Programmation Logique et Par Contraintes Avancée

Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes
treinen@pps.univ-paris-diderot.fr

Organisation du cours

- 11 semaines
- Cours/TP mardi 9h30 à 12h30. On commence normalement en salle de cours, puis on passe en salle de TP.

Organisation du cours

- Examen : normalement mardi 24 mars (2h)
- Contrôle continue : TP noté dans la deuxième moitié du semestre
- Note : 50% CC + 50% Examen
- http://www.pps.univ-paris-diderot.fr/~treinen/teaching/plpc/
- Des copies des transparents seront distribuées en cours.

Plan du module

- Oz : un langage multi-paradigme
- Programmation concurrente par flot de données
- Programmation logique en Oz (un peu)
- Programmation de contraintes : techniques avancées pour la résolution de problèmes combinatoires

Pre-requis du module

- Programmation fonctionnelle (OCaml ou Haskell ou Lisp ou Python ou ...)
  Style de programmation prévalent en Oz.
  Par exemple : Pattern matching, fonctions d'ordre supérieur comme `map`.
- Notions de base de la logique du premier ordre.
  Le modèle de la mémoire en Oz est basé sur la notion de *contraintes*
- Il n'est pas nécessaire de connaitre le langage Prolog.

Contenu chapitre 1

Un petit rappel : Prolog

- Abréviation de *programmation en logique*
- Développé au début des années 70 indépendamment par les équipes d'Alain Colmerauer (Marseille) et Robert Kowalski (Edinbourgh).
- Paradigme de la programmation *déclarative* : On utilise une logique pour *déclarer* la sémantique souhaitée du programme, puis le compilateur va se débrouiller pour trouver un moyen de l'exécuter efficacement (c'était au moins l'idée).
- Un programme Prolog définit des *prédicats*.
- Données : termes (éventuellement avec des variables), entiers.

Exemple d'un programme Prolog

Le prédicat $append(X, Y, Z)$ doit être vraie ssi Z est la concaténation des deux listes X et Y.

```
append([],Y,Y).
append([H|X],Y,[H|Z]) :- append(X,Y,Z)
```

À lire comme :

$$\begin{aligned} \forall Y: &\quad append([], Y, Y) \\ \forall H, X, Y, Z: &\quad append(X, Y, Z) \Rightarrow append([H|X], Y, [H|Z]) \end{aligned}$$

Ingrédients de la sémantique opérationnelle

- Unification (vient de la logique du premier ordre) : elle explique comment résoudre des équations entre termes.
- Résolution (également de la logique du premier ordre, mais ici on a seulement besoin d'un cas particulier) : elle explique comment appliquer la définition d'un prédicat, en utilisant l'unification.
- Arbre de recherche et retour en arrière (angl. : *backtracking*) : technique de programmation classique pour les problèmes combinatoires. Les interpréteurs Prolog utilisent une implémentation très astucieuse (Warren Abstract Machine).

Unification

- Inventée par Jacques Herbrand, réinventée par John Alan Robinson (celui avec la résolution).
- Permet de résoudre un système équations entre termes symboliques.
- Exemple : $f(x, a) = f(b, y)$:
  Solution $x = a, y = b$.
- Exemple : $f(x, a) = f(y, b)$:
  pas de solution !

Unification

Exemple : $f(x, g(b)) = f(g(y), z)$:

- Une solution est : $x = g(b), y = b, z = g(b)$
- Une autre solution est : $x = g(g(b)), y = g(b), z = g(b)$
- *La solution la plus générale* : $x = g(y), z = g(b)$.
- En anglais : *most general unifier (mgu)*.
- Le mgu est unique quand il existe (à des équations entre variables près).
- Le mgu ne contient que des variables qui paraissent dans les équations de départ.
- Toute solution est une instance du mgu.

Unification

Exemple : $f(a, f(a, x)) = f(a, x)$

- Est-ce qu'il y a une solution ?
- Si on permet des arbres infinis : oui !
  $x = f(a, f(a, f(a, f(a, \ldots))))$
- Si on ne permet que des arbres finis : non.
- Dans le cas des arbres finis seulement : l'algorithme d'unification doit exécuter un *test d'occurrence* (angl. : *occur check*), ce qui est souvent pas fait par les interpréteur Prolog.

Résolution

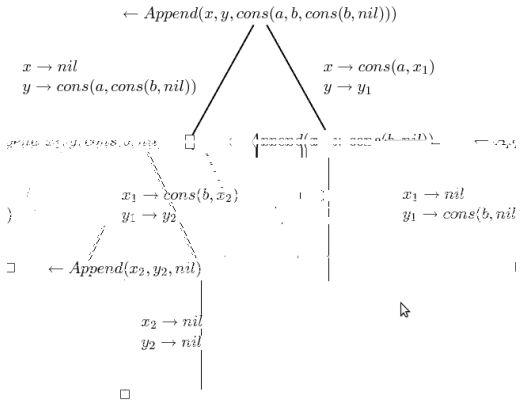
- Définie par J. Alan Robinson 1965 comme procédure de recherche de preuve dans la logique du premier ordre.
- Prolog : cas particulier car toutes les clauses sont Horn (les clauses du programmes, ainsi que la requête).
- *Requête* : les termes qu'on cherche à ≪ évaluer ≫ par rapport au programme (plus exactement : on cherche des valeurs de leurs variables qui constituent une solution).

L'arbre de recherche

- Étant donné le premier atome de la requête il y a en général plusieurs clauses du programme avec lesquels on peut résoudre.
- Il faut essayer toutes les possibilités pour ne pas perdre une solution potentielle.
- Pour chaque étape de résolution on peut avoir plusieurs alternatives, qui forment alors les arêtes dans un *arbre de recherche*.
- L'évaluation d'un programme Prolog consiste en un parcours de cet arbre de recherche qui est construit à la volée par l'interpréteur.

Exemple : le prédicat append

```
Append([],Y,Y).
Append([H|X],Y,[H|Z]) :- Append(X,Y,Z)
```



Une critique de Prolog (1)

- On a seulement droit à des clauses, tout calcul se fait pas résolution et construction d'un arbre de recherche.
- Or, dans un programme il y a normalement beaucoup de calcul déterministe qui ne nécessite pas de recherche.
- ☺ Il y a des techniques de compilation pour exécuter très efficacement un programme Prolog quand le calcul est déterministe (WAM).
- ☹ Il est quand même très pénible de programmer tout avec des prédicats et avec des clauses !

Une critique de Prolog (2)

- Pas de types ! !
- ☺ Cela permet de faire en Prolog quelques astuces, comme par exemple confondre des termes avec des atomes, écrire un interpréteur Prolog en quelques lignes de Prolog.
- ☹ Dans l'absence d'un système de typage (statique, c.-à-d. avant l'exécution du programme) il est beaucoup plus difficile de détecter les erreurs de programmation (comparer avec OCaml !).

Une critique de Prolog (3)

- Prolog ne fournit aucun moyen de *structuration* du programme en unités encapsulées :
  - des modules (sauf extensions dans certains compilateurs)
  - des objets
- ☹ En Prolog on est constamment obligé de violer toutes les bonnes principes du Génie Logiciel qui préconisent une structuration en modules, en classes, en types abstraits, etc.

Une critique de Prolog (4)

- La programmation déclarative est un idéal très noble ...
- ... dont (la pratique de) Prolog est très loin :
  - On ne peut pas éviter l'algorithmique, même pas en Prolog
  - cut et la ≪ négation ≫ de Prolog.
  - Prédicats ≪ méta-logiques ≫ qui permettent une introspection de termes (comme le prédicat *var* qui teste si un terme est une variable).

Extensions de Prolog

- ☹ Prolog ≪ classique ≫ ne connaît que les termes et l'unification comme mécanisme de résolution de contraintes.
- ☺ Les Prolog modernes (Prolog III, GNU Prolog, Yap, ...) intègrent aussi des autres systèmes de contraintes (équations linéaires, domaines finis, ...)
- ☹Toute fois cela n'est pas suffisant, il manque :
  - La possibilité de définir de nouveaux systèmes de contraintes
  - La possibilité de définir sa propre stratégie de recherche

Alors quoi faire avec la programmation logique ?

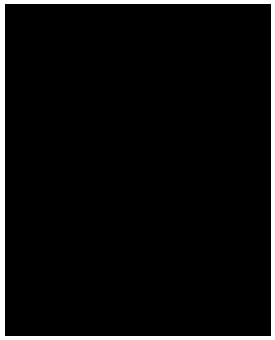
- La programmation par recherche est très utile pour certaines applications : optimisation, planification, ordonnancement, ...
- Prolog est un langage *minimaliste* pour la programmation par recherche qui manque des constructions qui existent dans d'autres langages de programmation.
- Il faut intégrer la programmation par recherche avec des autres concepts utils de langages de programmation,
- $\Rightarrow$ soit un langage multi-paradigme (Oz, Alice)
- $\Rightarrow$ soit une bibliothèque pour la fonctionnalité de recherche (GeCode, ILOG solver library)

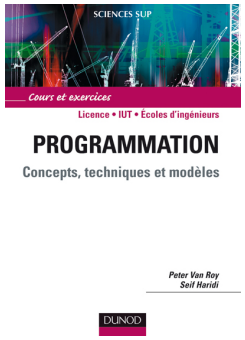
Oz et Mozart

- *Oz* est le nom du langage de programmation
- *Mozart* est le nom de l'implémentation du langage
- ☺ Mozart est un logiciel libre !
- Né en 1991 (Gert Smolka et. al.)
- Contributions essentielles de l'université de Saarbrücken (Allemagne), SICS (Suède), UC de Louvain (Belgique).
- Développement aujourd'hui piloté par un groupe international.

Le livre

- *Concepts, Techniques, and Models of Computer Programming*, par Peter Van Roy et Seif Haridi.
- Version française abrégée ! (ne parle pas des contraintes !)





Documention sur Mozart/Oz disponibles en ligne

- `http://www.mozart-oz.org` $\rightarrow$ Documentation, en particulier le *Tutorial* et *The Oz Base Environment*.
- Il y a les mêmes documents sur les PC de l'UFR à l'adresse `/usr/local/doc/mozart`

Oz

- Oz est un langage multi-paradigme : il permet
  - programmation fonctionnelle
  - programmation par objets
  - programmation logique et par contraintes
  - programmation concurrente *dataflow*
  - programmation distribuée
  - programmation des interfaces graphiques
  - ...

Oz

- La syntaxe de Oz est influencée par LISP :
  L'application d'une fonction (procédure, méthode) s'écrit

$$\{func \quad arg_1 \quad \ldots \quad arg_n\}$$

- La syntaxe permet des expressions composées, comme

$$\{f \quad a_1 \quad \{g \quad a_2 \quad a_3\}\}$$

- Typage *dynamique* (malheureusement) : des erreurs de typage sont détectées seulement pendant l'exécution.

L'environnement de programmation

- Basé sur GNU Emacs
- Lancez l'environnement par la commande oz, cela ouvre une session d'emacs avec une fenêtre d'édition (Oz) et une fenêtre (`*Oz Compiler*`) où le compilateur affiche ses messages.
- Quand on exécute un programme Oz, les affichages paraissent dans une autre fenêtre `*Oz Emulator*`.
- Utiliser la fonction *Show/Hide -> Emulator* du menu Oz pour basculer.

L'environnement de programmation

- Utiliser le menu Oz d'Emacs quand il est dans le mode oz
- Utiliser les raccourcie claviers
  (C-x : appuyez les touches `Ctrl` et x au même moment) :
  - `C-. C-l` Exécuter la ligne courante
  - `C-. C-r` Exécuter la région courante
  - `C-. C-p` Exécuter le paragraphe courant (délimité par des lignes vides)
  - `C-. C-b` Exécuter le tampon (buffer) courant

[Demo 1]

Déclaration de variables

`local X Y Z in S end`

déclare les variables X, Y, Z, leur portée est S.

`declare X Y Z in S`

déclaration ≪ ouverte ≫ : la portée de X, Y, Z est globale,leur portée n'est pas limitée (comme un `let x = ...;;` en OCaml).

Les noms des variables commencent toujours sur des lettres en MAJUSCULES.

[Demo 2]

Variables et types

- Variables sont à affectation *unique* : on peut leur donner une fois une valeur (comme `let x = ... in ...` en OCaml), mais cette valeur peut être un terme composé qui contient des variables (comme dans la programmation logique).
- Les *valeurs* (pas les variables !) portent des types, l'application d'une opération à des valeurs du mauvais type déclenche une erreur.
- Affectation d'une variable :
  `VARIABLE = valeur`

[Demo 3]

Les types

- Il y a des types primaires et de types secondaires
- Types primaires : Leur union couvre tout l'univers de valeurs
- Deux types primaires sont soit sous-type un de l'autre, soit ils sont disjoints.
- Les types primaires les plus importants (pour l'instant) :
  - `Number`, avec des sous-types `Int` et `Float`
  - `Record`, avec le sous-type `Tuple`
  - `Procedure`

Valeurs numériques

- une valeure numérique peut être entière (`Int`) ou flottante (`Float`). Les caractères (`Char`) sont un sous-type du type `Int`.
- le moins unaire s'écrit $\sim$
- Les flottants doivent être notés avec un point décimal :

$$\sim 3.141, \quad 4.5E3, \quad \sim 12.0e \sim 2$$

- pas de conversion automatique entre Int et Float.

Affectations

variable = *valeur*

Exemple :

```
local I F C in
   I = 5
   F = 5.5
   C = &t
   {Browse [I F C]}
end
```

Littéraux

sont des « mots » atomiques. Il y en a deux sortes :

- *atomes* : séquence de caractères alphanumériques, commencant sur une minuscule, ou une chaine arbitraire entre apostrophes ´
- *noms* : des mots uniquement, peuvent seulement être engendrés par la procédure `NewName`

Exemple d'atomes :

```
a   foo   '='   ':='   'OZ 3.0'   'Hello World'
```

[Demo 4]

Enregistrements (angl. : *Records*)

- Pour l'instant : seulement enregistrements « fermés » (tous les champs sont définis au même moment)
- Un record consiste en un *label*, et une séquence de paires d'une clefs et d'une valeur.
- On peut utilise la même clef dans des enregistrements différents (contrairement à OCaml) puisqu'il n'y a pas d'inférence de type.

Example :

```
tree(key: I value: Y left: LT right: RT)
```

Tuples

Si les clefs d'un enregistrement sont $1, 2, \ldots, n$ alors on n'est pas obligé de les écrire explicitement. Ainsi

```
tree(I Y LT RT)
```

est la même chose que

```
tree(1:I 2:Y 3:LT 4:RT)
```

Opérations sur des enregistrements

- Sélectionner la valeur de x avec la clef c : `x.c`
- Arité de x (liste des clefs de x) : `{Arity x}`
- Label de x : `{Label x}`
- Ajouter une paire à un enregistrement : `{AdjoinAt R F X}` donne $R1$ avec en plus la paire (F,X). Quand 1 a déjà la clef F alors le résultat est $R1$, sauf que la valeur à la clef F est X.
- Il y a des variantes (joindre deux enregistrements, joindre une liste de paires à un enregistrement, ...). Voir la doc.

Listes

Les listes sont un type secondaire.
Deux syntaxes équivalentes :

```
1|2|3|nil
[1 2 3]
```

Ceci n'est qu'une abréviation pour des tuples imbriqués, donc une autre façon d'écrire la même liste est

```
'|'(1 '|'(2 '|' (3 nil)))
```

Chaînes de caractères

Abréviation pour des *listes* de caractères. Les trois valeurs suivantes sont égales :

```
"OZ 3.0"
[&O &Z &  &3 &. &0]
[79 90 32 51 46 48]
```

Déclaration de procédures

```
local Max X Y Z in
  proc {Max X Y Z}
    if X >= Y then Z = X else Z = Y end
  end
  X = 5
  Y = 10
  {Max X Y Z} {Browse Z}
end
```

Procédures

- `proc {P X1 ... Xn} ... end`
  est une instruction qui lie l'identificateur P à une procédure.
- Définition de procédures : liaison statique (angl. *lexical scoping*)
- Les procédures sont des valeurs de première classe, on peut donc les passer comme argument à une autre procédure

Filtrage par motif

Une instruction peut aussi être un filtrage par motif :

```
case E of Pattern_1 then S1
    [] Pattern_2 then S2
    [] ...
    else S end
```

La partie `else` est optionnelle.
Les variables dans le motif sont des variables locales, sauf si précédée par `!`.

Exemple

```
declare Length in
proc {Length L Result}
   case L
   of nil then Result=0
   [] H|R then local Restlength in
                  {Length R Restlength}
                  Result=Restlength+1
       end
   end
end
```