ---------
                               q(z)        phi(z)
> c1 := ":
```

À noter que dans un autre cas, ce serait beaucoup plus compliqué, car la structure des expressions algébriques n’est pas directement la structure visible. Ainsi on a

```
> op(2, a/b);
                                        1/b
> op(2, a - b);
                                        -b
```

Une méthode plus sûre pour récupérer le coefficient consiste à écrire une fonction plus générale que `coeff`, que nous appelons `gcoeff`:

```
> gcoeff := proc(e, c) local __COEFF; subs(c = __COEFF, e); coeff(",
__COEFF) end:
> c1 := gcoeff(lhs(Eq13), diff(y(z), z));
                                            d
                                            -- phi(z)
                                 p1(z)      dz
                          c1 := ----- + 2 ---------
                                 q(z)        phi(z)
```

Pour retrouver une équation similaire à l’équation initiale, il faut que ce coefficient si durement extrait soit de la forme `tau(z)/q(z)` où `tau(z)` est un polynôme de degré au plus 1 (comme `p1(z)`). Si on écrit

```
> c1 = tau(z)/q(z);
```

On peut résoudre “en phi” (trouver une condition sur phi):

```
> solve(", diff(phi(z), z)/phi(z));
                                   -p1(z) + tau(z)
                               1/2 ---------------
                                         q(z)
> A := ";
```

Il faudrait maintenant injecter cette valeur pour la dérivée logarithmique de `phi` dans l’équation `Eq13`. Cela n’est pas tout à fait facile... La substitution par `subs` étant purement syntaxique, on a par exemple:

```
> subs(a/b = A, c*a/b);
                                        c a
                                        ---
                                         b
```

Il faut donc utiliser une fonction de substitution plus élaborée. La version V.4 de *Maple* propose la fonction `algsubs` pour cela. Elle n’est cependant pas d’une utilité immédiate:

```
> algsubs(diff(phi(z), z)/phi(z) = A, Eq13);
Error, (in algsubs) cannot compute degree of pattern in, phi(z)
```

Il faut, encore une fois, contourner le problème. Puisqu’il semble que le système n’apprécie pas le `1/phi(z)` on le remplace par une variable temporairement:

```
> subs(1/phi(z) = T, Eq13):
```

et ensuite un appel à `algsubs` fonctionne bien comme on s’y attendait

```
> algsubs(T*diff(phi(z), z) = B, ");
/                                                          / 2        \
|                                    /d       \        2   |d         |      2
|p1(z) y(z) B q(z) + p2(z) y(z) + 2 |-- y(z)| B q(z)   + |--- y(z)| q(z)
|                                    \dz      /            |  2       |
\                                                          \dz        /

                                  \               / 2         \
                /d       \         |   /      2    |d          |
     + p1(z) |-- y(z)| q(z)|   /  q(z)   + T |--- phi(z)| y(z) = 0
                \dz      /         |  /            |  2        |
                                  /               \dz         /
> Eq14 := ":
```

et les dérivées premières de `phi(z)` ont bien toutes disparues. Il est à noter que substituer directement par la valeur de `T*diff(phi(z), z)` (`A` dans notre session *Maple*) ne marche pas. On obtient:

```
/  /d       \                  \ /d         \
|2 |-- y(z)| q(z) + p1(z) y(z)| |-- phi(z)|
\  \dz      /                  / \dz        /
-------------------------------------------
                q(z) phi(z)
                                           / 2      \
             /d      \                     |d       |     2
       p1(z) |-- y(z)| q(z) + p2(z) y(z) + |--- y(z)| q(z)
             \dz     /                     |  2     |
                                           \dz      /
     + ----------------------------------------------------
                                  2
                              q(z)
         / 2        \
         |d         |
     + T |--- phi(z)| y(z) = 0
         |  2       |
         \dz        /
```

Il reste encore à faire disparaître la dérivée seconde de `phi`. Pour cela, on essaye de dériver l'expression suivante:

```
> diff(diff(phi(z), z)/phi(z), z);
                                             2
                                            d
                             /d        \2   --- phi(z)
                             |-- phi(z)|      2
                             \dz       /    dz
                           - ------------ + ----------
                                     2        phi(z)
                               phi(z)
```

et on essaye d'obtenir une expression fonction de la dérivée logarithmique

```
> solve(", diff(phi(z), z, z));
                                  /d        \2
                                  |-- phi(z)|
                                  \dz       /
                                  ------------
                                     phi(z)
```

On peut donc injecter cette valeur pour la dérivée seconde de `phi` dans l'équation courante

```
> subs(diff(phi(z), z, z) = ", Eq14);
/                                                       / 2      \
|                                   /d      \       2   |d       |     2
|p1(z) y(z) B q(z) + p2(z) y(z) + 2 |-- y(z)| B q(z)  + |--- y(z)| q(z)
|                                   \dz     /           |  2     |
\                                                       \dz      /
                                           /d        \2
                           \             T |-- phi(z)|  y(z)
             /d      \     |   /     2     \dz       /
     + p1(z) |-- y(z)| q(z)|  /  q(z)  + ------------------- = 0
             \dz     /     | /                 phi(z)
```

Il nous reste encore quelques efforts à faire pour éliminer la dérivée première de `phi` restante

```
> subs(1/phi(z) = T, ");
/                                                       / 2      \
|                                   /d      \       2   |d       |     2
|p1(z) y(z) B q(z) + p2(z) y(z) + 2 |-- y(z)| B q(z)  + |--- y(z)| q(z)
|                                   \dz     /           |  2     |
\                                                       \dz      /

                           \
             /d      \     |   /     2    2 /d        \2
     + p1(z) |-- y(z)| q(z)|  /  q(z)  + T  |-- phi(z)|  y(z) = 0
             \dz     /     | /              \dz       /
                           /
```

et ensuite

```
> algsubs(T*diff(phi(z), z) = B, ");
/
|      2     2                                      /d      \       2
|y(z) B  q(z)  + p1(z) y(z) B q(z) + p2(z) y(z) + 2 |-- y(z)| B q(z)
|                                                   \dz     /
\

       / 2      \                             \
       |d       |     2         /d      \     |   /     2
     + |--- y(z)| q(z)  + p1(z) |-- y(z)| q(z)|  /  q(z)  = 0
       |  2     |               \dz     /     | /
       \dz      /                             /
```

Il ne reste plus qu'à clarifier le résultat, en commençant par remplacer B par sa valeur:

```
> expand("): collect(", [diff(y(z), z), y(z)]);
                                                             / 2      \
     /      p1(z)\ /d      \   / 2   p1(z) B   p2(z)\        |d       |
     |2 B + -----| |-- y(z)| + |B  + ------- + -----| y(z) + |--- y(z)| = 0
     \      q(z) / \dz     /   |      q(z)          2|        |  2     |
                               \               q(z) /        \dz      /
```

et pour finir:

```
> subs(B = A, ""): expand("): collect(", [diff(y(z), z), y(z)]);
  /d      \
  |-- y(z)| tau(z)   /           2             2        \        / 2      \
  \dz     /          |      p1(z)        tau(z)    p2(z)|        |d       |
  ---------------- + |- 1/4 ------ + 1/4 ------- + -----| y(z) + |--- y(z)|
= 0
        q(z)         |          2             2        2|        |  2     |
                     \      q(z)          q(z)     q(z) /        \dz      /
```

Un dernier effort nous permet d'avoir un dénominateur unique

```
> map(factor, ");
    /d      \
    |-- y(z)| tau(z)               2         2                   / 2      \
    \dz     /               (-p1(z)  + tau(z)  + 4 p2(z)) y(z)   |d       |
    ---------------- + 1/4 ----------------------------------- + |--- y(z)|
          q(z)                                2                  |  2     |
                                          q(z)                   \dz      /
```

Le but initial est maintenant atteint. L'équation obtenue est bien du même type que l'équation de départ: le coefficient de y est bien une fraction rationnelle dont le numérateur et le dénominateur sont de degré au plus 2 et le coefficient de y' est bien une autre fraction rationnelle mais dont numérateur et dénominateur sont de degré 1. Notons que la "démonstration" précédente occupe moins de deux pages dans un ouvrage de référence [NU88].

3.3 Catégorisation des opérations

Les fonctionnalités d'un système sont nombreuses et variées (plusieurs centaines de fonctions dans des systèmes comme *Maple* ou *Mathematica*. Elles regroupent des opérations de nature très différentes comme des opérations "mathématiques" (factorisation, intégration, résolution d'équations, calcul du déterminant,...), des opérations de "mise en forme" d'expression (développement d'un produit, mise sous forme polynomiale) ou encore des opérations produisant d'autres résultats que des expressions (tracer de courbes et de surfaces, production de code *C* ou *fortran*). Comme point de départ à notre réflexion sur la conception d'interface, nous pouvons essayer de donner une catégorisation des opérations "élémentaires" du calcul formel. Les opérations que nous allons catégoriser ne correspondent pas directement aux fonctionnalités présentes dans les systèmes, mais bien aux "pas de calculs" qui devraient être disponibles depuis une interface plus naturelle. On peut distinguer les catégories suivantes:

Substitutions

La substitution est le remplacement d'un objet par un autre dans une ou plusieurs expressions. Il peut s'agir de substitutions simples (une variable par une expression) ou "générales" (une expression par une autre), sur une ou plusieurs occurrences. A priori, le cas où l'on substitue toutes les occurrences (dans une expression) semble le plus utile. On notera que l'opération de substitution peut réaliser aussi l'évaluation numérique d'une formule (par exemple, substituer n par 1 dans $n^2 + n + 1$ donne 3).

Distributions

La distribution consiste à appliquer la distributivité sur une expression produit et la transforme donc en somme (si l'un des termes est une somme). On peut agir sur un seul ou tous les produits d'une expression. C'est une partie du comportement de la fonction `expand` de *Maple*.

Mise en ordre

La “mise en ordre” a pour but de réordonner les termes des opérateurs commutatifs (sommes et produits) dans une expression. Elle peut se faire d’une manière locale (on permute des termes d’une somme ou d’un produit particulier dans une expression) ou globale. Elle nécessite, dans ce dernier cas, la spécification d’un ordre. Les ordres les plus simples sont les ordres “polynômiaux”: on voit l’expression comme une somme, dont les termes sont de la forme $c_i\, x_i$ que l’on ordonne suivant les puissances croissantes ou décroissantes de x. Comme souvent dans la suite, il faut entendre “variable” au sens large (celui des noyaux de *Reduce*): ce peut être aussi bien une variable comme x qu’une expression comme sin x. Ainsi on peut ordonner le polynôme trigonométrique $sin\ x + a + b\ sin^2\ x$ en $b\ sin^2\ x + sin\ x + a$.

Mise sous forme polynomiale

La mise sous forme polynômiale transforme une expression en un polynôme en une ou plusieurs variables, ces variables pouvant être elles-même des expressions. Les fonctions `collect` de *Maple* ou `Collect` de *Mathematica* sont des exemples de “mise sous formes polynômiales”. Une mise sous forme polynômiale doit réaliser l’utilisation de la distributivité (fonction `expand` de *Maple*) quand c’est nécessaire (ce n’est pas le cas pour la fonction `collect` de *Maple* par exemple).

Compatibilités opérations/prédicats

Certaines opérations (somme, produit) sont compatibles avec une relation (égalité, inégalité). Cette compatibilité peut être conservée lors du déplacement des termes d’un coté à l’autre de la relation. Les opérations qui implémentent cette compatibilité entre opérations et prédicats permettent typiquement des transformations comme $a\ x > b$ devient $x > b/a$. Dans le cas où une division est introduite, ces transformations peuvent donner lieu à l’introduction de cas particuliers (dans l’exemple précédent, $a = 0$).

Résolutions simples

Les résolutions simples sont celles dont l’implémentation est quasiment triviale. Des exemples de telles résolutions sont la résolution d’une équation linéaire ou quadratique en une variable, ou bien les cas où l’application d’une fonction inverse suffit. Par exemple, la résolution en x de a ln $x = y$ donne simplement $x = e^{y/a}$. Ces transformations peuvent également introduire des cas particuliers et des ensembles de solutions.

Application de règles

L'application de règles est une opération très générale où l'on utilise une formule connue pour changer la valeur d'une sous-expression. En général, cette formule est une identité comme $\ln xy = \ln x + \ln y$. Elle permet de remplacer un sous-terme comme $\ln a(b + 1)$ par le terme $\ln a + \ln (b + 1)$, ou en sous-terme comme $\ln a + \ln b$ par $\ln ab$. Cette reconnaissance de motif doit s'entendre "syntaxiquement". On peut appliquer $sin^2 x + cos^2 x = 1$ à la bonne sous-expression de $1 + a + sin^2 (a + b) + cos^2 (a + b)$ mais pas à $1 + sin^2 (a + b) + a + cos^2 (a + b)$ ou à $1 + a + sin^2 (a + b) + cos^2 (b + a)$ sans une étape de mise en ordre des termes. On peut faire rentrer dans ce cadre assez général bon nombre de manipulations: la mise à un dénominateur commun, la simplification d'un facteur commun, la racine d'un carré qui devient la valeur absolue....

Opérations "complexes"

On regroupe ici toutes les autres opérations "non-triviale" d'un point de vue implémentation, dont l'exécution peut prendre un temps assez long et aussi quelques autres (comme la différentiation) :

- factorisation
- intégration et sommation
- résolution d'équations différentielles
- résolution générale
- différentiation
- calculs de limites
- développements limités

Certaines de ces fonctionnalités ne sont pas disponibles dans la plupart des systèmes, comme par exemple le fait de pouvoir donner un ordre arbitraire aux termes d'une somme (un système doit garder un ordre "canonique"). On peut aussi noter que les systèmes s'intéressent essentiellement à la dernière catégorie d'opérations. L'intérêt de cette classification réside bien entendu dans son aptitude à rendre compte des besoins et l'adéquation des fréquences du découpage que nous avons réalisé. Nous allons donc la confronter, dans le paragraphe suivant, avec un certain nombre d'exemples représentatifs de ce que nous voulons résoudre.

3.4 Des exemples

Dans ce paragraphe, nous allons nous pencher sur la façon dont les systèmes de calcul formel sont effectivement utilisés ou du moins pourraient l'être. Pour cela, nous avons consulté un certain nombre d'articles d'applications du calcul formel, les quelques livres où le calcul formel est appliqué (quelquefois à un domaine précis comme l'économie et la finance) et un cours d'université conçu autour de l'utilisation d'un système *Maple*.

Quelques exercices de base

Un premier jeu d'exemples que nous avons examiné provient d'un recueil d'exercices de l'université de Waterloo (Ontario, Canada). Il est à noter que Waterloo est l'université où *Maple* a été conçu. Il s'agit d'un ensemble d'exercices destinés à être résolus entièrement grâce à *Maple*. Ces exercices sont assez courts (typiquement entre une demi-douzaine et une vingtaine de lignes de code *Maple*). On trouve y aussi quatre problèmes plus longs (avec entre douze et cinquante deux opérations nécessaires à leurs résolutions).

Après avoir analysé les solutions proposées (le code *Maple* de la solution est donné), deux sortes d'observations peuvent être faites: d'abord sur l'importance relative de certaines opérations, et ensuite sur l'adéquation entre les opérations fournies par *Maple* et l'usage qui en est fait. Notons bien que le fait que *Maple* soit le système cible n'est en aucun cas fondamental ici. Transposer les solutions en *Mathematica* (par exemple) mènerait aux mêmes conclusions.

En ce qui concerne la fréquence des opérations, il est évident qu'il ne s'agit pas de discuter de la fréquence de `limit` par rapport à `diff`, ce qui n'aurait évidemment pas grand sens. Par contre, il est intéressant de noter la fréquence d'apparition des opérations des autres catégories. On remarque une fréquence relative assez forte des substitutions. Plus de 30% des opérations dans ces exercices sont des substitutions. La résolution d'équation par `solve` représente environ 13% des opérations. Les opérations de simplification comme `simplify` ou `normal` interviennent pour environ 15%. Il faut cependant noter que cet échantillon est très orienté “calcul”. Les expressions manipulées sont assez petites et très peu d'efforts sont faits pour mettre les résultats sous une forme particulière ou pour effectuer des simplifications. Remarquons aussi qu'aucun exercice ne fait intervenir plus de dix opérations différentes.

En ce qui concerne l'adéquation des opérations fournies à la tâche, on peut dire qu'elle est en général mauvaise. La plupart des `expand` sont suivis de `collect` et ne sont donc que des mises sous forme polynômiale. Les quelques occurrences de `factor` sont souvent aussi des mises sous forme polynômiale quand les “variables” sont en fait des expressions.

Des articles d'applications

Nous avons analysé neuf articles ([AAF82], [Boyd93], [CJ90], [GQ86], [Koto86], [Mack94], [Ogil82], [RDBE87] et [Wyan81]) décrivant des applications du calcul formel. Trois de ces articles décrivent des résultats originaux (au sens où ils étaient nouveaux et n'ont pu être obtenus que grâce au calcul formel), quatre montrent comment le calcul formel permet de faire des calculs plus facilement (de retrouver des résultats) et les deux autres sont des articles plus généraux sur l'application du calcul formel à un certain domaine assez vaste (applications en chimie physique et en génie chimique). Ces articles utilisent une large gamme de systèmes: *Macsyma*, *Maple*, *Reduce*, *Scratchpad* (ancêtre d'*Axiom*) et *muMath* (ancêtre de *Derive*). Il semble que tous les auteurs soient des spécialistes du domaine de l'article et aucun réellement spécialiste du calcul formel.

Parmi les remarques que l'on peut faire sur le contenu "calcul formel" de ces articles, notons:

- Seul un petit nombre de fonctionnalités est en général utilisé pour résoudre un problème donné. Le nombre maximum d'opérations appelées (au sens de `factor`, `simplify`...) ne dépasse pas 15 dans notre échantillon. Il est, en moyenne, plus proche de 6. Il était de 7 dans notre exemple de référence. Ceci contraste avec le très grand nombre d'opérations disponibles dans des systèmes comme *Maple* ou *Mathematica* (plusieurs centaines).

- Il est possible que l'utilisateur ait à taper des formules relativement grandes. On atteint quelquefois une trentaine d'opérateurs (en comptant les opérations arithmétiques et les références à des fonctions), ce qui représente plus d'une ligne d'une interface "classique".

- Un effort substantiel peut être nécessaire pour mettre les résultats sous une forme donnée ou pour obtenir que certaines simplifications soient faites.

3.5 Conception d'une interface

Plusieurs aspects interviennent dans la conception d'une interface "pour les mathématiques":

- saisie des formules

- expression des opérations de bases (menus, séquences de touches, manipulations à la souris...) et leurs actions sur l'interface

- organisation des calculs
- présentation des résultats

Ces différents points vont être développés dans les paragraphes suivants.

Evaluation et validation

Il est évidemment important de disposer d'une méthode pour évaluer une interface, qui puisse permettre de comparer effectivement deux interfaces. L'idée de base est bien sûr que les opérations les plus simples doivent pouvoir se faire le plus facilement possible...

Si on essaye d'appliquer un modèle classique comme le modèle *Keystroke* [CMN87] à l'évaluation de ce type d'interface, il faut être prudent. Les actions dans ce type d'interfaces sont beaucoup plus "réfléchies" et l'adéquation des fonctionnalités de base aux besoins est un point très important qui dépasse le simple cadre traditionnel des interfaces homme-machine.

A la lumière de ce que nous avons vu dans les sections précédentes, nous pouvons proposer l'utilisation conjointe d'un modèle *Keystroke* et d'un jeu de tests. Le modèle *Keystroke* se prête bien à l'évaluation de certains aspects de l'interface: facilité de saisie des formules et facilité de correction d'une erreur et déplacement dans une formule.

La composition d'un bon jeu de tests est un point non trivial. En ce qui concerne les problèmes de saisie, il est relativement simple d'exhiber des formules tests qui soient suffisamment représentatives. Pour les aspects plus complexes de résolution d'un problème mathématique, la sélection de cas tests représentatifs est plus délicate. De bonnes sources sont des ouvrages "calculatoires" ou des recueils d'exercices du niveau classes préparatoires aux grandes écoles.

3.6 Saisie

La fonctionnalité de base offerte par l'interface est la saisie des formules mathématiques. Bien que la plupart des formules saisies par l'utilisateur ne dépasse en général pas une ligne de texte (en notation mathématique usuelle) l'utilisation d'une notation "informatique" unidimensionnelle peut rendre les choses assez complexes. Par exemple, une intégrale comme:

s'écrit dans *Maple*:

```
int((1 + x)/(x*sqrt((x - a)*(x - b)*(x - c))), x);
```

La saisie d'objets intrinsèquement bidimensionnels comme les matrices est bien sûr encore plus délicate. La matrice:

est entrée comme:

```
array(1..6, 1..6, [[2, -1, 0, 0, 0, 0], [-1, 2, -1, 0, 0, 0], [0, -1, 2,
-1, 0, 0] [0, 0, -1, 2, -1, 0], [0, 0, 0, -1, 2, -1], [0, 0, 0, 0, -1, 2]]);
```

Les notations précédentes rendent la correction délicate. Tout cela milite pour une saisie sous la forme bidimensionnelle habituelle. Il existe de nombreuses variantes dans les méthodes de saisie des formules. Plusieurs études ont déjà été effectuées dans ce domaine. Les plus complètes, dans le cadre de la saisie de formules en deux dimensions, sont celles de Neil Soiffer [Soif91],[KS94] et de Ken Rimey [Rime95]. Le problème principal semble être le conflit entre la nature structurée des notations mathématiques et notre tendance à penser en terme de caractères frappés de la gauche vers la droite. Considérons l'expression:

$$ax^2+bx+c$$

Le premier objet visible constituant la formule est le symbole `a`. C'est aussi celui que l'on commencerait à écrire si on notait cette formule sur une feuille de papier. Pourtant la structure sous-jacente est différente. Cette expression est un arbre constitué de trois sous-arbres, deux produits et un symbole, et dont la racine est l'opérateur + :

Cette structure arborescente est la représentation interne classique des expressions dans un système de calcul formel. La méthode consistant à forcer l'utilisateur à manipuler la formule sous cette forme arborescente n'est pas efficace. L'effort de concentration qu'il faut fournir pour penser cette formule en terme d'arbre est encore raisonnable mais il devient pénible dès que la profondeur de l'arbre augmente un peu. La saisie doit donc s'effectuer en respectant l'habitude de penser les expressions mathématiques de gauche à droite. Mais il n'est pas simple de concevoir un éditeur qui utilise la structure arborescente de la formule en permettant à l'utilisateur de la manipuler sous cette forme plus linéaire. De gros progrès ont été réalisés dans ce domaine. Des éditeurs comme celui de Kaava [Rime95] ou de FrameMaker [FTC88] permettent par exemple de saisir la formule:

$$ax^2+b$$

Figure 2 : Arbre syntaxique de l'expression *ax*2+*bx*+*c*

assez naturellement par la frappe des touches a, *, x, ^, 2, <espace>, <espace>, + et b. La touche espace permet de préciser sur quel ensemble de termes porte la somme. Il nous semble qu'une saisie encore plus naturelle est envisageable. En prenant des décisions en fonction du contexte dans lequel une touche est frappée, il est possible de rapprocher encore un peu plus la saisie d'une formule de la façon de la penser. Toujours en prenant l'exemple de la saisie d'un polynôme, la frappe des signes multiplié est facultative. De même, si l'on admet que les identificateurs de variable sont réduits à une lettre, le signe ^ est inutile. Il suffit alors de frapper:

$$ax^2+3$$

Cette série de touches correspond exactement à la façon de prononcer cette formule. Il est difficile d'imaginer plus intuitif. Ce genre de conventions est très bien adapté à la saisie de la plupart des expressions usuelles, mais elle peut bien sûr être prise en défaut. Par exemple, pour saisir la formule:

$$(a-b)\,x^{a+b}+c$$

il faut indiquer un parenthésage explicite pour éviter que la convention du multiplié implicite ne crée l'expression:

$$a-bx^a+b+c$$

Toutefois, il faut noter que ce type de parenthésage correspond aux pauses effectuées lorsque l'on dicte une formule de ce type. L'expérience montre que cette méthode est assez naturelle. Il serait intéressant de quantifier plus finement son impact sur la vitesse de frappe et le nombre d'erreurs commises.

Ce type d'amélioration de la saisie nécessite un moteur d'analyse très souple permettant notamment d'adapter la saisie au contexte. Des détails plus techniques sont donnés dans le paragraphe 4, page 79.

3.7 Manipulations

La manipulation des formules ne s'effectue pas toujours simplement ni intuitivement dans les interfaces habituelles des systèmes de calcul formel. De nombreuses opérations peuvent être appliquées à une formule mathématique. Les actions classiques comme couper, copier, coller sont les plus simples. Mais des opérations plus mathématiques comme la simplification, l'évaluation, la factorisation ou la substitution peuvent aussi être vues comme des manipulations de formules. Ces types de manipulations plus sémantiques doivent pouvoir s'effectuer tout aussi simplement. La sélection permet une grande richesse d'expression et peut être utilisée avec profit dans ce contexte.

Sélection simple

Pour des opérations qui ne nécessitent qu'un seul paramètre, la sélection d'un seul fragment de formule permet de préciser ce paramètre. Par exemple, la simplification ou l'évaluation numérique peuvent s'appliquer à une sélection. Pour illustrer ce principe, il suffit d'imaginer une interface proposant des boutons (ou des raccourcis clavier) qui remplacent la dernière zone sélectionnée par le résultat de son évaluation ou de sa simplification. Ce genre de manipulation est rapidement assimilée, même par des utilisateurs novices.

Sélection multiple

Des opérations plus complexes sont envisageables grâce aux sélections multiples. Par exemple, le développement d'un polynôme en plusieurs variables nécessite de connaître le polynôme mais aussi la variable par rapport à laquelle le développement doit s'effectuer. Dans ce cas, la sélection multiple permet de préciser les différents paramètres. Avec ce genre de mécanisme, il est facile de réaliser un grand nombre d'opérations.

Sélectionner et coller

Certaines opérations à plusieurs paramètres s'expriment plus naturellement en utilisant à la fois la sélection et le collage. Par exemple, la substitution d'une variable par une expression peut se réaliser en sélectionnant l'expression, puis en la collant sur la variable cible.

Sélectionner et déplacer

Une autre opération intéressante à implémenter est le déplacement sémantique tel que réalisé par [Avit90], [PI88]. Par exemple, si on déplace la variable b dans l'équation $a + b = 0$ du membre de gauche vers le membre de droite, la formule devient $a = - b$. Le passage de la variable `b` de part et d'autre de l'égalité provoque la modification de l'équation. Cette transformation conserve le sens mathématique de l'équation. Toutes les opérations décrites ici peuvent être effectuées par les mécanismes habituels des systèmes de calcul formel, mais une bonne interface doit permettre de les réaliser plus rapidement. Les techniques basées sur la sélection offrentS l'expressivité suffisante pour réaliser un grand nombre de manipulations complexes.

3.8 Les opérations et leurs paramètres

Les opérations

Il y a deux sortes d'opérations: celles pour lesquelles une notation mathématique existe (une notation "naturelle") et celles pour lesquelles aucunes notations n'existent. Parmi les premières, on trouve la différentiation ou l'intégration. Parmi les secondes, on trouve la factorisation, les substitutions, la résolution...

Effectuer n'importe quelle opération qui a une notation naturelle peut donc s'exprimer d'une manière uniforme. Il suffit "d'écrire" la notation et d'effectuer/évaluer par un bouton ou une séquence de touches. Par exemple, pour calculer une primitive, on peut saisir l'intégrale correspondante et appuyer sur un bouton "évaluer" ou "effectuer".

Pour les opérations sans notations, deux possibilités s'offrent. La première est d'utiliser une notation fonctionnelle comme dans les interfaces classiques et d'écrire quelque chose comme:

$factor\ (x^3 - 3\ x^2 + x - 3)$

avec une action explicite pour déclencher le calcul comme dans le cas précédent. L'inconvénient en est que l'on perd l'élégance de la notation et que l'opération est placée devant l'expression.

Une autre possibilité pour pallier ces inconvénients est d'utiliser l'opération comme un "modifieur" séparé de la partie mathématique. Le déclenchement du calcul peut alors être implicite.

Une troisième solution est de ne pas écrire l'opération mais d'utiliser un bouton (éventuellement dans un menu), une séquence de touche ou une manipulation à la souris. Ainsi, dans l'exemple précédent, on pourrait rendre le polynôme "actif" puis presser un bouton "factor" pour obtenir sa factorisation.

Les paramètres

En général, une opération s'applique sur une ou plusieurs expressions, avec un ou plusieurs paramètres.
Dans le cas le plus fréquent, on a une expression "principale" à laquelle l'opération va s'appliquer et un ou plusieurs paramètres. Par exemple, dans le cas d'une substitution simple, l'expression dans laquelle on substitue est l'argument principal, la variable à l'intérieur de l'expression un paramètre, et ce par quoi on la substitue un autre paramètre. Dans ce type de cas, la spécification des paramètres peut se faire par sélection. La possibilité de sélections multiples est donc primordiale.

Retour sur nos catégories

Voyons maintenant comment on pourrait exprimer les opérations que nous avons catalogué.

Ici, "appuyer sur un bouton" peut signifier effectivement appuyer sur un bouton dans l'interface, ou bien sélectionner un élément dans un menu (qui peut apparaître dynamiquement sur l'expression, pour éviter de porter la souris vers une barre de menus), ou encore entrer une certaine séquence de touches.

En général, les opérations effectuées sur une expression s'appliquent à "l'expression courante". Le positionnement de l'expression courante peut se faire implicitement (dernière expression rentrée) ou explicitement ("clic" souris).

- ❏ Substitutions: Une substitution s'applique sur une expression avec deux paramètres, une variable (ou une sous-expression) et une expression par quoi la remplacer. Une fois l'expression courante choisie, il est possible de sélectionner une variable ou une sous-expression à l'intérieur, de sélectionner ailleurs une expression et de "porter" cette sélection sur celle de l'expression courante. On peut tout aussi bien appuyer sur un bouton au lieu de "porter".

- ❏ Distributions: Pour distribuer un produit, il faut d'abord sélectionner le produit correspondant (la sous-expression), avant d'appuyer sur un bouton.

- ❏ Mises sous formes polynômiales: Ici aussi une expression courante est nécessaire. On sélectionne ensuite une variable (ou une expression) comme "variable principale" dans cette expression puis on active (menu, tableau de bord) l'opération de mise sous forme polynômiale.

- Mises en ordres: Pour une “mise en ordre” d’une expression entière, le choix d’une expression courante est utile. La multi-sélection peut là encore servir à donner les variables dans l’ordre.

- Compatibilités: La compatibilité peut s’exprimer assez naturellement par un “glissé” d’une sous-expression d’un coté vers l’autre d’une égalité (ou inégalité).

- Résolutions simples: Le “glissé” peut aussi servir dans ce cas: sélectionner une variable (ou une sous-expression) et porter la sélection de l’autre coté de l’égalité (sur le 0). A noter que cet usage du “glissé” n’entre pas en conflit avec l’usage précédent (dont on peut considérer que c’est un cas particulier pour les égalités à zéro).

- Applications de règles: Ici plusieurs solutions sont possibles pour présenter les règles: comme des expressions ou dans des palettes. A noter qu’il est raisonnable de pouvoir transformer une règle dans une palette en une expression dans l’interface. Cela semble être une première solution au problème de l’organisation des règles. La partie de l’expression à laquelle on veut appliquer la règle doit être sélectionnée (sinon c’est l’expression courante). Ensuite on peut appuyer sur un bouton.

- Tracés de courbes et de surfaces: C’est l’un des rares cas où il semble nécessaire de faire apparaître une boite de dialogue/formulaire étant donné le nombre de paramètres.
  A noter que le tracé lui-même devrait être un objet “actif” où l’on devrait pouvoir spécifier des choses comme un angle de vue différent ou un zoom (sélection d’une région). On peut mentionner ici les possibilités du “*graphing calculator*” de *R. Avitzur* (notamment la représentation des inégalités et des paramètres).

- Opérations complexes: La façon d’effectuer une telle manipulation dépend de l’opération proprement dite. Comme nous l’avons déjà mentionné, pour toutes les opérations possédant une notation “mathématique” (intégration, dérivation, sommation, équations “sans paramètres”), un simple bouton “effectuer” suffit.

3.9 Organisation des calculs

Un aspect fondamental des interfaces qui nous intéressent est la façon dont les calculs y sont organisés.

L’aspect exploratoire s’accommode assez mal de l’organisation linéaire classique. Souvent des résultats intermédiaires de grande taille remplissent l’unique fenêtre d'interaction avec le système, et les résultats précédents sont très difficiles à retrouver. Une organisation en arbre avec des "tableaux" qu'on peut ouvrir à volonté semble donc indispensable. La possibilité de travailler simultanément sur plusieurs tableaux peut constituer un environnement plus proche du mode de travail traditionnel offert par les feuilles de papier et le crayon. En outre, il est fréquent d'avoir besoin d'effectuer des calculs temporaires dans un environnement vierge ne prenant pas en compte les liaisons courantes des variables. Il est donc souhaitable de disposer d'un mécanisme permettant de lier un groupe de tableaux à une même session du système de calcul formel sous-jacent. Les effets de bord (affectation de variables) effectués dans un tableau ne doivent être visibles que dans les tableaux du groupe correspondant. Cette organisation en tableaux permet également de disposer d'un (ou plusieurs) tableau brouillon contenant des sources de formules toujours disponibles (visibles) pour d'éventuelles sélections.

A rapprocher de la présentation d'un calcul, la présentation des étapes du calcul semble être un point important. Pour être plus précis, on peut dire que l'interface d'un système de calcul formel affiche un certain nombre d'objets mathématiques (au sens large, formules, systèmes, ensemble de solutions, graphes de fonction...). La plupart des objets ont un lien naturel avec un ou plusieurs autres objets. Un tel lien peut être un lien "question/réponse" du type "est le résultat de l'évaluation de" ou "est le résultat de la factorisation de" ou encore "est le résultat de la simplification d'une sous-expression de". Ces liens peuvent également exprimer des relations plus mathématiques entre les objets comme: (cette expression est valide) "sous la condition" (donnée par cette relation). L'interface doit pouvoir faire apparaître ces liens.

La possibilité de mélanger du texte et des formules mathématiques au sein de l'interface nous semble une fonctionnalité nécessaire, mise en œuvre par certains systèmes avec les "*notebooks*" et autres "*worksheets*". Associer des commentaires aux expressions mathématiques est sans conteste utile et naturel, et cette possibilité peut s'agrémenter d'opérations de type pliage-dépliage qui peuvent cacher temporairement un ou plusieurs commentaires pour donner à un tableau un aspect plus condensé. La présence de texte permet d'effectuer des recherches, surtout si un texte peut être associé à une formule particulière.

3.10 Présentation des résultats

La présentation des résultats est évidemment importante pour leur bonne exploitation. On peut y voir deux niveaux, le premier est purement typographique, le second concerne la présentation des résultats de grande taille, qui peuvent être assez fréquents dans les systèmes de calcul formel.

A mi-chemin entre ces deux niveaux, la césure n'est nécessaire que pour un rendu final "papier". L'élision n'est vraiment utile que si elle est couplée à un mécanisme sémantique (présentation des polynômes ou sa généralisation par ordre de grandeur). La clef est sans doute de fournir des outils d'utilisation simple pour l'exploration d'une formule (réorganisation des termes, élision explicite de sous-termes...). Plus généralement, on pourra voir l'article de *D. Stoutemyer* "Qualitative analysis of mathematical expressions" [Stou76].

Quelle que soit la méthode implémentée pour "résumer" une expression, il faut un moyen "dynamique" de la déclencher à certains niveaux. Il serait aussi intéressant d'avoir une taille maximum par défaut pour les expressions générées telles que toute expression plus grosse enclenche le résumé.

Elision

L'utilisation de l'élision est la méthode la plus simple pour condenser un gros résultat. Cette méthode purement syntaxique consiste à remplacer les sous-expressions au delà d'une certaine profondeur (ou les sous-termes d'un opérateur n-aire au delà d'une certaine position) par un seul symbole particulier, en général "...". Par exemple, une somme comme $x + y + \sin(x + mn)$ peut être rendue comme $x + y + \sin(...)$.

Si l'élision est une technique très simple à mettre en œuvre, son caractère purement syntaxique en limite l'intérêt.

La fonction `reveal` de Macsyma

Le système *Macsyma* dispose d'une fonction `reveal` qui essaye de montrer la forme d'une expression. `reveal(expr, level)` "révèle" la structure de `expr` jusqu'au niveau `level`. La structure est décrite à l'aide des opérateurs de certains opérateurs particuliers, comme `SUM` et `PRODUCT`. `SUM(n)` représente une somme de `n` termes, `PRODUCT(n)` un produit de `n` termes. Ainsi, en prenant l'exemple simple de la primitive d'une fraction rationnelle, on a:

```
(C4) integrate(1/(x^3 + 2), x);
                                                    1/3
                                             2 X - 2
                                       ATAN(------------)
                  2     1/3       2/3           1/3                        1/3
             LOG(X   - 2     X + 2    )        2     SQRT(3)     LOG(X + 2    )
(D4)       - ----------------------- + ------------------ + -------------
                         2/3                  2/3                      2/3
                      6 2                  2     SQRT(3)            3 2
```

et `reveal` avec de petites valeurs pour son niveau, donne les informations suivantes:

```
(C5) reveal(d4, 1);
(D5)                                Sum(3)
(C6) reveal(d4, 2);
(D6)                     Negterm + Quotient + Quotient
(C7) reveal(d4, 3);
                                     ATAN         LOG
(D7)                 - Quotient + ---------- + ----------
                                  Product(2)   Product(2)
(C8) reveal(d4, 4);
                        LOG       ATAN(Quotient)   LOG(Sum(2))
(D8)                - ---------- + -------------- + -----------
                      Product(2)     Expt  SQRT       3 Expt
```

On peut évidemment pousser plus loin (et remarquer que certains résultats sont en fait plus gros que l'expression initiale...)

```
(C9) reveal(d4, 5);
                                          Sum(2)
                                   ATAN(----------)
                 LOG(Sum(3))           Product(2)     LOG(X + Expt)
(D9)           - ----------- + ---------------- + -------------
                   6 Expt         2/3                      2/3
                                 2     SQRT(3)          3  2
(C10) reveal(d4, 6);

                                             Product(2) + Negterm
                                        ATAN(---------------------)
        LOG(Expt + Negterm + Expt)                 Expt  SQRT
(D10) - -------------------------- + --------------------------
                      2/3                       2/3
                   6 2                         2    SQRT(3)
              1/3
   LOG(X + 2    )
 + -------------
           2/3
        3  2

(C11) reveal(d4, 7);
                                              2 X - Expt
                                         ATAN(------------)
           2                  2/3               1/3                      1/3
      LOG(X   - Product(2) + 2    )           2    SQRT(3)     LOG(X + 2    )
(D11) - -------------------------- + ------------------ + -------------
                      2/3                    2/3                      2/3
                   6 2                      2    SQRT(3)           3  2
(C12) reveal(d4, 8);
                                                   1/3
                                              2 X - 2
                                        ATAN(------------)
             2             2/3              1/3                      1/3
        LOG(X   - Expt X + 2    )          2    SQRT(3)     LOG(X + 2    )
(D12)   - ----------------------- + ------------------ + -------------
                      2/3                  2/3                     2/3
                   6 2                    2    SQRT(3)          3  2
```

et le niveau suivant (9) donne l'expression initiale.

En fait, cet exemple n'est pas très bon pour montrer l'intérêt de `reveal` mais c'est un bon exemple pour comprendre son fonctionnement. *D. Stoutmeyer* [Stou76] utilise une fonction `revelation` qui permet de borner la taille du résultat de `reveal`.

On peut s'interroger sur les différences réelles entre élision et `reveal`. En comparant les résultats de `reveal` aux niveaux 2 et 3 avec les résultats d'une élision, on voit que les différences sont finalement minimes bien que `reveal` semble donner une information un peu meilleure en général.

Il est bien entendu toujours instructif de comparer le résultat d'un système aux résultats donnés par d'autres. D'abord par *Maple*:

```
> int(1/(x^3 + 2), x);
                 1/3         1/3            1/3      2        1/3     2/3
            1/6 2     ln(x + 2    ) - 1/12 2     ln(x   - x 2     + 2    )
                        1/3  1/2                 1/2    2/3
                 + 1/6 2    3     arctan(1/3 3     (2     x - 1))
```

et ensuite par *Mathematica*:

```
In[1]:= Integrate[1/(x^3 + 1), x]
               -1 + 2 x
        ArcTan[--------]                                 2
               Sqrt[3]      Log[1 + x]   Log[1 - x + x ]
Out[1]= ---------------- + ---------- - ---------------
            Sqrt[3]             3                6
```

Comme on le voit `reveal` ne révèle qu'une partie de la vérité...

Polynomialisation

Sous ce néologisme, nous introduisons une autre technique pouvant être vue comme parente de `reveal`, au sens où elle privilégie les sommes et les produits. Cela consiste à montrer la structure polynômiale d'une expression. Il s'agit d'une structure polynômiale au sens large, où les variables peuvent être des expressions “non polynômiales”.

Si nous reprenons l'exemple de la primitive de la fraction rationnelle précédente, telle qu'exprimée par *Macsyma*, on peut dire que le résultat est un polynôme en arc tangente et logarithme, dont les arguments sont des polynômes en `x`. On peut ajouter l'information supplémentaire du degré et du domaine des coefficients (extension du corps des rationnels, $Q[\sqrt{3}, 2^{1/3}]$). On pourrait, par exemple, dire qu'il s'agit d'un polynôme en ln $(x^2 + ...)$, $atan(x + ...)$, $\log(x + ...)$ à coefficients dans $Q[\sqrt{3}, 2^{1/3}]$.

Il y a donc deux problèmes: choisir parmi les différentes possibilités de polynomialisation et trouver comment les exprimer d'une manière suffisamment concise.

3.11Conclusion

Dans cette partie, nous avons montré, à l'aide de plusieurs exemples, que la manipulation d'expressions n'est pas aisée dans les systèmes de calcul formel. Capables d'exécuter des algorithmes très complexes, ces systèmes n'offrent pourtant que peu d'opérations élémentaires pour organiser les calculs et gérer la résolution d'un problème. Après une tentative de classification de ces opérations, nous avons proposé divers mécanismes pour les rendre facilement accessibles via l'interface. Dans le modèle que nous proposons, des opérations alors fastidieuses ou à la syntaxe obscure peuvent être effectuées très simplement et intuitivement en combinant sélections et activations de boutons dans l'interface. Nous avons présenté également des fonctionnalités nouvelles qui devraient permettre à l'utilisateur de mieux organiser l'ensemble des calculs, qui mènent à la solution d'un problème. Nous verrons dans les chapitres suivants comment ces fonctionnalités peuvent être effectivement implémentées dans une interface.

4 Éléments pour la réalisation d'une interface

4.1 La réutilisation de l'interface

Les interfaces des systèmes de calcul symbolique disponibles ne sont bien entendu pas utilisables en tant qu'interfaces seules. L'interface est complètement liée au système. Les prototypes d'interfaces indépendantes que nous avons expérimentées ne sont que difficilement réutilisables. Ce manque de réutilisabilité nuit pour plusieurs raisons à la qualité des interfaces:

- les efforts fournis pour développer une interface ne peuvent pas être factorisés. Pour chaque nouveau SCF, il faut recréer une interface complète. Comme cette tâche est loin d'être aisée, il est compréhensible que les développeurs n'aient pas toujours le temps de la mener à bien.

- la multiplicité des interfaces s'accompagne d'un manque d'homogénéité dans la façon de les utiliser. Dans certains cas, l'utilisateur doit employer plus d'un système pour résoudre un problème, car du fait de leurs spécificités, chacun d'eux n'apporte qu'une partie de la solution. L'utilisateur est donc confronté à plusieurs interfaces, ce qui augmente le temps d'apprentissage. Une approche plus unifiée aurait donc sûrement des retombées bénéfiques.

Malgré de nets progrès, la solution d'une interface unique n'est peut-être pas encore envisageable [DGS94]. Par contre, l'emploi d'un même composant de base permettrait d'avoir un comportement similaire pour les différentes interfaces. Pour être utilisable dans ce contexte, le composant d’édition doit pouvoir s’adapter à différentes notations.

Adaptation aux notations.

Les besoins des différents systèmes en terme d'édition sont variés. Certains systèmes sont destinés à réaliser de petits calculs alors que d'autres peuvent produire des résultats d'une longueur impressionnante. Les manipulations de formules qui s'en suivent ne sont bien sûr pas du même type. Le style des symboles à éditer est aussi très variable. Ces différents points nécessitent une forte paramétrabilité du composant d'édition qui devra, non seulement permettre l’affichage d’un grand nombre de notations, mais aussi leur édition. De plus, il n'est pas réaliste d'espérer prévoir tous les types de notations utiles. Le composant devra donc être modifiable pour en intégrer de nouvelles.

Intégration dans différents environnements

Pour être pleinement efficace, le principe de partage du composant d'édition doit être appliqué le plus souvent possible. Ceci implique que le composant devra s'intégrer dans des interfaces de conceptions très diverses. Par exemple, il ne devra pas imposer une bibliothèque graphique pour la réalisation des interfaces.

Certains SCF sont accessibles sous plusieurs systèmes d’exploitation[1]. *Maple*, par exemple, est porté sur une trentaine de plates-formes et existe par exemple en version *Unix, Windows (NT, 3.11, 95)*, *MacOS* ou même *AmigaOS*. Pour chacun de ces OS, un environnement graphique différent doit être utilisé. Idéalement, le composant d'édition devrait donc être adapté à tous les environnements graphiques modernes. C'est un problème technique très délicat. L'écriture d'une interface est, pour le moment encore, difficile à abstraire du système de fenêtrage qu'elle emploie. Toujours dans le cas de *Maple*, les interfaces pour les différents environnements graphiques sont très différentes. Une version nouvelle a dû être développée pour chaque OS.

Le composant d’édition doit donc pouvoir s’intégrer facilement dans de nombreux environnements de développement et être rapidement adaptable aux spécificités de

[1] Dans la suite du document l’expression système d’exploitation sera souvent remplacée par OS (pour Operating System)

nombreuses interfaces. Ces deux aspects ne doivent toutefois pas pénaliser la qualité de l'affichage proposé.

4.2 Un affichage soigné

Intérêt de l'affichage bidimensionnel

Une des caractéristiques souhaitables du composant d'édition est de permettre une saisie la plus proche possible de celle à laquelle sont habitués les utilisateurs: le papier et le crayon (ou suivant affinité le tableau et la craie). Dans cette optique, les notations du style de celles employées par *Maple* ne sont pas satisfaisantes. En linéarisant artificiellement la saisie, elles ne permettent plus la présentation habituelle des formules. C'est pourtant bien cette présentation bidimensionnelle qui est la plus rapide à assimiler puisqu'elle est employée sur les supports autres que l'écran. L'exemple de la figure 3 est tiré de l'interface V.4 de *Maple* et illustre bien l'intérêt des notations bidimensionnelles, qui commence maintenant à être adopté par tous les systèmes.

Figure 3 : Notations mono et bi-dimensionnelles.

Il faut aussi remarquer que les notations qui tirent partie des deux dimensions du support peuvent permettre une meilleure lisibilité. En regroupant les éléments à mettre en rapport sur une zone plus compacte, ces notations facilitent la lecture.

Problèmes typographiques

Un affichage agréable nécessite une typographie soignée. Les écrans modernes à technologie bitmap[1] permettent un affichage bien plus respectueux des conventions

[1] L'image est produite par une matrice de points colorés (plus connus sous le nom de pixels).

typographiques que ce qui pouvait être obtenu sur les terminaux alphanumériques. Il est maintenant possible de tracer un symbole de forme quelconque à toute position de l'écran (dans les limites de la définition de l'écran). L'évolution des moyens d'affichage et des goûts des utilisateurs impose une qualité typographique bien meilleure que par le passé. Ce n'est pas sans poser de nouveaux problèmes.

- La définition[1] des écrans n'est pas encore à la hauteur de celle des moyens d'impression. Le nombre de points par centimètre, qu'un bon écran associé à une carte graphique haut de gamme peut afficher, est de l'ordre de 35 ppcm[2], tandis qu'une imprimante laser d'entrée de gamme propose déjà du 120 ppcm. Pourtant, les utilisateurs attendent d'un affichage sur écran une finesse comparable à celle qu'ils apprécient dans les documents imprimés.

- Les notations mathématiques classiques utilisent un très grand nombre de symboles différents. Dans l'ouvrage [AS72], et sans tenir compte des caractères alphanumériques usuels, nous en avons dénombré de l'ordre d'une centaine[3]. Certains de ces symboles sont, de plus, de tailles variables. Pour respecter la typographie conventionnelle, il faut adapter leur taille au contexte et soigner les alignements.

Une interface de bonne qualité pour l'édition de formules mathématiques doit résoudre ces problèmes de typographie.

4.3 La gestion de grandes formules

Un système de calcul formel peut produire des formules de très grande taille même en réponse à de petites requêtes. L'évaluation de l'expression de la figure 4 provoque une réponse de cinq pages qui est difficilement interprétable en l'état. Les solutions envisageables sont:

- tenter d'extraire des éléments qui se répètent pour réduire la taille de la formule. Ce n'est possible que si la formule possède effectivement des éléments identiques. C'est une solution employée par les SCF qui codent les formules sous forme de

[1] le nombre de points distincts affichables simultanément à l'écran. Pour les stations de travail la définition des écrans est de l'ordre de mille par mille points.

[2] ppcm: points par centimètre. L'unité la plus fréquemment employée est le dpi (dot per inch) ou ppp (point par pouce). Un dpi vaut 2.54 ppcm.

[3] Ils sont répartis en grandes familles comme: les symboles de grande taille (intégrale, somme...), les symboles diacritiques (prime, tilde...), les symboles d'opérateurs (dé rond, est inclus dans, barre de fraction...), les caractères grecs, les délimiteurs en tous genres (crochets, norme, accolades...).

Figure 4 : Une petite requête peut générer une longue réponse.

DAG[1] comme *Maple* car l'identification des éléments communs est alors immédiate. Le schéma suivant figure 5, page 53 représente le DAG associé à l'expression x+1-x/(x+1). Le sous arbre x+1 qui apparaîtrait deux fois dans la structure arborescente classique est partagé.

- appliquer une césure la moins gênante possible pour disposer la formule verticalement sur une suite de pages. Ce n'est pas toujours possible dans de bonnes conditions. Par exemple, dans le cas d'une formule ayant une structure verticale, comme une fraction, une césure est difficilement réalisable et la lisibilité en souffre. Notez la barre de division de la figure 6 qui est peu aisément repérable dans la masse

[1] Direct Acyclic Graph. La formule n'est plus simplement représentée par son arbre syntaxique. Les sous arbres identiques de la représentation classique sont regroupés. De ce fait, les racines de ces sous arbres ont plusieurs pères. L'arbre est devenu un graphe orienté.

Figure 5 : Exemple d'équivalence entre arbre classique et DAG pour une formule.

des termes de la fraction. Cette technique a été par exemple décrite par *Norbert Kajler* dans son éditeur *CasPi* [Kajl92],[Kajl93].

Figure 6 : Un exemple où la césure n'est pas une bonne solution.

- permettre (par exemple grâce à des barres de défilement ou scrolling) de se déplacer rapidement dans la formule qui est alors formatée sans césure mais qui occupe plus d'une page en largeur. La difficulté est alors d'obtenir un affichage suffisamment rapide pour rester compatible avec les déplacements de la fenêtre.

- ne pas afficher les termes de la formule s'ils sont à un niveau trop bas dans l'arbre syntaxique. Cette approche n'est possible que si la formule a une structure profonde et suffisamment équilibrée. Par exemple, un grand polynôme ne se prête pas à ce genre de traitement.

- offrir une vue à une échelle différente de la formule qui permette de se faire une idée d'ensemble de la formule. La création de cette vue réduite risque d'être longue et il faut prendre soin de mettre en évidence les caractéristiques de la formule. Un simple changement d'échelle ne suffit pas. Il faut par exemple repérer une matrice triangulaire pour la remplacer par un symbole distinctif ou formater une clause logique pour mettre en évidence les imbrications de conjonctions et de disjonctions.

- ne montrer que certains termes de la formule. Soit dans le cas d'un très grand polynôme, n'afficher qu'un monôme sur cinq, que les monômes de plus haut coefficients, ou bien uniquement les monômes en une variable. Cette approche nécessite une bonne connaissance de la structure mathématique de la formule pour prendre des décisions adaptées aux besoins de l'utilisateur.

Ces différentes approches de la gestion des grandes formules devront être testées pour déterminer les plus efficaces. Parmi les six solutions évoquées, seules les quatre premières sont utilisées dans les interfaces actuelles et il reste beaucoup à faire pour les améliorer [Lamp93], [KP81]

4.4 Une édition plus agréable

La qualité de l'édition des formules est certainement un point clef de l'ergonomie d'une interface pour le calcul formel. C'est pourtant un des aspects pour lequel il nous semble que le plus de progrès sont réalisables.

Le droit à l'erreur

Il est bien rare d'effectuer une saisie complète sans erreurs. Pourtant, dans les éditeurs actuels, il n'est pas fourni de mécanisme efficace pour corriger une erreur. En fait, il n'est en général proposé que l'annulation de la dernière frappe. C'est insuffisant car certaines erreurs ne sont détectées qu'après un certain temps. Il est alors trop tard pour revenir en arrière. L'éditeur Emacs[1] propose une technique intéressante. Il conserve la liste des

[1] Emacs est un éditeur de texte très prisé des utilisateurs d'*UNIX* [Stal81]. Les premières versions ont été développées vers la fin des années 1970 par *Richard M. Stallman*.

opérations effectuées au cours de l'édition et permet donc de retourner à tous les stades antérieurs (du moins depuis la dernière sauvegarde).

Ce point peut paraître à première vue de moindre importance. En fait, il est probablement un des facteurs les plus importants lors de l’apprentissage d’une interface. L’utilisateur qui découvre un nouveau système est tenté de procéder à son exploration par tentatives successives. Comme pour bien d’autres disciplines, l’apprentissage le plus aisé se base sur l’exploration par essais et erreurs successives. La difficulté de corriger une tentative infructueuse est donc très pénalisant et ralentit, voire inhibe totalement, l’assimilation des fonctionnalités non triviales d’une interface.

Rester réactif

Une interface doit rester réactive. L’utilisateur doit toujours avoir la confirmation que ses actions sont prises en compte [HH89]. La durée maximum entre le déclenchement d'un événement par l’utilisateur et la confirmation visuelle de sa prise en compte doit rester inférieure au dixième de seconde. Le composant d'édition idéal garantit ce grain de réactivité, ce qui provoque certaines difficultés:

❑ Les algorithmes à employer pour assurer un affichage bidimensionnel de bonne qualité sont complexes. C’est le cas, par exemple, des calculs à effectuer pour déterminer la position des caractères sur la page. Maintenir à jour un affichage soigné en respectant la vitesse de frappe d'un utilisateur entraîné peut donc s’avérer difficile.

❑ Le composant est destiné à être employé pour réaliser une interface à des systèmes de calcul formel. Les calculs effectués par de tels systèmes sont souvent longs. Il faut donc prévoir un moyen de conserver la réactivité de l'édition durant ces périodes. Ce n'est pas le cas dans les interfaces classiques (voir le tableau 1). Pourtant, l’utilisateur peut vouloir mener plusieurs raisonnements en parallèle. Par exemple, après avoir demandé la résolution d'un problème, il peut profiter du temps de calcul pour préparer l'étape suivante de son raisonnement et même y intégrer le résultat en cours de calcul.

Pour résumer, une bonne interface propose un affichage soigné tout en maintenant une bonne réactivité.

4.5 Déplacements rapides

Une des actions les plus souvent effectuées dans une interface d'édition est le déplacement du curseur. Pour ne pas pénaliser l'utilisateur, ces mouvements du curseur doivent être rapides. Voici les principales circonstances dans lesquelles l'utilisateur souhaite se déplacer dans la zone d'édition de l'interface.

- Lors de la saisie d'une formule, il est fréquent de revenir en arrière pour corriger une faute de frappe ou rajouter un terme oublié. Ces déplacements sont, le plus souvent, de faible amplitude. Les interfaces proposent deux solutions.
  La première est d'utiliser un dispositif de pointage[1] pour désigner la nouvelle position, ce qui est très intuitif mais assez long. Le modèle *Keystroke*, décrit dans [CMN83] et [CMN87], indique les durées de réalisation des différentes actions proposées par les interfaces homme-machine actuelles. Il précise notamment que le temps nécessaire pour effectuer l'opération de pointage avec la souris est de l'ordre de 1.5s, dont 0,4s pour déplacer une main du clavier à la souris.
  La seconde solution est de déplacer le curseur d'élément en élément pour le rapprocher du point visé, en employant, par exemple, les flèches disponibles sur les claviers modernes. Pour atteindre un point proche, cette méthode peut être plus rapide que la précédente, car il n'est pas nécessaire de lever les mains du clavier. La difficulté est de rendre le déplacement du curseur le plus naturel possible. Cette question sera détaillée plus loin dans cette section.

- Une autre manipulation fréquemment réalisée est la sélection de fragments de formules. L'intérêt de ces sélections est détaillé au paragraphe 3.7, page 40. Pour réaliser une sélection, il faut désigner deux points dans la formule. Là encore les deux méthodes de pointage et de déplacement pas à pas sont possibles.

- Quand un grand nombre de formules sont présentes dans l'interface, elles ne peuvent pas toutes apparaître dans la zone d'édition. Il faut donc pouvoir se déplacer pour faire réapparaître une formule trop éloignée. Pour ce type de déplacement de grande amplitude, la méthode la plus souvent employée est le saut page à page ou les barres de défilement (scrolling).

Pour permettre un déplacement efficace dans tous ces cas, il faut impérativement disposer d'un affichage rapide. De ce point de vue, la réalisation du saut page à page est critique car il faut s'assurer que le temps nécessaire à l'affichage d'une page complète reste faible et, si possible, inférieur au 1/10ème de seconde [Card87]. De plus, dans ce cas, les

[1] tels que: souris, trackball, glidepoint, etc.

algorithmes basés sur des copies de zones d'écran pour accélérer le défilement ne sont plus applicables.

Les problèmes liés au déplacement pas à pas sont plus particuliers. La vitesse d'affichage n'est plus un réel problème car des algorithmes dits incrémentaux s'appliquent bien dans ce contexte. Ils sont décrits au paragraphe 4, page 110. Par contre, pour ne pas perdre de temps, l’utilisateur doit pouvoir imaginer rapidement le chemin le plus court (mesuré en nombre de touches à frapper) entre le point courant et le point visé. Les interfaces actuelles pour l'édition de formules ne proposent que des déplacements pas à pas basés sur la structure arborescente de la formule. Par exemple, demander un déplacement vers le haut provoque un déplacement du curseur vers le père du noeud courant. Cette solution n'est pas satisfaisante. En effet, cette représentation mentale de la formule sous forme d'arbre n'est pas naturelle. Il n'est pas facile d'imaginer cette structure en voyant une formule à l'écran. Pourtant, pour pouvoir prévoir les mouvements du curseur, il faut établir la correspondance entre les positions des symboles à l’écran et leur place dans l'arbre. Nous avons pu vérifier auprès des utilisateurs que les déplacements basés sur cette méthode étaient désagréables car trop difficiles à prédire. La solution adoptée par l’utilisateur est donc en général de l'abandonner au profit d'un dispositif de pointage pourtant plus lent dans la majorité des cas.

Pour résoudre ce problème, il suffit de rendre les mouvements du curseur plus prévisibles. La phase très coûteuse d’élaboration mentale de l’action à effectuer est alors nettement plus courte. Le modèle *Keystroke* décrit bien l’importance, souvent négligée par les concepteurs d’interface, de cette phase de visualisation mentale de l’action à effectuer. Il est assez difficile d’évaluer sa durée, mais *S. K. Card* [CMN87] propose une durée moyenne de 1.35s. Cette évaluation doit bien sûr être révisée à la hausse dans le cas du déplacement syntaxique qui nécessite un réel effort mental pour imaginer la trajectoire du curseur en réponse aux actions du clavier. Pour éviter ce désagrément, une solution envisageable est de les baser, non plus sur la structure interne de la formule, mais sur son aspect externe qui est directement visible pour l'utilisateur. Ce type de déplacement du curseur est dit spatial car il correspond directement à la disposition des éléments de formule dans l'espace de la zone d'édition. Ainsi, un déplacement vers le haut tendra à faire remonter le curseur vers le haut de l'écran. Les déplacements spatiaux sont utilisés couramment dans les éditeurs de textes. La raison pour laquelle ils n'ont pas été adoptés pour l'édition de formules nous échappe. Il faut tout de même noter qu'il est plus facile d'implémenter ce type de déplacements dans le cadre d'un éditeur où les éléments sont bien alignés (du texte par exemple) que dans celui où ils sont plus dispersés (c'est le cas pour les formules mathématiques présentées sous forme bidimensionnelle). Les algorithmes développés pour obtenir des déplacements spatiaux intuitifs sont décrits au paragraphe 4, page 110.

Pour mieux illustrer la différence entre les déplacements syntaxiques et spatiaux, les trois figures suivantes proposent un exemple concret de déplacement au sein d'une formule.

Sur la figure 7, les deux structures sont bien visibles: à gauche, la structure bidimensionnelle telle que perçue par l'utilisateur devant son écran et à droite, la structure syntaxique sous-jacente.

Figure 7 : Arbre syntaxique et représentation graphique d'une formule

La figure 8 montre la succession de touches à frapper pour faire passer le curseur du x au 1 en utilisant la structure de l'arbre. Il faut remarquer le nombre élevé de manoeuvres à effectuer (cinq touches) et le manque de naturel de certains sauts du curseur (durant la première étape, le curseur passe du x à la barre de fraction).

Figure 8 : Déplacement syntaxique.

La figure 9 illustre le déplacement spatial. Cette fois, le nombre de déplacements est plus faible (2 touches). Il est plus facile de prévoir l'effet d'un déplacement et donc de planifier la suite de touches à frapper.

Ce mode de déplacement est plus agréable à employer et il est beaucoup plus efficace pour tous les déplacements de faible amplitude. Quelques tests précis sont donnés au paragraphe 3, page 133.

Figure 9 : Déplacement spatial.

4.6 Évolutivité

Convenir à tous les utilisateurs

Le nombre croissant d'utilisateurs des systèmes de calcul formel s'accompagne d'une diversification de leurs besoins. Pour s'en convaincre, il suffit de comparer deux cas: un lycéen qui souhaite vérifier rapidement le résultat d'un exercice et un chercheur qui utilise le système à longueur de journée. Dans le cas de l'étudiant, la saisie doit être la plus naturelle possible pour lui épargner les erreurs de manipulation. Pour le chercheur par contre, il faut privilégier la puissance d'expression sachant qu'il a le temps de se familiariser avec des manipulations plus complexes.

De façon plus générale, il doit être possible d'adapter les comportements standard de l'interface ou même d'en créer de totalement nouveaux. De ce point de vue, l'exemple d'*Emacs* est frappant. C'est sa très grande paramétrabilité qui est, pour une bonne part, à l'origine de son succès. Le nombre de modules développés par des utilisateurs pour améliorer le comportement de l'éditeur est supérieur au millier (en ne comptant que les modules qui ont été officialisés). Nul doute qu'un tel effort de développement n'aurait pas été pris en charge par les concepteurs d'Emacs. Il faut toutefois reconnaître que cette très grande diversité de modules additifs peut entraîner des problèmes de génie logiciel. L'adaptation des modules aux modifications du noyau est parfois difficile et la compatibilité des modules entre eux n'est pas toujours acquise.

Dans le domaine du calcul formel, il n'existe à notre connaissance aucune interface possédant ce degré de paramétrabilité. Celà nous semble pourtant primordial pour permettre un enrichissement et une adaptation de l'interface aux utilisateurs.

Expérimenter facilement

Certains problèmes d'édition n'ont pas encore trouvé de solution satisfaisante. Un composant développé dans le cadre d'un travail de recherche doit proposer de nouvelles approches. Il faut pouvoir tester des idées en nombre suffisant pour espérer en découvrir d'intéressantes. Le cycle d'expérimentation doit donc être court ce qui n'est pas toujours aisé avec les environnements de développement actuels. Il faudra prévoir cet aspect dans l'architecture du composant.

5 Conclusion

Il n'est pas possible de prétendre réussir une interface idéale pour les systèmes de calcul formel, mais il semble que des progrès sensibles sont envisageables dans les domaines décrits dans ce chapitre. Il doit maintenant être possible de réaliser une interface (ou du moins un composant d'interface) plus facilement réutilisable et évolutif dans lequel l'édition serait plus naturelle, sans pour autant pénaliser la qualité de l'affichage. Tous ces problèmes seront repris dans les chapitres suivants en mettant cette fois l'accent sur les solutions qui ont été retenues.

Chapitre II
Présentation du serveur Emath

Ce chapitre présente de façon assez générale les idées mises en pratique durant la réalisation d'*Emath*. Une présentation du même type est disponible dans l'article [ADG97]. Sa lecture doit permettre une compréhension globale du problème et donc fournir un cadre pour les chapitres suivants. Les thèmes abordés ici sont approfondis dans les chapitres consacrés à l'architecture ou à l'implémentation.
Dans cette section, peu de justifications et aucunes conclusions ne sont données. Ces développements sont réservés aux chapitres suivants.

1 Architecture d'une application utilisant Emath

Les questions liées à la réalisation d'interfaces spécifiquement conçues pour les systèmes de calculs formels ont déjà été abordées [Kajl90], [Kajl90b], [Kajl93], [Soif91]. Une bonne interface utilisateur requiert un effort important de conception et d'implémentation. Il serait donc profitable de pouvoir bénéficier d'un composant facilement intégrable et paramétrable, qui fournirait les fonctions de base pour l'affichage et l'édition des formules mathématiques. Cette remarque n'est pas réservée aux interfaces pour les systèmes de calcul formel, elle s'applique aussi à d'autres logiciels, comme par exemple ceux d'édition

de documents incluant des formules.
Emath est conçu pour remplir ce rôle. Il propose bon nombre de fonctionnalités: un affichage bidimensionnel soigné et incrémental, une gestion de plusieurs fenêtres partageant ou non les mêmes données, des mécanismes variés de pointage et sélection, etc... Pourtant, il convient de souligner qu'*Emath* n'est pas une interface en soi. Ce composant est conçu pour être employé dans toute application nécessitant un affichage ou une édition de formules mathématiques.

Pour illustrer ce principe, voici son application au cas d'une interface utilisateur pour un système de calcul formel. La figure 10 présente son architecture.

Figure 10 : Relation entre application, Emath et serveur X.

Emath est un processus indépendant et se comporte comme un serveur. L'application utilise une bibliothèque de fonctions pour s'y connecter, puis communiquer. Elle délègue ensuite une ou plusieurs fenêtres à *Emath*. L'interface peut alors y affecter des formules, dont l'affichage et éventuellement l'édition sont pris en charge par le serveur. Ainsi, l'interface est déchargée des nombreuses tâches accomplies par *Emath*. En pratique, *Emath* effectue l'affichage des formules puis gère tous les évènements X relatifs aux fenêtres qui lui sont sous-traitées.

Pour pouvoir communiquer avec le serveur, l'application doit assurer la traduction entre sa représentation interne des formules et celle d'*Emath*. Dans le cas où les comportements par défaut d'*Emath* correspondent aux besoins de l'application, c'est le seul travail que l'interface doit effectuer. Actuellement, l'interface et le serveur échangent les informations grâce à un formalisme *Lisp*. En fait, un interprète *Lisp* complet est inclus dans *Emath* et assure, pour une bonne part, la paramétrabilité du composant.

2 Utilisation d'Emath

En principe, *Emath* interagit à la fois avec un utilisateur et avec une application. L'interaction avec l'utilisateur s'effectue par l'intermédiaire des évènements X classiques, (déplacements de la souris, touches enfoncées au clavier, etc...). Ces évènements sont générés par le serveur X et envoyés aux fenêtres sous le contrôle du serveur *Emath*. La communication entre l'interface et le serveur est réalisée entièrement par échange de s-expressions *Lisp*. Ce point est détaillé au paragraphe suivant.

2.1 Emath et l'application

communication

L'ensemble du protocole de communication entre le serveur et l'application est basé sur l'échange de s-expressions *Lisp*, assurant ainsi une très grande évolutivité. Cette approche, basée sur l'emploi d'un langage embarqué, facilite aussi l'échange de fragments de code entre application et serveur. Il est, par exemple, possible pour l'application d'envoyer au serveur un fragment de code *Lisp* pour modifier le comportement de base d'*Emath*.

formules, arbres et attributs

Les formules, représentées dans les fenêtres gérées par *Emath*, doivent être connues du serveur. Celui-ci dispose donc d'une représentation interne des formules qu'il affiche. Ces arbres de formules sont construits grâce aux fonctions de manipulation d'arbres qui ont été ajoutées au noyau de l'interprète *Lisp*. L'application qui souhaite afficher une formule grâce à *Emath* doit donc envoyer au serveur une s-expression décrivant la construction de la formule avant de l'associer à une fenêtre.

```
(setq test-formule (mkt '(/(+ a b) c)))
(display-formula test-formula current-window)
```

Dans cet exemple, le symbole test-formula est associé à un arbre représentant la formule (a+b)/c. Puis, cette formule est affichée dans une fenêtre préalablement prise en charge par le serveur.

Les attributs sont des symboles *Lisp* qui peuvent être associés à chaque noeud d'une formule et ce, pour chaque fenêtre. En fait, le mécanisme est plus riche. Une description précise en est donnée au paragraphe 5, page 86. Les attributs ont de nombreuses applications. En voici quelques exemples:

- Lors de l'échange de formules entre l'application et le serveur, les attributs peuvent être employés pour nommer des sous-formules et simplifier leur identification.
- Certains aspects graphiques sont précisés par attributs. Le style avec lequel une formule est affichée, dépend d'un ensemble d'attributs spécifiques.
- La modification d'un attribut déclenchant une méthode, il est possible de s'en servir pour réaliser un système du type 'notification' permettant d'informer qui le souhaite des modifications apportées à un attribut; par exemple, pour assurer la cohérence de plusieurs interfaces manipulant la même formule.

La manipulation de ces attributs par l'application est aussi simple que celle des formules. Un ensemble de fonctions de base permet de modifier ou de lire la valeur d'un attribut. Exemple:

(aget current-window current-selection)

L'évaluation de cette s-expression retourne une formule qui est la sélection courante dans la fenêtre courante.

gestion des évènements

L'ensemble des évènements *X* reçus par les fenêtres déléguées à *Emath* sont gérées par un mécanisme à base d'automates. Ces automates sont réalisés grâce à une extension des tables de translation *Xt*. Les motifs d'évènements sont associés à des actions. Des tables prédéfinies sont fournies pour réaliser les principaux comportements attendus par l'application. Mais il est facile de rajouter ses propres motifs ou actions, grâce à l'interprète embarqué dans *Emath*.

Si la gestion d'un évènement ne doit pas être réalisée dans le serveur, il est possible de le faire suivre à l'application. Par exemple, pour réaliser le déplacement sémantique décrit par [Avit90], les évènements du type '`drag`' sont traités par l'application qui utilise ses compétences mathématiques pour assurer la cohérence de l'équation. Dans l'égalité a-b=0, le déplacement du b vers la droite du signe '=', provoque l'envoi d'un évènement 'drag' à l'interface qui effectue un calcul pour transformer l'équation en a=b. Le comportement par défaut aurait été un simple déplacement syntaxique du b pour donner a=-b.

Saisir des formules dans un éditeur est plus compliqué qu'il n'y paraît. Plusieurs modèles ont déjà été étudiés et implémentés [Soif91] : le modèle classique (dit linéaire), celui basé sur des motifs et le modèle d'overlays. *Emath* n'implémente aucun de ces modèles en particulier, il donne plutôt la possibilité de les construire à partir d'une gestion plus uniformisée des évènements. Le système déjà évoqué plus haut, basé sur un automate à états finis, fournit les mécanismes indispensables au support, entre autres, de ces trois modèles.

2.2 Emath et l'utilisateur

L'utilisateur intéragit avec *Emath* par l'intermédiaire des interfaces classiques: clavier et souris. L'ensemble des évènements générés par ces interfaces sont redistribués par le serveur *X* aux fenêtres auxquelles ils sont destinées. L'automate, décrit précédemment, interprète ces évènements et détermine le comportement de l'interface en réaction aux actions de l'utilisateur.

saisie des formules

Du point de vue de l'utilisateur, l'aspect le plus apparent d'*Emath* est son comportement lors de la manipulation des formules. La saisie d'une formule se doit d'être la plus intuitive possible. Par exemple, la saisie de la formule (2a puissance 3b) s'effectue très simplement en tapant la suite de touches 2a^3b; alors que sous *Maple*, il faudrait taper: 2*a^(3*b). Cette simplification est obtenue grâce à l'automate, qui permet une gestion des évènements dépendant du contexte. Ce problème est étudié au chapitre paragraphe 4, page 79.

déplacements et sélections

L'utilisateur peut effectuer des déplacements en suivant la structure syntaxique de la formule, ou selon sa disposition spatiale. La représentation sous-jacente à la formule est un arbre, dans lequel les déplacements syntaxiques sont du type: monter, descendre d'un niveau, aller au fils suivant d'un opérateur, etc... La représentation, immédiatement perceptible par l'utilisateur, est l'image à l'écran de la formule. Les déplacements spatiaux s'appuient sur cette disposition en deux dimensions. Ils sont du type: aller une ligne vers le haut, un caractère vers la droite, etc...

L'utilisation de la structure syntaxique est bien adaptée aux sélections. Dans ce cas, elle assure que les blocs choisis sont syntaxiquement cohérents. Par contre, elle ne favorise pas un déplacement intuitif. En effet, l'utilisateur base ses déplacements sur l'aspect visible de la formule, plutôt que sur sa structure syntaxique sous-jacente. Contrairement aux interfaces d'édition de formules mathématiques classiques, *Emath* propose un jeu complet de déplacements spatiaux.

2.3 Intégration d'Emath dans une interface

Emath est un processus indépendant de l'interface, se comportant comme un serveur. Une fois le serveur opérationnel, l'application peut communiquer avec lui par échange de code *Lisp*. L'application doit déléguer une ou plusieurs fenêtres à *Emath* pour l'affichage et l'édition des formules. Cette délégation peut avoir lieu à tout moment.

En pratique, le programmeur d'une application, comportant une zone d'édition de formules, doit suivre les étapes suivantes pour utiliser *Emath*:

- Créer une hiérarchie classique de widgets comportant des menus, scrollbars, textes, boutons, et au moins un widget neutre (en *Motif* une "drawing area" est toute indiquée) dans lequel les formules seront affichées et manipulées. Puis, déléguer la fenêtre associée à ce widget au serveur *Emath*. *Emath* assure alors la gestion des évènements X associés à cette fenêtre (rafraîchissement, déplacements des curseurs, clic souris...).

- Si nécessaire, écrire du code *Lisp* qui adaptera *Emath* à ses besoins. Par exemple, en imaginant que les mécanismes d'édition proposés par défaut par *Emath* ne conviennent pas, il est possible de les modifier ou de les remplacer intégralement. Il existe une bibliothèque conséquente de fonctions facilitant ce type de développement.

- Assurer la traduction entre la représentation des formules dans l'application et celle utilisée dans *Emath*. Si *Emath* n'est utilisé que pour l'affichage de formules, ce travail se résume à transformer la formule telle que stockée dans l'application en une des formes compréhensibles par *Emath*. Dans le cas où l'application souhaite récupérer des formules en provenance d'*Emath*, le travail inverse doit aussi être effectué.

3 Affichage des formules

3.1 De la formule à l'écran

Pour passer de la représentation interne d'une formule à son image à l'écran, trois étapes sont nécessaires:

- Décider du format à utiliser pour représenter chaque opérateur. Par exemple, choisir qu'une fraction doit être affichée avec le numérateur au dessus du dénominateur, séparé par une ligne horizontale; ou encore, qu'un exposant se place en haut à droite, avec une taille réduite.
- Déterminer la taille et la position des caractères ou symboles qui apparaissent dans la formule. En général, cela se résume à choisir une fonte et une taille pour chaque caractère. Il faut toutefois tenir compte du cas particulier de certains symboles (tels que les parenthèses, crochets ou intégrales), dont la taille doit être adaptée au contexte.
- Utiliser ces informations pour afficher, au bon endroit dans la fenêtre, les images correspondant à ces caractères ou symboles.

Les deux premières étapes du processus, c'est-à-dire le choix du format puis le formatage, sont fortement paramétrables grâce aux attributs. Il existe, en autres, un attribut pour chaque type d'opérateur qui détermine la méthode à appliquer lors de son formatage. Dans le cas de l'opérateur `diviser`, cette méthode peut être de disposer verticalement numérateur et dénominateur, ou horizontalement, séparé par une barre oblique.

3.2 Modèle retenu

Lors de l'affichage d'une formule, l'efficacité et l'incrémentalité sont deux points clefs. L'efficacité est importante pour l'affichage de grandes expressions. Idéalement, le temps nécessaire pour l'affichage ne doit dépendre que de la taille de la partie visible de l'expression. L'incrémentalité autorise des délais de rafraîchissement compatibles avec la vitesse de frappe d'un utilisateur averti.

Comme la plupart des logiciels formatant des expressions, *Emath* s'appuie sur le modèle de [Knut86], [Mart71], employant un arbre de boîtes englobantes. Chaque boîte est un sous-arbre (contient une sous-formule), ou une feuille (contient un symbole). À chaque boîte est associée un ensemble de paramètres précisant ses dimensions et sa position relativement à son ancêtre. Pour assurer une bonne efficacité, seules les boîtes correspondant à la partie de formule visible à l'écran, sont construites et maintenues.

3.3 Multi-vue, multi-style

Emath est un éditeur multi-vue. Une même formule peut être manipulée dans plusieurs fenêtres; une modification effectuée dans une fenêtre est reportée dans les autres.
Emath est un éditeur multi-style. Un même type de formule peut être représenté de façons totalement différentes. Dans une même fenêtre, on peut, par exemple, mélanger des expressions sous leur forme infixe et préfixe.
Ces deux caractéristiques peuvent, bien sûr, se combiner pour afficher une même formule de différentes façons selon les fenêtres.

3.4 Elision et césure

L'affichage de grandes formules est un problème délicat. Trois techniques, parfois contradictoires, existent:

- Utiliser la césure pour tenter de visualiser la formule dans son intégralité. Cette méthode n'a d'intérêt que si la version découpée de la formule n'est pas plus haute que l'écran. De plus, certains opérateurs avec une structure verticale, telle une matrice, supportent mal la césure.

- Utiliser l'élision pour n'afficher que les termes les plus significatifs de l'expression. Cette méthode a le mérite de faire ressortir la structure générale de la formule. Mais le choix des termes à retenir est difficile et nécessite des compétences mathématiques approfondies, qui ne sont pas du domaine d'un éditeur.

- Fragmenter l'expression en sous-expressions de tailles plus raisonnables. Sur la figure 11, il est clair que l'utilisation des sous-expressions %1 et %2 facilite la lecture de la formule qui, au complet, serait trop étendue pour rester compréhensible. Dans le cas de certaines formules trop homogènes, cette méthode n'est pas toujours applicable.

À ces trois techniques, peuvent s'ajouter des solutions plus simples, telle que le défilement de la fenêtre le long de la formule, ou encore l'affichage à échelle réduite de l'expression.
Après avoir testé toutes ces solutions, nous avons décidé de ne rien imposer. *Emath* permet tous ces comportements et lorsque ses compétences mathématiques sont insuffisantes, il

Figure 11 : Utilisation de sous-expressions nommées pour faciliter la lecture des grandes formules.

laisse éventuellement décider l'application. Des tests pour obtenir une élision sophistiquée en collaboration avec *Maple* ont donné des résultats encourageants.

4 Implémentation

4.1 Interprète embarqué

Un des principaux atouts d'*Emath* est sa flexibilité. *Emath* a été conçu pour permettre la modification de tous ses comportements (de l'affichage des formules à la gestion des évènements *X*). Cette caractéristique est indispensable pour assurer l'adaptabilité d'*Emath* aux besoins de ses utilisateurs. Cela permet également de faire d'*Emath* une plate-forme d'expérimentation pour l'étude de nouveaux concepts.
Cette flexibilité est obtenue grâce à l'emploi d'un langage embarqué (cette aspect est détaillé au paragraphe 1, page 71). Nous avons retenu, après divers tests, *Klone* [Naha96], un dialecte de *Common Lisp*. *Klone* a été conçu pour être embarqué simplement dans des applications écrites en langage *C*.
Emath peut être vu comme un noyau de primitives *C* (agissant directement sur les structures de l'éditeur), et un ensemble de modules écrits en *Lisp* (implémentant des comportements de plus haut niveau). Cette architecture présente au moins deux avantages. Elle assure une bonne adaptabilité, autant pour l'utilisateur que pour le développeur d'applications

embarquant *Emath*. Elle permet aussi, lors du développement d'*Emath*, d'expérimenter rapidement (grâce à l'environnement interprété) un grand nombre de principes.

4.2 Réactivité et multithreading

Dans un modèle classique, l'arrivée d'un évènement déclenche une des trois opérations: modification, formatage ou affichage de la formule. Ces opérations étant bloquantes; il s'ensuit une période durant laquelle aucun autre évènement ne peut être traité. Malgré un effort important pour les rendre incrémentales, ces opérations sont longues. La réactivité de l'interface est donc mauvaise: un utilisateur averti risque de frapper plus vite que le temps de réaction. Il aura donc la désagréable impression que l'interface réagit avec de plus en plus de retard.
Pour pallier à ce grave inconvénient, *Emath* utilise un modèle différent, basé sur l'exécution parallèle de plusieurs tâches. La tâche de gestion des évènements étant distincte de celle réalisant les opérations plus longues, l'analyse des évènements n'est plus bloquée; une bonne réactivité est donc assurée. Cette séparation entre les différentes tâches est obtenue par la technique dite de multithreading (voir paragraphe 3, page 77).

5 Conclusion

Ces quelques pages ont donné un bref aperçu des différents thèmes abordés dans la suite du document. Le principal but de ce chapitre est de donner un cadre général, afin de faciliter la lecture des sections suivantes. Chacune des idées, exposées ici, sont reprises en détail dans les chapitres correspondants.

Chapitre III *Architecture d'Emath*

Dans le paragraphe 4, page 48, l'intérêt d'un composant réutilisable pour le développement de plusieurs interfaces a été démontré. Dans ce chapitre, nous exposons les aspects les plus intéressants de l'architecture d'*Emath*. L'utilisation d'un langage embarqué, une architecture de type client-serveur et l'emploi du multi-threading sont les principaux thèmes abordés.

1 Langage embarqué

1.1 Présentation

Les langages embarqués sont classiquement des langages interprétés. Après avoir rappelé le principe de fonctionnement d'un interprète, nous détaillerons les caractéristiques d'un langage embarqué. Dans cette section, nous employons indifféremment le terme langage pour le langage à proprement parler et pour l'implémentation qui en est faite.

Interprète

La définition d'un langage interprété a longtemps été fondée sur l'opposition qui existe entre langages interprétés et langages compilés. Cette distinction est de moins en moins évidente mais, pour fixer les idées, on peut admettre qu'un programme est écrit dans un langage interprété lorsque chaque instruction qui le compose est convertie en langage machine[1] lors de son exécution. Toute instruction est donc interprétée lors de son exécution. Par contre, pour un programme écrit dans un langage dit compilé, la traduction de toutes les instructions est effectuée une seule fois lors de la compilation. C'est donc le programme préalablement traduit en langage machine qui est exécuté.

Cette opposition entre langages interprétés et langages compilés est discutable. En effet, pour de nombreux langages, des implémentations interprétées et compilées sont disponibles. Il existe aussi une catégorie intermédiaire pour laquelle le programme est préalablement traduit dans une forme précompilée qui sera interprétée lors de l'exécution. Le langage Java [AG96] en est un bon exemple. Il faut également citer les travaux dans le domaine de la compilation à la volée [KEH94]. Dans ce cas, la compilation s'effectue par blocs d'instructions et lors de l'exécution afin d'améliorer la qualité du code machine produit.

Langage embarqué

La simple distinction interprété ou compilé ne suffit donc pas à déterminer les langages pouvant être embarqués. Pour définir un langage embarqué, il est donc plus judicieux d'employer les critères suivants:

- la possibilité de modifier, lors de son exécution, une partie d'un programme. C'est naturellement possible avec les langages interprétés, mais c'est maintenant aussi le cas des langages disposant de l'édition dynamique de liens[2].
- la possibilité de retrouver les instructions originales lors de l'exécution de leur traduction en langage machine. Ceci est particulièrement utile lors du retour aux sources pour le déboggage.
- la quantité de ressources nécessaires pour l'environnement d'exécution du langage. L'ajout d'un langage embarqué à un programme ne doit pas être trop coûteux en ressources. De ce point de vue, certains interprètes ne sont plus de bons candidats.

[1] Le langage machine est le seul qui soit directement compréhensible par le processeur. Pour exécuter un programme écrit dans un autre langage, il faut donc le traduire en langage machine.

[2] Pour les langages compilés, il a souvent été nécessaire que le fichier exécutable final contienne la totalité de la traduction d'un programme en langage machine. Il est maintenant possible de rajouter dynamiquement (lors de son exécution) des fragments de langage machine à un exécutable.

- la compatibilité des environnements d'exécution (runtime). L'environnement d'exécution du langage embarqué doit pouvoir coexister avec celui du programme principal. Certains langages ont des contraintes trop fortes au niveau de la gestion mémoire.

Pour des raisons historiques, le terme interprète embarqué reste prédominant dans la littérature. Pourtant au vu des remarques précédentes, il conviendrait mieux de parler de langage embarqué. Dans la suite de ce document, les deux expressions seront utilisées indifféremment.

1.2 Intérêt des langages embarqués

Pour répondre aux besoins évoqués dans le chapitre I et plus particulièrement au paragraphe 4.6, page 59, *Emath* dispose d'un langage embarqué. Quel est l'intérêt d'un tel langage dans le cadre d'une interface?

Dans ce contexte, le terme 'langage embarqué' signifie que la partie d'*Emath* codée dans ce langage est aisément modifiable et ce, même durant son exécution. Cette propriété est très appréciable dans le cadre d'un programme destiné à servir de plate-forme d'expérimentation. En effet, la modification dynamique du programme permet de réduire la durée du cycle de développement et donc d'effectuer un plus grand nombre de tests.
De plus, les langages embarqués améliorent l'évolutivité d'un programme en incitant les utilisateurs à l'adapter à leurs besoins. L'un des exemples les plus connus dans le monde *UNIX* est probablement l'*ELisp*[1] utilisé au sein de l'éditeur *Emacs*. La plupart des utilisateurs d'*Emacs* l'emploient pour enrichir les fonctionnalités de base de l'éditeur ou plus simplement pour modifier certains de ses comportements. Cette possibilité d'adaptation à des besoins particuliers est à notre avis indispensable pour les utilisateurs expérimentés.

Dans le paragraphe 1, page 94, une étude des langages embarqués disponibles permet de justifier le choix effectué pour *Emath*. Des détails sur l'intégration de ce langage à *Emath* et une conclusion plus technique sur cette expérience y sont aussi donnés.

1.3 Conclusion

Malgré un léger surcoût dû à l'interfaçage des fonctions exportées par le noyau d'*Emath*, le temps de développement a été considérablement réduit grâce à l'emploi d'un

[1] *ELisp* (ou *Emacs Lisp*) est un *Lisp* dérivé de *Common Lisp* très puissant car il a été adapté aux besoins de l'éditeur *Emacs*.

langage embarqué. Disposer d'un tel langage facilite le prototypage. Pour *Emath*, de nombreux modules ont été complètement réécrits lorsque la première approche choisie est apparue inefficace ou trop peu évolutive. Trois versions différentes de l'automate pour la gestion des évènements ont pu ainsi être développées dans un délai raisonnable.

Pour la réalisation de certaines fonctionnalités élaborées, un langage embarqué s'est révélé indispensable. Par exemple, il est possible pour un client d'*Emath* d'installer dynamiquement de nouvelles fonctions dans le serveur.

Enfin, la possibilité pour un utilisateur final de modifier rapidement les fonctions standard ou même d'ajouter ses propres modules est un plus indéniable. Il est facile avec un langage embarqué, d'adapter à son goût le comportement d'une application. Et, espérons le, *Emath* saura susciter de telles adjonctions.

2 Emploi du modèle client serveur

Emath est un composant regroupant les fonctionnalités de base pour la réalisation d'interfaces intégrant l'édition de formules mathématiques. Les développeurs doivent pouvoir l'utiliser pour réduire le temps nécessaire à la construction d'une interface manipulant des formules, celle d'un SCF par exemple. Nous avons décidé de fournir ces fonctionnalités sous la forme d'un serveur auquel s'adressent les interfaces.

2.1 Avantages du modèle client-serveur

Séparation des codes

Le modèle client-serveur permet, dans ce contexte, de bien dissocier le code écrit spécifiquement pour une interface de celui déjà intégré dans *Emath*. Cette séparation en deux processus facilite la mise au point en limitant les interactions imprévues entre le code en cours de développement et celui, en principe plus stable, du serveur. Du point de vue du génie logiciel, cette séparation a un autre intérêt. Elle permet de contrôler plus strictement les interactions entre le client et le serveur puisqu'elles ont lieu au travers d'un canal de communication bien défini et selon un protocole préétabli. L'étude des messages transitant sur ce canal peut alors servir à la mise au point du client. Il faut tout de même noter que les communications entre client et serveur étant (pour des raisons d'efficacité) asynchrones, le déboggage pas à pas est plus délicat. Un mode synchrone a donc été prévu à cet effet.

Intégration

L'intégration à un plus grand nombre d'environnements de développement est aussi facilitée si l'architecture client-serveur est respectée. En effet, le client peut être programmé en utilisant tous les langages disposant d'une bibliothèque de communication inter-processus. En imaginant qu'*Emath* soit défini sous la forme d'une bibliothèque de fonctions à inclure dans le code de l'interface, de nombreux problèmes d'intégration se seraient présentés. Il n'est pas toujours aisé de regrouper des fragments de programmes réalisés dans des langages différents.

Partage de ressources

Emath est conçu pour permettre à plusieurs interfaces de profiter de ses services au même instant. Ainsi, plusieurs applications s'exécutant sur la même machine, utilisent un seul serveur *Emath*, de même qu'elles partagent un seul serveur *X*. Cette approche permet de partager des ressources. Par exemple, la mémoire employée pour accélérer l'affichage étant utilisée très temporairement, il est possible de l'affecter tour à tour aux différents clients. La consommation de mémoire peut donc être réduite d'autant.

Distribution

Enfin, une des qualités les plus appréciées de l'architecture client-serveur est la possibilité d'exécuter le client et le serveur sur des machines différentes. C'est un point à ne pas négliger dans les environnements modernes où la tendance à la distribution des applications sur le réseau est forte.

Ces différents points justifient probablement les efforts supplémentaires de développement consentis pour faire d'*Emath* un serveur. La section suivante précise la méthode retenue pour permettre au serveur de prendre en charge les fonctions d'édition d'une interface.

2.2 Gestion de la zone d'édition par un serveur

L'interface d'un SCF doit disposer d'une zone d'édition. C'est dans cette zone d'écran que s'effectuent l'entrée des calculs à réaliser et l'affichage des résultats obtenus. Cet élément commun à toutes les interfaces pour la manipulation de formules peut être pris en charge par *Emath*. L'intérêt de cette prise en charge est multiple (comme détaillé au paragraphe 4, page 48). Pour mémoire, le développeur d'une nouvelle interface est dispensé de l'écriture des fonctions d'édition offertes par le serveur. L'utilisation du même composant d'édition par plusieurs applications simplifie leur prise en main. Les utilisateurs retrouvent les mêmes comportements dans chacune des interfaces. Par exemple, la suppression d'une sous-expression s'effectuera toujours par la même séquence de touches.

Principe de la délégation de fenêtre

Le développeur d'une interface pour un SCF va pouvoir intégrer cette zone d'édition gérée par le serveur dans son application. Une interface se compose habituellement d'une hiérarchie d'éléments assurant différentes fonctions. Par exemple, une interface classique dispose entre autres d'une barre de menu, d'une zone d'édition, de barre de scrolling et d'un panneau d'icônes. La construction de cette hiérarchie se fait grâce à des bibliothèques de haut niveau. Les plus fréquemment utilisées sont *Motif* [OSF95] et *Tk* [Oust94]. Il suffit de prévoir dans la hiérarchie un élément neutre de la taille souhaitée pour la zone d'édition et de le déléguer à *Emath*. Cette délégation se fait en transmettant son identifiant au serveur[1]. Une fois connu l'identifiant d'une zone qui lui est déléguée, *Emath* prend en charge la gestion des évènements relatifs à cette partie de l'interface.

Cette gestion est détaillée au paragraphe 4, page 79. Cette technique a le mérite de s'adapter à toutes les bibliothèques pour la création d'interfaces basées sur *X*. C'est le cas de toutes celles disponibles actuellement dont entre autres *Motif*, *Tk*, *Athena* et *Fresco*.

2.3 Conclusion

L'utilisation d'une architecture client-serveur pour la réalisation d'*Emath* semble avoir porté ses fruits. Les points évoqués plus haut ont pu être vérifiés. Plusieurs interfaces utilisent *Emath*. Elles ont été réalisées dans des langages différents (*C*, *Lisp* et *Tcl*) et avec des bibliothèques graphiques variées (*Motif*, *Tk* et *Leda*). Grâce au principe de la délégation de fenêtre, l'intégration d'*Emath* dans ces environnements n'a pas posé de problèmes.

Certains utilisateurs ont toutefois regretté qu'*Emath* ne puisse pas être employé comme un widget classique [MA88] ou du moins comme un widget virtuel [Boehm93]. La taille du serveur[2] et le surplus de complexité engendré par l'existence d'un processus disjoint de l'interface apparaissent aux yeux de certains comme un défaut. Nous envisageons de développer une version simplifiée d'*Emath* pour répondre à ces critiques, mais un important travail de réflexion sur les fonctionnalités à retenir doit encore être réalisé. Cette approche complique aussi l'intégration au sein des différentes bibliothèques graphiques.

[1] Cet identificateur est le *Xid* associé à la fenêtre *X* de la zone déléguée.

[2] 378Ko. Ce nombre ne reflète que le segment texte du programme sous *Linux*. La taille effective du serveur durant son exécution est très variable.

3 Utilisation du Multi-threading

Un autre aspect original de l'architecture du serveur d'édition *Emath* est l'emploi du multi-threading. Cette décision est justifiée dans ce paragraphe. Quelques commentaires sur les différents systèmes multi-flot et sur l'application de cette technique à *Emath* sont donnés dans le paragraphe 2, page 99.

3.1 Assurer une bonne réactivité

Une des qualités souhaitables pour une interface homme-machine est la réactivité. L'interface doit réagir aux actions de l'utilisateur dans un délai raisonnable. Une action de l'utilisateur déclenche généralement une série de traitements. Par exemple, dans un éditeur, la frappe d'un caractère nécessite la remise à jour de l'écran. Dans la majorité des cas, les interfaces ne sont pas réceptives durant l'exécution de ces traitements. Aucune autre action ne peut donc être effectuée avant leur fin[1]. Cette limitation n'est en général pas trop contraignante. Pourtant, dans le cas de l'éditeur de formules, cela peut devenir une réelle gêne. Considérons l'action 'frappe d'une touche', qui déclenche la série de traitements nécessaire au rafraîchissement de l'écran, alors le temps durant lequel l'interface n'est plus réceptive peut être trop long[2]. La génération et l'affichage de la représentation bidimensionnelle soignée d'une formule reste en effet un traitement coûteux en temps de calcul malgré certaines optimisations présentées au paragraphe 5.2, page 115.

Pour pallier à cet inconvénient, plusieurs solutions sont envisageables. Leurs réactivités respectives sont illustrées par la figure 12:

- Une des plus simples consiste à signaler à l'utilisateur que l'interface n'est plus réceptive et à le laisser patienter. Par exemple, dans les interfaces d'Apple pour les modèles 'Macintosh' on voit apparaître une montre ou un sablier à la place du curseur habituel. Il faut remarquer que, dans ce cas, la réactivité n'est pas améliorée.

- Une autre méthode a été prévue pour les applications fonctionnant sous X. La définition de procédures spéciales dites 'WorkProcs' permet d'utiliser les temps morts entre l'arrivée des évènements. C'est ce qui est schématisé sur la figure 12. Ces 'WorkProcs' doivent toutefois être conçues pour "se terminer dans des délais compatibles avec la gestion des évènements"[3]. Ainsi, une longue procédure de

[1] Dans certains cas, les actions sont mémorisées mais leur exécution est différée.
[2] En fait, il doit rester inférieur au délai entre la frappe de deux caractères.
[3] Cité d'après le manuel des fonctions *Xt* pour la manipulation des Workprocs.

calcul devra-t-elle être transformée pour se dérouler en plusieurs appels successifs. Cette modification est délicate et complique considérablement les algorithmes à employer.

- Une autre façon de procéder consiste à interrompre le traitement en cours dès l'apparition d'un évènement, de gérer cet évènement, puis de reprendre le traitement où il en était resté. Cette reprise peut être abandonnée si elle n'est plus justifiée.

Figure 12 : Gestion des évènements, modèle classique, workprocs et coroutines.

Nous avons retenu cette dernière approche. Elle a le mérite d'être la plus réactive et de ne pas compliquer les fonctions de calcul qui n'ont pas à tenir compte des problèmes de gestion d'évènements.

Pour autoriser ce type de comportement, le serveur doit s'exécuter suivant plusieurs flots concurrents. La procédure qui gère l'arrivée d'un évènement doit pouvoir coexister avec celle qui effectue l'affichage. Ce genre de structure est dite multi-flot (plus connue sous l'anglicisme multi-threading). Les détails d'implémentation et une conclusion plus technique sont donnés au paragraphe 2, page 99.

3.2 Conclusion

L'utilisation d'un système de multi-threading pour assurer la réactivité des interfaces semble s'imposer. Aujourd'hui, les bibliothèques pour la construction d'interfaces comme *Xt* [AS90], [STW92] ou l'*Awt* [AG96] ont adopté cette solution. Dans le cadre d'*Emath*, cette approche a permis de conserver une bonne qualité logicielle en séparant clairement les procédures de gestion d'évènement, d'affichage, d'optimisation des structures ou de gestion mémoire.

Après cette discussion sur les méthodes pour améliorer la réactivité d'*Emath*, le chapitre suivant présente un autre aspect des interactions avec l'utilisateur: la gestion des évènements.

4 Automate pour la gestion d'évènements

La description des actions de l'utilisateur est basée sur la notion d'évènements telle que définie par *X*. Àchaque action, correspond un évènement qui contient toutes les caractéristiques nécessaires à son traitement. Ainsi, la frappe d'une touche génère un évènement '`keypress`' qui contient le code de la touche enfoncée. L'interface réagit à ces évènements pour modifier son état. Elle doit donc disposer d'un mécanisme pour associer ses réactions aux évènements. Nous développons cet autre élément de l'architecture d'*Emath* dans ce paragraphe.

4.1 Caractéristiques souhaitées

La méthode choisie pour associer l'arrivée des évènements aux modifications de l'interface doit présenter certaines caractéristiques:

- Il faut pouvoir décrire des motifs d'évènements assez riches. Par exemple, il est souhaitable de pouvoir exprimer une condition du genre: si l'utilisateur appuie rapidement deux fois sur le premier bouton de la souris après avoir enfoncé la touche '`Shift`' et avant de l'avoir relâchée, alors toute la formule dans laquelle se trouve le curseur est sélectionnée. Cet exemple démontre que les conditions qui doivent pouvoir être exprimées sont assez complexes et dépendent de l'arrivée de plus d'un évènement. Il faut aussi prévoir les recouvrements entre plusieurs motifs. Ainsi, la règle précédente peut interférer avec celle-ci: si l'utilisateur appuie sur le premier bouton de la souris, il faut alors placer le curseur le plus près possible du point visé. Dans ce cas, laquelle des deux règles doit s'appliquer?

- Ces associations entre motifs d'évènements et actions doivent être aisément modifiables. Lors du développement d'*Emath*, nous avons testé de nombreuses associations pour déterminer celles qui sont les plus naturelles pour l'utilisateur final. De plus, cet utilisateur peut souhaiter retoucher ces comportements prédéfinis pour les rendre plus à son goût. Les règles décrivant le comportement de l'interface doivent être facilement localisables, l'idéal étant de les regrouper dans un seul fichier. Elles devront être décrites dans une syntaxe la plus naturelle possible pour faciliter leur modification par un utilisateur novice. Enfin, toujours pour simplifier leur mise au point, il est souhaitable de pouvoir tester rapidement des modifications, elles devront donc être intégrables à une interface durant son exécution. Pour résumer, l'ensemble des règles d'association devra être regroupé, facilement compréhensible et si possible modifiable dynamiquement.

- Le système choisi doit permettre à *Emath* de gérer des évènements plus variés que ceux générés par le serveur *X*. *Emath* se comporte aussi comme un serveur et il est intéressant de considérer certaines actions de ses clients comme des évènements afin d'uniformiser leur gestion (ex: Si un client se déconnecte, afficher alors un message d'alerte). Certaines modifications de l'état d'*Emath* peuvent aussi être décrites par l'envoi d'évènements (ex: Si une formule est modifiée, le signaler alors au client qui l'a affichée).

4.2 Systèmes envisageables

Le traitement des évènements s'effectue en trois phases: la collecte, l'analyse et la distribution. *Emath* est conçu pour fonctionner dans l'environnement *X*; la collecte des évènements est donc réalisée par la *Xlib*. L'analyse et la distribution sont le sujet de ce paragraphe. L'étude des modèles les plus couramment employés permet de décider si l'un d'eux convient en l'état, ou si un nouveau développement est nécessaire.

La méthode d'analyse et de distribution la plus primitive est basée sur un traitement du flux des évènements dans une simple boucle. Selon le type de l'évènement qui arrive, une certaine fonction est déclenchée. Ce procédé est bien sûr insuffisant car les motifs comportant plus d'un évènement ne peuvent pas être traités simplement. Cette technique est pourtant trop fréquemment utilisée pour les applications basées sur la *Xlib* [SG90].

Pour enrichir le mécanisme de base proposé par la *Xlib*, les concepteurs de la *Xt* [MA88] ont imaginé le système de tables de traduction ('translation table'). Dans ce modèle, chaque association est décrite par une paire: motif d'évènement / action. Par exemple:

```
Ctrl<KeyPress>C: Clear()
```

La syntaxe qui permet de décrire les motifs d'évènements n'est pas d'une lisibilité exemplaire, mais elle a le mérite d'être compacte et relativement bien connue des utilisateurs d'UNIX. Il est cependant regrettable que rien n'ait été prévu pour ajouter des évènements à la liste prédéfinie par le serveur *X*. Il est donc impossible d'y intégrer les évènements propres à *Emath* tels que: "modification d'une formule" ou "sélection d'une formule".

La distribution des évènements est assurée par l'appel de l'action qui correspond au motif reconnu. Les actions sont en nombre limité car elles sont définies statiquement par le développeur du programme. Il n'est pas non plus possible de transmettre un paramètre à une action, ce qui limite sa généricité.

Certaines boîtes à outils pour la réalisation d'interfaces (comme *Tk*) proposent d'autres mécanismes d'analyse et de distribution basés sur le système précédent. En général, elles tentent de le simplifier. Elles ne peuvent donc pas convenir aux besoins détaillés au paragraphe paragraphe 4.1, page 79.

4.3 Extension des tables de traduction, choix d'un automate

L'approche qui nous a semblé la plus raisonnable repose également sur le mécanisme des tables de traduction fourni par la *Xt*. Son principe de motifs d'évènements associés à des actions est assez intuitif et peut être généralisé pour répondre aux besoins plus spécifiques d'*Emath*. Trois modifications majeures ont été réalisées: l'ajout de nouveaux évènements, l'emploi d'une notion d'état et l'enrichissement des actions.

Extension des motifs

Pour pouvoir traiter de façon équivalente les évènements standard du serveur X et ceux générés par *Emath*, il faudrait modifier des mécanismes internes à la bibliothèque *Xt*. Bien qu'envisageable, cette approche nous a paru peu compatible avec les contraintes d'une thèse. Pour obtenir un résultat satisfaisant, il suffit d'interdire le mélange des deux types d'évènements (standard ou produits par *Emath*) au sein d'un même motif. L'analyse des motifs classiques est faite par les mécanismes habituels de la bibliothèque *Xt*, alors que celle des motifs à base d'évènements ajoutés est faite par du code propre à *Emath*. Du point de vue de l'utilisateur, cette séparation n'est pas sensible et les motifs semblent avoir été enrichis par l'ajout de nouveaux évènements. Il faut noter que la sémantique des deux classes d'évènements est assez différente. Les actions de l'utilisateur (les évènements classiques *X*) n'ont pas grand chose à voir avec ceux signalant des changements dans *Emath*. Interdire leur mélange dans un même motif n'est, de ce fait, pas vraiment limitatif.

Notion d'état pour un automate

Il nous a semblé que le système classique de la *Xt* ne permet pas de décrire simplement certains comportements d'une interface. Il n'est pas possible d'exprimer qu'un même motif d'évènements déclenche deux actions différentes suivant l'état de l'interface. Par exemple, on peut imaginer les associations suivantes:

- Si l'utilisateur est en train d'entrer un nombre et qu'il frappe un chiffre
  Alors concaténer ce chiffre au nombre.
- Si l'utilisateur est en train d'entrer un indentifiant de variable et qu'il frappe un chiffre
  Alors considérer que ce chiffre est le début de l'exposant appliqué à la variable.

Sur cet exemple, il est clair que la première partie de la condition de ces clauses est un état alors que la deuxième est un évènement. Nous avons donc enrichi les motifs en permettant d'y introduire une notion d'état. De ce fait, le moteur d'analyse des évènements d'*Emath* est devenu un automate.

L'exemple précédent peut s'exprimer dans la syntaxe étendue d'*Emath* par:

```
num-S  <Key>[0-9] continue-token
symb-S <Key>[0-9] num-after-symb
```

num-S et symb-S sont des identificateurs d'état qui précisent respectivement que l'utilisateur est en train d'éditer un nombre ou un symbole. Ces symboles sont créés à la demande et leur nombre n'est donc pas limité. L'expression au centre de chaque ligne est une expression régulière qui dénote la frappe d'une touche numérique. C'est un motif comme on en trouve dans les tables de traduction *Xt* classiques. Enfin, le terme le plus à droite de chaque ligne est le nom de l'action à déclencher.

Pour rapprocher le vocabulaire employé jusqu'ici de la terminologie plus formelle des automates, voici les équivalences qui les lient:

- L'état de l'automate correspond à celui de l'interface, il est représenté par la partie gauche du motif.
- L'évènement correspond à la partie droite du motif.
- La transition dans l'automate s'accompagne de l'action qui distribue l'évènement aux fonctions qui en tiennent compte.
- La fonction de transition peut être assimilée à la table de traduction enrichie.

Afin d'obtenir la plus grande concision possible, plusieurs états peuvent apparaître dans un motif et la notation * permet de dénoter tous les états possibles, par exemple:

```
*             Ctrl<Key>f goto-right
num-S|symb-S !<Key>[0-9] continue-token
```

Toujours dans le souci d'apporter la plus grande puissance d'expression possible, nous avons souhaité que l'état qui apparaît dans le motif puisse être composé. Il nous a semblé que certaines caractéristiques de l'état de l'interface peuvent être orthogonales et nécessiter une notation qui le reflète. Par exemple, être en mode 'frappe avec aide' et être en train d'éditer un nombre, sont deux facettes bien distinctes de l'état d'une interface. Un état peut donc être composé de plusieurs facettes regroupées par des parenthèses, par exemple:

```
(num-S help)
```

Ce modèle plus général a été implémenté dans *Emath* en réalisant un prétraitement de la table de traduction enrichie pour la réduire à plusieurs tables classiques qui sont alors gérées par la bibliothèque *Xt*. Lorsqu'un évènement arrive, *Emath* décide suivant l'état courant quelle table classique doit être employée.

Pour fixer les idées la figure 13 présente un automate pour la saisie des expressions de base. Les automates utilisés par Emath contiennent un plus grand nombre d'etat et d'actions.

```
; Reading an expression
(expression "!<Key>[0-9]"     new-number)
(expression "!:<Key>[A-Za-z]" new-symbol)

; Reading a number
(number     "!<Key>[0-9]"     continue-number)
(number     ":<Key>[+*^/-]"   new-operator)
(number     "!:<Key>[A-Za-z]" implicit-number-symbol)

; Reading a symbol
(symbol     "!<Key>[0-9]"     implicit-symbol-number)
(symbol     ":<Key>[+*^]"     new-operator)))
```

Figure 13 : Un automate pour la saisie des expressions élémentaires.

Enrichissement des actions

Les actions associées aux motifs dans les tables de traduction classiques ont deux défauts majeurs. Leur nombre est fixé lors de la compilation de l'application (le programmeur déclare l'ensemble des actions utilisables), et leur appel ne peut pas s'accompagner d'un passage de paramètre, par exemple:

```
Shift<key>Delete deleteToken
```

Pour résoudre ces deux problèmes, l'utilisation d'un langage embarqué s'est révélé très efficace. Les actions d'*Emath* sont en fait des fragments de code exprimés dans le langage embarqué. De ce fait, leur nombre n'est pas fixé car il est possible d'ajouter dynamiquement des fonctions à ce type de langage. Le passage de paramètre est naturellement possible. Dans l'exemple qui suit, il faut noter que la fonction 'delete' peut très bien avoir été ajoutée ou modifiée par un client après sa connection à *Emath*. Les paramètres étant évalués lors de son appel, la fonction 'delete' peut être écrite de façon assez générique.

```
Shift<key>Delete (delete (lenght current-token))
```

4.4 Modification des actions prédéfinies

Emath propose un certain nombre de tables de traduction prédéfinies: pour l'édition classique de formules par exemple, ou encore pour la manipulation de formules à échelle réduite, ou pour l'affichage de formules non éditables mais sélectionnables, etc. En règle générale, il suffit à un client délégant une fenêtre à *Emath*, d'y associer une des tables prédéfinies. Il n'est évidemment pas possible de prévoir les besoins de tous les clients qui utiliseront *Emath*. Il faut donc leur permettre de modifier les tables classiques.

Ajout d'une nouvelle table de traduction

L'ajout d'une nouvelle table de traduction est la méthode la plus générale, bien qu'elle implique vraisemblablement la duplication d'une bonne partie d'une table prédéfinie. Un client peut parfaitement déclarer une nouvelle table de traduction et l'associer à ses fenêtres. Pour des raisons évidentes de sécurité, il est interdit à un client de remplacer une table employée par d'autres clients. Dans cette hypothèse, la nouvelle version n'est associée qu'au client qui l'a installée. La version originale est toujours accessible pour les autres clients.

Remplacement d'une action

Les actions qui sont déclenchées lorsqu'un motif est reconnu dans le flot des évènements sont, elles aussi, remplaçables. En fait, puisqu'il s'agit de fragments de code écrits dans le langage embarqué, elles peuvent être intégralement remplacées. Là aussi, la sécurité exige que cette modification reste localisée au client qui la réalise. Pour cela, il suffit de l'effectuer sur une copie de la table et de laisser l'originale à disposition pour les autres clients. De manière plus générale, la modification d'un comportement standard d'*Emath* ne doit être effective que pour le client qui la réalise. Ce point n'est pas toujours aisé à assurer. Dans l'exemple suivant, le code prédéfini pour traiter l'arrivée du meta évènement '`selection`' est remplacé par un nouveau code.

```
(replace-action 'selection '(some-new-code))
```

Modification d'une action

Plutôt que de remplacer complètement une action standard, il est souvent préférable de conserver son comportement prédéfini pour n'y ajouter que quelques lignes de code. Par exemple, un client peut souhaiter enrichir le mécanisme prédéfini de sélection pour y ajouter un son. Dans ce cas, il est inutile de dupliquer tout le code original de l'action '`selection`' pour n'y ajouter que la ligne qui produit le son.

Nous nous sommes inspirés du langage CLOS [Paepk93] pour fournir une manière simple d'enrichir une action. Dans cette extension orientée objet de *Common Lisp*, un objet peut enrichir une méthode dont il a hérité. Il demande à ce qu'un fragment de code soit exécuté en plus de celui dont il a hérité, en précisant de plus l'ordre dans lequel ces deux exécutions doivent se dérouler. Les deux ordres possibles sont utiles (le code ajouté est exécuté avant ou après le code prédéfini). Par exemple, il est intéressant en exécutant un test préliminaire d'éviter qu'une action classique ait lieu, si elle est jugée incorrecte ou inutile par un client spécial. À l'opposé, en exécutant le fragment ajouté après celui prédéfini, il est possible d'utiliser la valeur de retour de celui-ci. Nous avons repris cette méthode pour l'intégrer dans *Emath*. Il n'est bien sûr plus question d'objet mais juste d'action. Voici un exemple de la syntaxe employée dans *Emath*:

```
(add-action-before 'selection semantic-check)

ou:

(add-action-after 'selection beep)
```

4.5 Conclusion

De nombreux automates ont été testés pour déterminer les comportements les plus intuitifs pour l'utilisateur final. Le système de notation décrit précédemment offre une bonne puissance d'expression. Nous n'avons pas rencontré de comportement qui soit impossible à décrire. Grâce à certaines conventions, les tables qui décrivent les automates restent relativement concises et donc assez lisibles.

Certains points sont moins positifs. Malgré son intérêt théorique, l'aspect multi-facette des états de l'interface n'a pour l'instant pas trouvé d'utilisation convaincante. La syntaxe qui décrit les évènements en provenance du serveur *X* n'est pas très intuitive. Nous avons tenu à la conserver pour profiter du moteur d'analyse fourni par la bibliothèque *Xt* et pour ne pas obliger les utilisateurs à apprendre une syntaxe supplémentaire.

Le bilan reste tout de même très positif. Des comportements complexes ont été décrits par des utilisateurs novices et l'expérience prouve que la notion d'état, qui a été ajoutée aux tables de traduction classiques, est indispensable. Nous ne comprenons pas pourquoi cet aspect a été oublié par les concepteurs de la bibliothèque *Xt*, car il n'est pas aisé de le simuler avec les éléments fournis.

5 Attributs et paramétrabilité

L'un des points les plus soignés lors de la réalisation d'*Emath* est la paramétrabilité. *Emath* est un composant de base pour la réalisation d'interfaces. Il faut donc pouvoir modifier le plus grand nombre possible d'aspects de l'apparence ou du comportement d'*Emath*. Un des choix architecturaux les plus décisifs pour assurer cette capacité est l'emploi des attributs. Une présentation succincte du système de paramétrage employé sous *X* sert d'introduction au système d'attributs retenu pour *Emath*. Nous donnons ensuite une description plus détaillée du principe de fonctionnement des attributs et de leurs applications. Certains points concernant leur implémentation sont abordés au chapitre IV, page 106.

5.1 Les ressources X

Les techniques employées lors de la réalisation d'*Emath* sont, autant que possible, compatibles avec celles prônées par le consortium *X*. Le système retenu doit donc, si possible, être une extension du mécanisme classique des ressources *X*.

système classique

Le paramétrage des applications graphiques utilisant la bibliothèque *X* est basé sur l'emploi des ressources *X*. Ces ressources sont des couples qui associent un motif d'application et une chaîne de caractères. Le motif donne le nom du paramètre et décrit l'ensemble des composants graphiques auquels il s'applique. La valeur du paramètre est la traduction de la chaîne de caractères en un type de donnée prédéfini. Par exemple, l'utilisateur peut vouloir signifier que les fenêtres portant le nom de 'input' pour les applications de la classe '*Maple*' doivent avoir un arrière-plan de couleur grise.

```
Maple.input.background : grey
```

Ces paramètres sont décodés lors de l'initialisation d'une application et servent à fixer certains de ses aspects. Les mécanismes de décodage sont détaillés dans [SG90] et de bons exemples d'application sont donnés dans [Léon95]. Ce mécanisme est, bien sûr, utilisable pour une application réalisée avec *Emath*. Mais nous avons souhaité une plus grande souplesse.

plus de souplesse

Un plus grand niveau de finesse est souhaitable. Le motif qui dénote le domaine d'application d'un paramètre ne peut pas décrire d'ensembles plus précis que les fenêtres *X*. Dans une interface manipulant des formules, une granularité plus fine est souhaitable. Par exemple, on peut vouloir préciser la couleur d'une seule formule et non de toutes les formules présentes dans une fenêtre.

Les paramètres sont décodés lors de l'initialisation de l'interface. Il n'est donc pas possible pour un client du serveur de modifier ces paramètres durant l'exécution de son interface. Là aussi, le mécanisme standard est insuffisant. Par exemple, lorsqu'un utilisateur édite une formule qui est le résultat d'un calcul effectué par un SCF, il est utile de pouvoir mettre en évidence les zones qu'il a modifiées en changeant leur couleur.

Ce sont ces deux points qui ont été améliorés par l'emploi d'attributs. Après avoir donné un éventail des aspects intéressants à paramètrer dans une interface, nous détaillerons le modèle retenu et l'implémentation correspondante.

5.2 Aspects à paramètrer

Le comportement de l'interface est décrit par les tables de traduction (translation tables). Nous n'avons pas rencontré de cas pour lesquels une table de traduction devait avoir un niveau d'application plus fin que celui d'une fenêtre. En général, le client indique à *Emath* la table à employer grâce aux ressources *X* conventionnelles.

L'aspect visuel nécessite un paramétrage plus fin. De nombreux paramètres ont un sens jusqu'au niveau d'un terme de formule. Par exemple, il est utile de pouvoir exprimer la fonte à employer pour l'affichage d'un morceau de formule, ou encore, d'indiquer la méthode de formatage à appliquer sur un terme particulier. Le moteur qui réalise l'affichage des formules est le plus générique possible pour convenir à une grande variété d'interfaces.

Il existe bien sûr d'autres types de paramètres envisageables. Le mécanisme choisi doit autoriser leur ajout. Par exemple, un client peut vouloir indiquer qu'une zone de formule doit agir comme un lien hypertexte et préciser les paramètres du lien associé.

5.3 Partage des paramètres

Emath permet d'avoir simultanément plusieurs vues d'un même document ou de voir plusieurs documents. Il faut donc proposer un mécanisme pour partager les valeurs des paramètres. Par exemple, on peut souhaiter que toutes les fenêtres dans lesquelles l'édition est impossible aient un fond d'écran grisé pour les distinguer rapidement des autres. La spécialisation d'un ensemble de paramètres est aussi un point important à traiter. Pour reprendre l'exemple précédent, on peut souhaiter que toutes les fenêtres non éditables partagent un ensemble de paramètres, sauf une fenêtre qui aurait un fond rouge pour marquer qu'elle contient les messages d'erreur.

Les deux exemples donnés s'appliquent au partage des paramètres par des fenêtres mais il faut prévoir que le mécanisme de partage et spécialisation puisse s'appliquer jusqu'au niveau des termes d'une formule. Ainsi, tous les termes d'une formule peuvent être affichés en noir, sauf certains qui seront dessinés en rouge pour dénoter qu'ils ont été modifiés. Dans ce cas, le système employé peut tirer parti de la hiérarchie naturelle des termes liée à la structure en arbre des formules.

5.4 Systèmes des attributs

Un attribut est un nom auquel est associée une valeur. *Emath* utilise par exemple l'attribut nommé 'background' qui a comme valeur une couleur. Les attributs sont placés dans une structure qui rassemble les fenêtres et les termes en une seule hiérarchie.

Le système retenu dans *Emath* pour réaliser le partage ou la spécialisation des ensembles de paramètres est assez proche des mécanismes d'héritage classiques. Lors de la recherche d'un paramètre pour réaliser l'affichage d'un terme de formule, la hiérarchie est parcourue de façon ascendante et la première occurrence du paramètre est retenue. Dans la suite du document, le terme paramètre est remplacé par celui d'attribut qui illustre mieux le fait que des valeurs sont associées à des éléments d'une structure.

hiérarchie de fenêtres

Les fenêtres gérées par *Emath* sont regroupées en ensembles par une hiérarchie arborescente de groupes. Toutes les fenêtres appartiennent au groupe racine. Lors de la descente dans l'arbre des groupes, les ensembles de fenêtres rencontrés sont de plus en plus spécialisés. Un exemple de ce type de hiérarchie est donné dans la figure 14. Le client définit un arbre de groupes et y dispose les fenêtres qu'il délègue. Chaque fenêtre est finalement associée à un groupe dans lequel elle est seule. Ce niveau de précision n'est pas suffisant pour descendre jusqu'aux termes d'une formule, une autre hiérarchie doit y être associée.

Figure 14 : Exemple d'une hiérarchie de groupes.

hiérarchie de termes

La solution la plus homogène avec celle adoptée pour les fenêtres eût été de définir des groupes de termes et d'insérer chaque terme dans cette hiérarchie. Cette solution est élégante mais elle oblige à préciser lors de la création de chaque noeud, à quel groupe il appartient. De plus, l'expérience a montré que les groupes de termes que l'on est amené à constituer, respectent en général la structure de la formule et sont souvent orthogonaux aux groupes de fenêtres. Un exemple permet de mieux illustrer cette idée. On peut vouloir distinguer une formule des autres (pour marquer le fait qu'elle soit le résultat d'un calcul par exemple) en la soulignant. Pourtant, cette même formule peut apparaître dans des fenêtres éditables ou non. Dans ce cas, la hiérarchie des groupes de fenêtres ne convient plus, il faut une autre structure qui vienne s’y superposer pour marquer ce genre de relations entre formules. Pour ne pas compliquer la gestion des formules, nous avons décidé de conserver leur structure naturelle. L’héritage est effectué suivant la hiérarchie de l’arbre des termes dans la formule (voir figure 15). Cette structure coexiste avec celle des groupes de fenêtres comme décrit au paragraphe suivant.

Figure 15 : Hiérarchie de termes avec attributs.

hiérarchie globale

Lors de l’impression d’un terme, la valeur de chaque attribut reconnu par le moteur d’affichage doit être déterminée. Pour trouver la valeur d’un attribut pour un terme, la recherche s’effectue en remontant dans la hiérarchie à partir de la position de ce terme. C’est la première occurrence de l’attribut qui est retenue et sa valeur est utilisée. La hiérarchie des fenêtres (basée sur les groupes) et celles des termes (basées sur l’arbre des formules) sont réunies en une seule structure par des noeuds spéciaux dits ’racines’. Toutes les formules apparaissant dans une fenêtre donnée, sont regroupées sous un même noeud racine. En plaçant ce noeud sous le groupe simple de la fenêtre considérée, le lien entre les hiérarchies est fait.

L’algorithme complet qui effectue la recherche d’un attribut est donné au paragraphe 3, page 106.

5.5 Conclusion

Ce système d’attributs est très largement employé dans *Emath* et il est implémenté de façon raisonnablement efficace pour les cas traités jusqu’à présent. Une attention toute particulière a toutefois été apportée à certains attributs utilisés par le moteur d’affichage (ce point est détaillé au paragraphe 8, page 124).

Des applications imprévues ont été découvertes aux attributs. Ils sont, par exemple, utilisables par le client qui souhaite annoter certains termes des formules qu'il fait afficher par *Emath*. Lors de la manipulation d'un fragment de ces formules, les attributs qui y sont associés servent au client. Ce principe assez général a de nombreuses applications. Par exemple, un système d'hyper texte est facilement réalisable. Le client positionne un attribut '`hyper-link`' sur certains termes et modifie la table de traduction standard pour que le click souris suive le lien ainsi défini. Le client peut aussi utiliser les attributs pour identifier certaines sous-expressions. Ainsi, lorsqu'elles lui seront retournées par *Emath* (par exemple pour demander une factorisation), le protocole de transmission des formules pourra être amélioré en ne transmettant plus que les attributs identifiant les formules déjà connues par le client.

Les attributs sont aussi très largement utilisés par les fonctions internes d'*Emath* pour le stockage de variables locales aux fenêtres. Les sélections, par exemple, sont stockées dans l'attribut '`selection`' sur le noeud racine de chaque fenêtre.

6 Conclusion

Ce chapitre a présenté les points les plus originaux de l'architecture d'*Emath*. L'utilisation d'un langage embarqué s'est révélée indispensable pour atteindre le niveau de souplesse requis par un travail de recherche et pour obtenir une évolutivité compatible avec les besoins de la communauté très diversifiée des utilisateurs de système de calcul formel.

L'architecture client-serveur n'est pas foncièrement originale mais son application à la réalisation d'un serveur d'édition est, à notre connaissance, unique.

Ce choix nous a, de plus, libéré de certaines contraintes liées à l'utilisation de bibliothèques graphiques de haut niveau et nous a permis d'employer une technique multi-flot pour l'exécution des tâches du serveur. L'implémentation d'un répartiteur spécialisé pour la gestion des interfaces à partir d'une bibliothèque de coroutines assure à *Emath* une excellente réactivité, sans pour autant dégrader la qualité logicielle de son code.

La réflexion menée sur les méthodes de gestion des évènements nous a convaincu de l'intérêt d'une amélioration des mécanismes standard. L'extension des tables de traduction *X*, implémentée dans *Emath* a prouvé son efficacité lors de la réalisation d'automates complexes tels que ceux nécessaires pour une manipulation intuitive des formules. De plus, il est facile pour un client de modifier les tables de traduction prédéfinies par *Emath*, ce qui apporte un degré supplémentaire d'évolutivité.

Toujours dans le but de proposer un système le plus souple possible, l'utilisation d'un modèle d'attributs adapté aux besoins d'une interface simplifie le partage et la spécialisation des paramètres du serveur. Le nombre de modifications apportées dans *Emath* aux mécanismes classiques démontre, nous l'espérons, que de nouvelles études sont nécessaires pour améliorer l'architecture des interfaces.

Dans le chapitre suivant, des points plus techniques sont évoqués. Nous espérons que notre expérience pourra être utile lors de la réalisation d'autres systèmes comparables à *Emath*.

Chapitre IV
Implémentation d'Emath

Nous avons déjà mentionné l'absence d'un composant réutilisable pour l'édition de formules mathématiques. Ce manque nous a contraint à développer *Emath* avant de pouvoir tester de nouvelles approches pour la manipulation de formules. Plutôt que de concevoir au plus vite une bibliothèque de fonctions juste suffisante pour nos tests, nous avons préféré investir plus de temps dans la réalisation d'un composant réellement utilisable comme base à d'autres travaux. Le soin apporté à la réalisation d'*Emath* explique la part importante de ce chapitre dans le document.

Dans ce chapitre, nous aborderons des aspects assez techniques mais que nous espérons être assez originaux pour retenir l'attention: l'interprète embarqué, le multi-threading, le système d'attributs, les déplacements spaciaux et les techniques d'affichage.

1 Klone comme langage embarqué

1.1 Critères de choix pour un langage embarqué

Tout au long du développement d'*Emath*, nous avons souhaité que le résultat final soit utilisable comme base pour de futurs travaux. Il ne s'agissait donc pas seulement de valider des idées concernant l'édition de formules mais aussi de réaliser un programme véritablement fonctionnel. Dans cette optique, il est important que l'interprète embarqué ne nuise pas à la qualité globale du produit. Il faut donc vérifier les critères suivants:

Apprentissage aisé

Un langage ésotérique n'est pas un obstacle pour les développeurs de modules fournis avec *Emath*. Puisqu'ils vont le manipuler fréquemment, le temps d'apprentissage n'est pas significatif. Par contre, les utilisateurs occasionnels qui auront à retoucher les modules standard ne pourront pas investir le temps nécessaire pour se familiariser avec un langage inconnu ou trop complexe.

La documentation doit être de bonne qualité et si possible être accessible en ligne. C'est l'un des principaux facteurs de réussite de l'apprentissage d'un langage, particulièrement si le but est d'écrire rapidement de petits programmes.

Efficacité et fiabilité

Comme il a déjà été mentionné au paragraphe paragraphe 1, page 71, le langage doit être efficace en terme de ressources. *Emath* est composé d'un noyau particulièrement optimisé écrit en *C*. Les modules implémentant les comportements les plus expérimentaux ou ceux qui sont accessibles par l'utilisateur final sont écrits dans le langage embarqué. L'exécution du code contenu dans ces modules ne doit pas dégrader trop significativement les performances globales d'*Emath*. Il faut donc choisir un langage embarqué s'exécutant rapidement et avec une petite empreinte mémoire. L'article [BS95] a guidé notre réflexion.

Le langage choisi doit bien sûr avoir fait ses preuves et être absolument fiable. Au vu du nombre croissant de personnes impliquées dans l'écriture de modules pour *Emath*, le langage embarqué doit avoir certains atouts pour le génie logiciel. Une bonne modularité est facilitée si le langage dispose d'instructions du type '`require`'[1]. Une gestion mémoire la moins contraignante possible est appréciable [Spir90]. Dans ce but, l'emploi d'un

[1] Ce genre d'instruction permet d'établir des liens entre modules qui sont résolus dynamiquement. Lors de l'exécution d'un module, ceux qui sont prérequis pour son fonctionnement sont chargés dans le cas ou ils ne sont pas déjà présents en mémoire.

ramasse-miettes[1] est souhaitable. Enfin, la production automatique de documentation est utile.

Environnement de développement

La fiabilité et l'efficacité d'un programme ne relèvent pas uniquement du langage dans lequel il est écrit. Les conditions dans lesquelles il est développé et testé sont elles aussi cruciales. Que ce soit pour le programmeur principal ou pour les utilisateurs occasionnels, l'environnement de développement du langage choisi doit être complet. La présence d'un deboggeur symbolique[2] est indispensable. Une bonne bibliothèque de fonctions prédéfinies doit être disponible. La présence d'éditeurs adaptés au langage est souhaitable.

Adaptation au noyau d'Emath

Pour être utilisé comme extension au noyau d'*Emath*, le langage embarqué doit permettre un bon interfaçage avec le langage *C*. L'appel depuis les modules additifs des fonctions primitives écrites en *C* doit rester simple. Le passage des structures de données entre le langage embarqué et le *C* doit se faire le plus naturellement possible.

Plus spécifiquement encore, il est important que le ramasse-miettes employé soit du type incrémental car la recherche des zones mémoires inutiles s'effectue de façon non bloquantes [Spir90]. Dans le cas d'une interface, le temps de latence de certains ramasse-miettes n'est pas tolérable. L'utilisateur doit toujours bénéficier de la réactivité maximale et un blocage dû à l'exécution du ramasse-miettes n'est pas acceptable.

1.2 Comparatif des langages embarqués

Après avoir réalisé cette liste de critères, nous avons cherché un langage s'y conformant parfaitement. Le comparatif du tableau 2 montre que rien de tel n'existe (ou tout du moins, n'existait au moment de notre recherche). Nous avons donc envisagé d'écrire un nouveau langage. Il existe de nombreux outils pour le développement des langages comme les générateurs d'analyseurs lexicaux et syntaxiques ou les générateurs d'environnement de programmation comme *Centaur* [BCDI88], [JRG92]. Après réflexion, il est apparu que nous ne disposions pas du temps nécessaire à la réalisation d'un langage respectant tous les critères détaillés au paragraphe 1.1, page 94. Nous avons donc retenu un langage déjà bien établi pour l'intégrer comme langage embarqué à *Emath* Le tableau ci-dessous regroupe

[1] Plus connu sous l'appellation Garbage Collector ou GC, le ramasse-miettes permet de libérer les programmeurs des soucis de désallocation de zone mémoire. Les zones qui ont été allouées mais qui ne sont plus nécessaires au déroulement du programme sont automatiquement détectées et libérées.

[2] Les deboggeurs symboliques permettent de suivre le déroulement d'un programme d'après ses sources durant son exécution. Ils autorisent aussi la consultation des variables d'après leur nom.

l’évaluation, selon les critères déjà mentionnés, des principaux langages embarqués disponibles. (De nombreux autres langages embarquables existent, mais ils nous ont semblé trop éloignés des critères de choix pour être sérieusement considérés. Entre autres: *oscheme* [BDC92], *XLisp* [Betz88], *ELI* [Glic91] et *Elk* [LB94].)

rappel des critères

Les critères portés dans ce tableau sont ceux déjà évoqués dans les chapitres précédents.

- connu — Le langage est-il connu d’un large public? Est-il facile à apprendre?
- doc — La documentation est-elle de bonne qualité?
- efficacité — L’exécution est-elle rapide? L’empreinte mémoire est-elle petite?
- fiabilité — Le langage est-il stable? Est-il propice à une bonne qualité logicielle?
- gc — Dispose-t-il d’un ramasse-miettes?
- debug — Le deboggage est-il facile? Dispose-t-on d’un deboggeur symbolique?
- bibli — Les bibliothèques fournies avec le langage sont-elles assez riches?
- éditeur — Existe-t-il des outils adaptés à l’édition de ce langage?
- C interface — L’interfaçage entre des fonctions *C* et ce langage est-il facile?
- gc incr — Le ramasse-miette est-il incrémental?

	connu	doc.	efficacité	fiabilité	gc	debug	bibli.	éditeur	C interface	gc incr.
ELisp	**	***	*	***	*	**	**	**	**	
TCL	***	***	*	*		*	***	*	***	
Guile	*	**	**	*	*	*	**	*	**	*
Python	*	*	*	***	*	**	***	*	***	*
Perl	**	*	**	**		**	***	*	**	
Klone	*	**	***	***	*	***	***	**	***	*

Table 2: Comparatif de certains langages embarqués.

Plus le nombre d’étoiles présentes dans une case du tableau est élevé, plus le critère considéré est de bonne qualité. L’absence d’étoile signifie que la caractéristique n’est pas présente. Par exemple, Perl n’a pas de ramasse-miettes.

langages embarqués testés

Seuls les langages les plus proches de celui recherché sont comparés dans le tableau tableau 2.

- *ELisp* [LL90] est l’interprète *Lisp* embarqué dans l’éditeur *Emacs*. Il n’est pas possible de l’extraire simplement de l’éditeur pour le réutiliser dans une autre application mais il a été inclus dans le comparatif pour servir de référence. Parmi les interprètes embarqués largement utilisés, c'est probablement le plus ancien.

- *TCL* [Oust90] est un langage de script comparable aux shells d'UNIX. Il fonctionne d'ailleurs aussi en ne manipulant que des chaînes de caractères. Il est très connu car, en association avec la bibliothèque *TK*, il permet de construire des interfaces graphiques assez simplement. De notre point de vue, il ne répond pas aux critères d'efficacité souhaités. C'est un des interprètes les plus lents du comparatif. De plus, il ne permet pas de développer avec une qualité logicielle correcte de grandes applications.

- *Guile* [Gal96] est un nouveau venu. Il est en cours de développement par GNU[1]. Le but de ces travaux est de fournir à la communauté UNIX un interprète embarqué de bonne qualité. Guile est basé sur le langage *Scheme* [Dybv87] et aurait pu convenir à nos besoins. Malheureusement, aucune version stable n'était disponible lorsque le développement d'*Emath* a commencé et, même aujourd'hui, une version finalisée tarde à apparaître.

- *Python* [Ross93] est un langage de script orienté objet. Il est développé au *CWI*[2] à Amsterdam et mérite l'attention. Il est bien conçu pour l'interfaçage avec *C* ou *C++*. Pourtant, sa syntaxe avec une indentation significative nous paraît un peu déroutante. Ceci reste un défaut notable tant que la majorité des utilisateurs n'est pas familière de ce type de langage.

- *Perl* [Schwa95] est aussi un langage de script orienté objet. Là aussi, malgré une bonne efficacité et un interfaçage avec *C* assez facile, la syntaxe reste le point faible. Les programmes *Perl* sont peu lisibles, ce qui est très gênant pour assurer une bonne évolutivité du code. *Perl* est bien adapté à l'écriture de petits scripts mais son usage comme interprète embarqué est plus discutable.

- *Klone* [Naha96] est un dialecte de *Lisp* très proche de *Common Lisp* [Stee90]. Il bénéficie donc d'une grande communauté d'utilisateurs et d'une importante littérature. La documentation plus spécifiquement dédiée à *Klone* est de bonne qualité. Elle est accessible en ligne sous l'interprète ou sous forme hyper-texte. Les performances de l'interprète *Klone* sont très bonnes comme le montre le tableau 3. Sa fiabilité n'est plus à démontrer puisqu'il sert de base à de très nombreux développements, dont certains à caractère industriel[3] réalisés par l'entreprise *BULL*. *Klone* dispose d'un ramasse-miettes de conception assez ancienne mais très efficace. Il est, de plus, incrémental. L'interprète *Klone* dispose d'un deboggeur avec retour aux sources bien plus élaboré

[1] GNU est une association qui fournit le monde UNIX en logiciels facilement utilisables. En effet les protections juridiques qui y sont associées n'empêchent pas de les modifier pour les réutiliser.

[2] Le CWI est le centre de recherche informatique hollandais: *Centrum voor Wiskunde en Informatica.*

[3] Par exemple, le système d'administration de réseau ISM, développé par BULL est écrit en Wool (la version industrielle de Klone). Ce système comporte plus de 100 000 lignes de code.

que ceux des autres langages du comparatif. Toutes les fonctions systèmes sont disponibles et les bibliothèques sont assez riches[1]. *Klone* bénéficie des efforts investis dans les environnements de programmation *Lisp*. Il hérite ainsi, par exemple, des modes d'édition *Lisp* d'*Emacs*. L'interfaçage entre le noyau d'*Emath* écrit en *C* et le langage *Klone* est aisé. C'est un des points qui ont été particulièrement soignés lors de sa conception.

	consommation mémoire (Ko)	temps d'exécution (ms)
ELisp	-[a]	4888
TCL (7.4a-p1)	291	1421
Guile (IV)	305	310
Python (1.2)	1159	1860
Perl (5.001)	472	350
Klone (1.6)	190	107

Table 3: Efficacité de certains langages embarqués.

[a] il est difficile de séparer l'interprète de l'éditeur et la mesure n'est donc pas significative.

L'interprète *Lisp Klone* nous a paru être le choix le plus adapté. Nous disposons de plus, d'un support technique excellent car *Klone* est activement développé au sein de l'équipe *KOALA*[2] sur le site même de l'*INRIA* Sophia-Antipolis. Les conclusions sur l'emploi d'un interprète et plus particulièrement sur l'utilisation de *Klone* sont données au paragraphe suivant.

1.3 Conclusion

La conclusion sur les avantages généraux d'un langage embarqué est donnée en fin de chapitre. Voici quelques points plus spécifiques à *Klone* et à son intégration à *Emath*.

Lisp est probablement un bon choix car sa syntaxe est très homogène. Certains ont pourtant regretté sa lisibilité assez médiocre[3]. Heureusement, le *Lisp* est largement connu au sein de la communauté scientifique, peut-être du fait de son ancienneté (le *Lisp*, apparu en 1964, est le résultat des travaux de *Mc. Carthy*).

[1] Par exemple, Klone dispose de bibliothèques pour le traitement des expressions régulières, pour la communication par sockets ou encore pour la gestion de mémoire partagée.

[2] L'équipe Koala: Kit d'Outils pour un Atelier Logiciel Avancé.

[3] Le nombre élevé de parenthèses et une indentation mal standardisée sont les points qui rebutent le plus les novices.

L'interprète *Klone* s'est avéré assez rapide pour gérer une bonne partie des mécanismes de traitement d'évènements. L'automate est entièrement codé en *Klone* et seules les fonctions d'affichage ou de gestion des structures graphiques ont dû être développées en *C*.

L'environnement de développement est de bonne qualité même si le deboggeur n'est pas encore aussi confortable que celui disponible pour le langage *C*. L'interfaçage avec le noyau en *C* n'a pas posé de problème. On peut toutefois regretter que le travail assez fastidieux qui permet de déclarer des fonctions *C* pour les utiliser depuis *Klone* n'ait pas été automatisé. Cette difficulté devrait être résolue dans les prochaines versions de *Klone*.

Il semble donc que l'emploi d'un langage embarqué et plus particulièrement celui de *Klone* se soit révélé un bon choix.

2 Emath et le multi-threading

Nous souhaitons disposer d'un système permettant l'exécution concurrente de plusieurs traitements au sein du serveur *Emath* afin d'assurer, entre autres, la meilleure réactivité possible lors de la gestion des évènements en provenance de l'interface.

2.1 Les coroutines et les threads

Plusieurs systèmes sont envisageables pour obtenir une exécution concurrente. Cette section les décrit en tentant de préciser leurs limitations.

Multi processus

La solution la plus naturelle dans les environnements de programmation multi-tâches est d'associer un processus à chaque traitement. Outre la relative lourdeur d'une telle solution[1], le problème majeur est la complexité de partager les données entre processus. Dans le cas du serveur *Emath*, presque tous les processus doivent avoir accès aux structures comme l'arbre de la formule ou les ensembles d'attributs. Ainsi, le processus de traitement des évènements y effectue des modifications et celui d'affichage y trouve ses paramètres. Pour réaliser ce partage des structures entre processus, la solution la plus viable est l'emploi de la mémoire partagée[2]. Ce mécanisme a deux inconvénients majeurs. L'allocation des zones partagées de mémoire est délicate et en général restreinte à une taille donnée. Les

[1] Malgré des progrès indéniables dans ce domaine, le surcoût à l'exécution lié à un grand nombre de processus n'est pas négligeable.

[2] Il est possible d'employer des zones mémoires communes à plusieurs processus. Le partage des structures stockées dans de telles zones est alors obtenu sans trop de difficulté.

manipulations au sein de ces zones doivent s'accompagner de beaucoup de précautions pour éviter les problèmes de section critique (abordés plus loin dans ce paragraphe). Bien qu'envisageable, cette solution nous a semblée un peu trop complexe à mettre en oeuvre.

Signaux

Le mécanisme des signaux UNIX peut aussi être utilisé pour réaliser ce type de comportement. En effet, l'arrivée d'un signal détourne le contrôle au profit d'une procédure spéciale[1] puis restitue le contrôle à la fonction interrompue [Bach86]. On dispose alors de deux flots d'exécution qui peuvent être alternés. Cette approche n'est toutefois pas suffisante dans le cas d'une interface où plusieurs procédures de calcul peuvent avoir à s'exécuter simultanément[2]. Dans ce cas, il faut plus de deux flots d'exécution.

Coroutines

Les coroutines ont été conçues pour permettre l'exécution concurrente de plusieurs traitements. Une description de ce type de procédures, pouvant être interrompues et relancées à volonté, a été réalisée par *Conway* dans les années 50. Les procédures classiques (routines) induisent une dissymétrie dans le contrôle d'exécution. La procédure appelante n'a pas le même statut que la procédure appelée. Il existe une hiérarchie dans l'arbre des appels. Dans le modèle des coroutines, toutes les procédures travaillent comme une équipe dans laquelle chaque routine peut choisir de passer le contrôle à toute autre coéquipière.

Figure 16 : Comparaison des flux de contrôle entre les routines, les coroutines et les threads.

[1] Cette procédure (appelée handler) est chargée de traiter la condition spéciale qui vient d'être signalée.
[2] Le rafraîchissement concurrent de plusieurs fenêtres par exemple.

Une procédure peut donc exécuter toute autre procédure connue. Ce mode de transmission du contrôle offre une grande souplesse mais peut se révéler gênant. Par exemple, une procédure ayant terminée sa tâche peut souhaiter passer le contrôle à une coéquipière encore active; ce qui nécessite une connaissance généralisée des procédures et de leur état.

Threads

Ce problème de distribution des tâches est résolu par l'existence d'une coroutine spécialisée qui répartit les temps d'exécution. Les autres coroutines placées sous le contrôle de ce répartiteur ou 'scheduler' sont appelées 'threads'[1]. Le répartiteur choisit un thread à exécuter et lui passe le contrôle d'exécution. Ce thread effectue tout ou partie de sa tâche avant de restituer le contrôle au répartiteur. Ce cycle se poursuit tant qu'il reste des threads n'ayant pas terminé leur tâche. Suivant que la restitution du contrôle au répartiteur s'effectue à l'initiative de celui-ci ou à celle du thread exécuté, on parle de système préemptif ou non préemptif. Il existe aussi des variantes selon la stratégie de choix du prochain thread à exécuter (avec priorités, fifo, round robin etc...).

Préemptif ou non?

En majorité, les systèmes de threads développés aujourd'hui sont préemptifs. Pourtant, cette solution n'est pas sans inconvénient.

Un système de thread est dit préemptif quand le répartiteur choisit le moment d'interrompre la thread en cours d'exécution. Dans ce modèle, l'accès à des ressources partagées ou exclusives doit être géré avec soin. L'exemple classique de l'utilisation d'une variable partagée par deux threads est illustré dans la figure 17. La variable P est partagée entre les threads. Elle n'existe qu'en un exemplaire et sert de compteur. La variable L est locale à chaque thread et existe en autant d'exemplaires que de threads. Si deux threads (A et B) exécutent la fonction 'incp' en même temps, alors la manière dont le répartiteur va entrelacer leur déroulement peut produire des résultats incorrects. En suivant l'entrelacement proposé, il est visible que la variable P n'a été incrémentée qu'une fois alors que la fonction 'incp' a été exécutée deux fois (une fois pour chque *thread*).

La solution pour éviter ce type de problème est d'indiquer au répartiteur que la zone dangereuse (ligne 4 à 6), aussi dite 'section critique', est non-interruptible. Il faut modifier le code pour y introduire des instructions spécifiques qui définissent des exclusions mutuelles (nommées mutex).

[1] Le terme thread a été employé par analogie avec le 'fil' d'une exécution.

Figure 17 : Exemple d'exécution erronée du fait de l'entrelacement incorrect de deux threads.

Ce problème existe pourtant encore si les threads font appel à des fonctions qui n'ont pas été correctement conçues. En fait, un grand nombre de fonctions contenues dans les bibliothèques standard *UNIX* posent problème[1]. Elles ont été écrites sans tenir compte de ces contraintes et ne sont donc pas 'thread-safe'. Il faut alors encadrer chacun de leurs appels par des mutex, ce qui est vite fastidieux. De nouvelles versions de ces fonctions sont en cours de développement chez les principaux fournisseurs d'UNIX mais la situation est encore loin d'être stable.

Les threads employées dans *Emath* sont non-préemptives, ce qui signifie que la thread qui est en cours d'exécution n'est pas interrompue par le répartiteur. Une instruction spéciale nommée 'yield'[2] effectue le retour au répartiteur. C'est donc le programmeur de la thread qui inclut des yields dans son code pour choisir quand la thread suspendra son exécution et transmettra le contrôle à une autre. Ce choix est avantageux pour les systèmes où peu de threads coexistent car il simplifie la gestion des sections critiques. L'emploi explicite des mutex n'est plus nécessaire. Pour reprendre l'exemple de la figure 17, il suffit de ne pas placer de yield entre les lignes 4 et 6. En général, choisir où placer un yield est assez facile. Il suffit d'éviter les endroits où les structures de données sont instables. De plus, le nombre de yield est assez faible; sept instructions yields pour tout le code d'*Emath* suffisent à assurer une bonne interruptibilité des traitements.

[1] Les fonctions `malloc`, `open` ou `printf` sont de bons exemples de fonctions non thread-safe.

[2] Yield signifie 'céder le passage' en anglais.

Cette approche a aussi des inconvénients. Dans le cas préemptif, le répartiteur définit avec une bonne précision les instants où il reprend le contrôle. En général, les interruptions sont faites à un rythme constant, tous les dixièmes de seconde par exemple. Cela signifie que deux threads ayant même priorité partageront équitablement les ressources puisqu'elles seront entrelacées à la même fréquence. Par contre, pour des threads non-préemptives, le répartiteur ne peut pas forcer une thread trop égoïste à partager. Par exemple, si une thread effectue un traitement long sans y inclure de 'yield', elle va monopoliser les ressources au détriment d'autres threads qui ont pourtant peut-être la même priorité. Il faut prendre soin de placer des yields pour entrecouper les procédures les plus longues. Ce n'est bien sûr pas possible si elles sont incluses dans des bibliothèques dont les sources ne sont pas disponibles. Dans le cas d'*Emath*, cet aspect n'est pas pénalisant. Les instructions yields du répartiteur ont un coût quasi négligeable[1] et peuvent donc être exécutées fréquemment.

2.2 *Application à Emath*

Le principe du multi-threading nous est apparu intéressant à intégrer au serveur *Emath*. Il assure une très bonne réactivité de l'interface sans pour autant compliquer les algorithmes à employer pour l'affichage. Il faut toutefois s'assurer que ce choix architectural ne nuit pas à la qualité de la réalisation finale. L'implémentation choisie pour le multi-threading doit donc être efficace et portable.

Choix d'une bibliothèque

Pour réaliser un système à base de coroutines, il faut être capable de reprendre l'exécution d'une procédure là où elle a été interrompue. En pratique, cela implique de pouvoir replacer la pile d'exécution et les registres du processeur dans l'état qu'ils occupaient au moment de l'interruption. Ce sont des opérations délicates nécessitant d'être codées en assembleur. Ce travail de portage est donc réalisé pour chaque nouvelle plate-forme. Il existe heureusement un certain nombre d'implémentations.

- Un effort de standardisation important a été effectué. Il existe une norme POSIX pour les threads. OSF1[2] propose une interface aux *DECthreads* [DEC93] conforme à cette norme. Mais nous n'avons pu trouver aucune implémentation complète des

[1] Sur un Pentium100 en Linux2.0.20 on atteint 250 000 changements de thread (context switch) à la seconde sans optimisation. Il est possible d'améliorer encore ce résultat en testant une horloge avant d'effectuer le yield. Dans Emath, le retour effectif au répartiteur n'a lieu au plus tôt que 1/25$^{\text{ème}}$ de seconde après la reprise d'une thread. Dans ces conditions, un appel à la fonction yield ne coûte plus en moyenne que 0.5µs.

[2] Le système opératif fourni avec les machines du type *Dec Alpha*.

threads POSIX sur toutes les plate-formes standards. Par exemple, lors de l'évaluation de cette bibliothèque (en janvier 1994) seuls les portages pour OSF1 et SunOS4 étaient disponibles. La version SunOS4 était, par ailleurs, très instable. Il faut reconnaître que la situation a évolué depuis. Grâce aux efforts de *Frank Mueller* [Muel93], cette bibliothèque est maintenant devenue un standard tout à fait fiable.

- Sun offre une bibliothèque de threads nommée '*Leight Weight Processes*', [PKBS91]. Malheureusement, il s'agit là d'une interface propriétaire qui n'est disponible que sur les architectures SunOS4 et SunOS5 récentes.

- Une bibliothèque baptisée *CThreads* [CD88] propose les mécanismes de base pour un répartiteur. Malheureusement, elle ne s'est pas révélée suffisamment portable pour être retenue.

- Les *QuickThreads* [Kepp93] proposent, par contre, une interface minimaliste (des coroutines en fait) mais très portable. Actuellement, des versions stables existent pour sept grandes familles de processeurs: *axp*, *80x86*, *sparc*, *mips*, *vax*, *m88k*, *ksr1*. D'autres sont régulièrement développées car la zone de code qui doit être adaptée est très réduite (de l'ordre de trente lignes d'assembleur).

Nous avons finalement retenu les *QuickThreads* qui nous ont permis d'écrire un nouveau système de threads adapté aux problèmes d'ordonnancement de tâches posés par les interfaces à base de gestion d'évènements. Leur intégration à *Emath* et quelques points d'implémentation sont détaillés au paragraphe suivant.

Intégration à Emath

L'intégration de cette bibliothèque de coroutines dans *Emath* a été très enrichissante. Nous l'avons utilisée pour développer un répartiteur spécialisé.

Ce répartiteur utilise un système classique de priorités pour décider quelle thread doit être exécutée. Lorsqu'une thread interrompt son exécution pour retourner le contrôle au répartiteur, celui-ci choisit parmi les threads en attente la prochaine à exécuter. Chaque thread possède une priorité qui reflète son importance pour l'interface. C'est donc la thread la plus prioritaire qui est choisie. Dans le cas où la même priorité est associée à plusieurs threads, il faut s'assurer que toutes auront le droit de s'exécuter. Elles sont donc choisies tour à tour et s'exécutent en profitant à parts égales des ressources de la machine. Ce type d'algorithme est dit "strict priority round-robbin". Pour donner un exemple, la thread qui assure la gestion des évènements générés par l'utilisateur est très prioritaire alors que celle qui effectue les tâches de ramasse-miette a une des plus faibles priorités.

Dans le cas du serveur *Emath*, le nombre de threads s'exécutant au même moment reste assez faible et les problèmes complexes que les répartiteurs plus généralistes doivent résoudre sont évités. Par exemple, il n'y a pas de gestion spécifique des sections critiques, ni de politique pour l'héritage des priorités lors de la création de nouvelles threads. Par contre, le répartiteur que nous avons développé pour *Emath* répond à des problèmes spécifiques aux interfaces. Par exemple, la priorité de certaines threads est modifiée en fonction des déplacements du curseur à l'écran. Les threads d'affichages voient ainsi leur priorité varier en fonction du focus[1]. C'est la fenêtre possédant le focus qui sera rafraîchie la plus vite. La fonction 'yield', déjà évoquée plus haut, est elle aussi adaptée pour assurer une réactivité satisfaisante sans provoquer un trop grand nombre de retours au répartiteur. Chaque changement de thread provoque un surcoût qu'il convient d'éviter aussi souvent que possible. Dans *Emath*, les appels à la fonction 'yield' sont effectifs avec une fréquence maximum de l'ordre de 20 fois par seconde[2]. Ainsi, la réactivité est maintenue à un niveau agréable pour l'utilisateur tout en minimisant le surcoût dû à l'appel du répartiteur.

L'emploi des threads dans le serveur *Emath* améliore le confort des utilisateurs de l'interface. Pour assurer la réactivité maximum, il donne le contrôle à la thread de gestion d'évènement avec un délai très faible (toujours inférieur au vingtième de seconde[3]). Il faut noter que ce chiffre est une moyenne obtenue en conditions réelles d'utilisation. Les évènements pris en compte arrivent donc notamment pendant des procédures d'affichage complexes.

La réactivité d'une interface est perçue par l'utilisateur comme le temps qui s'écoule entre le moment où il génère un évènement et celui où la gestion en est terminée: par exemple, le temps entre la frappe d'une touche et l'apparition du caractère à l'écran. Une bonne réactivité du répartiteur ne suffit donc pas. Il faut assurer que le temps de traitement de l'évènement reste faible. Les algorithmes de gestion des évènements et de réaffichage doivent donc être rapides.

2.3 Conclusion

L'utilisation d'un système multi-flot dans *Emath* s'est avéré très intéressant. Des solutions élégantes aux problèmes de réactivité ont été implémentées, sur la base d'un nouveau répartiteur bien adapté aux besoins d'une interface.

Pourtant, le choix d'une bibliothèque de bas niveau comme les QuickThreads n'est pas sans poser de problèmes. Des incompatibilités étonnantes avec certains ramasse-miettes[4]

[1] Une fenêtre possède le focus lorsque les évènements générés par le clavier lui sont adressés. C'est donc dans la fenêtre ayant le focus que les touches frappées au clavier doivent produire un effet.

[2] Cette fréquence est obtenue grâce à la fonction '`setitimer`' d'UNIX.

[3] De l'ordre de 40ms sur un Pentium100 avec une carte vidéo *Diamond Stealth 964*.

ont été constatées. La gestion des débordements de piles est à la charge du développeur, ce qui l'oblige ainsi à porter un soin particulier aux problèmes de profondeur de récursion. Enfin, le déboggage des programmes utilisant le multi-threading est assez délicat et *Emath* ne fait pas exception à la règle.

3 Implémentation des attributs

Les attributs ont déjà été présentés au paragraphe 5, page 86. Un attribut est une valeur associée à un nom et l'occurrence d'un attribut est placée sur un terme dans un groupe. L'algorithme suivant permet de parcourir la hiérarchie globale pour déterminer l'occurrence la plus prioritaire étant donné une fenêtre et un terme initial. Cette recherche est effectuée pour tous les termes d'une formule durant son affichage. Il est donc indispensable de fournir une implémentation efficace à ce système.

3.1 Algorithme de recherche d'attributs

Un attribut est associé à un terme d'une formule et à un groupe. Ainsi, la remontée dans la hiérarchie globale pour trouver un attribut A pour un terme T0 dans une fenêtre F s'effectue comme décrit dans la figure 18.
Cette méthode peut sembler complexe à première vue mais l'exemple suivant permet de mieux comprendre son intérêt.

Imaginons que, lors de son affichage, l'attribut `couleur` de chaque noeud soit utilisé pour définir la couleur des caractères à l'écran. Sans système d'héritage, il faudrait stocker cet attribut sur chaque noeud de la formule. Le nombre important[1] d'attributs utilisé par Emath provoquerait donc une consomation mémoire inutilement forte. L'héritage se fait en suivant l'arbre de la formule. Pour reprendre l'exemple de la couleur, sur la figure 19, on voit que la couleur noire placée sur le noeud racine est héritée par tous les noeuds de l'arbre sauf par ceux placés sous le noeud `^` qui redéfinit la couleur à `gris`.

Le principe d'héritage au travers de l'arbre des groupes permet une factorisation d'un autre type. Par exemple, il est possible d'imaginer un groupe `Client` contenant toutes les fenêtres produites par un même client. Il est alors facile, en placant un attribut dans ce groupe, d'en faire profiter toutes les fenêtres du client. Ainsi, une formule marquée par

[4] Malgré l'aide de leurs auteurs, il n'a pas été possible de faire cohabiter paisiblement les QuickThreads et le GC conservatif de Boehm et Demers [Boehm93].

[1] Actuellement, *Emath* utilise un nombre d'attributs supérieur à 100. Sur ce nombre, une vingtaine sont effectivement utiles au niveau du noeud, les autres sont globaux à tout un groupe ou même à tout le contexte d'un client.

```
;; G est le groupe courant
;; T est le terme courant

1      G <- le groupe de la fenêtre F.
2      T <- le terme initial T0

3      S'il existe une occurrence de l'attribut A placée sur G et T
4      alors
5         retourner la valeur de l'attribut pour cette occurrence
6      sinon
7         Si T n'est pas un noeud racine
8         alors
9            T <- le père de T
10        sinon
11           Si G a un père dans l'arbre des groupes
12           alors
13              G <- le père de G
14              T <- T0
15           sinon
16              aucune occurence de A n'existe en partant de T0 et F

17     recommencer en 3
```

Figure 18 : Algorithme pour la recherche d'attribut.

Figure 19 : Application de l'algorithme de recherche d'attribut à un cas concret.

l'attribut sélection dans le groupe Client apparaîtra sélectionnée dans toutes les fenêtres du client.

3.2 Structure de données et optimisations

La structure retenue pour le stockage des attributs est une table de hachage dont la clef est basée sur l'identifiant du noeud et le nom de l'attribut. Chaque groupe possède une table qui contient toutes les occurrences d'attributs placés sur une des formules visibles au travers d'une de ses fenêtres. Cette structure permet ainsi d'accéder en temps quasi constant à la valeur d'un attribut à partir de la clef, c'est-à-dire réaliser le test principal de l'algorithme précédent: «existe-t-il une occurrence de l'attribut recherché pour le terme T et le groupe G?»

D'après cet algorithme, le test principal peut, dans le pire des cas, être effectué pour tous les ancêtres du terme T0 et pour tous les groupes auquels appartient la fenêtre F. Si la profondeur moyenne des arbres de formules et de groupe est respectivement f et g, alors la complexité de l'algorithme est en O(f*g). Le nombre de groupes reste faible (la profondeur g de l'arbre des groupes est de l'ordre de quatre). Par contre, puisqu'*Emath* se destine à afficher des formules construites notamment par des systèmes de calcul formel, la profondeur de l'arbre d'une formule peut être très élevée. Par exemple, les polynômes manipulés dans ce contexte ont souvent des degrés supérieurs à cent. Or, un polynôme exprimé sous sa forme de Horner a la profondeur de son plus haut degré.

Nous avons constaté que beaucoup de noeuds ne portent aucun attribut car ils héritent de ceux placés sur leurs parents. Cette observation permet de proposer une optimisation de l'algorithme de base. Un booléen est associé à chaque terme, il indique si au moins un attribut est placé sur ce terme dans au moins un groupe. Par ailleurs, une table de hachage dont la clef est basée sur le nom des attributs est associée à chaque groupe. Elle indique en temps constant si un attribut existe dans ce groupe, et si oui, quel est le niveau N le plus bas de l'arbre des formules où il apparaît. Grâce à ces deux éléments, la recherche dans la hiérarchie des termes est nettement accélérée. L'algorithme présenté dans la figure 20 est la version employée dans *Emath*:

```
1   N <- le niveau le plus bas où l'attribut A est positionné dans G
2   Remonter dans l'arbre de formule depuis T jusqu'au niveau N
3   S'il existe au moins une occurrence d'un attribut placée sur T et
4      S'il existe une occurrence de l'attribut A placée sur G et T
           même corps de boucle que l'algorithme de base (lignes 4 à 16)
5   recommencer en 1
```

Figure 20 : Optimisation pour l'algorithme de recherche d'attributs.

La ligne 1 réduit considérablement le nombre de tests principaux (ligne 4) à effectuer. En effet, les attributs ont tendance à être placés sur des noeuds hauts dans l'arbre pour profiter de l'héritage. Dans le cas contraire, ce sont des attributs de spécialisation qui sont placés plutôt bas. Le cas très fréquent d'un attribut placé sur le noeud racine d'un groupe (comme la table de traduction) est particulièrement optimisé.

Le test très rapide de la ligne 3 évite aussi un grand nombre d'accès plus coûteux à la table de hachage.

La complexité maximum de cette version modifiée reste identique à celle de la version de base, mais appliquée aux attributs employés en pratique par *Emath*, elle est nettement plus performante (le rapport constaté est de 1 à 17 sur les formules du test déjà mentionné.

3.3 Autres aspects

Les attributs ont une contrepartie textuelle: leur structure interne peut être convertie en une chaîne de caractères et inversement. Ils sont donc facilement manipulables pour la transmission ou la sauvegarde. Par exemple, l'expression spécifiant que les fenêtres du groupe 'edit' doivent avoir un fond gris pour tous leurs termes est:

```
(edit root background grey)
```

Ainsi, il est possible de reprendre une session de travail dans l'état exact où elle a été quittée ou de communiquer aisément des modifications d'aspect entre les clients et le serveur.

Des fonctions permettent de connaître tous les attributs définis pour un groupe ou un noeud. De plus, chaque attribut peut être doté lors de sa déclaration d'une documentation et de marqueurs pour préciser son niveau d'importance (par exemple, s'il est réservé au serveur ou s'il est visible par les clients). Ces fonctionnalités peuvent servir de base à un composant d'interface pour l'édition des préférences de l'utilisateur, ce qui évite de lui imposer la compréhension des ressources X.

3.4 Conclusion

Certains utilisateurs ont regretté que le mécanisme d'héritage ne soit pas plus intuitif. Nous espérons qu'il s'agit là uniquement d'un problème de documentation, car ce système permet d'exprimer des relations vraiment utiles entre les groupes de fenêtres et les sous formules.

Les accès à la structure des attributs sont très fréquents. Les efforts consentis pour leur assurer une bonne efficacité ont permis un gain très net de réactivité. L'affichage a été clairement accéléré et c'est un des points les plus sensibles pour l'utilisateur final. Le compromis entre la puissance d'expression, la généralité des attributs et leur coût à l'exécution est donc raisonnable. La part de temps consacrée au traitement des attributs reste faible comparée à celle nécessaire pour les fonctions d'affichage de plus bas niveau. De nouvelles optimisations sont pourtant envisageables. Il est probable que des tables de hachage plus spécialisées que celles actuellement employées pourraient améliorer les performances d'*Emath*. Un système permettant d'accéder plus rapidement aux attributs les plus fréquemment utilisés devrait être intéressant à imaginer. Toutes ces directions seront probablement explorées si *Emath* suscite un intérêt suffisant de la part de la communauté des concepteurs d'interfaces.

4 Déplacements spaciaux

Les logiciels classiques d'édition de formules mathématiques proposent des déplacements peu pratiques. En général, ils s'appuient sur la structure syntaxique de la formule pour déterminer les mouvements du curseur. Il en résulte une trajectoire du curseur difficilement prévisible et donc des déplacements peu intuitifs (ce thème a déjà été abordé au paragraphe 4.5, page 56). *Emath* dispose de mécanismes permettant des déplacements plus cohérents. Ces déplacements sont dit 'spaciaux' car ils s'appuient sur la disposition des symboles dans l'espace de l'écran. C'est donc l'image de la formule plutôt que sa structure syntaxique invisible qui guide les mouvements du curseur.

4.1 Algorithme

L'édition d'une formule a pour effet de modifier la structure syntaxique de celle-ci. Les insertions de termes ou la suppression de sous formules provoquent des modifications de l'arbre syntaxique, qui est la représentation interne de la formule. Les déplacements doivent, par contre, utiliser la représentation externe de la formule; c'est à dire son image à l'écran. Pour permettre cette dualité de l'accès aux formules, *Emath* permet un va et vient assez souple entre ces deux aspects de la structure.

Lors de son édition, une formule contient un point d'insertion. Ce noeud de l'arbre de la formule a une image à l'écran qui se distingue par un fond différent ou un curseur clignotant. Du point de vue de l'utilisateur, le point d'insertion est donc un point de l'écran et non un noeud de l'arbre. Déplacer le point d'insertion consiste donc à trouver en quel point de l'écran l'utilisateur souhaite se rendre, puis d'en déduire le noeud de l'arbre correspondant.

Pour déterminer le nouveau point d'insertion à partir de sa position courante et d'une direction, *Emath* utilise un algorithme assez simple. Nous déterminons l'ensemble des noeuds affichés dans un cône ayant pour origine le point d'insertion courant et pour orientation la direction souhaitée du déplacement. Dans cet ensemble de noeuds, le plus proche du point d'origine est retenu comme nouveau point d'insertion. Ce principe est exposé sur la figure 21, page 111. Les boîtes graphiques correspondant à la formule -x+y^2 sont représentées. Le cône pour un déplacement à droite est figuré par les deux droites partant du point d'insertion courant (la boîte grisée). Le nouveau point d'insertion choisi est la première boîte dans le cône, c'est à dire le symbole +.

Figure 21 : L'algorithme de base pour un déplacement du point d'insertion.

Pour raffiner cette approche, *Emath* gère plus spécifiquement les déplacements en chaîne. Nous appelons déplacements en chaîne une série ininterrompue de déplacements dans la même direction. Ces enchaînements de micro mouvements du curseur sont fréquents, car c'est un moyen simple pour se déplacer rapidement entre deux zones peu éloignées de l'écran. Par exemple, remonter dans la page pour revenir à une formule quelques lignes plus haut ou revenir vers la gauche dans la formule en cours d'édition pour se placer sur un des premiers termes de la ligne. La méthode consistant à choisir toujours le noeud le plus proche dans le cône partant du noeud courant peut écarter le curseur de la trajectoire imaginée par l'utilisateur. Par exemple, un mouvement vers la droite peut amener le curseur sur une boîte plus haute sur la page que la boîte de départ. L'accumulation de ces petits écarts provoque un déplacement plus difficile à anticiper. La colonne de droite de la figure 22, page 112, illustre ce problème. Les boîtes figurées sont celle par exemple de la formule x^2^1+y. Après chaque déplacement à droite, le point d'insertion change. Le curseur s'éloigne progressivement de la ligne de base de la formule et la trajectoire semble être oblique plutôt qu'horizontale.

Dans ces séries de déplacements, *Emath* conserve donc la position de départ de la série, ce qui permet de limiter l'erreur commise. Dans le cas d'un déplacement horizontal, la position verticale initiale est mémorisée et sert de point de référence pour favoriser les boîtes placées sur la même ligne lors des déplacements suivants. La colonne de droite de la figure 22, page 112, illustre l'intérêt de ce perfectionnement de la méthode de base. Le curseur ne s'éloigne plus de la ligne de base de la formule.

4.2 Conclusion

Ce type de déplacement est très naturel et correspond à l'image mentale que nous construisons en prévoyant un déplacement. Les mouvements du curseur dans l'interface sont donc très rapides et retrouvent la simplicité que l'on apprécie dans les logiciels d'édition à base de lignes régulières (comme les traitements de texte). L'emploi des touches

Figure 22 : Gestion particulière des déplacements enchaînés.

de déplacement se substitue donc avantageusement au pointage à la souris pour tous les déplacements de faible amplitude (de l'ordre d'une dizaine de micro mouvements).

5 *Typographie soignée*

Des problèmes de typographie apparaissent pour tous les types d'interface [Hers93], mais ils sont particulièrement difficiles à gérer dans le cas d'une interface pour la manipulation de formules mathématiques. En effet, les règles typographiques associées aux mathématiques sont très nombreuses et complexes. Le nombre de symboles employés et les contraintes de positionnement variées nécessitent des algorithmes élaborés. L'emploi de ces algorithmes coûteux ne doit pourtant pas dégrader la vitesse d'affichage, point clef d'une interface réactive.

Dans ce paragraphe, nous présentons l'importance du formatage bidimensionnel pour les interfaces construites avec *Emath* et les contraintes qui y sont attachées.

5.1 Formatage bidimensionnel

Il a déjà été fait référence au formatage bidimensionnel dans le paragraphe 4.2, page 50. Cette section va détailler les points préalablement évoqués en mettant l'accent sur leur application au sein d'*Emath*.

Intérêt du formatage bidimensionnel

Le formatage est dit bidimensionnel lorsque les éléments d'une même formule sont disposés en employant les deux dimensions de la page. En fait, c'est la manière classique d'écrire les mathématiques par opposition aux méthodes simplifiées employées par certaines interfaces et qui obligent à linéariser les notations. La saisie et l'affichage linéarisés (ou mono-dimensionnels) sont faciles à réaliser, mais ils ne sont pas satisfaisants. La notation classique est préférable car:

- Elle est, bien sûr, plus familière aux utilisateurs puisque c'est cette notation qui est employée pour les documents manuscrits ou les documents correctement typographiés.
- Elle tire aussi partie des deux dimensions de la page et il est donc possible de rassembler sur une surface plus compacte les termes en relation. Il suffit de voir l'exemple de la figure 23 pour se convaincre du manque de lisibilité de la version linéaire.

Figure 23 : Mise en page mono et bidimensionnelle d'une même expression.

- Certains objets sont intrinsèquement bidimensionnels. Les matrices ou certains diagrammes sont naturellement disposés selon deux dimensions. Leur version linéaire est en général illisible (l'exemple de la figure 24 doit permettre de s'en convaincre).

Figure 24 : Un objet intrinsèquement bidimensionnel et son affichage linéaire.

Les interfaces modernes pour l'édition de formules mathématiques sont donc de plus en plus nombreuses à proposer un formatage bidimensionnel. En temps que composant pour la réalisation de telles interfaces, *Emath* se doit d'offrir cette possibilité.

Difficultés à résoudre

Les règles typographiques concernant les formules mathématiques sont nombreuses. Il faut intégrer toutes les notations standard dans le serveur. Certaines notations ont des caractéristiques communes. Par exemple, les parenthèses, crochets, valeurs absolues et normes[1] partagent la même structure: deux symboles (pas forcément identiques) encadrent un corps en adoptant sa hauteur. Ce genre de classe de notation doit être géré par le serveur sans duplication excessive de code. Des techniques pour instancier une notation générique en plusieurs variantes doivent être fournies par le serveur.

L'architecture retenue pour *Emath* assure la possibilité de réaliser une visualisation multi-vue et multi-style. Pour respecter cet autre aspect de la multiplicité des notations, le moteur d'affichage doit pouvoir mélanger sur une même page, différentes conventions de notation.

Certaines relations entre les éléments d'une notation ne sont pas triviales à assurer. Par exemple, la disposition correcte des éléments d'une matrice requiert un algorithme en

[1] D'autres notations partagent cette structure mais la liste ne peut pas être exhaustive sous peine d'être vraiment longue!

plusieurs passes [1]. Un soin tout particulier est donc accordé à l'efficacité des algorithmes de formatage pour assurer que les délais d'affichage restent compatibles avec les besoins d'une interface. Nous avons déjà évoqué ce problème. Par exemple, le temps d'affichage d'une page doit rester inférieur au dixième de seconde, si l'on souhaite conserver une fluidité satisfaisante lors des déplacements page à page.

Un autre point, peut-être moins directement lié à la typographie, mais qui est important pour assurer le confort visuel de l'utilisateur, est l'emploi d'une mémoire auxiliaire de travail (appelée plus communément 'double-buffering'). En effet, aussi optimisés soient-ils, les algorithmes exécutés sur les machines actuelles ne peuvent pas assurer un affichage assez rapide pour rester imperceptibles à l'oeil humain. La persistance rétinienne a tendance à prolonger dans le temps les images reçues par l'oeil. Or, pour effectuer une mise à jour de l'écran, il faut commencer par l'effacer. L'image de l'écran vide alterne donc avec celle de l'écran à jour. Il s'ensuit une impression de papillonnement de l'écran [BG89]. Pour éviter cette sensation désagréable, il suffit de réaliser la mise à jour sur une copie de l'écran hors de la vue de l'utilisateur. Lorsque cette opération assez longue est terminée, la version à jour de l'écran est copiée dans la zone visible. Cette copie étant effectuée dans un temps beaucoup plus bref et sans apparition d'un écran vide, le papillonnement disparaît. *Emath* utilise cette méthode pour assurer un confort visuel maximum. Les références techniques pour l'utilisation de cette méthode dans l'environnement X sont données dans [EW94], [FUCH95].

5.2 Algorithmes employés et performances

Pour assurer le niveau de performance maximal, le moteur d'affichage d'*Emath* est codé en langage C. Il effectue la traduction de la structure en arbre des termes d'une formule en une image, suivant un ensemble de règles typographiques. Cette transformation s'effectue en trois phases:

- L'arbre des termes est converti en un arbre de boîtes dites GBox[2]. Le type d'une GBox indique son comportement vis-à-vis de ses boîtes filles, par exemple, si elle va les disposer horizontalement ou verticalement. Cette information géométrique est nécessaire aux étapes suivantes. Cette conversion va déterminer le type de notation utilisée. Deux exemples de conversion sont donnés dans la figure 25.

[1] La position d'un terme de la matrice n'est connu qu'après avoir déterminé, par une recherche plus globale, la largeur de chaque colonne de la matrice et la hauteur de chaque ligne.

[2] Le terme de GBox sera employé pour décrire cet élément de base des structures d'affichage. Il est l'acronyme de Graphic Boxes ou boîtes graphiques.

- L'arbre de GBox est reparcouru pour calculer les positions exactes qu'elles occuperont sur la page lors de leur affichage.

- Enfin, l'arbre de Gbox est affiché en utilisant les positions qui viennent d'être calculées.

Figure 25 : Deux conversions possibles pour une même formule.

Cette technique est très classique. Ce système nous est apparu satisfaisant et nous n'avons pas modifié son principe. Par contre, son déroulement en trois parcours de l'arbre, comme il l'est décrit ici, n'est pas efficace. L'affichage doit être rapide. Or, l'arbre des formules affichées peut être considérable[1]. Plusieurs améliorations sont possibles:

- Lors de l'affichage d'une nouvelle formule, il peut être inutile de parcourir tout son arbre si son extension après affichage est plus grande que la zone visible par l'utilisateur. Par exemple, un polynôme peut occuper plusieurs pages. Il est alors inutile de construire tout son arbre de GBox pour n'en afficher qu'une partie. Des évaluations partielles ou retardées permettent d'optimiser ce cas. Les GBox à grande extension ne construisent pas leurs filles si elles ne sont pas visibles. Cette approche améliore considérablement les performances du moteur d'affichage mais elle complique son code. Certains cas particuliers doivent être traités. Par exemple, la GBox employée pour disposer verticalement les équations d'un système, doit assurer que le symbole d'accolade placé à leur gauche a bien la taille de tout le système. Or, si l'on souhaite ne pas évaluer la taille des équations qui ne sont pas visibles, ce calcul ne peut être effectué avec exactitude. Pour résoudre ce genre de difficulté, certaines heuristiques sont employées. Dans l'exemple du système d'équations, la taille réelle de l'accolade est estimée à partir de la taille des équations qui sont effectivement visibles à l'écran. Pour ces boîtes, les dimensions exactes sont connues, leur taille moyenne est utilisée comme approximation de la taille des boîtes non encore évaluées. Par la suite, si leur taille exacte est calculée

[1] Emath affiche sans difficulté des formules contenant cent mille termes.

(au cours d'un déplacement du curseur par exemple), l'approximation initiale sera améliorée.

- Lors de la modification d'une formule déjà affichée, seule une petite zone change effectivement d'aspect à l'écran. C'est particulièrement vrai lors de la saisie d'une formule car, dans ce cas, le point d'insertion des caractères est placé à droite de la formule. La partie gauche évolue donc très peu. De manière générale, l'algorithme de construction, de formatage et d'affichage de GBox implémenté dans *Emath* est incrémental [SH95]. Seuls les nouveaux termes d'une formule sont traduits en GBox. Ils sont alors raccrochés à l'arbre préexistant. Lors du formatage, les informations de position d'une GBox sont données relativement à sa mère. De ce fait, le déplacement d'une GBox n'invalide pas les positions de ses filles. Là aussi, le calcul des positions n'est effectué que si nécessaire. Enfin, l'affichage est doublement incrémental. En effet, seules les boîtes nouvelles ou modifiées sont affichées et un système de tampon permet d'améliorer cette étape. C'est le point suivant.

- Lorsqu'un fragment de formule est déplacé à l'écran, il est inutile de le reconstruire intégralement. Si son image est conservée, il suffit de la réafficher à la nouvelle position. Ces déplacements sont plus fréquents qu'il n'y paraît. Par exemple, lorsqu'on insère des termes à gauche d'une formule, il suffit de décaller toute la partie droite. Pour assurer que ce genre d'optimisation est effectuée aussi souvent que possible, *Emath* dispose d'heuristiques.

- Il est possible d'accélérer les déplacements de faible amplitude de la fenêtre le long de la formule[1]. Lorsque l'affichage de la partie visible de la formule est terminé, *Emath* précalcule une partie des GBox qui environnent la partie visible. Ainsi, il est possible d'effectuer plus vite les petits déplacements (qui sont assez fréquents). Le système multi-flot permet de réaliser ce genre de calcul sans dégrader la réactivité. Ce sont les temps morts entre l'arrivée des évènements qui sont utilisés. Une thread spéciale est activée lorsque l'affichage est terminé; elle est interrompue dès qu'un évènement est en attente. Si les fragments de formule qu'elle traite ne sont pas modifiés lors de la gestion des évènements, elle reprend son déroulement au prochain temps mort.

L'efficacité des algorithmes employés pour l'affichage ne se mesure pas uniquement à leur vitesse. L'occupation mémoire des structures de données qui leur sont nécessaires doit aussi être considérée. Voici quelques techniques utilisées dans *Emath* pour limiter la

[1] Par exemple, lorsque l'on souhaite faire remonter la fenêtre de quelques lignes vers le haut.

consommation mémoire du moteur d'affichage:

- La construction partielle de l'arbre de GBox déjà évoquée plus haut diminue aussi la consommation mémoire. Une GBox occupe 52 octets[1] en mémoire et il faut en moyenne 1,7 GBox par terme de formule[2]. L'économie réalisée en ne traduisant en GBox que la zone visible plutôt que la totalité de l'arbre est donc très importante. Le nombre de GBox présentes en même temps en mémoire est à peu prêt constant. Il dépend[3] du nombre de fenêtres gérées par *Emath*, de leurs surfaces et du nombre moyen de symboles par unité de surface[4].

- L'optimisation (mentionnée dans la première partie de ce paragraphe) qui consiste à essayer de déplacer l'image d'une sous-formule plutôt que de la reconstruire à partir des symboles de base, peut aussi être employée pour réduire le nombre de GBox. Si l'image d'une sous-formule est conservée, les GBox qui s'y rapportent peuvent être détruites. Elles seront reconstruites si besoin, par exemple si l'image n'est plus valable car un de ses éléments a changé. Cette optimisation permet de diviser le nombre de GBox employées par 6 environ. Pour éviter d'allonger le temps d'affichage de la formule, cette élimination des GBox inutiles est effectuée durant les temps morts par une thread spécialisée.

- Une autre optimisation est envisageable mais elle n'a pas été réalisée dans *Emath*. Lorsque deux fenêtres visualisent avec les mêmes conventions de notation la même zone d'une formule, les GBox peuvent être partagées entre les deux fenêtres. Le cas nous est apparu peu fréquent et trop difficile à identifier pour retenir cette technique.

5.3 Conclusion

L'affichage bidimensionnel implique une gestion assez complexe du formatage des symboles. Les mécanismes à base de GBox, tels qu'ils ont été implémentés dans *Emath*, offrent de bonnes performances. Le délai nécessaire à l'affichage d'une page est en moyenne de 0.3s[5]. Ces résultats sont dûs au nombre important d'optimisations qui ont été réalisées. La version originale n'offrait pas des performances suffisantes pour être utilisable. Plusieurs versions du moteur d'affichage ont été développées. La version

[1] Cette valeur a été obtenue sur une architecture 32 bits classique.

[2] Certains termes se traduisent par plusieurs GBox. Par exemple, la somme de deux chiffres nécessite quatre GBox; celle qui structure horizontalement les caractères et les trois correspondant aux chiffres et au symbole +.

[3] En fait c'est approximativement le produit de ces termes.

[4] Cette valeur est plus difficile à évaluer, elle dépend du type de notation choisie et de la taille de la fonte de base.

actuelle nous semble offrir un compromis valable entre vitesse, consommation mémoire et paramétrabilité.

6 Les caractères de taille variable

De nombreuses règles pour la typographie des mathématiques impliquent l'utilisation de caractères à taille variable. Les plus simples sont les caractères dont la taille change sans modification de leur aspect. C'est, par exemple, le cas des indices ou des exposants. Aucune difficulté autre que technique n'est apparue pour leur affichage. Par contre, il existe une famille de caractères dont l'aspect change en même temps que leur taille. C'est le cas des parenthèses, des crochets, ou du symbole intégrale. Ce paragraphe présente quelques idées sur le sujet.

6.1 Formats matriciels (bitmap)

Lors de son affichage à l'écran, un caractère est constitué de points. C'est pour cette raison que les premiers formats de description de caractères ont utilisé une grille de points. Ainsi, chaque caractère est décrit par une matrice de booléens indiquant les points à afficher. Ce type de format, dit bitmap, est encore très fréquemment utilisé pour les fontes destinées aux périphériques de résolution moyenne.

6.2 Formats utilisant les contours (outline)

De nouveaux types de périphériques à plus haute résolution, comme les imprimantes laser, ont imposé un autre format [Knut86], [Karo87]. La place mémoire occupée par le format bitmap croissant comme le carré de la résolution, il a fallu mettre au point un format plus compact. Celui-ci est basé sur la description de la forme du caractère. Son contour est exprimé par un ensemble de courbes. Cette approche résout efficacement le problème de stockage des fontes. Il améliore aussi l'indépendance de la fonte par rapport à la résolution. Il est possible d'afficher ce type de fontes, dit outline, quel que soit la résolution du périphérique visé ou la taille finale souhaitée. Lors de l'affichage, il faut toutefois convertir cette forme de caractères en un ensemble de points qui seront affichés. Cette étape peut être assez coûteuse et induit de nouvelles difficultés lorsque la résolution finale est faible. La discrétisation en pixels de la surface décrite par les courbes est délicate. De nombreux artefacts en résultent. Sur la figure 26, par exemple, on peut noter des différences entre la

[5] Cette valeur est indicative. Le délai effectif varie assez nettement suivant les conditions de l'affichage. Il est de l'ordre d'une seconde si la page n'a jamais ete affichée et inferieur au 1/10$^{\text{ème}}$ de seconde s'il s'agit d'un simple défilement vertical d'une ligne .

version outline du 'm' et la version bitmap qui en découle. Les barres verticales sont d'épaisseurs inégales et le sérif de la première barre est absent. Ces défauts dégradent sensiblement la lisibilité des caractères générés. De nombreuses indications pour résoudre ces problèmes sont données dans [AP92] et [AB89], mais le domaine reste encore très ouvert et aucune solution totalement satisfaisante ne semble se dégager.

Figure 26 : Artefacts dus à la discrétisation d'un contour pour une faible résolution.

6.3 Application aux notations mathématiques

Pour afficher des caractères de taille variable (tels que les crochets) il est possible, en première approximation, d'utiliser les fontes outlines. A partir de la description de la forme du crochet, il est possible de générer son image à l'écran pour toutes les tailles souhaitées.

Le système de fenêtrage X propose, depuis la version R5, la possibilité d'utiliser de telles fontes grâce à un serveur de fontes [Lemk91], [SG90]. Malheureusement, les résultats obtenus sont assez décevants. Les techniques modernes pour appliquer une fonte outline aux résolutions typiques des écrans ne sont pas encore assez diffusées.

Nous avons fait quelques expériences pour implémenter de telles techniques mais l'effort de développement requis nous est vite apparu prohibitif pour un gain difficile à évaluer. *Emath* dispose tout de même d'un format spécial de fontes outlines basé sur des courbes de Bézier et destiné à être affiché en nuances de gris sur un périphérique de basse résolution (de l'ordre de 100dpi). Nous avons pris soin d'employer les méthodes les plus efficaces qui soient connues pour le moment (voir [Conc89], [NN89] et [Fahl89]). Les ouvrages [AH89], [AM91] et [FDFH93] contiennent un grand nombre d'informations utiles sur ce thème. Dans l'état actuel, cette approche n'est plus employée pour deux raisons:

- Du fait de la lenteur des opérations de manipulation de pixels en X, la génération de l'image d'un caractère d'après son contour est longue. Le système de cache de pixmap, déjà mentionné, ne compense qu'en partie cette lenteur car les caractères de taille variable sont plus fréquemment modifiés que la moyenne. Par exemple, une parenthèse encadrant une expression peut voir sa taille varier avec chaque modification de l'expression.

- Les notations mathématiques ont fréquemment recours aux symboles de petite taille (indices, exposants,...). C'est justement dans ce cas que les méthodes basées sur des contours ne parviennent pas à produire un affichage comparable à ceux des fontes classiques.

D'autres règles typographiques compliquent l'affichage des symboles mathématiques de taille variable. L'article [AV93] présente en détail ces difficultés. Pour résumer, la taille n'est qu'un des paramètres à prendre en compte. L'épaisseur ou la forme de tels caractères doit aussi varier. L'emploi de fontes outlines ne résout pas directement ces problèmes. Nous avons aussi expérimenté sur ce thème, mais les délais imposés par un travail de recherche n'ont pas permis de pousser très loin. Comme résultat préliminaire, les fontes outlines d'*Emath* permettent de générer de nouvelles formes de symboles par interpolation entre deux (ou plus) contours connus (figure 27).

Figure 27 : Génération de nouveaux symboles par interpolation entre des contours connus.

6.4 Solution adoptée

Les tentatives d'utilisation de fontes outlines pour générer les symboles de taille variable s'étant révélées peu concluantes, nous avons retenu une approche moins satisfaisante sur le plan intellectuel mais efficace en pratique. Les symboles de taille variable étant, somme toute, assez peu nombreux[1], ils sont gérés au cas par cas. Lors de l'affichage d'un caractère, le serveur *Emath* utilise une meta-fonte. Ces meta-fontes recouvrent les fontes X classiques, qu'elles soient bitmap ou outline et les fontes internes qui sont en fait procédurales. Ces fontes internes regroupent un ensemble de fonctions capables de générer l'image à l'écran de certains symboles de taille variable. Cette construction de l'image d'un symbole s'effectue en tenant compte de certains paramètres comme la dimension souhaitée ou l'épaisseur du trait à employer. Le résultat est tout à fait correct et s'adapte bien aux contraintes de vitesse imposées par l'interactivité de l'affichage.

6.5 Conclusion

Le résultat de ces expérimentations est assez décevant. Nous n'avons pas pu mettre en place un système satisfaisant pour la génération des caractères de grande taille ou de taille variable dans l'environnement *X*. Les fontes outlines, bien qu'a priori les meilleures candidates, n'ont pas pu être employées pour tous les symboles de taille variable. Il est probable qu'avec un peu de maturation, ces techniques remplaceront avantageusement la solution un peu trop ad-hoc retenue dans *Emath*. Certaines bibliothèques comme *FreeFonts*[1] semblent être des candidats intéressants. Nous n'avons pas encore pu obtenir de version assez stable pour effectuer des tests significatifs, mais ce type de contribution va probablement combler une grave défaillance de l'environnement de développement *X*.

7 Ajout de nouvelles notations

Emath propose un large éventail de notations prédéfinies, mais il n'est pas raisonnable d'espérer recenser tous les types possibles. Il est donc nécessaire de pouvoir en ajouter simplement. Ces mécanismes, permettant la définition de nouvelles notations, sont décrits dans la section suivante.

7.1 Dynamicité

Emath permet d'enrichir ou de modifier à tout moment les catégories de noeuds qu'il sait manipuler. La représentation interne d'une formule dans *Emath* est basée sur le modèle classique d'arbre dans lequel chaque noeud possède un type indiquant ses caractéristiques. Chaque noeud est donc associé à un type de notation mathématique. Ce type n'est pas entièrement décrit par son aspect graphique. La définition d'un nouveau noeud doit

[1] Les grandes familles sont: les délimiteurs (parenthèse, crochet, valeur absolue...), les symboles spéciaux (intégrale, intégrales multiples, barre horizontale de division...) et certains diacritiques (vecteur, complémentaire...).

[1] FreeFonts est une bibliothèque pour l'affichage des fontes TrueType sous X. En se substituant à Emath pour la gestion des caractères outline cette bibliothèque devrait améliorer l'esthétique globale.

préciser, entre autres, son arité ou sa précédence pour permettre de gérer le parenthésage implicite. La ligne suivante illustre l'ajout de l'opérateur '`sqrt`' d'arité 1 avec une précédence maximale et une méthode d'affichage prédéfinie.

```
(node:define "sqrt" :arity 1 :priority 'max :display `((,bm:sqrt)))
```

Cette dynamicité prend tout son sens, par exemple, lors du chargement d'un module dans un SCF. Cet ajout d'opérateurs dans le SCF peut alors s'accompagner d'un enrichissement des notations employées par *Emath*.

Les caractéristiques intrinsèques d'un opérateur, comme son arité ou sa précédence, sont identiques dans toutes les fenêtres d'une même application pour éviter de possibles confusions. Par contre, les aspects plus graphiques peuvent être différents.

7.2 Aspect graphique

Chaque type d'opérateur est muni d'une méthode qui effectue sa traduction en GBox. C'est l'application de ces méthodes sur tout l'arbre d'une formule qui produit son équivalent en Gbox. Une fois cette traduction effectuée, les mécanismes d'affichage prennent le relai. L'ensemble des méthodes de traduction fournies par *Emath* est assez réduit, mais il est possible de les composer ou de leur transmettre des paramètres qui modifient leur comportement. Par exemple, la traduction d'un vecteur en GBox s'effectue en composant une méthode qui affiche des délimiteurs de taille variable (ici les symboles de parenthèse pris dans la fonte interne) avec la méthode qui aligne verticalement des expressions en intercalant des espaces. Soit:

```
(bm:delim if:left-bracket if:right-bracket)
(bm:v-nary if:space bm::center)
```

La liste des méthodes à composer pour la traduction d'un type de noeud est stockée par attribut. Il est donc très facile de modifier l'aspect graphique que prendra un type de noeud d'une fenêtre à l'autre ou au sein même d'une formule. Dans l'exemple suivant, les noeuds de type "`multiplié`", pour toutes les fenêtres du groupe `debug-windows`, perdent leur aspect traditionnel pour être affiché avec la méthode par défaut (une notation préfixée de type `lisp`).

```
(asets debug-windows *root-node* "mult-bm" bm:default)
```

La liste des méthodes prédéfinies est assez réduite. Elle est très nettement inspirée des travaux de l'équipe *CROAP*[1] de l'*INRIA* sur le langage *ppml* [BCDI88]. Voici la liste des

[1] Conception et Réalisation d'Outils d'Aide à la Programmation.

grandes familles de méthodes:

- disposition: pour aligner verticalement, horizontalement ou suivant une grille régulière...
- spéciales: pour certains opérateurs comme la racine carrée ou la division...
- finales: pour afficher les symboles de taille variable ou plus simplement un caractère.

Il est, bien sûr, possible d'ajouter de nouvelles méthodes de base en les incluant au code *C* du serveur. Cette solution, bien qu'ayant le mérite d'exister, n'est pas satisfaisante. Nous envisageons de rendre accessibles en *Lisp* les fonctions *C* de base permettant de construire dynamiquement de nouvelles méthodes de traduction. En fait, pour le moment toutes les notations que nous avons eu à traduire ont pu l'être grâce au jeu déjà implémenté dans *Emath*, mais il est très probable que cela ne soit pas toujours le cas. L'approche, permettant de créer dynamiquement de nouvelles méthodes de traduction, autrement que par composition de celle prédéfinies, sera implémentée ultérieurement.

7.3 Conclusion

Nous avons trop peu d'expérience dans le domaine de l'ajout de notation pour tirer une réelle conclusion. Il semble que les mécanismes choisis soient suffisamment souples pour permettre un enrichissement aisé de la gamme des notations connues d'Emath. L'éventail actuellement proposé par Emath a été défini en utilisant ces même mécanismes. Le résultat est satisfaisant mais l'avenir nous apportera probablement son lot de déconvenues.

8 Spécialisation de certains attributs

Les mécanismes d'héritage pour les attributs, décrits au chapitre paragraphe 3, page 106, se sont montrés très efficaces en général. Toutefois, dans le cas des attributs contrôlant l'aspect visuel d'une formule, certains comportements n'ont pu être réalisés. Des considérations d'efficacité ont aussi imposé une spécialisation du mécanisme général.

8.1 Problème d'efficacité

Lors de l'affichage de chaque symbole à l'écran, un certain nombre d'attributs (tels que la couleur de l'arrière-plan, la couleur du caractère, la fonte à employer...) doivent être obtenus. Ce nombre très important d'appels à la fonction '`aget`' pénalise lourdement l'affichage. Dans la version initiale, près de 40% du temps d'affichage était dû à ces requêtes. Un algorithme plus efficace a donc été implémenté.

8.2 Comportements souhaitables

Les attributs influant sur l'aspect de la formule sont utilisés pour indiquer différents types d'informations. La couleur des caractères peut différencier les formules qui ont été introduites par l'utilisateur de celles produites en réponse par un SCF. Le style de la fonte peut dénoter les zones ayant des propriétés hyper-textes. L'image d'arrière-plan peut délimiter la sélection courante. De nombreuses autres applications sont envisageables, mais dès que leur nombre dépasse celui des attributs graphiques, il n'est plus possible d'assurer leur indépendance. Si dans l'exemple précédent, on souhaite aussi visualiser la dernière zone sélectionnée, il faut utiliser un attribut déjà employé, par exemple la couleur des caractères. Dans ce cas, il faut décider, lors de l'affichage, comment lever la surcharge apparue. La figure 28 permet de comparer le résultat actuel et le résultat obtenu avec le mécanisme d'héritage classique des attributs. La première ligne montre la formule avec une zone de lien hyper-texte (le terme b_i) et une partie sélectionnée ($a_i x^2$). Le lien est visible car la couleur et le style des caractères qui le composent sont différents de ceux employés habituellement (style italique et couleur grise plutôt que style droit et couleur blanche). La sélection est délimitée par une zone où le texte apparaît en noir sur fond plus clair. Sur les deux lignes suivantes, la zone de sélection a été aggrandie pour couvrir toute la formule.

- Dans le premier cas, les attributs 'italique' et 'couleur grise' continuent d'avoir effet pour le b_i. La couleur du fond est, par contre, obtenue en héritant de celle placée plus haut dans l'arbre de formule (comme le stipule l'algorithme d'héritage d'attributs donné au paragraphe 3.1, page 106). La lisibilité n'est pas bonne car le contraste entre le b_i et le fond n'est pas suffisant.

- Dans le second cas, le système de faces permet de spécifier que la couleur de la sélection est prioritaire sur celle du lien hyper-texte. La couleur placée sur le b_i est donc surchargée par celle héritée de la sélection. La lisibilité est meilleure.

Figure 28 : Affichage obtenu avec la méthode classique d'héritage et la variante destinée aux faces.

8.3 Solution retenue

Un sous-ensemble des attributs a été isolé pour être groupé sous le nom de face[1]. Une face est un ensemble d'attributs, dont les valeurs pourront être retrouvées en un seul appel à 'aget', et qui bénéficient d'un mécanisme d'héritage un peu différent. Chaque face possède une priorité qui va influencer la résolution des surcharges.

Pour obtenir la valeur d'un attribut A (contenu dans une face) pour un noeud N et un groupe G, l'algorithme suivant est appliqué (voir figure 29):

```
Construire la liste F des faces applicables au noeud N dans le groupe G
par les mécanismes classiques d'héritage. C'est à dire: remonter l'arbre
des noeuds depuis N en ajoutant à F chaque nouvelle face rencontrée et ce
pour chaque groupe (suivant la hiérarchie des groupes depuis G).

Trier cette liste des faces suivant leur priorité en conservant l'ordre
initial pour les faces de même priorité.

Parcourir la liste F et retourner la première valeur trouvée pour A. (Cer-
taines faces pouvant ne pas contenir A.)
```

Figure 29 : Algorithme de recherche adapté aux faces.

Cet algorithme donne un résultat très intuitif. La face de plus haute priorité est appliquée. La hiérarchie liée à l'arbre de la formule est respectée pour les faces de même priorité. Les valeurs absentes d'une face sont retrouvées dans des faces moins prioritaires. Une face par défaut est placée à la racine de l'arbre des formules, dans le groupe racine de la hiérarchie des groupes. Une valeur est donc toujours retournée.

Sur l'exemple donné au début de cette section, il suffit donc d'indiquer que la face utilisée pour dénoter la sélection est plus prioritaire que celle de l'hyper-texte pour obtenir un arrière-plan correct, quel que soit l'ordre des deux faces dans l'arbre. De plus, la couleur du texte n'étant pas spécifiée dans ces deux faces, celle provenant de la face indiquant l'origine de la formule est employée.

Cet algorithme peut sembler coûteux. En pratique, il est facile de le rendre complètement incrémental. Le coût d'un accès à la valeur d'un attribut contenu dans une face est très nettement inférieur à celui d'un attribut classique[2]. La technique permettant ce gain de vitesse n'est malheureusement pas applicable pour tous les attributs car elle implique une consommation mémoire plus importante.

---

[1] Par analogie avec les 'faces' apparues dans les versions modernes d'Emacs.

[2] Sur un Pentium100 pour l'affichage d'une pleine page type Maple, on trouve en moyenne 24 000 accès à un attribut classique par seconde contre plus de 1 100 000 pour une face soit un rapport de 1 à 46.

8.4 Conclusion

La perte d'homogénéité dans la gestion des attributs entraînée par la spécialisation des faces, nous semble justifiée par le gain en expressivité et en efficacité qui en résulte. L'expérience a prouvé que, sans le système à base de priorité des faces, il était très difficile de mettre au point la palette de style d'une interface complexe. L'emploi des faces pour intégrer des fonds d'écran avec images, des animations graphiques et des icônes lors de l'affichage des formules permet de réaliser des interfaces riches et variées.

9 Conclusion

Dans le cadre d'un travail de recherche, il n'a pas été possible d'investir assez de temps pour assurer la portabilité d'*Emath* dans tous les environnements de développement. Nous nous sommes volontairement restreints au domaine des plate-formes *UNIX*. Le système de fenêtrage sur lequel est basé *Emath* est donc 'X Window system' [1]. Afin de disposer de plus de liberté possible pour expérimenter les solutions aux problèmes d'affichage, nous avons décidé de ne pas employer de bibliothèque de haut niveau pour le graphique. Comme discuté au paragraphe 2.2, page 75, cela n'interdit pas de les employer conjointement à *Emath*.

Pour un composant tel qu'*Emath*, la qualité de la typographie réalisée nous paraît être un point crucial. Nous avons dû investir beaucoup de temps en expérimentations dans ce domaine. Les résultats sont assez satisfaisants. L'affichage bidimensionnel respecte bien la typographie habituelle, tout en restant assez rapide pour être utilisé durant l'édition. La possibilité de construire dynamiquement de nouvelles méthodes d'affichage à partir de celles existantes est très utile. Il reste tout de même beaucoup de points à améliorer. Nous regrettons de ne pas avoir pu employer plus globalement les fontes dites outlines. Il pourrait être intéressant d'évaluer l'utilité des fontes avec des nuances de gris afin d'améliorer la définition subjective des images produites. La vitesse de l'affichage peut sûrement encore être augmentée.

Sur le plan purement technique, *Emath* emploie des solutions assez originales, telles que l'interprète embarqué, le multi-threading ou les attributs. Le résultat obtenu est assez convaincant. Il reste toutefois de nombreux points pour lesquels une étude plus approfondie permettrait d'améliorer les performances. Ces travaux d'optimisation ne s'insèrent pas facilement dans le cadre d'une thèse et nous avons préféré les laisser de coté pour nous consacrer finalement aux développements des fonctions de plus haut niveau. Nous

[1] 'X Window system' est le système de fenêtrage mis au point par le consortium X pour les plates-formes UNIX.

considérons toutefois que dans l'état actuel, les performances observées sont compatibles avec le développement d'applications réellement utilisables. Le serveur *Emath* permet de construire des interfaces dont les caractéristiques techniques sont suffisantes pour les rendre compétitives avec ce qui peut être proposé dans le commerce. Nous espérons donc que les solutions validées durant cette étude seront retenues lors de la conception d'autres interfaces.

Chapitre V
Applications et Validation

Comme nous l'avons expliqué dans les deux chapitres précédents, *Emath* est un composant d'édition de formules mathématiques destiné à être intégré dans diverses applications. Le prototype d'interface pour les systèmes de calcul formel que nous avons réalisé est un exemple d'une telle application. Cette interface, baptisée *MathBoard*, utilise Emath comme un serveur et lui délègue la gestion des fenêtres d'édition. Tous les comportements spécifiques de *MathBoard* sont programmés (et donc modifiables) en utilisant les fonctionnalités initiales d'Emath. *MathBoard* met en œuvre la plupart des principes et des idées exposés au chapitre I. Cependant, afin de valider le caractère intégrable et réutilisable d'Emath, nous avons réalisé une autre application basée sur ce composant, un éditeur de formules multi-format.

1 EdMa, un éditeur de formules

EdMa, est un Editeur Mathématique basé sur *Emath* qui permet d'éditer des expressions mathématiques quelconques et d'en générer le code correspondant dans différents formats. Il est également possible de charger une formule représentée dans un de ces formats dans *EdMa*. Les formats connus par *EdMa* sont actuellement *LaTex*, *OpenMath* (l'encodage ASCII), ainsi que la syntaxe d'entrée de *Maple* et de *Mathematica*. Une formule encodée dans un des ces formats peut être lue et chargée dans l'éditeur, modifiée interactivement, puis transformée à nouveau dans un des formats disponibles. *EdMa* bénéficie de toute la puissance d'édition et de manipulation d'Emath: édition incrémentale en deux dimensions, déplacements spatiaux, sélection multiple, etc. En outre, *EdMa* offre des fonctionnalités originales comme le copier-coller inter-application qui permet, par exemple, de sélectionner une formule dans la fenêtre d'édition puis d'en coller la forme *LaTeX* par un simple clic souris dans une fenêtre de l'éditeur *Emacs*. Ce genre de fonctionnalité n'est pas intégrée a priori dans *Emath* mais peut être ajoutée et définie par l'application.

2 *MathBoard, une interface pour le calcul formel*

Pour valider notre étude sur l'utilisation des systèmes de calcul formel, nous avons réalisé un prototype d'interface implémentant la plupart des principes imaginés durant notre étude sur les interfaces. Ce prototype, appelé *MathBoard*, est basé sur le composant Emath, et constitue une plate-forme d'expérimentation très souple pour l'édition et la manipulation de formules dans le cadre du calcul formel. Tous les comportements sont modifiables et de nouvelles fonctionnalités peuvent être ajoutées sans grande difficulté. L'interface possède plusieurs possibilités originales comme la sélection multiple ou la gestion de feuilles de calcul et peut accéder de façon concurrente à plusieurs systèmes de calcul formel. Les caractéristiques de *MathBoard* se répartissent en trois groupes:

2.1 *L'édition des formules:*

Les fonctionnalités d'édition de *MathBoard* démontrent la puissance et la souplesse du composant Emath. L'édition se caractérise par plusieurs points:

- **saisie incrémentale**: les expressions mathématiques sont saisies de manière incrémentale par l'utilisateur, ou bien sont produites par le système de calcul formel. Il n'y a donc aucune différence entre les "requêtes" soumises par l'utilisateur et les "résultats" produits par le système. L'utilisateur manipule uniformément des expressions mathématiques. Malgré l'existence de menus proposant des modèles d'expressions, la saisie peut s'effectuer entièrement au clavier, ce qui constitue un gain de temps très significatif pour l'utilisateur entraîné.

- **utilisation d'automate**: la saisie interactive s'effectue à l'aide de combinaisons de touches au clavier. La sémantique des touches est déterminée par un automate qui peut être modifié à volonté. Ceci permet de disposer de plusieurs modes de saisie, et de spécialiser un mode à un type ou une classe d'objets particuliers. Par exemple, il existe dans *MathBoard* un mode `polynôme` dans lequel la saisie du polynôme $3x^2y^3+5x^3+1$ s'effectue simplement à l'aide la séquence de caractères `3 x 2 y 3 + 5 x 3 + 1`. Il faut noter que cette suite de touches correspond exactement à la prononciation du polynôme et est donc très intuitive. Plusieurs modes sont disponibles par défaut, mais rien n'empêche l'utilisateur averti de définir ses propres modes de saisie. Cette paramètrisation très fine des fonctionnalités d'édition est rendue possible par l'utilisation d'un automate qui décrit le comportement de l'éditeur en fonction des évènements et de l'emplacement du curseur dans la formule en construction.

- **sélection**: il est possible de sélectionner des formules ou des sous-formules, de les déplacer, les couper, les copier et les coller. La sélection multiple permet de sélectionner plusieurs formules à l'aide de la souris et de les considérer ensuite comme arguments de certaines actions ou commandes. La sémantique de la sélection peut être programmée finement, et permet par exemple d'effectuer une substitution en sélectionnant l'expression à substituer, puis en cliquant sur la variable à substituer. La plupart des commandes standard que l'on trouve dans les systèmes de calcul formel généraux comme *Maple* ou *Mathematica* sont accessibles par la combinaison de la sélection d'une formule et d'un bouton dans un menu. Grâce à la sélection multiple, ce principe est étendu aux fonctions à plusieurs arguments.

- **déplacements**: les déplacements constituent une fonctionnalité importante et ne peuvent être remplacés par le pointage, qui est en général beaucoup plus lent. *MathBoard* propose plusieurs types de déplacements. À l'intérieur d'une formule, les déplacements syntaxiques permettent de parcourir la structure syntaxique d'une formule. Ce n'est pas le mode de déplacement le plus intuitif, mais c'est celui que l'on trouve dans la plupart des éditeurs de formules. *MathBoard* possède un autre type de déplacement, les déplacements spatiaux. Le déplacement spatial permet de parcourir la structure graphique d'une formule. C'est un type de déplacement très intuitif puisqu'il correspond exactement à ce que l'utilisateur voit sur l'écran. Enfin, *MathBoard* permet d'effectuer sur certaines formules des déplacements sémantiques, c'est à dire des déplacements qui tiennent compte de la sémantique de la formule. Si on déplace la variable b dans l'équation $a + b = 0$ du membre de gauche vers le membre de droite, la formule devient $a = -b$. Ce genre de comportement n'est pas intégré au cœur du composant d'édition, mais il est programmé dans l'application *MathBoard* qui exploite la grande paramétrabilité d'*Emath*.

2.2 L'organisation des calculs:

A l'inverse des systèmes de calcul formel qui ne présentent qu'une interface non structurée et linéaire, *MathBoard* possède plusieurs fonctionnalités permettant d'organiser les différentes étapes qui constituent un calcul:

- **feuilles de calcul**: il est possible de travailler simultanément sur plusieurs feuilles de calcul, c'est à dire dans plusieurs fenêtres. Un groupe de fenêtres est lié à une même session du système de calcul formel sous-jacent. Les effets de bord (affectation de variables) effectués dans une fenêtre ne sont visibles que dans les fenêtres du groupe correspondant.

- **étiquetage des formules**: l'utilisateur peut associer des informations (du texte commentant le calcul par exemple) à une formule qui n'apparaissent à l'écran que sur demande. Il possible d'effectuer une recherche sur ces informations pour retrouver une formule ou une expression éditée auparavant.

- **liens entre formules**: durant les différentes phases de la résolution d'un problème, les calculs intermédiaires sont liés par des relations du type "est le résultat de", "est la factorisation de", etc. *MathBoard* permet à l'utilisateur d'établir ces liens entre les objets manipulés dans l'interface. Il est fréquent, par exemple, d'effectuer certaines séquences de calcul plusieurs fois avec des paramètres différents. *MathBoard* permet de lier ces calculs entre eux, de propager les modifications et d'effectuer ainsi ces calculs avec de nouveaux paramètres.

- **calculs asynchrones**: *MathBoard* accède de manière asynchrone à un ou plusieurs systèmes de calcul formel. Durant un calcul, l'interface n'est pas bloquée et l'utilisateur peut continuer à éditer des formules ou même effectuer un autre calcul de façon concurrente. La connection entre *MathBoard* et les systèmes de calcul formel est réalisée par le *Central Control* [DGS94] qui assure la communication avec ces différents systèmes dans un format homogène. Il est intéressant de noter que la partie communication est complètement indépendante de la partie édition, et qu'elle peut être assurée par différents protocoles. À titre d'expérimentation, nous avons testé une version de *MathBoard* qui communique avec *Maple* et *Reduce* en utilisant le protocole *OpenMath*.

2.3 La présentation des résultats:

MathBoard propose, non seulement, l'affichage bidimensionnel classique des résultats, mais aussi quelques autres fonctionnalités intéressantes.

- **attente du résultat**: Comme il a déjà été mentionné, *MathBoard* ne bloque pas l'édition lors d'une requête. Le point où la réponse à une requête apparaîtra est donc marqué par une animation figurant le système de calcul formel sollicité. Dès son arrivée, la réponse est affichée.

- **plier-déplier**: Toute formule apparaissent dans *MathBoard* peut être repliée pour clarifier la page de calcul. Dans ce cas, un symbole marque l'existence d'une zone repliée. Un simple clic sur ce symbole restitue la formule initiale. Cette technique est très utile pour masquer les résultats trop volumineux. Un mécanisme similaire permet de replier une zone de formule, par exemple, pour faire disparaître un terme encombrant mais peu significatif.

- **la césure**: *MathBoard* prend le parti de ne pas tenter de césure sur les expressions complexes. Dans ce cas, les mécanismes habituels de déplacement (grâce aux scroll-bars) sont satisfaisants. Dans le cas particulier de grands produits ou sommes, une césure simple permet de les maintenir dans la page visible.

- **l'élision**: Toujours dans le but de clarifier l'affichage, il est possible d'appliquer des mécanismes d'élision sur une formule. Plusieurs algorithmes sont proposés et il est, bien sûr, possible d'en ajouter. Par exemple, pour un grand polynôme, il est possible de faire disparaître, en les remplaçant par un symbole spécial, tous les termes indépendants d'une variable.

L'affichage des résultats est un point particulièrement critique dans les interfaces classiques. *MathBoard* propose une esquisse de solution à certains problèmes et nous espérons pouvoir continuer nos expérimentations dans ce domaine très prometteur.

3 Validation

Même si le temps nous a manqué pour valider de façon satisfaisante les fonctionnalités de haut niveau de *MathBoard*, nous avons cherché à comparer les fonctions de base avec celles proposées par les interfaces plus conventionnelles.

3.1 Vitesse d'édition

La qualité globale de l'édition proposée par une interface a pour conséquence directe la vitesse d'édition. Pour comparer la qualité de l'édition dans différentes interfaces, il est donc possible de mesurer le temps de saisie d'un jeu de formule. Pour ce test, nous avons composé un ensemble de formules en consultant des ouvrages les plus variés possible et en privilégiant la diversité des formules. Il s'agit de formules réelles et non d'exemples artificiels qui auraient pu biaiser le test. Le jeu complet est donné en annexe.

MathBoard (et plus généralement *Emath*) peut être employé pour la saisie de formules à des fins d'édition de document ou pour interagir avec un SCF. Les interfaces comparées doivent donc couvrir ces deux aspects de l'édition. Les systèmes les plus employés dans ce contexte sont *Maple*, *Mathematica* et *LaTeX*.

Le test a été effectué avec la coopération de six personnes toutes relativement familières du calcul formel. Pour limiter le biais dû à leur connaissance de certaines interfaces, chaque test a été effectué trois fois. Ainsi, les tâtonnements dus à la découverte d'une nouvelle interface sont moins fréquents lors des deux derniers tests. Ce sont les seuls à intervenir dans les moyennes présentées ici.

- Le nombre d'erreur de saisie est une bonne mesure de l'intuitivité d'une interface. Plus les comportements sont complexes, plus le nombre d'erreurs augmente.

- Le nombre d'erreur syntaxiques (à différencire des erreurs de saisie) révèle la fiabilité du contrôle de l'interface. Il est impossible d'éditer une formule incorrecte en *MathBoard*. Il est par contre très fréquent d'oublier une parenthèse en mode linéaire.

- Le temps de saisie révèle l'efficacité de l'édition. Pour une interface à un SCF, il s'agit bien sûr de minimiser cette valeur afin d'accélérer le traitement des problèmes.

- Enfin le confort d'utilisation est un critère plus subjectif mais qui caractérise de manière plus globale l'ergonomie de l'édition. Une interface doit bien sûr être efficace, mais elle doit aussi rester simple d'emploi et ne pas forcer l'utilisateur à un effort de concentration trop soutenu. Les utilisateurs ont noté chaque interface sur une échelle variant de 1 à 4 pour pénible, inconfortable, normal et agréable.

Cette validation est actuellement en cours. Le nombre et la durée omportatnte (plus de vingt minutes par essai) des tests nécessaires à une validation statistiquement significative necessitent des expérimentateurs motivés et les candidats ne sont pas légion.

Les premiers résultats semblent toutefois démontrer que la saisie bidimensionnelle proposée par *Maple* ou *Mathématica* est inconfortable. Les utilisateurs notent mieux les modes linéaires pour ces deux interfaces. Les temps obtenus en mode linéaire sont comparables entre toutes les interfaces (LaTeX est un peu moins bon à cause du nombre plus important de caratères à frapper). La variabilité du nombre d'erreurs commises dans ces trois modes linéaires est essentiellement due à l'utilisateur. Certains connaissaient mieux les primitives de *Mathematica* que de *Maple* par exemple.

L'édition naturellement bidimensionnelle de *MathBoard* est bien notée. Le temps de saisie est le meilleur du comparatif. Les notes moyennes obtenues en `confort` et `nombre d'erreurs` par *MathBoard* sont probablement dues au manque d'entraînement des utilisateurs qui sont habitués depuis longtemps aux modes de saisie linéaire. Le même test effectué après plusieurs jours d'utilisation intensive devrait être plus favorable. Il est aussi envisagé de modifier quelques points du mode d'édition par défaut d'*Emath* pour répondre aux critiques émises lors des tests.

Toutes ces remarques ne sont pour le moment que le fruit de tests partiels et sont donc sujettes à modification.

3.2 Modes spéciaux d'édition

Emath permet une très grande souplesse dans la définition de l'édition. *MathBoard* exploite cette possibilité en proposant deux modes d'édition. Le premier est général et permet la saisie de toutes les formules du test sans difficulté. Le deuxième est plus spécifique aux polynômes et assure une plus grande efficacité dans ce cas particulier. Ce mode a été décrit au paragraphe 3.6, page 37.

Il est possible de calculer le nombre de touches à frapper pour la saisie d'un polynôme dans les différents systèmes précédemment testés. Si le polynôme est de la forme:

{[0-9]*[a-z]$^{[0\text{-}9]*}$ \+}*

par exemple:

$2x^9+4x^7+6x^5+8x^3$

Si le nombre de caractères des coefficients (resp. des exposants) est de C_c (resp. C_e) et si le nombre de termes est N_t , alors le nombre de touches à frapper si aucune erreur n'est commise est porté dans le tableau suivant:

Système	Nombre de touches	Sur l'exemple
MathBoard (poly)	$2N_t$-1+C_c+C_e	15
MathBoard (std)	$3N_t$-1+C_c+C_e	19
Mathematica	$3N_t$-1+C_c+C_e	19
Maple	$4N_t$-1+C_c+C_e	23
LaTeX	$3N_t$-1+C_c+C_e	19

Table 4: Frappe d'un polynôme

Cette formalisation n'a pas d'autre intérêt que de montrer que les modes spéciaux qui peuvent être construits grâce à *Emath* sont utiles. Dans le cas des polynômes, l'amélioration de la vitesse d'édition est significatif (25% par rapport au meilleur concurrent).

4 Conclusion

Au delà de l'expérimentation, *MathBoard* est un prototype utilisable et fonctionnel dont les performances sont comparables à celles des logiciels commerciaux. *MathBoard* offre un grand nombre de fonctionnalités nouvelles qui, selon nous, améliorent significativement l'ergonomie habituelle des interfaces pour le calcul formel. Enfin, l'utilisation de protocoles de communication conçus pour l'échange d'expressions mathématiques permet à *MathBoard* d'accéder de manière unifiée à plusieurs systèmes (à ce jour, *Maple*, *Reduce* et *Mathematica*).

Conclusion

Nous avons présenté dans ce mémoire: la conception, la réalisation et l'utilisation d'*Emath*, un composant logiciel pour l'édition et la manipulation de formules mathématiques. Intégrable dans une application dans un style client-serveur, *Emath* est au cœur de *MathBoard*, un prototype d'interface pour les systèmes de calcul formel. Influencé par une étude préalable des interfaces existantes et de l'activité de l'utilisateur du calcul formel, *MathBoard* offre des fonctionnalités assez variées dans un environnement d'édition confortable. Aujourd'hui, le champ d'application du calcul formel est de plus en plus large et l'interface devient un composant crucial dans un système de calcul formel. Les utilisateurs de tels systèmes sont en nombre croissant et appartiennent à des milieux très variés. L'apprentissage de logiciels tels que *Maple* ou *Mathematica* est au programme de toute formation d'ingénieur et la formation au calcul formel commence souvent dès les premières années d'enseignement supérieur. Paradoxalement, les interfaces n'ont que peu évolué et restent bien en dessous des normes définies dans le domaine de l'interface homme-machine. Ce déficit de l'interface en terme d'ergonomie peut considérablement ralentir l'utilisateur expérimenté et bien souvent rebuter le débutant. Ce constat, qui a motivé nos travaux, est encore vrai à l'heure de leur conclusion. Les interfaces du commerce ont certes progressé, mais elles sont, de notre point de vue, encore largement insuffisantes. Le composant *Emath* et les quelques interfaces expérimentales qui ont servi à sa validation peuvent donner un avant-goût de ce que devraient être les logiciels pour la manipulation de formules mathématiques dans les années à venir. Nous souhaitons, bien sûr, que les composants logiciels qui résultent de nos travaux aient une vie après la thèse.

Le serveur *Emath* a, en tout cas, bénéficié de tous nos soins pour offrir les garanties de portabilité et d'évolutivité nécessaires à des applications autres qu'expérimentales.

Le développement d'une plate-forme d'expérimentation pour nos idées en matière de manipulation de formules mathématiques s'est transformé au cours des mois en une mise au point d'un véritable composant réutilisable par tout logiciel souhaitant offrir une édition ou un affichage soigné de formules. Cette évolution a principalement été motivée par l'absence de tels composants. De nouvelles expérimentations devraient avoir lieu notamment dans le domaine des actions sémantiquement riches, comme les manipulations à base de sélections multiples ou les déplacements sémantiques. Pour faciliter de telles expériences, il nous a semblé judicieux de fournir aux futurs intervenants la base logicielle qui nous a tant fait défaut. *Emath* a été conçu en gardant cet objectif présent a l'esprit. Certaines de ses caractéristiques en sont le reflet immédiat.
Par exemple, l'interprète embarqué est l'un des choix architecturaux les plus novateurs dans ce domaine. Avec *Klone,* il est possible de fournir au noyau du serveur la flexibilité requise pour de nouvelles expérimentations. L'ensemble des fonctions d'affichage et de manipulation des arbres de formule est facilement accessible pour adapter les comportements standard aux besoins d'un utilisateur imaginatif.
Toujours dans l'optique d'une adaptation au plus grand nombre d'application, le moteur d'*Emath* bénéficie d'un système d'attributs qui contrôle la quasi totalité des paramètres de formatage ou d'affichage. Ces attributs et le système d'héritage qui leur est associé ont prouvé leur intérêt. Ils servent bien sûr à modifier le style de l'affichage mais aussi à stocker des éléments utiles aux applications clientes d'*Emath*. Grâce à quelques optimisations, cette très grande souplesse du moteur d'affichage a pu rester compatible avec les contraintes de vitesse imposées par une bonne réactivité.
Un des points fort d'*Emath* est de proposer une édition qui reste réactive quelles que soient les conditions d'édition. L'architecture multi-flot du noyau d'*Emath* assure, en effet, que les évènements en provenance de l'utilisateur sont pris en compte dans un délai constant. Ainsi, les sensations désagréables de retard à la frappe sont évitées. Même si ce point peut sembler anecdotique dans le cadre des interfaces classiques, il est plus sensible dans le contexte des interfaces pour l'édition de formules mathématiques. Dans ce cas, une saisie bidimensionnelle requiert des temps de calcul plus conséquents et les délais d'affichage s'en trouvent allongés. Nous espérons avoir fait la preuve que cette approche originale du traitement des évènements est intéressante. Il faut noter que la tendance actuelle semble nous donner raison car les systèmes basés sur les boîtes à outils modernes le proposent en général. Par exemple, avec *Java*, la gestion des évènements relatifs à l'affichage est intrinsèquement multi-flot.
Toujours sur un plan technique, nous avons tenté des améliorations dans le domaine de l'affichage des caractères. Les résultats sont moins satisfaisants et nous attendons beaucoup de l'apparition de nouveaux outils dans ce domaine. Il reste difficile de générer des

caractères de taille variable de qualité convaincante en environnement *X*.
Enfin, il reste à évoquer le mécanisme de gestion des événements. Nous avons fait le choix d'un automate à état fini, en espérant que ce modèle offre un cadre formel suffisamment clair. Après avoir écrit de nombreux automates adaptés à plusieurs types d'édition, il semble que ce choix soit justifié. Nous n'avons pas rencontré de cas où la structure d'automate se soit révélée trop rigide pour pouvoir être adaptée à une fonctionnalité d'édition. Tous les mécanismes de gestion d'évènement proposés par *Emath* sont réalisés grâce à un automate. Le choix le plus critiquable dans ce contexte est l'emploi de la syntaxe *Xt* pour la description des motifs d'évènements de l'automate. Il semble que cette syntaxe rebute la plupart des utilisateurs. La compatibilité avec les mécanismes classiques de gestion des événements *X* ne mérite peut-être pas ce sacrifice.

Si sur le plan technique ces différents points nous semblent acquis, il est plus difficile de se prononcer sur les choix ergonomiques mis en œuvre dans l'interface *MathBoard*. Là aussi, nous espérons avoir proposé des solutions intéressantes, bien que certains choix subjectifs n'aient pas encore été complètement validés dans la pratique. Le serveur *Emath* permet a priori un large éventail de comportements. Les manipulations les plus élaborées proposées par le client *MathBoard* ne sont bien sûr qu'un choix parmi d'autres.
La sélection multiple semble être un outil appréciable pour des opérations complexes. Le glisser déposer est une autre approche intéressante mais nous l'avons moins approfondie. Il reste de nombreuse voies à explorer avant de prétendre conclure sur le thème des manipulations de haut niveau destinées aux systèmes de calcul formel. *MathBoard* propose un éventail respectable de ces facilités et nous prévoyons de poursuivre nos investigations dans ce domaine.
Il est très agréable de pouvoir éditer une même formule au travers de plusieurs fenêtres ou dans des styles différents. Cette fonctionnalité est maintenant devenue assez répandue dans les systèmes d'édition de documents mais reste inconnue des interfaces pour le calcul formel. *Mathboard* propose par défaut ce type d'édition multi-vue et multi-style.
Le noyau d'*Emath* offre aussi les fonctionnalités de déplacement spatial. De notre point de vue (et de celui des utilisateurs qui ont pu le tester), le déplacement syntaxique est infiniment moins intuitif que le déplacement spatial. Nous ne comprenons pas l'absence de cette fonctionnalité dans les interfaces modernes. Associée à une édition intuitive, cette facilité donne une autre dimension à l'interface. Elle se rapproche de la convivialité d'une feuille de papier et permet enfin d'associer la puissance des systèmes de calcul formel à la simplicité du travail sur papier.

Il est évident que des progrès considérables doivent être effectués pour rendre les interfaces pour la manipulation de formules mathématiques plus facilement utilisables. Le travail effectué, avec pour objectif la production d'une interface aux systèmes de calcul formel, a vu son champ d'application s'élargir à tous les types d'interfaces proposant un affichage ou une édition de formules. Ainsi, les résultats de nos travaux peuvent s'appliquer aux éditeurs de documents mathématiques ou aux *note-books* qui proposent des outils d'accès aux bases de connaissances mathématiques plus conviviaux. Si le temps le permet, il sera intéressant de poursuivre la validation d'*Emath* en intégrant le serveur à des applications de ce type.

Annexe A: Principales Interfaces

Figure 30 : Scientific Workplace (de TCI)

Figure 31 : Maple V.4

Figure 32 : Mathematica 3.0

Figure 33 : CasPi, un éditeur expérimental.

Figure 34 : Une interface expérimentale basée sur Emath.

Références bibliographiques

[AAF82] M.J.ATKINSON, F.W.ARCUN andJ.C.Friedly
The Use of a Computer Algebra System in the Analysis of Chemical Engineering *Problems*
Computers and Chemical Engineering, Vol. 6, No. 2, pp. 169-175, 1

[AAF92] ARNON, I. ATTALI and P. FRANCHI-ZANNETTACCI
Language-Based Document Processing,
Rapport de recherche 1731, INRIA, Juillet 1992

[AB89] J. ANDRÉ and B. BORGHI
Dynamic Fonts,
In Raster Imaging and Digital Typograpy, Edited by J. André and R. D. Hersch, The Cambridge Series on Electronic Publishing, October 1989

[ABIW88] Dennis ARNON, Richard BEACH, Kevin MC ISAAC, and Carl WALDSPURGER
Carminoreal: an interactive mathematical notebook,
In EP'88 International Conference on Electronic Publishing, Document Manipulation and Typography, Cambridge University Press, Nice, France, April 1988

[ADG97] O. ARSAC, S. DALMAS et M. GAËTANO
The Design of a customizable Component to Display and Edit Mathematical Formulas,
To be Published In Journal of Symbolic Computation, vol. 11, 1997

[AG96] Ken ARNOLD and James GOSLING
The Java Programming Language,
The Java Series, ISBN 0-201-63455-4, Addison-Wesley, 1996

[AH89] Jacques ANDRÉ and Roger D. HERSCH
Raster Imaging and Digital Typography,
Proceedings of the International Conference Ecole Polytechnique de Lausanne, Edited by Jacques André and Roger D. Hersch, The Cambridge Series on Electronic Publishing, October 1989

[AM91] Jacques ANDRÉ and Robert A. MORRIS
Raster Imaging and Digital Typography II,
Proceedings of the International Conference Ecole Polytechnique de Lausanne, Edited by Robert A. Morris and Jacques André, The Cambridge Series on Electronic Publishing, 1991

[AP92] J. ANDRÉ and M. PALIX
Font Rasterization: The state of the Art,
Support de cours (CEE programme COMETT II) Didot, Edited by Roger D. Hersch, Juillet 1992

[AS72] Milton ABRAMOWITZ and Irene A. STEGUN
Handbook of mathematical functions,
Edited by Abramowitz and Stegun, Dover Publications Inc., New-York, 1972

[AS90] P.J. ASENTE and R. SWICK
X Window System Toolkit,
Digital Press X and Motif Series, 1990

[AV93] J. ANDRE and I. VATTON
Contextual Typesetting of mathematical Symbols Taking Care of Optical Scaling,
Publication interne no: 747, IRISA, June 1993

[Avit90] Ron AVITZUR
Suggestions for a friendler user interface,
In Proc. of the DISCO'90, LNCS 429, Springer-Verlag, pp. 282--283, 1990

[Bach86] M. BACH
The Design of the UNIX Operating System,
In Software, Prentice-Hall, 1986

[BCDI88] P. BORRAS, D. CLEMENT, T. DESPEYROUX, J. INCERPI, G. KAHN and B. LANG
CENTAUR: the system,
ACM SIGSOFT'88, (SDE3), Boston MA, November 1988

[BDC92] Gabe BEGED-DOV and Ellis S. COHEN
Implementing Drag & Drop for X11,
The X Resource, O'Reilly & Associates Inc., pp. 165--190, 1992

[Betz88] David Michael BETZ
XLISP: An Object-Oriented Lisp (Version 2.1),
P.O. Box 144, Peterborough, NH 03458, 1988

[BG89] P. BURGER and D. GILLIES
Interactive Computer Graphics,
Addison Wesley Publishing Company, 1989

[Boehm93] H. BOEHM
Space Efficient Conservative Garbage Collection,
Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices 28, pp. 197--2066, June 1993

[Bona87] A. BONADIO
Expressionist Reference Manual,
Prescience Corporation, 814 Casto St., San Francisco, CA 94114, 1987

[Bosh93] Peter BOSH
Inheritance vs. delegation: Is one better than the other ?,
*Unpushished, In the Python software documentation,
ftp://ftp.ibp.fr/pub/python/doc/delegation.ps.gz, 1993*

[Boyd93] J.P.BOYD
Chebyshev and Legendre Spectral Methods in Algebraic Manipulation Languages
Journal of Symbolic Computation,16:377-399, 1993

[Brya88] M. BRYAN
SGML an author's guide to the standard generalized markup language,
Addison Wesley Publishing Company, 1988

[BS95] Anselm BAIRD-SMITH
Interprête embarquable: Expérimentations,
Journées françaises des Langages Applicatifs, Janvier 1995

[BW89] A. BONADIO and E. WARREN
Theorist Reference Manual,
Prescience Corporation, 814 Casto St., San Francisco, CA 94114, 1989

[Card87] S. K. CARD
Human Limits and VDT Computer Interface,
In Readings in Human-Computer Interaction, a multidisciplinary approach, R. M. Baecker, W. A. S. Buxton, Morgan Kaufman Publishers, ISBN 0-934613-24-9, 1987

[CD88] Eric C. COOPER and Richard P. DRAVES
C THREADS,
Internal Report CMU-CS-88-154, DoD, June 1988

[CGGM88] B. W. CHAR, K. O. GEDDES, G. H. GONNET, M. B. MONAGAN and S. M. WATT
Maple Reference Manual, 5th edition,
Watcom Publications Limited, Waterloo, Ontario, Canada, 1988

[CMN83] S. K. CARD, T. P. MORAN and A. NEWELL
The psychology of Human-Computer Interaction,
Lawrence Erlbaum Associates, 1983

[CJ90] R.M.CORLESS and D.J. JEFFREY
Solution of a Hydrodynamic Lubrication Problem with Maple
Journal of Symbolic Computation,9:503-513, 1990

[CMN87] S. K. CARD, T. P. MORAN and A. NEWELL
The KeystrokeLevel Model for user performance time with interactive systems,
In Readings in Human-Computer Interaction, a multidisciplinary approach, R. M. Baecker, W. A. S. Buxton, Morgan Kaufman Publishers, ISBN 0-934613-24-9, 1987

[Conc89] J. CONCZAROWSKI
Fast generation of unfilled and filled outline characters,
In Raster Imaging and Digital Typograpy, J. Andre and R. D. Hersch, The Cambridge Series on Electronic Publishing, October 1989

[Coutaz90] Joëlle COUTAZ
Interface Homme-Ordinateur: Conception et Réalisation,
Dunod informatique, 1990

[CS86] J. R. COOKE and E. T. SOBEL
MathWriter,
Cooke edition, Ithaca, NY, 1986

[CZ93] F.M.CLARCK and X.ZHAO
Analytica, a theorem prover for Mathematica
The Mathematica Journal, 3(1):56-71, 1993

[DEC93] DIGITAL EQUIPMENT CORPORATION
DECthreads (Guide to DECthreads),
Technical Report: AA-PV6NA-TK, Maynard, Massachusetts, March 1993

[DGS94] Stéphane DALMAS, Marc GAËTANO and Alain SAUSSE
Distributed computer algebra: the Central Control approach,
PASCO'94, World Scientific, pp. 104--113, Linz, Austria, September 1994

[DR86] J. H. DAVENPORT and C. E. ROTH
PowerMath: A System for the Machintosh,
In Proc. of ACM SIGSAM, Symposium on Symbolic and Algebraic Computation, pp. 13--23, 1986

[DS96] DESIGN SCIENCE Inc.
MathType,
4028 Broadway, Long Beach, CA 90803, 1996
http://www.mathtype.com/

[Dybv87] Kent R. DYBVIG
The SCHEME programming language,
Prentice-Hall, ISBN 0-13-791864, 1987

[EHV89] S. Van EGMOND, F. C. HEEMAN and J. C. VAN VLIET
INFORM: an interactive syntax-based formulae editor,
In Journal of Systems and Software, 9: pp. 169--182, 1989

[EW94] Ian ELLIOT and David P. WIGGINS
Double Buffer Extension Specification, Protocol Version 1.0,
Draft Revision 1.8, X Consortium Public Draft, 1994

[Fahl89] O. FAHLANDER
A spline contour method with efficient filling,
In Raster Imaging and Digital Typograpy, J. Andre and R. D. Hersch, The Cambridge Series on Electronic Publishing, October 1989

[FDFH93] J. D. FOLEY, A. VAN DAM, S. K. FEINER, J. F. HUGUES and R. L. PHILIPPS
Introduction to computer graphics,
Addison Wesley Publishing Company, 1993

[Franc89] P. FRANCHI-ZANNETTACCI
Attribute Specifications for Graphical Interface Generation,
In IFIP 11th World Computer Congress, pp. 149--155, San Francisco, USA, 1989

[FTC88] FRAME Technology Corporation
Framemaker demonstration document,
Frame Technology Corporation Inc., 1988

[FUCH95] Matthew FUSHS
Escaping the event loop: an alternative control structure for multi-threaded GUIs,
Proceedings of the EHCI'95, Chapman-Hall, London, 1995
ftp://ftp.cerc.wvu.edu/pub/fuchs/gui.ps.z

[FZCL88] P. FRANCHI-ZANNETTACCI, B. CHABRIER and V. LEXTRAIT
GIGAS: a graphical interface generator for attribute specifications,
In Actes du Colloque " Le Génie Logiciel et ses Applications", Toulouse, Décembre 1988

[Gal96] Mark GALASSI
Guile User Manual
Los Alamos National Laboratory and Cygnus Support, 1996
http://nis-www.lanl.gov/~rosalia/mydocs/guile-user_toc.html

[GJ71] J. GRIESMER and R. JENKS
SCRATCHPAD/1- An Interactive Facility for Symbolic Mathematics,
In Proc. of the 2nd Symposium on Symbolic and Algebraic Manipulation, ACM, pp. 42--587, March 1971

[Glic91] Bob GLICKSTEIN
ELI, the Embedded Lisp Interpreter, 1991
Andrew Distribution, bob@andrew.cmu.edu

[GQ86] A.D.GARRAD and D.C.QUARTON
Symbolic Computing as a tool in Wind Turbine Dynamics
Journal of Sound and Vibration, 109(1), pp. 65-78, 1986

[GS91] Simon GARFINKEL and Gene SPAFFORD
Practical UNIX Security,
O'Reilly & Associates, 1991

[Hasc91] Laurent HASCOËT
FIGUE: An Incremental Graphic Formatter, User's Manual for version 1.,
INRIA, Sophia-Antipolis, Août 1991

[Hearn85] A.C. HEARN
Reduce User's Manual,
Rand publication CP78 edition, 1985.

[Hers93] Roger D. HERSH
Visual and Technical aspects of type,
Roger D. Hersch editor, Cambridge University Press, 1993

[HH89] Rex H. HARTSON and Deborah HIX
Human-Computer Interface Development: Concepts and Systems for its Management,
ACM computing Surveys, vol. 21, no 1, March 1989

[JG86] Fahimeh JALILI and Jean H. GALLIER
Building friendly parsers,
In Proc. 9th ACM Symposium on Principles of Programming Langages, pp. 196--206, ACM, 1986

[JRG92] I. S. JACOBS and L. RIDEAU-GALLOT
A Centaur tutorial,
Rapport Technique no: 140, Programme 2, INRIA, Juillet 1992

[JS92] R. D. JENKS and R. SUTOR
AXIOM, The Scientific Computation System,
Springer-Verlag, 1992

[Kajl90] N. KAJLER
Building Graphic User Interfaces for Computer Algebra Systems,
In Proc. of DISCO'90, LNCS 429, Springer-Verlag, pp. 235--244, Capri, Italy, April 1990

[Kajl90b] N. KAJLER
Éléments de spécification pour l'interface utilisateur d'un système de calcul formel,
Rapport Technique no: 1220, Programme 1, INRIA, Mai 1990

[Kajl92] N. KAJLER
CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems,
In Proc. of ISSAC'92, ACM Press, pp. 376--386, Berkeley, USA, July 1992

[Kajl93] N. KAJLER
Environnement graphique distribué pour le calcul formel,
Thèse, UNSA, 2 mars 1993

[Kapl92] Philippe KAPLAN
Sémantique des fenêtres, dans le système de fenêtrage X, pour les gestionnaires d'espace de travail,
Thèse de doctorat, UNSA, Décembre 1992

[Karo87] P. KAROW
Digital Formats for Typefaces,
URW Verlag, 1987

[KEH94] David KEPPEL, Susan J. EGGERS and Robert R. HENRY
A case for runtime code generation,
Technical Report UWCSE 91-11-04, University of Washington, 1991

[Kepp93] David KEPPEL
Tools and Techniques for Building Fast Portable Threads Packages,
Technical Report: UWCSE 93-05-06, University of Washington,1993

[Knut86] Donald E. KNUTH
Computers & Typesetting: Computer Modern Typefaces,
Addison Wesley Publishing Company, 1986

[Knut84] Donald E. KNUTH
The Tex Book,
Addison Wesley, 1984

[Koto86] W. P. KOTORYNSKI
Steady laminar flow through a twisted pipe of elliptical cross-section
Computers & Fluids, Vol. 14, No. 4, pp. 433-444, 1986

[KP81] D. E. KNUTH and M. F. PASS
Breaking Paragraph into Lines,
Software Practice and Experience, 11 (11) : pp. 1119--1184, 1981

[KS94] N. KAJLER and N. SOIFFER
Some human interaction issues in computer algebra,
ACM SIGSAM Bulletin, (28):pp. 18--28, Also in ACM SIGGHI Bulletin 28 (1), March 1994

[KS89] E. KANTOROWITZ and O. SUDARSKY
The Adaptable User Interface,
Communications of the ACM, vol. 32, number 11, November 1989

[Lamp93] Leslie LAMPORT
How to Write a Long Formula,
Digital System Research, 130 Lytton Av., Palo Alto, California 94301, December 25, 1993

[Lang86] B. LANG
The Virtual Tree Processor In Generation of Interactive Programming Environments,
CWI Report CS-R8620, Amsterdam, May 1986

[LB94] Olivier LAUMANN and Carsten BORMANN
Elk: The Extension Language Kit,
In Proceedings of USENIX Computing Systems, vol.7, no4, pp.419--449,1994
http://www.informatik.uni-bremen.de/~net/elk/doc/usenix.ps

[Léon95] Jean-Michel LÉON
End-User Customization of Xt Applications, 1995
Unpublished Paper: http://www.inria.fr/koala/jml/xres/xres.html

[Léon95b] Jean-Michel LÉON
Klm: A Simple Scripting Interface to Motif,
In The X Advisor, Vol. 1 No 2, July 1995

[Lemk91] Dave LEMKE
The X Font Server,
In Proc. of the eXug, Cambridge, Autumn 1991

[Lint92] Mark A. LINTON
Implementing Resolution Independance on top of the X Window System,
In Proc. of the 6th annual X Technical Conference, Boston, Massachussets, 1992

[LL90] Bil LEWIS and Dan LALIBERTE
GNU Emacs Lisp Reference Manual, version 18,
GNU, Free Software Foundation,1990

[MA88] Joël MCCORMACK and Paul ASENTE
Using the X Toolkit or How to Write a Widget,
In USENIX Conference Proceedings, Summer 1988

[Mack94] J.W.Macki (editor)
Classroom Notes
SIAM Review, Vol. 36, No. 4, pp. 642-648, December 1994

[Mart71] William A. MARTIN
Computer input/output of mathematical expressions,
In Second Symposium on symbolic and algebric manipulation, pp. 78--89, ACM, 1971

[MAS91] Joël MCCORMACK, Paul ASENTE and Ralph R. SWICK
X Toolkit Intrinsics C Langage Interface, 1991

[MG83] MATHLAB GROUP
Macsyma Reference Manual, version 10,
MIT edition, 1983

[MI89] MATHSOFT Inc.
MATHSTATION, Version 1.0,
201 Broadway, Cambridge, MA, 1989

[MI93] MATHSOFT Inc.,
MathCad 6.0
101 Main Street,Cambridge, MA 02142-1521, 1993
http://www.mathsoft.com

[Muel93] F. MUELLER
A Library Implementation of POSIX Threads under UNIX,
In Winter USENIX, San Diego, CA, January 25-29, 1993

[MS93] Kevin MULLET and Darrell SANO
Applying Visual Design : Trade secrets for elegant interfaces (or, How to look good in pixels),
Tutorial notes, X Technical Conference, Boston, January 16, 1993

[Naha88] C. NAHABOO
D'un paragrapheur à un éditeur syntaxique et graphique pour le langage ESTEREL,
Thèse, UNSA, 23 Novembre 1988

[Naha96] C. NAHABOO
KLONE Reference Manual,
Documentation provided with the Klone free Software package, 1996
ftp://www.inria.fr/pub/Klone/klone-ref.ps.gz

[NN89] M. NANARD and J. NANARD
An anti-aliasing method for low resolution fonts based on font structure,
In Raster Imaging and Digital Typograpy, J. Andre and R. D. Hersch, The Cambridge Series on Electronic Publishing, October 1989

[Newm95] Jan NEWMARCH
Embedding of X Applications,
In Proceedings of the 9th Annual X Technical Conference, X Resource issue 13, 1995

[NU88] A. F. NIKIFOROV and V. B. UVAROV
Special Functions of Mathematical Physics
Birkhauser, 1988

[Ogil82] J. F. OGILVIE
Applications of Computer Algebra in Physical Chemistry
Computers and Chemistry, Vol. 6, No. 4, pp. 169-172, 1982

[OSF95] OPEN SOFTWARE Foundation
OSF/MOTIF, Programmer's Guide, Release 2.0,
Prentice Hall, Inc., 1995

[Oust90] John K. OUSTERHOUT
Tcl: An Embeddable Command Language,
Winter USENIX Conference Proceedings, 1990

[Oust94] John K. OUSTERHOUT
Tcl and the Tk Toolkit
Addison-Wesley Publishing, ISBN: 0-201-63337-X ,1994

[Paepk93] Andreas PAEPCKE
Object-oriented programming: the CLOS perspective
MIT press, 1993

[PI88] PARACOMP Inc.
Milo User's Guide,
123 Towsend St., Suite 310, San Francisco, CA, 94107, 1988

[PKBS91] M. L. POWEL, S. R. KLEIMAN, S. BARTON, D. SHAH, D. STEIN and M. WEEKS
SunOS Multi-thread Architecture,
In Proc. of the USENIX Winter'91 Conference, Dallas, Texas, 1991

[Quin89] V. QUINT
Structured documents, chapter Systems for the representation of structured documents,
The Cambridge Series on Electronic Publishing, Cambridge University Press, pp. 39--73, Cambridge, 1989

[QVAR91] V. QUINT, I. VATTON, J. ANDRÉ and H. RICHY
Grif et l'édition de documents structurés: nouveaux développements,
Cahiers GUTemberg, 9:49-65, Juillet 1991

[RDBE87] M. L. REHAK, F. L.DIMAGGIO, H. BENAROYA and I. ELISHAKOFF
Random Vibrations with Macsyma
Computer Methods in Applied Mechanics and Engineering 61, pp. 61-70, 1987

[Rime95] Ken RIMEY
Template-based Formula Editing in Kaava,
Department of Computer Science, Helsinki University of Technology, 1995
rimey@cs.hut.fi

[Ross93] Guido van ROSSUM
Python Reference Manual,
Tech. Report CS-R9525, CWI, Centrum Voor Wiskunde En Informatica, Amsterdam,1993
ftp://ftp.cwi.nl/pub/CWIreport/AA

[Schwa95] Larry and Randal SCHWARTZ
Programming Perl
O'Reilly & Associates, ISBN: 0-937175-64-1, 1995
ftp://ftp.ora.com in /pub/examples/nutshell/programming_perl/perl.tar.Z

[Segd83] R. SEGDEWICK
Algorithms,
Addison-Wesley Publishing Company, 1983

[SG90] R. W. SCHEIFFER and J. GETTYS
X Window System (The complete reference to), 2nd Edition,
Digital Press X and Motif Series, 1990

[SH95] Bo STIG HANSEN
Towards a Theory of Incremental Document Formatting,
Draft, Department of Computer Science, Building 344, Technical University of Denmark,, DK-2800 Lyngby, 1995

[Smar89] Gene SMARTE
Voice Recognition or Music Synthesis, sounds processing has joined the rank of microcomputer,
BYTE, pp. 245--248, December 1989

[Soif91] Neil SOIFFER
The Design of a User Interface for Computer Algebra Sytems,
PhD thesis, Report UCB/CSD/ 91/626, University of California, Berkeley, CA, April 1991

[Spir90] E. SPIR
Gestion dynamique de la mémoire dans les langages de programmation,
Science Informatique, InterEditions, ISSN_ISBN: 2 7296 0329 8, 1990

[SS86] Carolyn J. SMITH and Neil M. SOIFFER
Mathscribe: A user interface for computer algebra systems,
In Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation, pp. 7--13, July 1986

[Stal81] Richard M. STALLMAN
EMACS- The Extensible, Customizable, Self-documenting Display Editor Production System,
SIGPLAN Notices, vol. 16, Association for Computing Machinery, New-York, 1981

[Stee90] Guy L. jr. STEEL
COMMON LISP, The Language, Second Edition,
Digital Press, ISBN 1-55558-041-6, 1990

[Stou76] D. R. STOUTEMEYER
Qualitative Analysis of Mathematical Expressions
Using Computer Symbolic Mathematics
Proc. of ACM Symposium on Symbolic and Algorithmic Computation, 1976

[STW92] C. SCHMIDTMANN, M. TAO and S. WATT
Design and Implementation of a Multi-Threaded Xlib,
In Proceedings of the XHibition 1992, pp. 65--74, San José, California, June 15-19, 1992.

[SWI90] SOFT WAREHOUSE Inc.
The Derive User Manual: Mathematical Assistant for Personal Computer, Version 2,
Honolulu, Hawaï, 1990

[TCI96] TCI Software Research
Scientific WorkPlace,
1190 Foster Road, Las Cruces, NM 88001, 1996
http://www.hearne.com.au/catalog/sciword.html

[Tekt88] TEKTRONIC Inc.
MATHSCRIBE User's Manual Version 1.0, 1988

[Trav91] D. TRAVIS
Effective Color Display, Theory and Practice,
Academic Press, ISBN 0-12-697690-2, 1991

[Wass90] A. WASSERMAN
Tool integration in software engineering environment,
Software Engineering Environments, LNCS 467, 1990

[Wolf96] S. WOLFRAM
The MATHEMATICA, Book 3rd edition,
Wolfram media, Cambridge University Press, 1996

[Wyan81] J. C. WYANT
Use of a symbolic math system to solve polarized light problems
Applied Optics, Vol. 20, No. 19, pp.3321-3326, October 1981

[YW87] Douglas A. YOUNG and Paul S. WANG
GI/S: A Graphical User Interface for Symbolic Computation Systems,
In Journal of Symbolic Computation, Academic Press, 4:pp. 365--380, 1987

Interfaces homme machine pour le calcul formel

Le sujet de ce mémoire est la définition et la réalisation d'une interface pour les systèmes de calcul formel. Jusqu'à présent, la majeure partie des efforts déployés dans le domaine du calcul formel concernait les algorithmes et leur implémentation, l'interface restant la plupart du temps minimale. Nous tentons donc de définir les caractéristiques d'une interface véritablement adaptée à la manipulation d'expressions mathématiques. Pour pouvoir tester ces idées, nous avons eu besoin d'une plate-forme d'expérimentation. Nous avons développé un composant réutilisable et fortement paramétrable pour l'édition de formules mathématiques. Ce composant, baptisé Emath, se comporte comme un processus serveur et possède une architecture assez sophistiquée. La gestion des évènements est définie par un automate pour permettre une bonne expressivité. Le noyau d'affichage est multi-flot pour une réactivité optimale. Un interprète Lisp embarqué assure une paramétrabilité maximum. Pour valider Emath, nous avons réalisé un prototype d'interface à l'ergonomie plus recherchée que les standards du moment. En nous appuyant sur des protocoles de communication d'objets mathématiques comme OpenMath, nous pouvons utiliser cette interface avec des systèmes de calcul formel comme Maple ou Mathematica.

Mots-clés : interfaces homme machine, calcul formel, éditeur de formules.

User interfaces for computer algebra systems.

This thesis describes the design and implementation of a new interface for Computer Algebra Systems (CAS). There is a clear need for a good front-end for CAS. Developers are used to invest mainly in efficiency of the CAS kernel, thus leading to relatively poor interfaces. Our aim was to design a new kind of interface, dedicated to mathematical expression manipulation. To experiment some new ideas, we developed a highly customizable formula editor. This component, called Emath, has a sophisticated architecture. Events are handled by an automaton to obtain good versatility. To insure a good responsiveness, Emath is multi-threaded and runs as a separate process (using Client/Server model). An embedded Lisp interpreter allows easy customization. To validate Emath, we built a prototype interface with sophisticated ergonomy. Thanks to formula exchange protocols like OpenMath, this interface may be used as a front end for major CAS (like Maple or Mathematica).

Keywords : human computer interface, computer algebra system, formula editor.