Un peu de programmation logique

—

Unification

Programmation logique

- origines: années 70, étude des preuves constructives en logique

  une fonction peut être vue comme une preuve d'une propriété de la forme $\forall \vec{x}.\exists y.P(\vec{x}, y)$

- Prolog est un *démonstrateur de théorèmes* ($\neq$ Coq)

  un programme $\equiv$ des axiomes et des règles (clauses)

```
addright(nil,X,cons(X,nil)).
addright(cons(A,B),X,cons(A,Z)) :- addright(B,X,Z)
rev(nil,nil).
rev(cons(X,Y),Z) :- rev(Y,W), addright(W,X,Z)
```

le :- se lit $\Leftarrow$

Prolog, "exécutions" d'un programme

```
addright(nil,X,cons(X,nil)).
addright(cons(A,B),X,cons(A,Z)) :- addright(B,X,Z)
rev(nil,nil).
rev(cons(X,Y),Z) :- rev(Y,W), addright(W,X,Z)
```

- on saisit une *requête*

```
rev([1,2,3],[3,2,1])   → true
rev([9,1,1],Y)         → Y=[1,1,9]
rev(X,[32,52])         → X=[52,32]
rev([1,2,A],[3,2,B])   → A=3, B=1
rev([1,2,C],[3,3,D])   → ∅
```

- pas vraiment de notion d'entrées/sorties
- programmation *très déclarative*
- résultat d'un programme: ensemble de solutions de la requête
  plus petit ensemble de faits satisfaisant axiomes et règles

Prolog – hello world

- famille

```
pere(X,Z) :- pere(X,Y), frere(Y,Z).
grandpere(X,Z) :- pere(X,Y), pere(Y,Z).
fils(X,Y) :- pere(Y,X).
pere(X,Y) :- fils(Y,X).
cousin(X,Y) :- pere(T,X), frere(T,U), pere(U,Y).

pere(jacques, arthur).
fils(mathieu, louis).
frere(louis, roger).
pere(maurice,roger).

grandpere(X,mathieu).
frere(jacques,Y).
```

- comment ça marche?

Un programme Prolog, formellement

- des axiomes et des règles:

  $c(t_1, \ldots, t_k).$
  $c(t_1, \ldots, t_k) \ :- \ p_1(u_{11}, \ldots, u_{1k_1}), \ldots, p_n(u_{n1}, \ldots, u_{nk_n}).$

  - les $c, p_1, \ldots, p_n$ sont des *prédicats*
  - les t_i, u_{ij} sont des *termes*
  - un terme est:
    - soit une *variable* *(majuscule)*
    - soit construit en appliquant un prédicat à un certain nombre (éventuellement nul) de (sous-)termes
- exemples: `nil` `Y` `rev(nil,nil)` `rev(cons(a,nil),X)`
- un prédicat peut être vu comme un constructeur

  il a une *arité*: on ne veut pas de `nil(rev,X)`, `rev(X,Y,Z)`, `cons(nil)`, `g(f(a,b), f(X))`

Exécution d'un programme Prolog

$c(t_1, \ldots, t_k).$
$c(t_1, \ldots, t_k) \ :- \ p_1(u_{11}, \ldots, u_{1k_1}), \ldots, p_n(u_{n1}, \ldots, u_{nk_n}).$

- lancer un programme Prolog: *requète* $q(v_1, \ldots, v_m).$
- on cherche si un axiome convient**convient**
- on cherche si une règle peut être appliquée
  (i.e. si sa conclusion convient)**convient**)

  on met face à face $c(t_1, \ldots, t_k)$ et $q(v_1, \ldots, v_m)$
  on **unifie** les deux termes
- unification: Robinson 1965, preuve de théorèmes

  "si on a montré $K \vee p$ et $K' \vee \neg p$, alors on peut déduire $K \vee K'$"

Unification – exemples

- unifier, c'est trouver un *deal*

conclusion	requète	solution
$c(X, Y)$	$c(f(a), g(a, b))$	$X \mapsto f(a), Y \mapsto g(a, b)$
$c(f(a), g(a, b))$	$c(X, Y)$	$X \mapsto f(a), Y \mapsto g(a, b)$
$f(T, c(e), d)$	$f(a, X, d)$	$T \mapsto a, X \mapsto c(e)$
$f(a, b, X)$	$f(Y, c, d)$	échec
$c(X, Y)$	$c(Z, T)$	$X \mapsto Z, Y \mapsto T$ (ou $T \mapsto Y$, ou $Z \mapsto X$…)
$c(X, a, b)$	$c(c, X, b)$	problème mal posé
$f(a)$	$f(a)$	oui
$p(X, c, X)$	$p(a, Y, a)$	$X \mapsto a, Y \mapsto c$
$f(X, g(X))$	$f(Z, Z)$	échec

- on fait de la *"résolution d'équations symboliques"*
  - *le principe: "coller" deux termes l'un en face de l'autre en attrapant le dur (prédicats) avec le mou (variables)*
  - exploration des deux arbres en parallèle, en engendrant une *substitution*
    - fonction partielle des variables vers les termes
    - souvent représentée comme une liste de couples
  - $f(X, g(a, d))$ et $f(h(c, Z), g(a, Y))$: ok
  - $f(g(X))$ et $f(h(Y))$ ne peuvent pas être unifiés

Unification, algorithme

- on travaille avec un *problème* P (ensemble d'équations) et une *solution* courante S
- si P est vide, **succès**, on renvoie S;
  sinon on pioche une équation; plusieurs cas:
  - **décomposition** $c(s_1, \ldots s_k) \stackrel{?}{=} c(t_1, \ldots t_k)$
    on ajoute les équations $s_i \stackrel{?}{=} t_i$ à P
  - **conflit** $c(s_1, \ldots, s_k) \stackrel{?}{=} d(u_1, \ldots, u_n)$ **échec**
  - **trivial** $t \stackrel{?}{=} t$, on continue avec P
  - **élimination de variable** $X \stackrel{?}{=} t$
    on continue avec $P_{[X \mapsto t]}, S_{[X \mapsto t]}$ (on note $X \mapsto t$) **si** $X \notin Vars(t)$
  - **cyclicité** $X \stackrel{?}{=} t$ **échec** si $X \in Vars(t), t \neq X$
  - **orientation** $t \stackrel{?}{=} X$, où t n'est pas une variable
    on ajoute $X \stackrel{?}{=} t$ à P
- pour unifier, on part de $P = \{c \stackrel{?}{=} q\}$, $S = \emptyset$

Algorithme d'unification, propriétés

- **décomposition** $c(s_1,\dots s_k) \stackrel{?}{=} c(t_1,\dots t_k)$
  on ajoute les équations $s_i \stackrel{?}{=} t_i$ à P
- **conflit** $c(s_1,\dots,s_k) \stackrel{?}{=} d(u_1,\dots,u_n)$ échec
- **trivial** $t \stackrel{?}{=} t$, on continue avec P
- **élimination de variable** $X \stackrel{?}{=} t$
  on continue avec $P_{[X\leftarrow t]}, S_{[X\leftarrow t]}$ **si** $X \notin \mathit{Vars}(t)$
- **cyclicité** $X \stackrel{?}{=} t$ échec si $X \in \mathit{Vars}(t), t \neq X$
- **orientation** $t \stackrel{?}{=} X$, où t n'est pas une variable
  on ajoute $X \stackrel{?}{=} t$ à P

- quand on applique une substitution $[X \leftarrow t]$, la variable X disparaît
- le processus termine (test de cyclicité)
- lorsqu'on élimine une variable, on *préserve l'ensemble des solutions*

Algorithme d'unification – propriétés

- intéressons-nous à une exécution qui réussit;
  elle est de la forme $\{t \stackrel{?}{=} u\}, \emptyset \Rightarrow \dots \Rightarrow \emptyset, S$
  - c'est déterministe (confluence: on pioche comme on veut)
  - correction: si $P, \emptyset \Rightarrow \emptyset, S$, alors S unifie les équations de P,
    i.e. $\forall \{t \stackrel{?}{=} t'\} \in P. S(t) = S(t')$
    - la substitution S est d'ailleurs idempotente
      (si $\{x \leftarrow t\} \in S$, alors x n'a pas d'autre occurrence dans S)
  - complétude: si σ unifie les équations de P, alors *tout calcul*
    $P, \emptyset \Rightarrow \dots$ se termine sur un S qui généralise σ
    (i.e. $\exists \eta. \sigma = S\eta$)
- donc $\{t \stackrel{?}{=} u\}, \emptyset \Rightarrow \dots \Rightarrow$ `erreur` ssi $t \stackrel{?}{=} u$ n'a a pas de solution
- conclusion: la procédure d'unification calcule un *unificateur le plus général* lorsqu'il existe

  (en d'autres termes, S décrit toutes les solutions au problème de départ)

Unification – complexité

- cet algorithme peut être exponentiel en espace et en temps
- représenter les données qu'on manipule (termes, substitutions) de manière efficace permet d'obtenir des algorithmes plus efficaces (temps $\mathcal{O}(n^2)$, espace $\mathcal{O}(n)$ – on utilise des DAGs *(directed acyclic graphs)*, des ensembles de Tarjan,...)
  ...et même des algorithmes en temps linéaire

- remarque: le filtrage en Caml c'est de la "demi-unification"
  - filtrage: le *motif* mange la *valeur* *(toutes les variables à gauche)*
  - unification: les deux acteurs (conclusion et requête) sont à égalité

Prolog – stratégie de recherche

$c(t_1,\dots,t_k).$
$c(t_1,\dots,t_k)$:- $p_1(u_{11},\dots,u_{1k_1}),\dots,p_n(u_{n1},\dots,u_{nk_n}).$ $\quad q(v_1,\dots,v_m).$

- si $q(v_1,\dots,v_m)$ peut être unifié avec un axiome, on renvoie la substitution résultante *(cf. les* `X = a, Y = b`*)*
- sinon on unifie $q(v_1,\dots,v_m)$ avec la conclusion d'une règle, et on ajoute les *sous-buts engendrés*, en fabriquant progressivement la substitution solution
- *backtracking:* si plusieurs règles appliquables, parcourir dans l'ordre en cas d'échec
- exécuter un programme $\leftrightarrow$ construire un arbre de preuve
- stratégie de recherche: en profondeur d'abord
  du coup, des divergences peuvent "cacher" des solutions

DÉMO `demo_gprolog.pl`