Une introduction à la programmation fonctionnelle : HOPE et ML

Martin WIRSING et Don SANNELLA

Présentation *Comment s'abstraire le plus possible des contraintes de la machine sans pour autant obtenir des programmes trop inefficaces ? La Programmation Fonctionnelle tente de fournir une réponse à cette vieille question en décidant résolument de considérer un programme comme une fonction. Cette idée, a priori très simple et séduisante, n'est pas sans poser de nombreuses questions : comment définir de telles fonctions sans tomber dans les lourdeurs d'écritures comme en Lisp ? peut-on améliorer la structuration de tels ensembles de fonctions par des mécanismes de typage et de modularisation ? peut-on toujours se passer de l'affectation sans risquer des acrobaties peu raisonnables ? Autant de questions auxquelles les différents auteurs de langages fonctionnels ont tenté de répondre, depuis les diverses versions de Lisp jusqu'aux langages fondés sur les types abstraits algébriques comme OBJ3, en passant par le fameux FP de J. Backus ou Miranda de D. Turner. Cet article de M. Wirsing et D. Sannella présente l'histoire de deux autres langages célèbres HOPE et ML développés à Édimbourg et dotés de propriétés très puissantes comme le polymorphisme. Nulle doute que cette synthèse permette au lecteur de TSI d'acquérir rapidement les fondements de ce style de programmation.*

J. Pierre Finance

I. Introduction

Un langage « fonctionnel » est un langage de programmation dont les objets principaux sont des fonctions. On dit qu'un tel langage est « applicatif » lorsqu'on veut souligner le rôle de l'opération « application d'une fonction à une expression ». L'avantage d'un programme fonctionnel par rapport à un programme écrit dans un langage impératif comme PASCAL ou ADA résulte du fait qu'un tel programme est mathématiquement plus simple et plus accessible à l'homme. La correction d'un programme fonctionnel peut-être (relativement) bien vérifiée [Boyer, Moore 77] et des techniques formelles de transformation et d'optimisation facilitent le développement de programmes corrects [Burstall, Darlington 77 ; Bauer, Wössner 82]. Un programme fonctionnel n'admet pas d'effets de bord et l'ordre d'exécution des différentes parties d'un programme n'est pas fixé à l'avance, une conséquence étant qu'un traitement parallèle peut lui être applicable. Ces propriétés sont à l'origine de l'intérêt porté aux langages fonctionnels depuis quelques années par les milieux de la recherche. Depuis peu l'industrie commence à les prendre en considération, certainement en raison de nombreuses et importantes initiatives internationales de recherche et développement comme la «5e génération d'ordinateurs » initiée au Japon ou le programme ESPRIT de la communauté européenne.

HOPE et ML sont deux langages de programmation fonctionnelle qui ont été développés à l'Université d'Édimbourg en Écosse. HOPE a été conçu aux environs de 1980 par Rod Burstall, David MacQueen et (l'un des auteurs de ce texte) Don Sannella [Burstall *et al.* 80]. Le nom HOPE a été choisi en référence à l'ancienne adresse de l'Institut pour l'Intelligence Artificielle d'Édimbourg : Hope Park Square. C'est un langage basé sur des équations récursives : chaque programme représente une ou plusieurs fonctions qui

sont définies par un ensemble (fini) d'équations récursives.

L'origine de ML, développé par Robin Milner en collaboration avec Malcolm Newey, Lockwood Morris, Michael Gordon et Christopher Wadsworth, remonte à 1974. Initialement ML n'était pas considéré comme un langage général mais plutôt comme étant adapté à une application bien déterminée : ML était le métalangage (le nom est une abréviation de l'expression anglaise « Meta-Language ») pour le projet LCF à Édimbourg. Le but de ce projet était la construction d'un système interactif pour la preuve formelle de fonctions récursives [Gordon *et al.* 79]. LCF est l'abréviation de « Logic of Computable Functions », une logique définie par Dana Scott qui permet de formuler et de prouver des théorèmes sur les fonctions calculables. Très rapidement dès l'implantation et l'expérimentation du système LCF on a pu constater que son succès provenait en grande partie de la flexibilité et de l'universalité de ML. On a alors commencé à utiliser ML de plus en plus indépendamment de LCF en tant que langage d'implantation de prototypes, par exemple pour la vérification automatique de programmes ou pour la construction de systèmes experts. A la suite de ces expériences le langage a été redessiné par un groupe de scientifiques comprenant les auteurs déjà mentionnés auxquels se sont joints Rod Burstall, Luca Cardelli, Robert Harper et David MacQueen. C'est ainsi que les idées principales de HOPE et un concept de modularisation ont été intégrés dans ML. La version consolidée de ML a été publiée sous le nom « Standard ML » en mars 1986 [Harper *et al.* 86]. D'autres travaux ont été simultanément menés sur ML tels que ceux développés en France à l'INRIA et qui ont conduit à CAML [Cousineau 85]. Cette version de ML est fondée sur le concept de machine catégorique (Categorical Abstract Machine) bien adapté au calcul fonctionnel ce qui conduit à une implantation particulièrement efficace.

Dans cet article HOPE et ML sont présentés informellement à partir d'exemples simples. Commençons tout d'abord par résumer les propriétés principales des deux langages :

— Puisque ce sont des langages de programmation fonctionnelle chaque programme est constitué d'une séquence de déclarations.
— Les fonctions sont définies inductivement par des équations récursives sur les constructeurs des types de données.
— Les deux langages admettent des fonction(elle)s d'ordre supérieur. Ceci signifie que les fonctions sont des « objets de première classe » qui peuvent également être utilisées en tant que paramètres ou résultats d'autres fonctions.
— Les deux langages sont strictement typés et permettent grâce au concept de type polymorphe des définitions schématiques de types de données et de fonctions.
— HOPE est un langage fonctionnel « pur » alors que Standard ML contient également des concepts non fonctionnels (« impératifs ») comme l'affectation, les pointeurs et un mécanisme de traitement d'exceptions.
— Standard ML peut être vu comme une extension de HOPE contenant en plus des concepts impératifs et surtout des mécanismes puissants de modularisation.
— Standard ML possède une sémantique formelle : une définition complète de sa sémantique opérationnelle peut être trouvée dans [Harper *et al.* 87].

2. HOPE

Le développement de Hope a été beaucoup influencé par LISP et ISWIM [Landin 66] ; un prédécesseur direct de HOPE est le langage NPL [Burstall 77].

Un programme écrit en HOPE consiste en une expression et une séquence de déclarations de fonctions et de sortes (c'est-à-dire noms de types de données) qui servent à spécifier les symboles fonctionnels apparaissant dans l'expression. Pour améliorer la structuration on peut grouper les déclarations dans des « modules ». Un module permet aussi de cacher des déclarations auxiliaires de telle façon que seules les déclarations nécessaires (pour l'évaluation de l'expression) soient visibles à l'extérieur.

Dans les sections 2.1-2.3 suivantes nous présentons successivement la déclaration de fonctions récursives, la déclaration de types de données et le polymorphisme en les illustrant par des exemples. Le concept de modularisation n'est pas présenté puisqu'il est contenu dans celui de ML lui-même présenté en section 3.

2.1. Déclaration de fonctions

HOPE est un langage strictement typé comme par exemple l'est aussi PASCAL. Le type d'une fonction (sa « déclaration ») est donné par le nom de la fonction et par sa fonctionnalité formée des sortes des paramètres et du résultat.

Par exemple le type du maximum de deux nombres naturels est décrit par la déclaration suivante :

```
dec max : num * num → num
```

dec est un mot réservé (abréviation de « déclaration ») ; la sorte num dénote le type de données des nombres naturels ; entre « : » et « → » sont précisées les sortes des paramètres, à droite de « → » les sortes des résultats (ici il n'y a qu'un résultat, num).

Les fonctions sont définies par des équations ; pour le maximum une seule suffit :

```
--- max(x, y) ⇐ x  if x > y
                   else y
```

Le symbole « --- » peut être lu comme « la valeur de ». Le membre gauche, max(x, y), de l'équation précise les paramètres formels x et y qui, dans un appel de la fonction, sont remplacés par les valeurs effectives. Le symbole « ⇐ » peut être lu comme « est définie

par ». Le membre droit de l'équation est une expression conditionnelle qui définit le résultat de la fonction: la valeur de max(x, y) est x si la valeur de la sous-expression $x > y$ est vraie ; sinon c'est la valeur de y.

Un programme permettant de contrôler la correction des types (en anglais « type-checker ») garantit que toutes les erreurs de types sont découvertes au moment de la compilation ; ce qui n'est par exemple pas le cas en PASCAL à cause des « variant records ». De plus contrairement aux langages impératifs l'ordre de l'exécution d'une expression conditionnelle n'est pas fixé dans HOPE. Ainsi avec un ordinateur qui admet des exécutions parallèles il est possible de calculer les trois sous-expressions et de choisir le résultat correct au moment où la valeur de la condition est connue.

La déclaration de max peut être utilisée par exemple dans l'expression simple suivante :

```
max(17, max(3, 5)) * max(1, 3)
```

avec le résultat

```
51 : num
```

où le terme à droite des deux points indique le type du résultat.

On peut bien sûr définir récursivement une fonction. Par exemple la définition de la factorielle d'un nombre naturel x est :

```
dec fac : num → num
---fac(x) ⇐ 1   if x = 0
          else x * fac(x − 1)
```

2.2. Types de données

L'une des idées de base de HOPE est de représenter tous les éléments d'un type de données par des termes formés avec des « constructeurs ». Pour déclarer un nouveau type de données il suffit donc d'en indiquer ses constructeurs. Par exemple le type de données num des nombres naturels peut être défini de la façon suivante :

```
data num == 0 ++ succ(num)
```

Les constructeurs de num sont la constante « 0 : num » et la fonction unaire « succ : num → num ». La sorte du paramètre de succ est mentionnée en parenthèses derrière l'indication du nom succ, la sorte du résultat des constructeurs est donnée par le nom du type en partie gauche de l'équation. Les éléments de num sont 0, succ(0), succ(succ(0)), ..., ou encore de façon équivalente 0, 1, 2, ...

La déclaration de num est une définition récursive ; num apparaît sur le côté gauche *et* sur le côté droit de l'équation. Le type de données suivant n'est pas récursif :

```
data color == white ++ black ++ red ++ green
```

Les constructeurs du type color sont les quatre constantes white, black, red et green qui constituent en même temps les quatre éléments de color.

HOPE contient aussi des types de données prédéfinis, en particulier ce sont num, truval (pour le type des booléens avec les constantes true et false), char (pour des symboles comme a, ..., z, 0, ..., 9, !, ?, ...), list (pour les séquences finies) et set (pour les ensembles finis). Tous ces types de données pourraient bien sûr être définis par l'utilisateur, mais des raisons d'efficacité ont conduit à les intégrer comme types standards du langage. Un autre exemple de type de données récursif est le type « numtree » qui représente des arbres dont les nœuds sont étiquetés par des nombres naturels :

```
data numtree = empty ++ node(numtree # num # numtree)
```

Un objet du type numtree est par exemple (voir aussi *fig.* 1).

```
node(node(empty, succ(0), empty),
     succ(succ(0)),
     node(empty,
          0,
          node(empty, succ(succ(succ(0))), empty))),
```

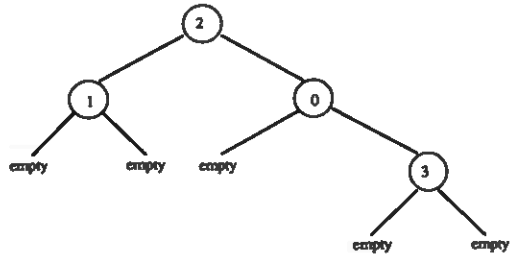

Figure 1. Un arbre du type numtree.

Étant donné un nouveau type de données la définition d'une nouvelle fonction peut être donnée par « induction structurelle » sous forme d'une suite d'équations : à chaque constructeur est associée (au moins) une nouvelle équation. Par exemple on peut décrire récursivement la fonction « flatten » qui définit pour chaque arbre la suite de ses (étiquettes de) nœuds de la façon suivante:

```
dec flatten : numtree → list(num)
--- flatten(empty) ⇐ nil
--- flatten(node(t1, n, t2)) ⇐ append(flatten(t1), n :: flatten(t2))
```

La constante « nil » ci-dessus dénote la suite vide, « append » représente la concaténation de deux suites et la fonction « :: » ajoute un élément en tête d'une suite. L'ordre des équations ne joue aucun rôle. En général chaque équation a la forme.

```
--- ⟨nom de fonction⟩(⟨construction⟩) ⇐ ⟨expression⟩
```

où ⟨construction⟩ dénote une expression qui est formée par l'application d'un constructeur à des variables libres (ou plus généralement à d'autres expressions de la forme ⟨construction⟩). Ainsi l'équation dans la déclaration de « max » est de cette forme.

Le symbole « , » entre x et y est le constructeur du produit cartésien « num # num ». Le résultat de flatten, appliquée à l'arbre $t1$ de la figure 1, est la liste [1, 2, 0, 3].

La fonction suivante, chartree, change en 1 toutes les étiquettes strictement positives d'un arbre. Dans la déclaration on utilise les constructeurs de numtree en plus ceux de num :

```
dec chartree : numtree → numtree
--- chartree(empty) ⇐ empty
--- chartree(node(t1, 0, t2)) ⇐ node(chartree(t1), 0, chartree(t2))
--- chartree(node(t1, succ(n), t2))
                    ⇐ node(chartree(t1), succ(0), chartree(t2))
```

Comme c'est le cas dans les exemples ci-dessus, une règle importante de la programmation fonctionnelle est que chaque ensemble d'équations doit être complet, ce qui signifie qu'à chaque cas possible doit correspondre une équation dans le système d'équations.

2.3. Polymorphisme

Les résultats de « flatten » sont des suites de nombres naturels de type list(num). list est un type de données polymorphe (on dit encore générique ou paramétré) : la définition de list permet d'obtenir list(num) (« suites de nombres ») ainsi que list(truval) (« suites de symboles booléens »), list(list(num)) « suites de suites de nombres » et même list(num → num) (« suites de fonctions unaires sur les nombres naturels »). Pour les fondements théoriques du polymorphisme voir [Milner 78]. De manière analogue les arbres peuvent être définis de façon polymorphe :

```
data tree(alpha == empty
          ++ node(tree(alpha) # alpha # tree(alpha))
```

Ci-dessus « alpha » dénote le paramètre formel qui peut être remplacé par des (noms de) types de données arbitraires, même polymorphes, par exemple par list(beta), tree(alpha) ou tree(list(beta)).

On peut aussi définir des fonctions avec des types polymorphes comme sortes de paramètres ou résultats. Par exemple la fonction

```
flatten : tree(alpha) → list(alpha)
```

peut être définie de façon polymorphe avec les deux mêmes équations que ci-dessus.

La fonction maptree ci-dessous est un autre exemple de fonction polymorphe. De plus maptree est une fonction d'ordre supérieur puisque son premier paramètre est lui-même une fonction. Pour une fonction unaire donnée f et un arbre t l'appel maptree(f, t) a pour résultat un arbre t' qui est obtenu à partir de t par l'application de f sur les étiquettes des nœuds de t.

```
dec maptree : (alpha → beta) # tree(alpha) → tree(beta)
--- maptree(f, empty) ⇐ empty
--- maptree(f, node(t1, x, t2))
          ⇐ node(maptree(f, t1), f(x), maptree(f, t2))
```

Chaque type de données peut être substitué à alpha ou beta. Par exemple l'appel suivant de maptree (voir *fig.* 2)

```
maptree(max, node(node(empty, (3, 4), empty),
                  (9, 3),
                  node(node(empty,
                            (6, 6),
                            node(empty,(7, 4), empty)),
                       (1, 2),
                       empty)))
```

donne comme résultat l'arbre suivant

```
node(node(empty, 4, empty),
     9,
     node(node(empty,
               6,
               node(empty, 7, empty)),
                    2,
                    empty))
```

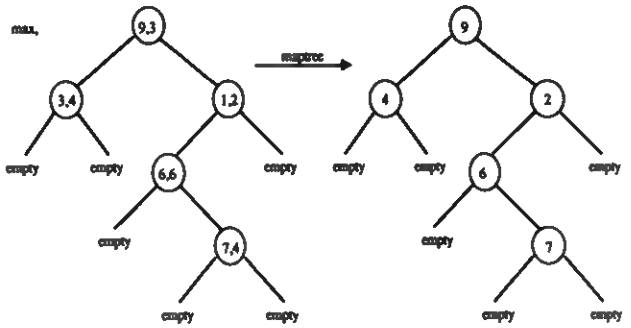

Figure 2. — L'appel de maptree(max, *t*).

Dans cet exemple on a substitué « num # num » à « alpha » et « num » à « beta ». La fonction chartree de la section 2.2 peut être définie à l'aide de maptree :

```
--- chartree(t) ⇐ maptree(ch, t)
```

où ch indique la fonction caractérisque de num :

```
dec ch : num → num
--- ch(0) ⇐ 0
--- ch(succ(n)) ⇐ succ(0)
```

Les fonctions sont des objets « de première classe » dans HOPE : non seulement elles peuvent apparaître comme paramètres ou résultats d'autres fonctions, mais elles peuvent également contenir d'autres fonctions (en tant que sous-données). Par exemple il est possible de définir des graphes avec les nœuds étiquetés par des nombres naturels de la façon suivante :

```
data graph == mkgraph(set(num) # (num # num → truval))
```

Dans cette déclaration du type de données « graph » un graphe est représenté par (l'application du constructeur mkgraph sur) un ensemble de nœuds (du type « set(num) ») et une fonction (de fonctionnalité « num # num → truval ») qui détermine s'il y a un arc entre deux nœuds.

Par exemple l'expression

```
mkgraph({1, 2, 3, 4}, tr)
```

où

```
dec tr : num # num → truval
--- tr(x, y) ⇐ true,
```

représente un graphe complet avec quatre nœuds: il existe un arc entre chaque paire de nœuds (voir *fig.* 3*a*).

L'expression

mkgraph({0, 1, 2, 3, 4, 5, 6, 7, 8}, pair)

où

```
dec pair : num # num → truval
--- pair(x, y) ⇐ true if y − x = succ(succ(0))
                 else false,
```

représente le graphe décrit par la figure 3*b*.

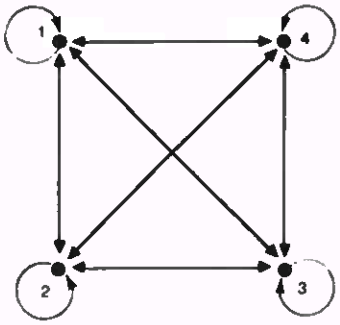

Figure 3a. — Le graphe complet mkgraph({1, 2, 3, 4}, tr).

Figure 3*b*. — Le graphe mkgraph({0, 1, 2, 3, 4, 5, 6, 7, 8}, pair).

3. ML

Standard ML peut être vu comme une extension de HOPE. Tous les concepts de section 2 peuvent être réalisés dans ML. La syntaxe de ML diffère peu de celle de HOPE et les différences devraient devenir claires en regardant les exemples suivants. Contrairement à HOPE il n'est pas nécessaire dans ML de donner la fonctionnalité d'une fonction car le système ML dispose d'un algorithme de typage permettant de calculer la fonctionnalité d'un objet ainsi que les constructeurs d'un type de données. La déclaration d'arbres polymorphes et de la fonction maptree se présente sous la forme suivante :

```
datatype 'a tree = empty | node of 'a tree * 'a * 'a tree
fun maptree(f, empty) = empty
|   maptree(f, node(t1, x, t2))
            = node(maptree(f, t1), f(x), maptree(f, t2))
```

(Dans la déclaration du type de données on écrit « **datatype** » au lieu de « **type** », « *'a* » au lieu de « alpha », « | » au lieu de « + », « * » au lieu de « # » et « **of** » au lieu des parenthèses autour des paramètres). Le système de ML répond à cette déclaration avec toute l'information à sa disposition :

```
datatype 'a tree = empty | node of ('a tree) * 'a * ('a tree)
con node = fn : (('a tree) * 'a * ('a tree)) → ('a tree)
con empty = empty : 'a tree
val maptree = fn : (('a → 'b) * ('a tree)) → ('b tree)
```

Notons que la déclaration du type « *'a* tree » est répétée. Le mot réservé « **con** » permet de préciser les deux constructeurs « empty » et « node » et leurs fonctionnalités. La valeur de « empty » est la constante (du même nom) « empty : *'a* tree » du type « *'a* tree », la valeur de node est une fonction (mot réservé « **fn** ») de la fonctionnalité « (('*a* tree) * *'a* * ('*a* tree)) → ('*a* tree) ». La valeur de maptree (mot réservé « **val** ») est aussi une fonction, mais comme la fonctionnalité la plus générale de *f* est « *'a* → *'b* » la fonctionnalité de maptree est « (('*a* → *'b*) * ('*a* tree)) → *'b* tree ». Puisque les fonctions ne possèdent pas de représentation standard (contrairement aux types de données et aux constantes) le système n'indique que le mot réservé « **fn** » et la fonctionnalité au lieu de la valeur.

3.1. Modules

Une des qualités nécessaires pour la construction et la manipulation de systèmes de programmes est l'habilité du programmeur à décomposer des programmes larges dans des petites unités relativement indépendantes. ML offre dans cette perspective un concept de modularisation qui est formé de trois notions : structure, signature et foncteur [MacQueen 85].

3.1.1. *Structures et Signatures*

Une structure encapsule des types de données et des fonctions dans une unité de programme. Par exemple les nombres naturels, la relation d'égalité et la relation « plus-petit-ou-égal » forment une structure nommée Nat :

```
structure Nat =
  struct
    datatype num = O | succ of num
    fun le(O, n) = true
    |   le(succ(m), O) = false
    |   le(succ(m), succ(n)) = le(m, n)
    fun eq(m,n) = le(m, n) andalso le(n, m)
  end
```

La « fonctionnalité » (ou le « type ») d'une structure *S* est appelée « signature » ; elle est donnée par tous les noms de types de données et de fonctions de la structure qui sont visibles à l'extérieur. La signature décrit l'interface d'une unité de programme avec les autres unités de programme dans un système. Des valeurs comme la déclaration d'un type de données ou l'indication « **fn** » pour les fonctions ne font pas partie de la signature. Comme pour les fonctionnalités on peut déduire les signatures automatiquement de la déclaration de la structure. Ainsi « Nat » a la signature suivante :

```
sig
  type num
  con O : num
  con succ : num → num
  val le : num * num → bool
  val eq : num * num → bool
end
```

On peut donner un nom à une signature :

```
signature NAT =
  sig
    type num
    con O : num
    con succ : num → num
    val le : num * num → bool
    val eq : num * num → bool
  end
```

De cette façon on peut indiquer explicitement l'interface « NAT » de « Nat » par la déclaration :

```
structure Nat : NAT =
  struct
    datatype num = O | succ of num
    fun le(O, n) = true
    |   le(succ(m), O) = false
    |   le(succ(m), succ(n)) = le(m, n)
    fun eq(m, n)=le(m, n) andalso le(n, m)
  end
```

Si l'on considère « le » simplement comme une fonction auxiliaire de la définition de « eq », on peut restreindre l'interface à la signature NAT1 suivante :

```
signature NAT1 =
  sig
    type num
    val O: num
    val succ : num → num
    val eq : num * num → bool
  end
```

On obtient une structure Nat1 qui est définie à l'aide de Nat, mais qui ne met pas la fonction « le » à disposition d'un utilisateur.

```
structure Nat1 : NAT1 = Nat
```

3.1.2. *Structures hiérarchiques*

La structuration hiérarchique est une des méthodes les plus simples de modularisation. Dans Standard ML on peut construire hiérarchiquement une structure à partir de structures existantes. La structure d'arbres suivante utilise par exemple la structure Nat :

```
structure Tree =
  struct
    structure Elem = Nat
    datatype tree = empty | node of tree * Elem.num * tree
    fun flatten(empty) = nil
    |   flatten(node(t1, n, t2)) = append(flatten(t1), n :: flatten(t2))
    fun put(n, empty) = node(empty, n, empty)
    |   put(n, node(t1, m, t2)) =
        if Elem.le(n, m)
          then node(put(n, t1), m, t2)
          else node(t1, m, put(n, t2))
  end
```

Les étiquettes des arbres de la structure Tree sont les objets de la structure primitive Elem (qui est égal à Nat). Dans une structure hiérarchique on peut identifier les fonctions et types de données d'une structure primitive en donnant le nom de la structure suivi du nom de la fonction ou du nom du type de données, par exemple « Elem.le », « Elem.num ». La signature d'une structure hiérarchique *S* contient les noms et signatures des structures ainsi que les noms de toutes les fonctions et de tous les types introduits par *S*. Par exemple la signature de Tree est comme suit :

```
sig
  structure Elem : NAT
  type tree
  con empty : tree
  con node : tree * Elem.num * tree → tree
  val  flatten : tree → Elem.num list
  val  put : Elem.num * tree → tree
end
```

Si l'on considère des arbres ordonnés, on remarque que les résultats de « flatten » et « put » appliqués à des arbres ordonnés sont de nouveau ordonnés ; par contre la fonction « node » ne respecte pas l'ordre. Il semble donc dangereux de mettre cette fonction à disposition de l'utilisateur. A l'aide de la signature suivante ORDTREE on peut exclure l'utilisation de « node » à l'extérieur de Tree :

```
signature ORDTREE =
  sig
    structure Elem : NAT
    type tree
    val  empty : tree
    val  flatten : tree → Elem.num list
    val  put : Elem.num * tree → tree
  end
structure Ordtree : ORDTREE = Tree
```

La structure Ordtree est la restriction de Tree à la signature ORDTREE. En utilisant les fonctions de Ordtree on ne peut générer que des arbres ordonnés.

3.1.3. *Foncteurs*

Pour décrire des arbres ordonnés il n'est pas important d'utiliser les nombres naturels. Chaque structure *X* munie d'une relation d'ordre serait appropriée pour l'étiquetage. Plus précisément la condition préalable est que *X* possède la signature suivante :

```
signature ORD =
  sig
    type elem
    val  le : elem * elem → bool
  end
```

Notons que les propriétés caractéristiques d'une relation d'ordre (réflexivité, antisymmétrie, transitivité et linéarité) ne peuvent pas être exprimées dans ML ; un langage de spécification serait adapté à ce propos (*cf.* par exemple [Sannella, Tarlecki 85]).

En associant « elem » à « num » à « elem » on obtient à partir de Nat une structure de la signature ORD :

```
structure Natelem =
  struct
    datatype elem = Nat.num
    val  le = Nat.le
  end
```

La structure Natelem consiste en l'ensemble des nombres naturels muni de la relation « plus-petit-ou-égal ».

Une autre structure ordonnée est l'ensemble des suites (d'entiers) muni de l'ordre lexicographique.

Dans la structure Ordstring ci-dessous on dénote la sorte des suites d'entiers par « elem » et l'ordre lexicographique par « le » ; alors ORD est une sous-signature de la signature de Ordstring.

```
structure Ordstring =
  struct
    datatype alphabet = int
    datatype elem = nil | :: of alphabet * elem
    fun le(nil, s) = true
    |   le(x :: s, nil) = false
    |   le(x :: s, y :: s') =
          if x < y then true
                   else if x = y then le(s,s')
                                 else false
  end
```

Le concept de « foncteur » permet de prendre en compte des structures ordonnées arbitraires pour étiqueter des arbres. Un foncteur définit une structure paramétrée dont le paramètre consiste en un ou plusieurs noms de structures munis de signatures. Le foncteur Xtree prend comme paramètre une structure ordonnée X (plus exactement : une structure X avec (sous-) signature ORD) et a comme résultat une structure d'arbres dont les étiquettes de nœuds sont des éléments de X :

```
functor Xtree(X : ORD) =
  struct
    structure Elem = X
    datatype tree = empty | node of tree * Elem.elem * tree
    fun flatten(empty) = nil
    |   flatten(node(t1, n, t2)) = append(flatten(t1), n :: flatten(t2))
    fun put(x, empty) = node(empty, x, empty)
        if Elem.le(x, y)
          then node(put(x, t1), y, t2)
          else node(t1, y, put(x, t2))
  end
```

Par instantiation avec Natelem on obtient la structure

```
Xtree(Natelem)
```

qui coïncide (presque) avec Tree ; cependant la constante 0 et les fonctions succ et eq sont absentes de Xtree(Natelem).

Une (représentation de la) signature de Xtree(Natelem) est TREE :

```
signature TREE =
  sig
    structure Elem : ORD
    type tree
    val  empty : tree
    val  node : tree * Elem.elem * tree → tree
    val  flatten : tree → Elem.elem list
    val  put : Elem.elem * tree → tree
  end
```

Il y a une différence subtile entre TREE et la signature de Tree. La dénotation du type des nombres naturels dans Tree est « Elem.num » ; par contre dans Xtree(Natelem) on a deux notations équivalentes, « Elem.elem » et « Natelem.elem » ; une notation équivalente pour la fonctionnalité de « node » est:

```
node : tree * Natelem.elem * tree → tree.
```

Une autre instantiation de Xtree est la structure

```
Xtree(Ordstring).
```

Cette structure décrit des arbres avec des suites comme étiquettes des nœuds. Xtree(Ordstring) a la signature

```
sig
  structure Elem : ORD
  type tree
  val  empty : tree
  val  node : tree * Elem.elem * tree → tree
  val  flatten : tree → Elem.elem list
  val  put : Elem.elem * tree → tree
end
```

Comme dans Xtree(Natelem) on dénote ici le type des arbres par « tree » et le type des marquages par « Elem.elem ». Un dernier exemple montre qu'il est possible de composer des structures. La structure Sum ci-dessous construit la somme disjointe des deux types d'arbres :

```
structure Sum =
  struct
    structure Tree1 = Xtree(Natelem)
    structure Tree2 = Xtree(Ordstring)
    datatype sum =
      nattree of Tree1.tree |
      stringtree of Tree2.tree
  end
```

La structure Sum a la signature

```
sig
  structure Tree1 : TREE
  structure Tree2 : TREE
  type sum
  con nattree : Tree1.tree → sum
  con stringtree : Tree2.tree → sum
end
```

Ici aussi on voit comme on peut distinguer dans un contexte plus large les deux différents types d'arbres bien qu'ils aient la même dénotation « tree » dans Xtree(Natelem) et Xtree(Ordstring).

3.2. Style impératif et traitement des exceptions

Standard ML contient aussi des concepts non fonctionnels comme l'affectation et les pointeurs. Mais contrairement aux langages de programmation conventionnels ces concepts ne sont pas centraux dans ML.

Les pointeurs sont introduits par un type de données standard à qui appartiennent l'affectation et la composition séquentielle en tant qu'opérations de base. Il s'agit d'un type polymorphe dénoté par « *'a* ref » (où « *'a* » ne peut être instantié que par des expressions de types sans variable). Ainsi donc les pointeurs sont également strictement typés. Par exemple « int ref » est le type des pointeurs sur les entiers et « int list ref » est le type des pointeurs sur les suites de type « int ».

L'affectation est une opération, notée classiquement « := », qui permet de changer la valeur d'un pointeur. Donc l'effet d'une affectation est un changement du « tas » de l'ordinateur, l'affectation n'a pas de résultat

propre. Pour en décrire la fonctionnalité on utilise le type standard « unit » qui ne contient qu'un seul élement, la figure 4 précise ce point.

```
ref   : 'a → 'a ref
!     : 'a ref → 'a
:=    : 'a ref * 'a → unit
;     : 'a * 'b → 'b
```

Figure 4. — Les opérations standards du type « 'a ref » des pointeurs.

Par exemple on peut déclarer un pointeur x sur l'entier 17 par « **val** x =ref 17 ». Le système de ML calcule le type « int ref » de x et répond « **val** x =ref(17) : int ref ». L'expression !x a alors la valeur « 17 : int ». L'affectation x := 0 est du type unit et a la valeur standard (), elle ne change que le contenu du pointeur ; en calculant !x on obtient « 0 : int ». L'expression x := 3 ; !x+1 » a la valeur « 4 : int » ; la composition séquentielle a dans ce cas la fonctionnalité « unit * int → int ».

Un concept intéressant de ML est le traitement des exceptions. Il permet d'obtenir des messages d'erreur explicites ainsi que de calculer des « valeurs d'erreur » (qui dans ML sont bien définies et strictement typées). Par exemple en cas de division par zéro le système de ML émet un message d'erreur prédéfini ; pour l'expression « 2/0 » ce message est « div : 2 ». Un autre exemple est la déclaration d'une fonction « hd » qui calcule la tête d'une suite. La tête de la suite vide n'a pas de valeur bien définie. Par la déclaration

```
exception headofnil : unit
fun hd(nil) = raise headofnil
|   hd(x :: s) = x
```

on introduit l'exception « headofnil » du type unit. Un appel de « hd nil » cause le message « Failure : headofnil » du système de ML. La fonctionnalité de hd est « 'a list → 'a ». Dans l'exemple suivant le mécanisme de traitement d'erreurs (en anglais « exception handler ») est utilisé pour définir une « valeur d'erreur » dont le type doit être de nouveau conforme aux types des autres résultats.

```
fun search(x, s) = lookfor(x, s) handle headofnil ⇒ 0
fun lookfor(x, s) = if x = hd(s) then 1
                                 else 1 + lookfor(x, tl(s))
```

La fonction « search » calcule la première position d'un élément x dans une suite s ; si x n'est pas dans la suite, on obtient la longueur de s comme résultat. Par contre la fonction auxiliaire « lookfor » émet dans la même situation le message d'erreur « headofnil ». Une introduction complète dans Standard ML est donnée dans [Harper 86].

4. Conclusion

HOPE et ML sont des langages de programmation fonctionnelle ; leurs programmes sont surtout formés par des déclarations récursives et des expressions construites à l'aide d'appel de fonctions. Des concepts impératifs n'existent pas (dans HOPE) ou sont introduits d'une manière « propre » en tant que types de données standards (dans ML). Cette méthode pourrait être aussi utilisée pour introduire le concept de parallélisme dans les programmes. Une extension de ML est en cours de développement dans laquelle le type de données standard « process » représente les processus parallèles ; dans cette approche la composition parallèle est une fonction binaire

|| : process * process → process.

Le polymorphisme introduit une forme de dépendance entre les types de données ; par exemple la sorte « int list » dépend de la sorte « int ». Une extension intéressante de ML serait une généralisation du polymorphisme introduisant le concept des « types dépendants ». Ce concept permettrait d'écrire des expressions à la place des paramètres de sortes polymorphes ; un exemple simple est l'expression « [0..n] list », où n est un pointeur, sur un nombre naturel. Ce type décrirait des suites de nombres naturels plus petits ou égaux à n. D'autres exemples sont des paires de la forme [int, – 3] ou [bool, true] où la première composante est une sorte et la seconde composante est une expression dont la sorte dépend de la première composante. Ce concept est réalisé dans le langage « Pebble » [Burstall, Lampson 84].

Implantations

Un interpréteur portable de HOPE, écrit en PASCAL, est distribué par le Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ.

Un compilateur de ML est distribué par le Laboratory for Foundations of Computer Science, University of Edinburgh, Edinburgh, EH9 3JZ. Ce compilateur est portable sur chaque ordinateur possédant un compilateur *C* (par exemple VAX, SUN, Orion, ...). Plusieurs versions de ML ont été mises au point à l'INRIA dans le cadre du projet FORMEL, elles sont réalisées en Le Lisp. La dernière de ses versions a déjà été mentionnée dans cet article, il s'agit de CAML qui possède une implantation très efficace et qui est dotée d'un environnement de programmation complet.

Remerciements

Les auteurs remercient Jean Pierre Finance pour avoir pris la peine de corriger notre français, Robert Stabl pour avoir calculé les exemples dans le système ML et Thomas Ramke et Robert Stabl pour avoir tapé le manuscrit.

Bibliographie

[Amadio, Longo 87] R. AMADIO, G. LONGO : *Type-free compiling of parametric Types.* In : M. Wirsing (ed.) : Formal Description of Programming Concepts III. Amsterdam : North-Holland, 1987, 377-396.

[Bauer, Wössner 82] F. L. BAUER, H. WÖSSNER : *Algorithmic Language and Program Development.* Berlin : Springer, 1982.

[Boyer, Moore 77] R. S. BOYER, J. S. MOORE : *A lemma driven automatic theorem prover for recursive function theory.* Proc. 5th Int. Joint Conference on Artifical Intelligence, Cambridge, Mass., 1977, 511-519.

[Burstall 77] R. M. BURSTALL : *Design considerations for a functional programming language.* Infotech State of the Art Conference : The Software Revolution, Copenhagen, 1977.

[Burstall, Darlington 77] R. M. Burstall, J. Darlington : *A transformation system for developing recursive programs.* J. ACM 24, 1977, 44-67.

[Burstall, Lampson 84] R. M. BURSTALL, B. LAMPSON : *A kernel language for abstract datatypes and modules.* In : G. Kahn, D. MacQueen, G. Plotkin (eds) : Semantics of Data Types C. Lecture Notes in Computer Science 173, Berlin : Springer, 1984.

[Burstall *et al.* 80] R. M. BURSTALL, D. MACQUEEN, D. SANNELLA : HOPE : *An experimental applicative language.* Proc. 1980 LISP Conference, Stanford, California, 136-143.

[Cousineau *et al.* 85] G. COUSINEAU, P. L. CURIEN and M. MAURRY : *The Categorical Abstract Machine.* In Functionnal Programming Languages and Computer Architecture, J. P. Jouannaud (ed), Lecture Notes in Computer Science 201, Berlin : Springer, 1985, 50-64.

[Gordon *et al.* 79] M. GORDON, R. MILNER, C. WADSWORTH : *Edinburgh LCF.* Lecture Notes in Computer Science 78, Berlin : Springer, 1979.

[Harper 86] R. HARPER : *Introduction to Standard ML.* Edinburgh Univ. Int. Report ECS-LFCS-86-14, 1986.

[Harper *et al.* 86] R. HARPER, D. MACQUEEN, R. MILNER : *Standard ML.* Edinburgh University, Int. Report ECS-LFCS-86-2, 1986.

[Harper *et al.* 87] R. HARPER, R. MILNER, M. TOFTE : *The semantics of Standard ML.* Draft report, Edinburgh University, 1987.

[Landin 66] P. Y. LANDIN : *The next 700 programming languages.* Comm. ACM 9, 1966, 157-166.

[MacQueen 85] D. MACQUEEN : *Modules for Standard ML.* Tech Report AT & T Bell Laboratories, 1985 ; also in [Harper *et al.* 86].

[Milner 78] R. MILNER : *A theory of type polymorphism in programming languages.* Journal of Computer and System Sciences 17, 1978, 348-375.

[Milner 83] R. MILNER : *A Proposal for Standard ML.* Report CSR-157-83, Computer Science Departement, University of Edinburgh (1983). Also Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, 1984, 184-197.

[Sannella, Tarlecki 85] D. SANNELLA, A. TARLECKI : *Program specification and development in Standard ML.* Proc. 12th ACM Symposium on Principles of Programming Languages, New Orleans, 1985, 67-77.

Né en 1948 à Bayreuth (RFA), Martin Wirsing a étudié les mathématiques à Paris et Munich. En 1976 il soutient une thèse de doctorat en logique. De 1975 à 1983 il est chercheur en informatique à l'Université Technique de Munich (TUM) et participe au projet CIP dont le thème est le développement par transformations successives de programmes à partir de spécifications formelles. Depuis 1983 il est professeur d'Informatique à l'Université de Passau où il est maintenant le Doyen de la Faculté de Mathématiques et d'Informatique. Actuellement Martin Wirsing est engagé dans deux projets ESPRIT de la Communauté Européenne : le projet METEOR qui vise à fournir une méthode intégrée de programmation et le projet DRAGON qui concerne la réutilisation du logiciel. Ses principaux thèmes de recherche sont la spécification formelle et le développement de programmes du point de vue des fondements théoriques (langages, méthodes et outils).

Martin Wirsing
Universität Passau Fakultät für Mathematik und Informatik
Postfach 2540 D - 8390 Passau.

Donald Sannella a obtenu son diplôme de « Bachelor of Science » à l'Université de Yale en 1977, son diplôme de « Master of Science » à l'Université de Californie de Berkeley en 1978 et son « Ph. D. » à l'Université d'Édimbourg en 1982, le tout en informatique.

Depuis 1985, il est « Lecturer » au département d'Informatique de l'Université d'Édimbourg après avoir été « Postdoctoral Rechearch Fellow » de 1982 à 1985 dans ce même département. Ses domaines d'intérêt de recherche concernent la spécification algébrique de programmes, les langages de programmation fonctionnelle et la démonstration automatique de théorèmes.

Don Sannella
University of Edinburgh Department of Computer Science
Mayfield Road Edinburgh EH9 3JZ.