SYNTHÈSE

# Programmation applicative
# État des lieux et perspectives


## Christian Queinnec
## Pierre Weis

*Christian.Queinnec@polytechnique.fr, École Polytechnique, 91128 Palaiseau,*
*Pierre.Weis@inria.fr, INRIA, BP 105, 78153 Le Chesnay*



*RÉSUMÉ. Après avoir rappelé la lignée des langages de programmation applicative, nous décrivons les qualités qui font leur succès. Nous brossons ensuite leurs perspectives d'avenir et donnons en annexe une liste des systèmes francophones disponibles.*


*MOTS-CLÉS : langages applicatifs, Caml, Scheme.*



*ABSTRACT. We recall the story of applicative programming languages. We then describe the qualities that make the success of these languages, in particular for educational purpose. We sketch some prospective trends. Some available free implementations are mentioned in appendix.*


*KEY WORDS : functional language, Caml, Scheme.*


## 1 Passé

La famille des langages applicatifs est née avec Lisp, en 1960, tout juste après Fortran. Cette lignée d'un autre type a eu la plus féconde descendance qui soit. Scheme et ML en sont aujourd'hui les branches maîtresses sans oublier les autres Hope, Miranda, Haskell, Gofer, COMMON LISP, Talk, Guile, Dylan et en excluant les multiples implémentations ou dialectes de ces mêmes langages.

De nombreuses expériences de conception de langages ont été menées en Lisp depuis sa naissance et dans de très nombreuses directions. Nous ne nous intéresserons ici qu'à la branche la plus classique, celle des langages stricts autorisant les effets mémoire (par opposition aux langages paresseux ou purement fonctionnels). Dans le courant des années 70, cette branche a vu éclore deux bourgeons remarquables : ML puis Scheme. ML a par dessus tout introduit l'inférence de types tandis que Scheme tirait la quintessence de Lisp en supprimant tout ce qui n'était pas primitif.

Syntaxiquement, ML a rejeté l'emploi des parenthèses propres à Lisp, au profit d'une syntaxe plus traditionnelle dans la droite lignée d'Algol. ML utilise également le filtrage en guise d'aiguillage multiple fondé sur la forme de la donnée filtrée. Grâce au typage fort à la compilation, les programmes ML ne produisent aucune erreur de type à l'exécution. De surcroît, ML autorise un polymorphisme de bon aloi puisqu'il permet l'écriture d'algorithmes indépendants du type des données qu'ils manipulent.

Scheme se réduit à quatre formes spéciales (`lambda`, `if`, `set !`, `quote`) et une fonction spéciale réifiant les continuations (`call/cc` capture la suite des calculs en suspens (représentée par la pile d'exécution (*sic*)) dont elle fait un objet manipulable). Ce noyau très réduit fait de Scheme une base de départ idéale pour l'étude de traits expérimentaux, par exemple les continuations, les objets, la réflexion (la capacité à manipuler dans le langage l'état du calcul en cours).

ML et Scheme forment en quelque sorte un "dipôle" linguistique : les langages sont fortement couplés mais bien distincts. Leur implémentation et leur compilation sont très proches, et ils partagent le même fondement théorique : le $\lambda$-calcul (théorie mathématique des fonctions). Le $\lambda$-calcul apporte le polymorphisme, les fonctions d'ordre supérieur, locales ou anonymes. Les langages fonctionnels considèrent les fonctions comme des valeurs à part entière que le programmeur manipule librement (en argument, en résultat ou dans les structures de données). Ce traitement des fonctions implique la création dynamique des fermetures qui les représentent. De façon plus générale, la manipulation de données symboliques ou complexes est explicitement prise en compte par le langage qui gère silencieusement les nécessaires manipulations de pointeurs. Le système de gestion automatique de la mémoire prend en charge l'allocation des valeurs et corrélativement, la libération des données devenues inutiles : c'est la notion de ramasse-miettes ou GC (glanage de cellules). Le GC supprime complètement les problèmes du programmeur confronté à la gestion manuelle de la mémoire, source d'erreurs subtiles, erratiques et malheureusement communes (pointeurs fous et fuites de mémoire). En outre, le modèle de mémoire assure la pertinence des accès aux données en opérant les vérifications dynamiques indispensables (bornes de tableaux par exemple).

Si Scheme et ML partagent les valeurs fonctionnelles, la gestion automatique de la mémoire et la vérification dynamique des opérations d'accès, ils conservent en revanche des philosophies nettement différentes. Scheme privilégie la liberté d'écriture et ne limite pas l'ensemble des programmes utiles concevables, mais suppose en contrepartie des utilisateurs avertis capables de maîtriser leurs erreurs. C'est une sorte d'assembleur de très haut niveau qui n'est pas incompatible avec une programmation de style ML à condition de typer "dans sa tête". Inversement, ML privilégie la sûreté et exige des programmes qu'ils soient bien typés avant d'envisager leur exécution. Cette discipline permet de diminuer dans des proportions impressionnantes le temps de mise au point en capturant, dès la vérification de types, les myriades de bévues habituelles.

Si la vérification de type de ML est peu contraignante pour le programmeur, puisqu'elle est entièrement gérée par le compilateur, elle induit une forte contrainte sur les implémenteurs du langage : la nécessité de fournir une version bien typée des nouveaux traits de programmation retarde parfois leur introduction dans le langage. Par exemple, les continuations n'ont été introduites en ML que bien après leur apparition en Scheme, et les extensions objets de ML sont une victoire toute récente de la recherche (voir Objective Caml).

Sur le plan syntaxique, l'écriture parenthétique de Scheme lui confère une régularité que ne possède aucune syntaxe usant de précédences, ce qui autorise les macros, c'est-à-dire un système d'extension de syntaxe fondé sur une représentation uniforme des programmes par des listes. Les macros sont un des traits distinctifs de Lisp qui lui ont permis de survivre en absorbant la plupart des styles linguistiques environnants et c'est un atout majeur pour l'introspection et la réification des programmes.

# 2 Présent

Les langages applicatifs sont actuellement largement utilisés comme premier langage de programmation dans les cours d'informatique ; par exemple, les Écoles Normales Supérieures, l'École Polytechnique et le CNAM enseignent Caml et/ou Scheme depuis de nombreuses années. Depuis la rentrée scolaire 1995-1996, Caml a été choisi comme langage support pour l'option informatique des classes préparatoires aux grandes écoles. Un grand nombre d'universités proposent aussi des cours basés sur les langages applicatifs à tous les niveaux de l'enseignement de l'informatique.

Ce succès repose sur la qualité pédagogique intrinsèque des langages applicatifs : ces langages sont plus faciles à apprendre parce que leur sémantique est simple et régulière. En outre ils possèdent toujours un système interactif : sitôt un programme fourni, il est vérifié, conditionné (compilé) puis évalué. Une réponse rapide aux actions des élèves a toujours été une qualité pédagogique essentielle. Grâce à ce contact immédiat avec le langage, le cycle essai-erreur est raccourci et le comportement des programmes plus immédiatement appréhendé.

Les langages applicatifs offrent une liberté de style de programmation hors d'atteinte des langages impératifs classiques. Compte tenu du classique effet d'imprégnation, on améliore ainsi l'arsenal programmatique des étudiants en ne les confinant pas à un style unique autant qu'étriqué. Ce gain est principalement dû à la présence de fonctions et de fermetures et à leur usage régulier comme structures de données. Cela facilite l'abstraction et accessoirement permet à l'informatique de s'appuyer sur la culture mathématique ambiante. Dominer un langage applicatif permet également, comme de multiples expériences pédagogiques l'ont montré, de mieux posséder les langages plus classiques.

Choisir entre ML et Scheme est autrement plus complexe et n'a pas fait l'objet d'études approfondies. Il nous semble que ce choix doit être fait en fonction de la population enseignée et des buts recherchés. Pour de futurs ingénieurs généralistes c'est-à-dire non-informaticiens, ML semble plus adapté car il apporte le concept de type. Pour des informaticiens, le choix nous semble moins clair et l'idéal est bien que les étudiants voient les deux : ML parce que l'inférence de types est un outil majeur de la programmation, Scheme car c'est le langage d'expérimentation linguistique permettant de lever facilement le rideau sur son implémentation. Mais ni ML, ni Scheme ne sont immuables et de nombreuses expérimentations visent à les améliorer.

L'indéniable succès pédagogique actuel des langages applicatifs devrait s'accompagner dans les cinq à dix ans d'une percée de ces langages dans le monde industriel. Cette diffusion ne sera pas sans rappeler celle de Pascal dans les années 80, à ceci près que les langages applicatifs procurent des traits plus puissants qui leur conféreront sans doute une plus grande longétivité.

# 3 Avenir

Deux voies nous semblent s'ouvrir actuellement au développement des langages applicatifs. La première, plus spéculative, consisterait à doter les langages applicatifs d'une puissance accrue en allant vers une fusion (ou un déplacement) vers les outils de calcul formel. Ce serait une démarche logique pour des langages ayant intégré d'abord les notions mathématiques de base (nombres rationnels, nombres complexes, matrices), puis la notion de fonction au plein sens mathématique, d'aller vers une mathématisation encore plus poussée en intégrant maintenant les notions d'ensembles, de types quotients, de simplification et de calculs formels sur des formules.

Pour le long terme, la seconde voie qui s'ouvre vise à satisfaire les besoins croissants de logiciels certifiés et la preuve de programme en général. Certains systèmes d'aide à la preuve mathématique, Coq par exemple, autorisent déjà l'obtention de programmes certifiés corrects par rapport à leurs spécifications, puisque le texte de ces programmes est produit automatiquement par la machine à partir de la preuve de correction de leur spécification. Ces programmes extraits de preuves sont des programmes purement fonctionnels, dont la taille et la complexité sont aujourd'hui limitées, avant tout, par la capacité des systèmes de preuves à aider le programmeur à prouver ses spécifications.

Cependant une nouvelle méthode se dessine à l'heure actuelle, notamment illustrée par Eiffel, qui consiste à fournir le programme au système de preuves pour le guider dans l'obtention de la preuve de correction. Pour cela il est nécessaire de fournir un programme qui comporte des annotations logiques apportant la preuve de certains lemmes ou propriétés du programme.

Mêler plus intimement le matériel de preuve aux programmes développés est certainement une voie d'avenir pour obtenir du logiciel fiable et bien compris. Cette voie est largement ouverte aux langages applicatifs dont la sémantique est simple et bien comprise, qualités rarement réunies par les langages de programmation traditionnels.

# 4 Conclusions

Les langages applicatifs sont maintenant des langages adultes, bien compris, à la technologie éprouvée.

La formation mathématique des étudiants en informatique leur fait appéhender facilement des langages à substrat mathématique fort. Réciproquement cette assise théorique confère aux langages applicatifs une rigueur qui permet d'aller plus loin et plus vite dans l'enseignement de l'informatique, sans être arrêté par des détails techniques sans rapport avec la résolution du problème.

L'avènement de la programmation sans erreur, indispensable lorsqu'un logiciel met en jeu des vies humaines, ne s'accorde que de preuves au sens mathématique. Pour écrire des programmes prouvables il faut des langages de programmation rigoureux, simples et sémantiquement bien définis. Seuls les langages applicatifs réunissent actuellement ces qualités.

# Travaux pratiques

Nous recensons quelques systèmes francophones de Scheme et de ML, tous facilement accessibles sur le réseau et distribués gratuitement. Tous tournent sous Unix, souvent sur PC et quelquefois sur Macintosh.

Trois bonnes implémentations de Scheme sont disponibles : Bigloo, Gambit et STk :

- Bigloo, œuvre de Manuel Serrano, est un compilateur performant de Scheme vers C. Il comporte bien sûr un interprète et procure également des extensions en matière de modules, de filtrage et de génération d'analyseurs lexicaux et syntaxiques. En outre, Bigloo illustre concrètement la profonde similarité entre les langages applicatifs, puisqu'il compile indifféremment Caml Light et Scheme dans les mêmes conditions de performance.
- Gambit est un compilateur/interprète véloce, disponible sur les trois architectures. Gambit produit du C portable et comporte une excellente interface sur Macintosh.
- STk est dû à Erick Gallesio (Université de Nice — Sophia Antipolis). C'est un interprète couplé à la bibliothèque graphique Tk de John Ousterhout et accompagné d'une couche d'objets à la CLOS.

Côté ML, on trouve principalement LCS et Caml :

- LCS qui est un langage dérivé de SML, développé à Toulouse par l'équipe de Bernard Berthomieu, et qui inclue la notion de tâche à la CCS.
- Caml est le principal dialecte de la famille ML d'origine française. Disponible également sur micros Macintosh et PC, il est développé à l'Inria par le projet Cristal. Caml comporte évidemment un mécanisme de filtrage

élaboré, des modules simples et performants, et des capacité de génération d'analyseurs lexicaux et syntaxiques à travers un système original de flots paresseux.

La dernière extension du langage est Objective Caml, qui comporte un puissant système de modules et la première implémentation d'objets compatible avec la discipline de typage de ML.

# References

[AD96] Laurent Arditi and Stéphane Ducasse. *La programmation : une approche fonctionnelle et récursive avec Scheme*. Eyrolles, 1996. ISBN 2-212-08915-5.

[AS85] Harold Abelson and Gerald Jay with Julie Sussman Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., 1985. Traduit en français chez InterÉditions.

[CM95] Guy Cousineau and Michel Mauny. *Approche fonctionnelle de la programmation*. Ediscience international, 1995.

[Huf96] Jean-Michel Hufflen. *Programmation fonctionnelle en Scheme. De la conception à la mise en oeuvre*. Masson, 1996.

[HV92] Thérèse Accart Hardin and Véronique Donzeau-Gouge Viguié. *Concepts et outils de programmation*. InterÉditions, 1992.

[Que94] Christian Queinnec. *Les langages Lisp*. InterÉditions, Paris (France), 1994.

[SJ93] Emmanuel Saint-James. *La programmation appplicative (de LISP à la machine en passant par le lambda-calcul)*. Hermès, 1993.

[WL93] Pierre Weis and Xavier Leroy. *Le langage Caml*. InterÉditions, 1993.